IMPLEMENTATION: Data Structures
Relational Information Systems Chapter 1.2
(Revised 99/9)

Direct Access Files

November 14, 1999

The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

While it is impossible to improve on the worst case performance of $\mathcal{O}(\log N)$ for a search for one record in N using *comparisons*, we can *expect* a search to cost less if we do not use comparisons. This is the secret of direct access: instead of comparing the search key to values already in the file, use the search key to *calculate* the address where that record is to be found.

Unfortunately, hardly any such calculation is completely accurate, and *collisions* arise in which two different search keys are mapped to the same address. So direct access techniques consist of two steps: the *key-to-address transformation*, and the *collision resolution*.

We look at two kinds of key-to-address transformation, *hash* functions, and *tidy* functions. We will consider collision resolution in the context of hash functions, because it is easier to understand collision resolution once we have a particular key-to-address transformation in mind.

Before we start, we should look at the problems inherent in key-to-address transformations. Consider the nine-digit social insurance numbers (SINs) used in a country of 10^7 people, and suppose that, on secondary storage, blocks will hold 100 records. The blocks will be the addresses, and so there will be 10^5 of them for the 10^9 possible keys. The objective is to map all these key values to addresses in such a way as to minimize collisions.

Figure 1 shows three key-to address transformations for this example. The first is straight scaling: divide each SIN by 10^4, to reduce the range from 10^9 values to 10^5 block addresses.

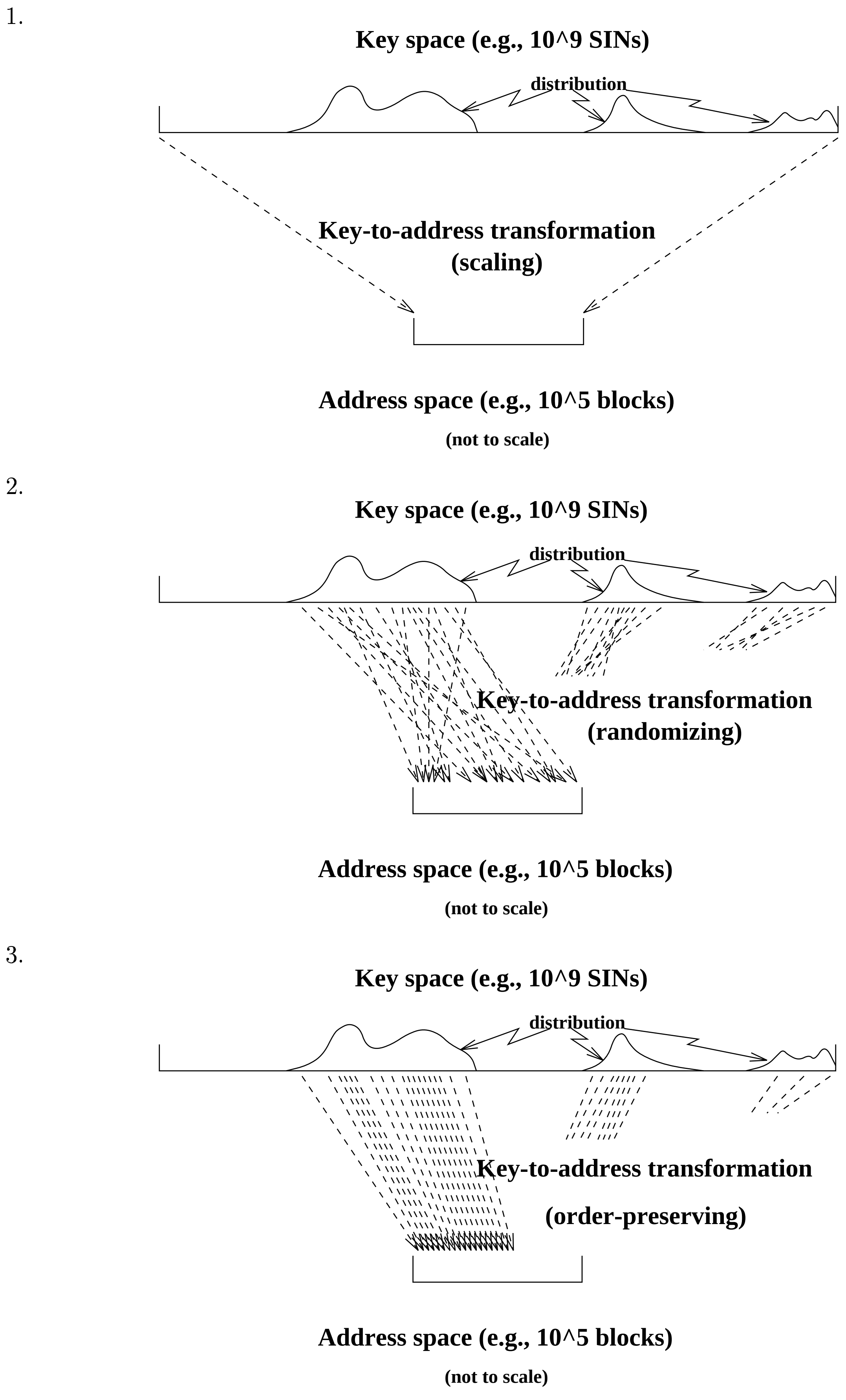

1.
Key space (e.g., 10^9 SINs)
distribution
Key-to-address transformation
(scaling)
Address space (e.g., 10^5 blocks)
(not to scale)
2.
Key space (e.g., 10^9 SINs)
distribution
Key-to-address transformation
(randomizing)
Address space (e.g., 10^5 blocks)
(not to scale)
3.
Key space (e.g., 10^9 SINs)
distribution
Key-to-address transformation
(order-preserving)
Address space (e.g., 10^5 blocks)
(not to scale)

Figure 1: Key-to-Address Transformations

This simple approach would be fine if each SIN were a multiple of 10^4: every address would be occupied and every SIN would have a unique address. However, data (including SINs) are usually clustered in irregular ways. This is shown by an (artificial) *distribution* curve in the key space. The scaling transformation would just squeeze these clumps by the factor of 10^4, leaving a lot of empty space and a lot of collisions in the address space.

One way of smoothing out the clumps while doing the transformation is to attempt to generate a *uniform* distribution in the address space. This can be done by a randomizing transformation, which is shown second in figure 1. This method is known as *hashing.*

Hashing unfortunately does not preserve order, a useful requirement for high activity, particularly range queries. So figure 1 finally shows an order-preserving transformation. This is called *tidying.*

1 Hashing

The goal of hashing is to spread the keys uniformly over the address space. It is difficult enough to achieve uniform distributions with a single random number generator; doing it for a set of keys with an arbitrary distribution of its own is particularly challenging. The choice of hash function requires a blend of theoretical considerations and experience with actual files. For primary memory, hash functions have been extensively studied ([Knu73], sect. 6.4). For secondary memory, requirements are less stringent: speed is of less concern, because calculating an address is negligible in comparison with seeking the block so designated; and the ability of the hash function to resolve collisions is also less important, because the statistics of large block sizes reduces the severity of collisions when they happen.

So we will discuss only two of the most successful hash functions, *division-remainder*, and *multiplicative.* Both assume the hash key is an integer: this is not a limitation, because any key can be represented as an integer after a little manipulation (such as exclusive-oring the bytes of the hash key, or treating letters as integers to base 26).

The division-remainder method is the basis for random number generators, and so we hope it will also smooth a set of keys into a uniform distribution. The hash function,

$$h(k) = k \bmod n$$

generates n addresses, $0 \ldots n-1$, and works best for a prime value for n. A prime number is chosen because a number with factors, particularly small ones, tends to preserve the regularities that are found in most data, thus causing collisions. For instance, if n is a multiple of 2, $k \bmod n$ will map even ks into even addresses and odd ks into odd addresses: an imbalance of evenness and oddness among the keys will appear among the addresses, with more collisions for whichever class has more keys. If n is a multiple of f, the keys and addresses will be partitioned into f classes, with no crossover, and with similar problems in the case of imbalances.

Multiplicative hashing was originally invented for computers with hardware multipliers but no division hardware, a common early design. It remains useful when file sizes are powers of 2, which is the other extreme from files with a prime number of blocks. The hash function,

$$h(k) = ((Ak) \bmod w) \mid m$$

returns the first m bits of $(Ak) \bmod w$, which is the first m bits of the less significant word of Ak, since w is meant to be the word size, e.g., $w = 2^{32}$. A is a large, predefined number, advisedly related to the "golden ratio", i.e., near $(\sqrt{5}-1)w/2$ (see [Knu73], sect. 6.4). (This is written with the mod operator for a semblance of formality, but of course it is *computed* by bit masking. The product, Ak, of two numbers occupying one word each needs two words, say lg w bits each, and all we do is return the most significant m bits of the less significant word.)

We will also discuss only two methods of collision resolution, *linear probing* and *separate chaining.* In primary memory, these represent two extremes of performance, with linear probing being simple to implement but having relatively high cost, especially when the file is quite full. On secondary storage, however, the large blocksize reduces the severity of the collisions and the performance of two methods is much more close than in RAM.

Linear probing simply backs along the file, one block at a time, until the record is found (or, when inserting, a place is found to put it). If a record is not found on its home block, at address $h(k)$, then block $h(k) - 1$ is tried, then block $h(k) - 2$, and so on. (The reason for decrementing instead of incrementing is also historical, and may be changed: register hardware is able to decrement, with a test for 0.) The decrementing is cyclical: when block 0 is reached, the next block is $n - 1$.

A problem arises for linear probing when the file gets full, or almost full. The overflow probing can work its way all, or almost all, the way around the file, accessing n blocks in the worst case. Thus, the collision resolution gives hashing linear ($\mathcal{O}(n)$) complexity, even though the *expected* cost is order 1.

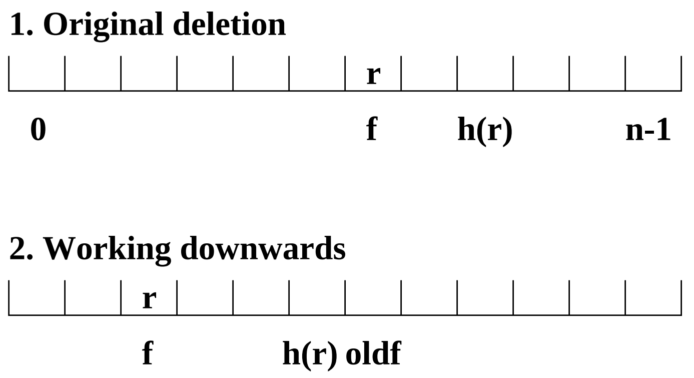
Separate chaining is more elaborate. In each block, pointers are maintained to (usually separate) overflow blocks, and a collision on the home block causes the pointer chain to be followed until the record is found, or can be placed. The coding makes a straightforward pointer, or double-pointer, chain. But it is more involved than linear probing, which requires no additional data structures.

Even separate chaining suffers from $\mathcal{O}(n)$ complexity, because every key might happen to hash to the same home block. This happens to linear probing, too, but is much less likely to arise than the case of probes to the whole file, discussed above.

With these two methods of collision resolution, we have six algorithms: three each for search, insertion, and deletion.

- Algorithm SS (Hash search with separate chaining)
- Algorithm SI (Hash insert with separate chaining)
- Algorithm SD (Hash delete with separate chaining)
- Algorithm LS (Hash search with linear probing)
- Algorithm LI (Hash insert with linear probing)
- Algorithm LD (Hash delete with linear probing)

These are all straightforward except for algorithm LD, and we leave them as exercises. In algorithm LD (figure 2), once the target record has been deleted, we must move up all overflow records in the chain for its home block, as long as we encounter full blocks on our way down from the home block. This is because deletion leaves an empty location, which is taken by the search and insert algorithms as indicating the end of the overflow chain. We must be sure we have *really* come to the end of the overflow chain. As step LD2 works down the blocks below the home block, decrementing f on its way, it is not restricted to moving up only records that hash to the original home block. It is free to move up any record, r, on current block f, as long as $h(r)$, its home block, does not lie between f and the block, *oldf*,

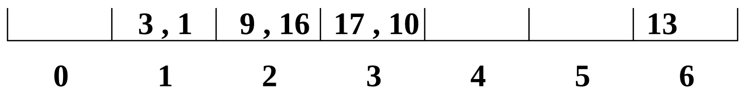


```
LD1 (Remove record) $\ell$ <− loc($r$); $oldf$ <−$f$; mark $\ell$ on $oldf$ empty

LD2 (Move later records up)
    If block $f$ not full (apart from deletion, if any, just made by LD1), stop.
    $f$ <− if $f = 0$ then $n - 1$ else $f - 1$.
    If $f$ = original home block, stop. /* else full file can thrash */
    For each record, $r$, on block $f$
        if ncycle($f, h(r)$, $oldf$)
        /* $h(r)$ not cyclically between $f$, $oldf$ or = $oldf$; i.e., $h(r)$ at or beyond
    $oldf$ */
            then {copy $r$ to $\ell$ on $oldf$; goto LD1}.
    Goto LD2.
```

Figure 2: Algorithm LD: Delete Record r Found on Block f

Example: delete 10 from



f	3	2		1	0
$h(r)$		2	2	3	
oldf		3			1

Figure 3: Algorithm LD: Delete Record 10

where the empty location has been left. For such a record lies on an overflow chain which includes both f and *oldf*.

Figure 3 shows the example of deleting 10 from a 7-block file in which only one record, 3, overflows, with a chainlength of 2 extra probes. The sequence of values of f, and, correspondingly, $h(r)$ and *oldf*, are shown in the table. The end result is that `3` is moved to where `10` was.

If block 4 had been full (say, with `18` and `25`), and block 1 had `4` instead of `3`, the `4` would have moved up to replace `10`, even though it is not on the chain from the block (3) where the original deletion of `10` had been done.

1.1 Volatility: Virtual and Linear Hashing

Algorithms SI, SD, LI, and LD permit a hash file to grow and shrink, but this volatility is limited by the number of addresses, n, specified in the hash function. To grow beyond this limit, the file requires another hash function. An unsatisfactory way of using a second hash function would be to rehash the entire file once all addresses allowed by the original hash function are full (or once some specified load factor, α_0, has been reached). Such a complete rehash is tantamount to changing the access structure, and thus violates the strategy we already laid down for dynamic files, *preserve the access method* (see section 1.1 of chapter 1.2, on B-trees).

The tactic we also discussed at that time, *split the blocks on growth*, also helps us here. If we had a hash function, $h(k) = k \bmod n$, and a second hash function which allows the file to double in size, $h'(k) = k \bmod 2n$, then any key, k, hashed by $h(k)$ to an address, a, will be hashed by $h'(k)$ to either of a or $a + n$. Thus, rehashing block a using $h'(k)$ as a new hash function, will split a into a and $a + n$. Similarly, on deletion, rehashing blocks a and $a + n$ with $h(k)$ would merge them into block a.

The search tactic is thus to use $h(k)$, somehow determine if the block a has been split, and if so try again with $h'(k)$.

For continued growth, we will eventually need a further hash function, say $h''(k) = k \bmod 4n$, which allows the file to double again. In order that matters not get out of hand, we impose the rule that *no more than two hash functions may be operative at any one time.* That is, we must split every block in the file, and so complete the doubling of the file size, and retire the first hash function, before we may introduce a third hash function.

Before we discuss systematic ways of doing all this, we should examine the criteria under which we split a block. One such criterion could be: *when the block overflows.* However, any one block may have very bad statistics and overflow repeatedly, or may not be split in half by the new hash function, while the new blocks created by splitting may reduce the load factor, α, to an unacceptably low percentage. Furthermore, if this split criterion wants us to split the same block twice in a row, we must not because of the rule limiting us to two hash

- Initially, $j = 0, p = 0, n = \nu$ and, for any k, $h_{-1}(k) = -1$.

LHI1 (Hash.) $a \leftarrow$ if $p = 0$ or $h_{j-1}(k) < p$ then $h_j(k)$ else $h_{j-1}(k)$. If not already there, store r in block a or as an overflow to block a. $N++$.

LHI2 (Split disallowed.) If $N/(n+1)b < \alpha_0$ then terminate.

LHI3 (Allocate.) If $p = 0$ then $j++$. Allocate block $p + 2^{j-1}\nu$. $n++$.

LHI4 (Split.) Rehash block p, including overflows, using h_j.

LHI5 (Increment pointer.) $p \leftarrow$ if $p \geq 2^{j-1}\nu$ then 0 else $p + 1$.

Figure 4: Algorithm LHI: Linear Hash Insert. Insert Record r with Key k

functions at a time; so we may have to split some other block.

We discuss two split criteria which use global considerations instead of being tied to particular blocks. One depends on the load factor, α, defined in section 2 of chapter 1.2. The other depends on the *probe factor*, $\pi = \frac{\text{total no. probes}}{N}$ (or on the *optimistic probe factor*, $\pi_{\text{op}} = 1 + \frac{\text{total overflows}}{N}$, which assumes that any record which overflows its home block can be found in exactly one further probe).

- (Lazy): split if $\pi > \pi_0$;
- (Greedy): split unless this makes $\alpha < \alpha_0$.

The first tests the probe factor every time an insertion causes a new overflow, and if it now exceeds some threshold, π_0, allows a split. The second tries to split after every insertion, unless this would make the load factor drop below a predetermined threshold, α_0.

The first is difficult to evaluate. We must keep track of the total number of overflows to date, and the total number of records, and modify these on each insertion and on each split. Moreover, it has the difficulties encountered above: splitting a block may not affect the probe factor at all, if its statistics are bad, but yet the load factor will be reduced in an uncontrolled way.

The second is easy to evaluate. $\alpha = \frac{N}{nb}$. After an insertion it will be $\frac{N+1}{nb}$, and after insertion and split it will be $\frac{N+1}{(n+1)b}$. Furthermore, it affects α exactly as the appropriate one of these expressions says, and it is as likely to have a beneficial effect on the probe factor as the lazy criterion.

We prefer the greedy split criterion. But neither one dictates the particular block to be split. So we can just split them in some systematic way, such as block 0 first, then block 1, and so on up to block $n - 1$. This is called *linear hashing*, and requires us to store, in addition to the data, only a pointer to the next block to be split. Figure 4 shows the algorithm to insert using the greedy criterion. Here, p is the number of the next block to be split, and is (re)set to 0 as soon as the file size has doubled and the splitting cycle is starting again. ν is the original file size. The hash function, $h_j(k) = k \bmod 2^j\nu$, is an anomaly, using division-remainder based on an address size with a power of 2 as a factor. The powers of 2 permit the doubling, and the division-remainder permits the splitting.

Figure 5 shows an example of linear hash growth, with a blocksize $b = 2$ and inserting the keys `3, 7, 2, 5, 6, 11, 4, 1, 9` in that order.

n	j	p	k	$h_j(k)$	$h_{j-1}(k)$	a			
1	0	0	3	0		0	$_0\boxed{3}$	$1/4 < 0.8$	no split
			7	0		0	$_0\boxed{3,7}$	$2/4 < 0.8$	no split
			2	0		0	$_0\boxed{3,7}$ 2	$3/4 < 0.8$	no split
			5	0		0	$_0\boxed{3,7}$ 2,5	$4/4 \geq 0.8$	split!
2	1	0					$_0\boxed{2}$ $_1\boxed{3,7}$ 5		
			6	0		0	$_0\boxed{2,6}$ $_1\boxed{3,7}$ 5	$5/6 \geq 0.8$	split!
3	2	1					$_0\boxed{}$ $_1\boxed{3,7}$ 5 $_2\boxed{2,6}$		
			11	3	1	1	$_0\boxed{}$ $_1\boxed{3,7}$ 5,11 $_1\boxed{2,6}$	$6/8 < 0.8$	no split
			4	0	0	0	$_0\boxed{4}$ $_1\boxed{3,7}$ 5,11 $_1\boxed{2,6}$	$7/8 \geq 0.8$	split!
4		0					$_0\boxed{4}$ $_1\boxed{5}$ $_2\boxed{2,6}$ $_3\boxed{3,7}$ 11		
			1	1	1	1	$_0\boxed{4}$ $_1\boxed{5,1}$ $_2\boxed{2,6}$ $_3\boxed{3,7}$ 11	$8/10 \geq 0.8$	split!
5	3	1					$_0\boxed{}$ $_1\boxed{5,1}$ $_2\boxed{2,6}$ $_3\boxed{3,7}$ 11 $_4\boxed{4}$		
			9	1	1	1	$_0\boxed{}$ $_1\boxed{5,1}$ 9 $_2\boxed{2,6}$ $_3\boxed{3,7}$ 11 $_4\boxed{4}$	$9/12 < 0.8$	no split

Figure 5: Example for Algorithm LHI

- Initially, j, p, and n are as they were left by the last call to LHI or LHD.

LHD1 (Hash.) $a \leftarrow$ if $p = 0$ or $h_{j-1}(k) < p$ then $h_j(k)$ else $h_{j-1}(k)$. If found, remove r from block a or from the overflows to block a. $N - -$.

LHD2 (Merge disallowed.) If $N/nb \geq \alpha_0$ then terminate.

LHD3 (Decrement pointer.) $p \leftarrow$ if $p = 0$ then $2^{j-1}\nu - 1$ else $p - 1$.

LHD4 (Merge.) Rehash blocks p and $p + 2^{j-1}\nu$ using h_{j-1}.

LHD5 (Deallocate.) Deallocate block $p + 2^{j-1}\nu$. $n - -$. If $p = 0$ then $j - -$.

Figure 6: Algorithm LHD: Linear Hash Delete. Delete Record r with Key k

To delete, we reverse the process, merging instead of splitting. The merge criteria are the converse of the split criteria: greedy becomes lazy, and vice versa.

- (Greedy): merge unless this makes $\pi > \pi_0$;
- (Lazy): merge if $\alpha < \alpha_0$.

We are either trying to keep π below (but near) π_0, or α above (but near) α_0, as before. And, as before, the α criterion is easier and more effective. Figure 6 shows the merging algorithm under this criterion.

Figure 7 shows an example of linear hash shrinkage, with a blocksize $b = 2$ as before and deleting the keys `7, 6, 2, 1, 11` in that order.

Linear hashing [Lit80] was preceded by *virtual hashing*, in which a block is split not because its turn has come in a simple linear order, but because an insertion was being made in it when the split criterion was satisfied (still subject to the rule that no more than two

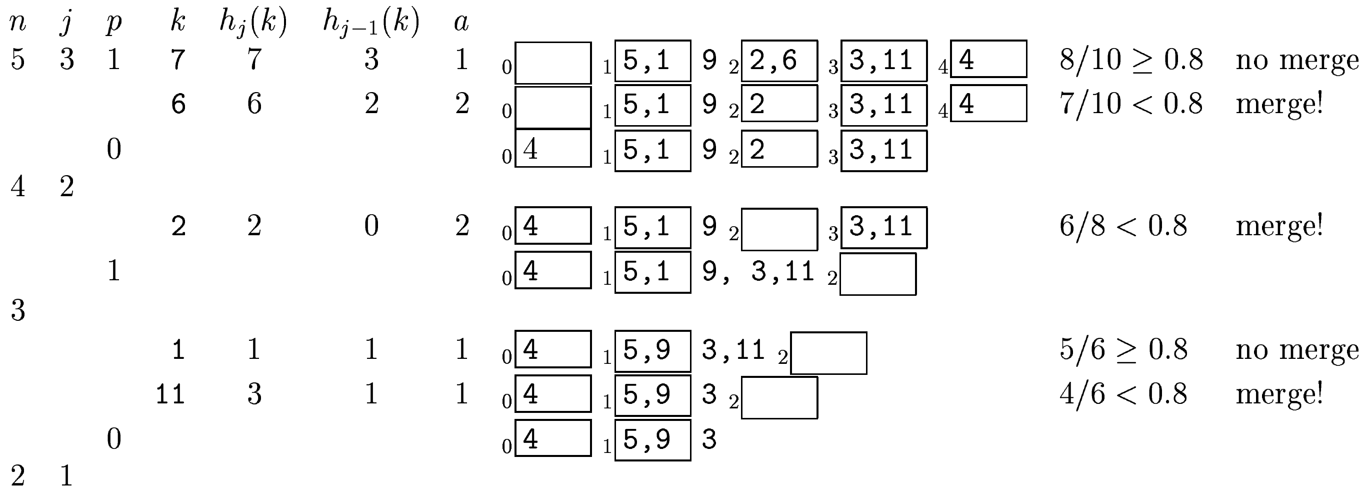


Figure 7: Example for Algorithm LHD

hash functions may be operative at once). So new blocks are created in any order, generally leaving gaps in the second half of the doubling file, until all blocks in the first half have been split. Not only is this inelegant, but also a bit map of the whole file is needed to determine which blocks have been split (n bits in RAM) instead of a simple counter ($\log n$ bits in RAM).

1.2 Symmetry: Multidimensional Hashing

Since a hash function can work on only one key, it would appear to be difficult to use hashing for multidimensional queries such as partial match. In fact, we can hash independently on each of several keys, to get a coordinate for each one. We can then use these coordinates to address an array of blocks, using the usual "row-major" or "column-major" addressing functions to convert the multiple coordinates to a single, one-dimensional address.

Figure 8 shows a two-dimensional and a three-dimensional array and the correspondence between coordinate pairs or triples and the disk address of each block. The row-major addressing functions are $a(i, j) = wj + i$ (2D) and $a(i, j, k) = hwk + wj + i$ (3D). Here, w is 4 and h is 3, so in any column the jump is 4 between rows, and the jump between corresponding positions in the three-dimensional planes is 12.

This figure illustrates an exact-match search for the two-dimensional key, `(21, 35)`, and partial-match queries for `(37, *)` and `(*, 23)`, where `*` means "any value". These hash, respectively, to `(1, 2)`, `(1, *)`, and `(*, 2)`, using division-remainder. The last two are identified as "segment"s in figure 8, and the first is the intersection of these two segments. So the exact-match query can be done in one access (apart from overflows), the partial-match query for `(37, *)` requires four accesses, and the partial-match queries for `(*, 23)` needs three accesses.

(An asymmetry is introduced in that blocks 8, 9, 10, and 11, for the first partial-match query, are contiguous on disk, while blocks 1, 5, and 9, for the second partial-match query, are not. With luck, we could save seek times after block 8 in the first case, and the four blocks could be retrieved faster than the three in the second case. But in both cases, only the blocks *needed* will be retrieved, and this is much more symmetrical than fetching, say, only four for the first query, but all blocks for the second.)

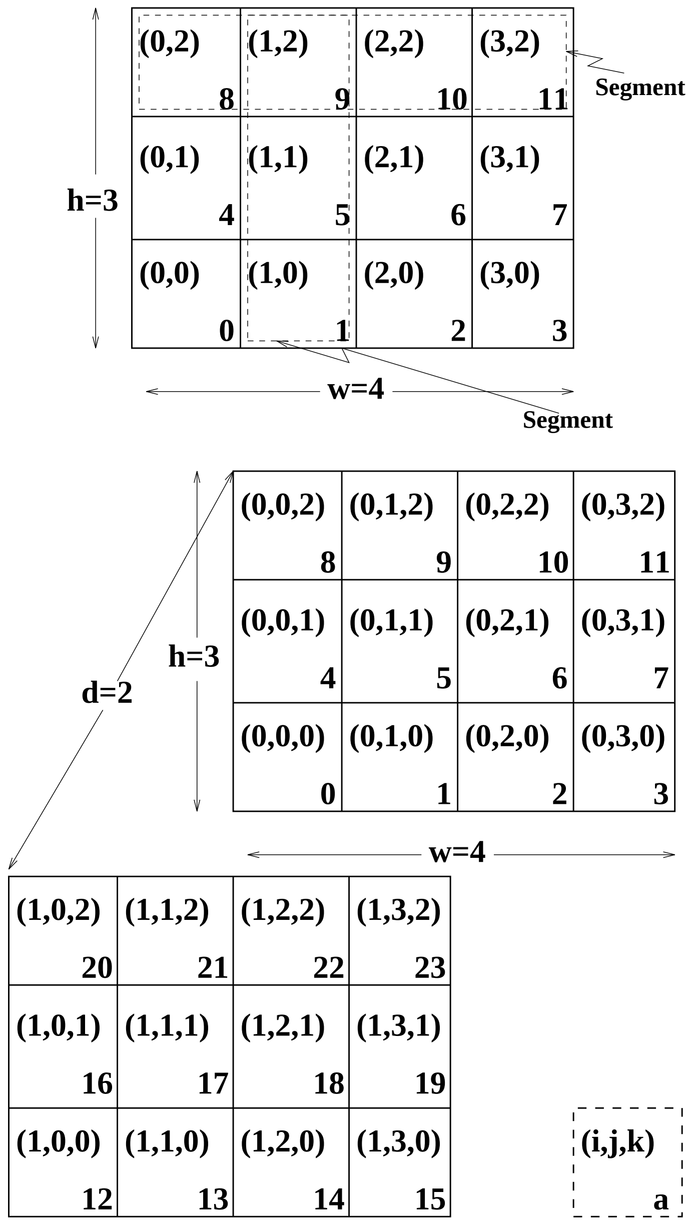


Figure 8: Array Addressing Functions in Two and Three Dimensions

1.3 Activity: Hash Merge

Hashing is clearly good for low activity, but suppose we want to retrieve many records from a hash file. The simple-minded approach is to hash repeatedly, once for each record. Because many records are stored at any addresss (block), we run the risk of accessing the same block repeatedly, leading to inefficiency factors of up to b, the number of records per block.

How can we organize our search so that repeated accesses are not made to the same block? We need to group the *queries* according to the blocks to which they will be directed. This could be achieved by sorting the queries into block order, which is hash-function order. Then the queries and the file can be *merged*, comparing block addresses to hash values. If the set of queries is too large for RAM, such a merge will be an especially effective way of doing the high-activity search.

Two difficulties with this approach are: (1) that it is messed up by overflows, especially if they are stored downwards from their home block by linear probing; and (2) that it cannot work for range queries, in which not every key sought is explicitly provided, but only the first and last keys in the ordered range.

2 Tidying

For range queries and other kinds of high-activity processing, especially when values sought are not present in the file, we need to preserve order. Can we have *Order-Preserving Key-to-Address Transformations*? We could call them *tidy* functions, as opposed to hash functions. In the 1936 edition of *Roget's Thesaurus of English Words and Phrases*, "tidy" falls under the category of "reduction to order". Since key-to-address transformations usually reduce a key spece to a smaller address space, and tidy functions will be designed to preserve order, this meaning of the word seems to capture our intention. Besides, we must avoid unwieldy appelation, such as the acronym "OPK2AX".

Figure 9 illustrates the two key-to-address transformations and how well they do with a range query. The range of key values requires us to look everywhere in the hash file, but only at a fraction of the blocks in the tidy file. So even if we could anticipate all the values in the range, and do an explicit hash merge, as discussed in the previous section, the tidy function would beat the hash function by several times.

The question is, how can we do this? The scaling key-to-address transformation at the beginning of this chapter is a possibility, but we saw that it does not smooth out lumps in the key distribution. If we think of the key-to-address transformation as a function plotted with the keys distributed along the abscissa and the addresses along the ordinate, we see from figure 10 that the *cumulative distribution* makes a perfect tidy function.

The interpretation of the cumulative distribution function for a field, A, is

$$D_A(a) = \text{probability}(A \leq a)$$

. As a probability, $D_A(a)$ increases monotonically from 0 to 1. To scale this up to the integer addresses for n blocks, we must first multiply by n and then take the ceiling. This gives the tidy function

$$t(a) = \lceil nD_A(a) \rceil$$

.

Of course, the practical solution is not so simple. To *store* the cumulative distribution would require as much space as to store all the key values. So we must approximate.

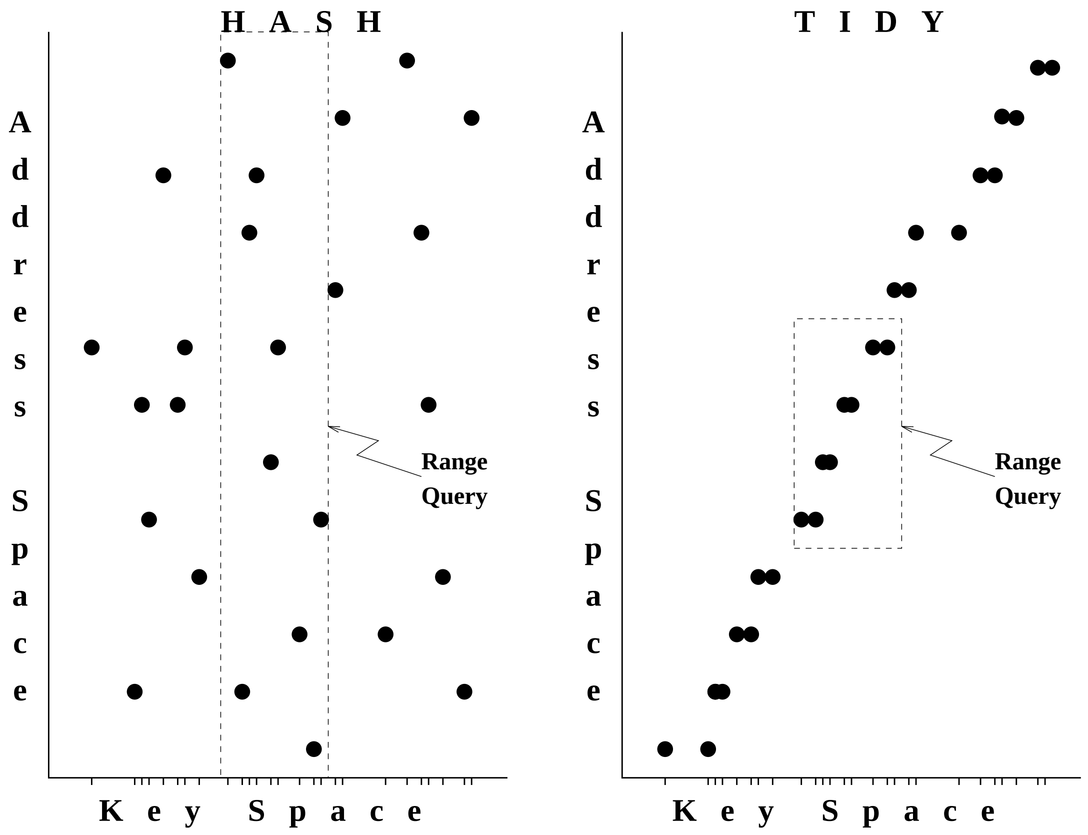


Figure 9: Tidying versus Hashing for Range Queries

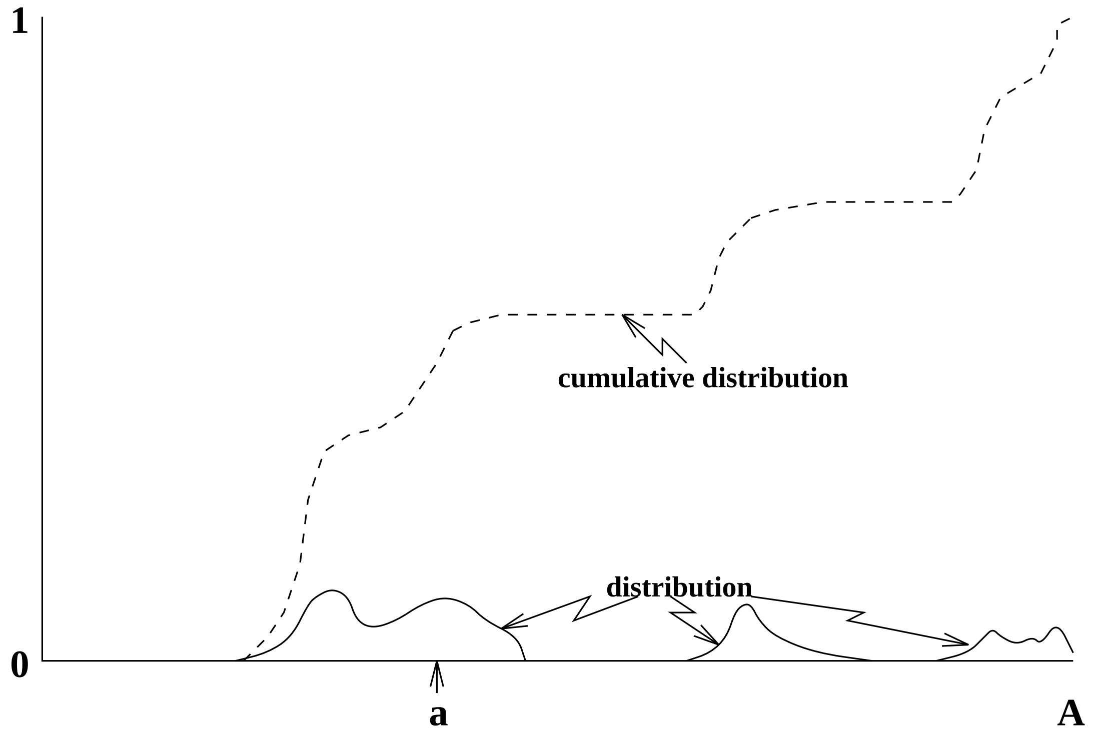


Figure 10: Cumulative Distribution as Tidy Function

The easiest approximation is piece-wise linear, using straight lines. To motivate the following discussion, we anticipate. Figure 11 shows the optimal fit of four linear pieces to 33 key values, which happen to be the first two digits of a set of telephone numbers. We see that the cumulative distribution is a step function, increasing one unit up the ordinate every time it passes the value on the abscissa of an actual key.

The tidy function of figure 11 consists of four pieces, so we must be able to fit in RAM the five endpoints of all the pieces, at two coordinates each (the page number, and the key value of the last record on that page). They must be stored in RAM, otherwise the tidy function does not support direct access. (On secondary storage, we have the advantage that, although the tidy function may take much longer to calculate than hash functions, for instance, do, this is usually negligible compared to doing the accesses to secondary storage for the data.)

To design such a tidy function for given data, we must determine how much room we can allow in RAM for coordinate pairs, use this to decide how many linear pieces we can have in the approximation, then find the best approximation using this many pieces.

The question is, how do we find the best approximation?

We now proceed to investigate this, starting with a simple case. The ordinate of figure 12 shows six records stored in three blocks, two per block: `1, 4, 9, 16, 25, 36`. The cumulative distribution is the parabola shown as a dotted line. It is approximated by the straight line.

A perfect tidy function, the parabola itself, would show that blocks end at keys `4,16`, and `36`. But this would need an index entry for each block. To save space, we sacrifice accuracy. The linear approximation is fine for queries in the ranges `0`–`4`, `13`–`16`, and `25`–`36`; but queries for keys in the ranges `5`–`12` and `17`–`24` will be incorrectly directed.

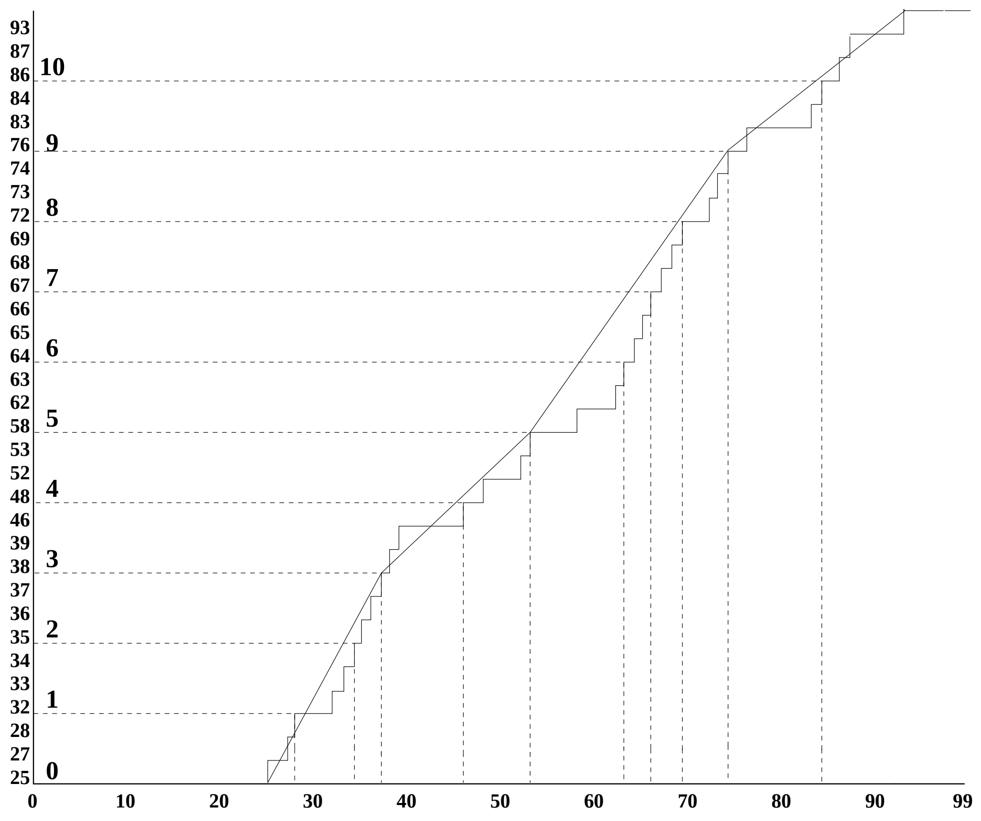


Figure 11: Optimal Approximation to Cumulative Distribution of Phone Numbers

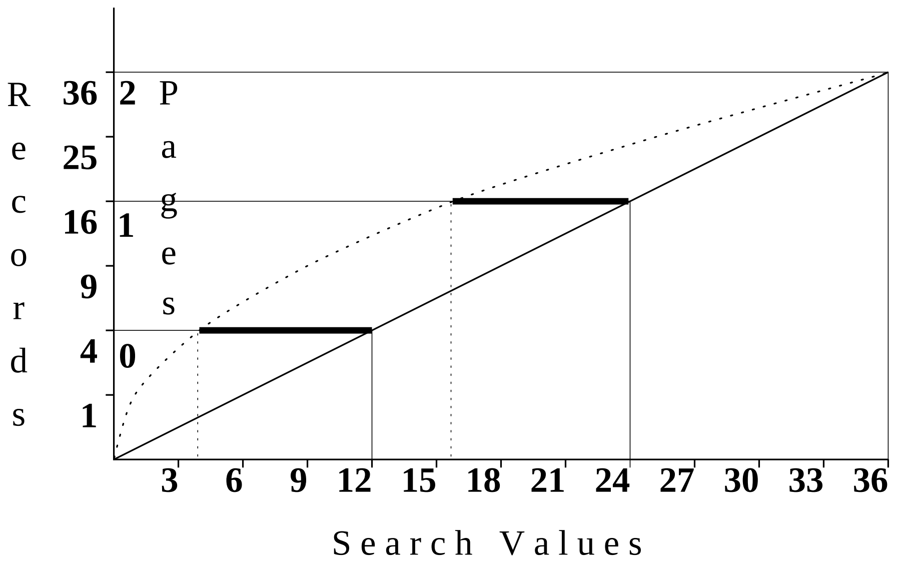


Figure 12: A Simple Key Distribution with Linear Approximation

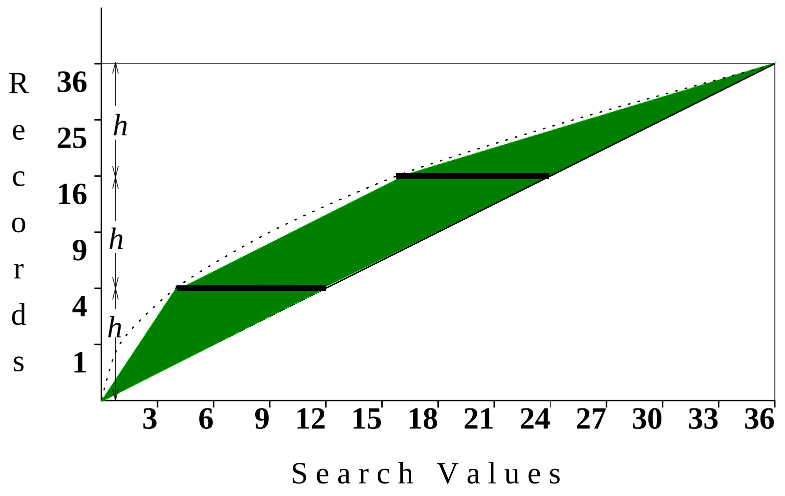


Figure 13: The Error $\propto$ the Area Between the Curves

These errors may be called "overflows", although no records are out of place in the blocks of the file, and although "overflows" will also include searches for records that are not in the file. Figure 12 makes it clear that incorrectly directed query ranges can be answered by (linear) probing *upwards* in the file. Thus, a search for key 5 will be directed by the approximation to block 0, while it could only be found, if at all, on block 1.

The number of these "overflows" is proportional to the length of the heavy horizontal lines shown between the true curve and the approximate curve. We can see from figure 13 that these are in turn proportional to the area between the curves $= (l_1/2 + (l_1 + l_2)/2 + l_2/2)h = h\Sigma l_i$.

In many cases, more than one probe may be needed before we find *any* of the records that the approximation says are on the page. Figure 14 shows cases of up to triple overflows, indicated by thicker horizontal lines. The error is still essentially the area between the true curve and the linear approximation. Where horizontal lines overlap, they do not count twice. The upper lines may be thought of as being shadowed by the lines below. On top of each bottom line, we build a number of blocks of height h equal to the number of overflows represented by the line (in the figure, indicated by the thickness of the line). We can see that $h\times$ the number of extra probes $\approx$ the area between curves.

As well, figure 14 shows a case where the overflow probes must go downwards, and we see that probes go up if the exact curve is above the approximation and down if it is below.

In some cases, we must do more than one probe before we find the page the record might be on. To save the extra probes, we store a pointer on each page showing how far down to skip before the search can possibly succeed.

We could even store a second pointer on each page showing where the probes must stop. Then, instead of linear searches on the pages between the two extremes, we could even do binary searches, since the records are always ordered. This reduces the worst-case complexity to $\mathcal{O}(\log n)$ from $\mathcal{O}(n)$, and speeds up the average search.

We have used single straight lines to approximate curves and have shown that the overflows incurred are proportional to the area between the two lines. Now let us see if we can find the combination of more than one linear piece that minimizes this area, and will thus be optimal. Figure 15 shows a parabolic cumulative distribution mapping to six pages, and two different

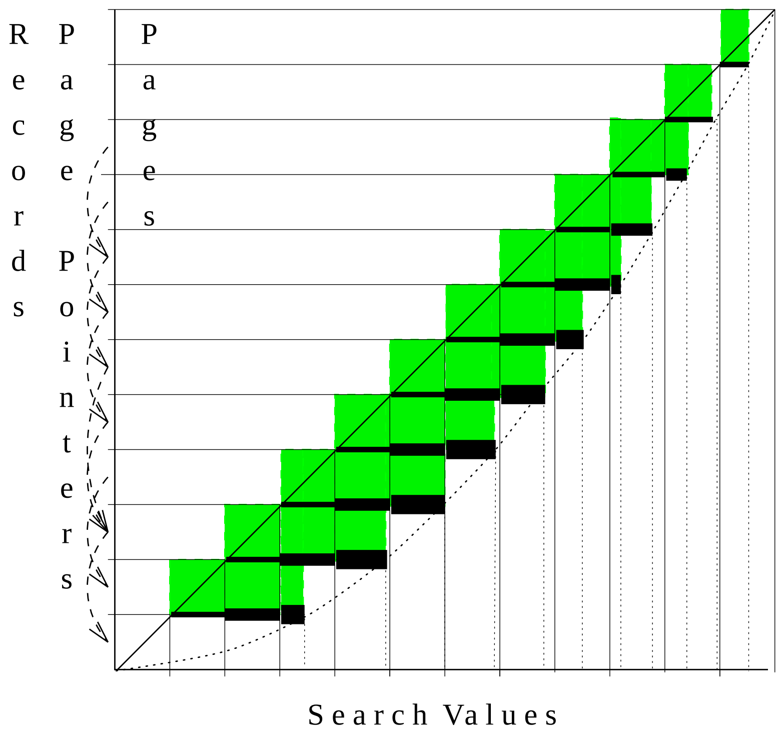


Figure 14: The Error Still $\propto$ the Area Between the Curves

optimal approximations, one of two pieces, and one of three.

To construct an optimal approximation of p linear pieces to n pages requires a search through all possible ways of connecting point 0 to point n with $p-1$ intermediate stops at the $n-1$ remaining points. Since we must explore all possible linear pieces, we need to know $a(i,j)$, the difference in area between the exact curve and a straight line, between point i and point j. For the parabola of figure 15, this area is $(\sqrt{j}-\sqrt{i})^3$ and we get a matrix of $\frac{n(n+1)}{2}$ values.

$a(i\backslash j)$	0	1	2	3	4	5	6
0		1.0	2.8	5.2	8	11.2	14.7
1			0.07	0.4	1	1.9	3.0
2				0.03	0.2	0.6	1.1
3					0.02	0.1	0.4
4						0.01	0.1
5							0.01
6							

(In practice, we would not have an analytical formula for the areas, but we can always calculate numbers like these.)

For $p=2$, we can work directly from this table of $a(i,j)$s. We need the minimum of the five ways of getting from 0 to 6, stopping at 1, 2, 3, 4, or 5, respectively.

$$\begin{aligned} m(2,j) &= \min_k a(0,k)+a(k,j) \\ m(2,6) &= \min(1.0+3.0,\ 2.8+1.1,\ 5.2+0.4,\ 8.0+0.1,\ 11.2+0.01) \end{aligned}$$

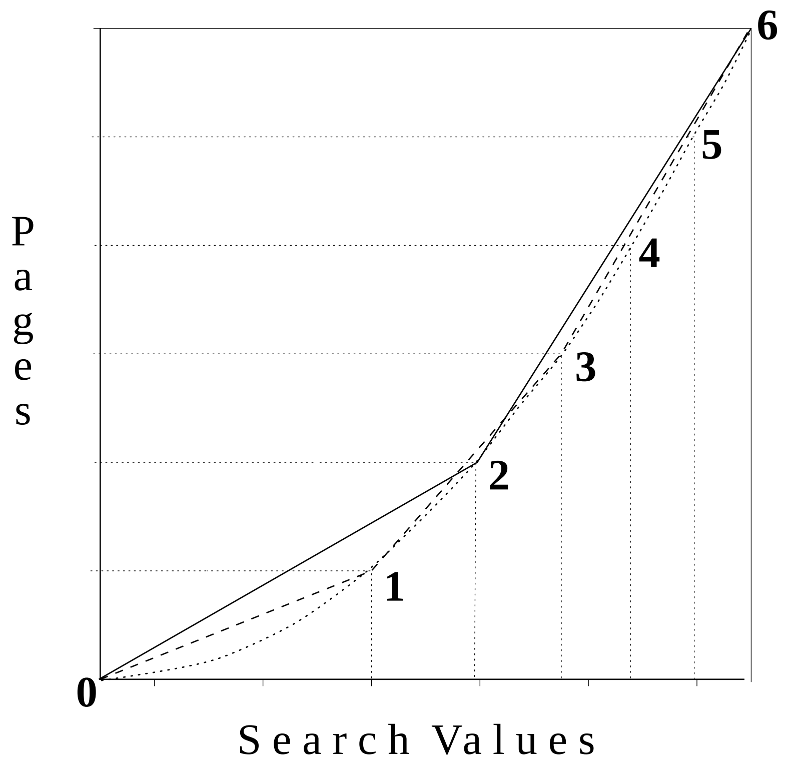


Figure 15: Optimal Approximations with 2 or 3 Linear Pieces

$= 3.9$ via point 2

where $m(p,j)$ is the minimum cost of breaking $0..j$ into p pieces; and where we must remember, when we find this minimum, how we got there.

More generally, we must construct a table of $m(p,j)$s. We can start with all the 1-piece costs, $m(1,j) = a(0,j)$. For $p = 3$ and $n = 6$, for instance, we could write recursively

$$
\begin{array}{llll}
\multicolumn{3}{l}{m(3,6)\ \text{(10 subproblems)} = \min:} & \\
\quad m(2,5) & & & +a(5,6) \\
 & \text{(4 subproblems)} = \min: & & \\
 & & m(1,4)+a(4,5) & \\
 & & m(1,3)+a(3,5) & \\
 & & m(1,2)+a(2,5) & \\
 & & m(1,1)+a(1,5) & \\
\quad m(2,4) & & & +a(4,6) \\
 & \text{(3 subproblems)} = \min: & & \\
 & & m(1,3)+a(3,4) & \\
 & & m(1,2)+a(2,4) & \\
 & & m(1,1)+a(1,4) & \\
\quad m(2,3) & & & +a(3,6) \\
 & \text{(2 subproblems)} = \min: & & \\
 & & m(1,2)+a(2,3) & \\
 & & m(1,1)+a(1,3) & \\
\quad m(2,2) & & & +a(2,6) \\
 & \text{(1 subproblem)} = \min: & & \\
 & & m(1,1)+a(1,2) &
\end{array}
$$

Such a recursive approach requires

$$\sum_{k=p-2}^{n-2} \binom{k}{p-2} = \binom{n-1}{p-1} = \mathcal{O}(n^p)$$

calculations, an exponential. But many of these calculations are repeated, so we could re-use the results (a process called *memoizing*) and reduce the cost. This is called *dynamic programming* and reduces the complexity from exponential to cubic, a gain which makes the difference between impossibility on one hand and, on the other, feasibility for problems small enough to fit into RAM. (For a big problem on secondary storage, we will have to reduce the cost further, which we discuss later.)

Here are the first two rows of the table for $m(p,j)$. The first row is the same as the first row of $a(i,j)$. The second row is $m(2,j) = \min_{0<k<j} m(1,k) + a(k,j)$. We subscript each result with the point that gave the minimum cost. We see that $m(2,6)$ is 3.9 via point 2, as we found before. So the optimal 2-piece partition bends at point 2.

m($p\backslash j$)	0	1	2	3	4	5	6
1		1.0	2.8	5.2	8	11.2	14.7
2			1.07_1	1.4_1	2_1	2.9_1	3.9_2

The remaining entries in the second row are needed to go on to build the third row, which we now do: $m(3,j) = \min_{1<k<j} m(2,k) + a(k,j)$.

m($p\backslash j$)	0	1	2	3	4	5	6
1		1.0	2.8	5.2	8	11.2	14.7
2			1.07_1	1.4_1	2_1	2.9_1	3.9_2
3				1.1_2	1.27_2	1.5_3	1.8_3

From this we see that $m(3,6)$ is 1.8 via point 3, which we got to ($m(2,3)$) via point 1. The optimal 3-piece partition bends at points 1 and 3.

We could continue with further rows if we wanted to investigate approximations of more than 3 pieces. (Note that the diagonal, $m(i,i)$, is just the cumulative sum of the diagonal of a.) The calculation for p pieces must find $\frac{n(n+1)}{2} - \frac{(n-p)(n-p+1)}{2}$ different $m(i,j)$s, and is at worst cubic in complexity.

Practical problems do not have analytic cumulative distributions (which is why we cannot use calculus to find the minimum area), but the procedure is the same once the $a(i,j)$s are found. For figure 11, the areas are $2\times \mid$ *step* $-$ *triangle* $\mid$, and are shown for the eleven pages in the upper part of the table below.

	28	34	37	46	53	63	66	69	74	84	93
	1	23	56	221	380	662	767	890	1121	1649	2188
		12	3	114	231	453	540	645	846	1314	1799
34	13_{28}		3	54	129	291	360	447	618	1026	1457
37	4_{28}	16_{34}		3	36	138	189	258	399	747	1124
46	59_{37}	7_{37}	19_{37}		9	33	66	117	228	516	839
53	92_{37}	40_{37}	16_{46}	28_{46}		18	3	30	111	339	608
63	194_{37}	92_{46}	40_{46}	34_{53}	46_{53}		3	12	63	231	446
66	245_{37}	95_{53}	43_{53}	19_{53}	31_{53}	49_{63}		3	18	126	287
69	314_{37}	122_{53}	70_{53}	46_{66}	22_{66}	34_{66}	52_{66}		9	39	146
74	449_{46}	203_{53}	113_{66}	61_{66}	37_{66}	31_{69}	43_{69}	61_{69}		12	41
84	719_{53}	353_{69}	161_{69}	109_{69}	73_{74}	49_{74}	43_{74}	55_{74}	73_{74}		1
93	988_{53}	460_{69}	244_{74}	154_{74}	102_{74}	74_{84}	50_{84}	44_{84}	56_{84}	74_{94}	

The lower part of this table is the *transpose* of the $m(p,j)$s, with $m(1,j)$ omitted because it is the same as $a(0,j)$. It is calculated by dynamic programming, and goes all the way up to 11 pieces (cost 74). From this we see that the optimal four-piece approximation shown in figure 11 is via points 74 (cost 244), 53 (cost 203), and 37 (cost 92).

The cubic dynamic programming problem is still too big. n is the number of pages, which could be in the millions.

We might divide the whole problem into a number of equal-sized smaller problems, then run the optimizing algorithm on each of these. For example, a 1 Gbyte file of 1 Kbyte pages would have $n = 10^6$ pages. Dividing this into 10,000 subproblems of 100 pages each would require 10,000 $\mathcal{O}(100^3)$-sized optimizations, instead of one $\mathcal{O}((10^6)^3)$-sized optimization.

2.1 Symmetry: Multidimensional Paging

As with hashing, tidying can be applied to more than one field at once by simply tidying the axes of the multidimensional space. Because $n^{1/d}$ is significantly smaller than n, even for d=2 dimensions, we can suppose that the complete tidy function along each axis will fit into RAM and there is no need to approximate. If this supposition is not true, we can make optimal linear approximations as in the one-dimensional case.

The problem of finding the partitioning of points (records) in multidimensional space that puts an equal number of points in each page, is constrained by the need to keep the partitions

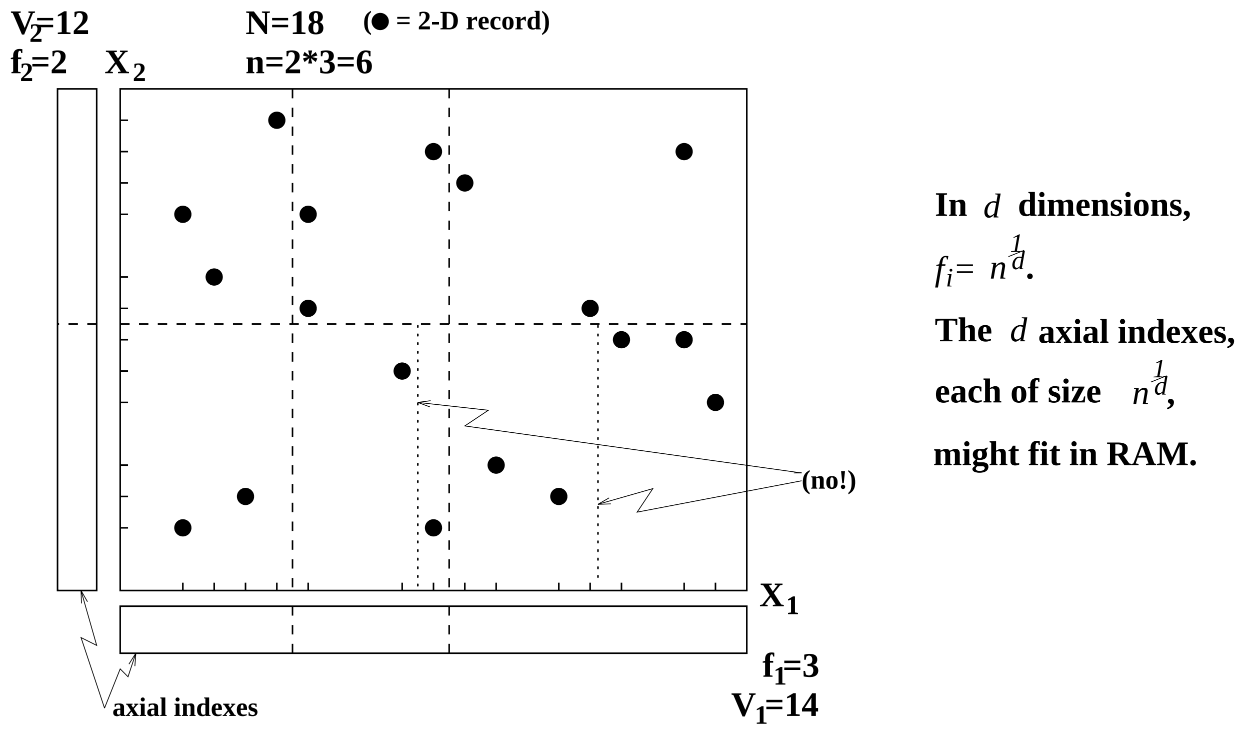


Figure 16: Rectilinear Partitioning for Efficient Addresing

rectilinear, so that the addressing can be done entirely via the axes. Figure 16 shows a two-dimensional distribution of records and a three-by-two rectilinear partitioning, requiring 3 + 2 = 5 index entries for the 3×2 = 6 pages (dashed lines). This partitioning is probably not optimal, and certainly does not have exactly three records per page, which would be the equalized number. An attempt to equalize page contents might split one of the pages further, as shown by dotted line, or even shift the boundary of only one page, also shown by dotted line. However, this would mess up the addressing altogether, because the pages would have to be addressed by a map (index) with size proportional to n instead of to $dn^{1/d}$.[1]

Before going on to discuss the construction of multidimensional tidy functions ("multipaging"), we review the straightforward retrieval process.

Algorithm MPS (Multipage Search)

MPS1 Use axial indices to find coordinates for page(s) that can hold the data requested.

MPS2 For each page needed (coordinates $i, j, k, ..$), use an array addressing formula to give the page address. (See section 1.2.)

We discuss the multipage construction algorithm (figure 17) step by step. The quantities N, n, and α have their usual meanings, and f_i and V_i, the partition factor and the number of different field values, respectively, for axis i, are illustrated in figure 16.

[1]A technique derived from multipaging, "grid files", missed this insight and uses the larger map. It accordingly requires two accesses to retrieve any record. In its favour, it *guarantees* two accesses, while multipaging access costs will be seen to depend on how the records are distributed in the space. On the other hand, multipaging storage overhead is negligible, while grid file storage requirements are sensitive to data distribution and can become enormous.

MP1 For each axis, $i = 1..d$, find axial distributions and V_i. (d sorts: $\mathcal{O}(dN \log N)$)

MP2 Given approximate values for b (blocksize) and α (load factor), choose partitioning factors, $f_i, i = 1..d$. ($\mathcal{O}(1)$)

$$n = \Pi_1^d f_i;$$

$$\text{heuristic}: \frac{V_i}{f_i} = \text{const}$$

MP3 For each axis, i, find candidate(s) for axial partition (scan forward then back, cost $2V_i$)

MP4 Build histograms for all combinations of axial partitions by 1 pass of the data. Do π-α comparisons to find the best. ($\mathcal{O}(N)$)

Figure 17: Algorithm MP: Construct Multipage Tidy Function Given N Records

The philosophy behind this algorithm is to find a good partitioning for a file in reasonable time and to assess the result in terms of π and α. In this way, if the file distribution makes an acceptable result impossible, it has not cost us too much to find out. The most expensive step is MP1. The critical steps are MP3, a linear-time dynamic programming search, and MP4, which embodies the heuristic that the optimal partitions found for each axis in MP3 will combine to give a good partition of the whole space.

The example we use is the two-dimensional file of toys and manufacturers shown in figure 18. This figure shows a record distribution (1s in the two-dimensional space) which is something of a challenge to multipaging.

This figure also illustrates step MP1. The two axial indices are shown, which count the number of records for each different value of *Toy* and of *Maker*, respectively. In general the process of finding these is straightforward, but is the most expensive step of algorithm MP. For each axis, the file must be sorted and scanned to get the counts.

Step MP2 needs a few iterations to determine the number of partitions, f_i, into which we wish to split the ith axis. We are given N, the number of records, and we have an idea, from other considerations, what values we would like for b, the blocksize, and α, the load factor.

These three then yield a value for n, the number of blocks, $n = \lceil N/b\alpha \rceil$, and we thus have a constraint on the f_is, $\Pi f_i = n$. This is one equation in d unknowns, and we need a heuristic to provide the other $d - 1$.

The heuristic we use is $V_i/f_i = c$, a constant. This has the attraction of making the shape of the address space, $f_1 \times f_2 \times .. \times f_d$, the same as the shape of the value space, $V_1 \times V_2 \times .. \times V_d$. It also gives optimal expected retrieval time for certain categories of query.

We now have d equations in the d unknowns, f_i, and we can determine c, the constant, and then, using V_i, f_i.

$$\begin{aligned} f_i &= cV_i \\ n &= c^d \Pi V_i \\ c &= (\Pi V_i / n)^{1/d} \end{aligned}$$

Unfortunately, this does not give integer values for f_i, and if we round them all, or take the floor or ceiling of them all, the resulting product may be nowhere near n. The best we can do is search the 2^d possibilities of taking the floor of some and the ceiling of the others

*Maker**Toy*	Caboose	Calico Cat	Car	Locomotive	Toy Train	Tractor	Tricycle	Truck	Ukulele	
Amloco Toys				1						1
Canloco Ltd.				1						1
Dink Inc.			1	1						2
Extrafun		1	1	1						3
Fischerman	1		1	1						4
General Toy Corp.	1	1	1	1	1	1	1	1	1	9
Mettal Toys	1		1	1		1	1	1		6
Noisy Toys		1	1	1	1	1				5
Playloco				1						1
	3	3	6	9	4	2	2	2	1	

Figure 18: Toys and their Makers

to see which product comes closest to n. There will likely be a new n as a result, with corresponding adjustments for b or α.

All these operations take place in RAM, or in the head of the designer, and do not add to the computational cost of algorithm MP.

For example, in the toys distribution of figure 18, $N = 32, V_1 = 9$, and $V_2 = 9$. If we took $b = 2$ and $\alpha = 1.0$, we would have $n = 16$. Since $V_1 = V_2$, then f_1 should $= f_2$. The solution is a 4×4 partition. There would be 4 *segments* of toys, with $8{=}N/f_{Toy}$ records each, and 4 segments of makers, also with 8 records each.

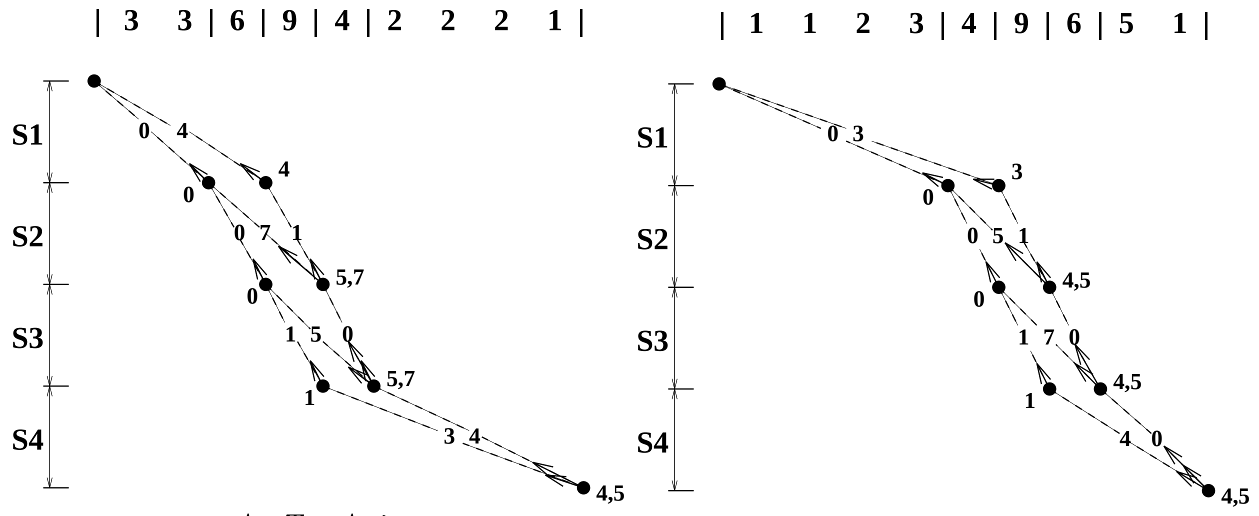
Step MP3 is performed independently for each of the d axes. The objective is to divide the V_i different values up into f_i partitions. This means placing $f_i + 1$ partition boundaries in the $V_i + 1$ positions to the left or right of each value. Since there is a boundary at each end, the number of ways we can do this is

$$\binom{V_i - 1}{f_i - 1} = \mathcal{O}(V_i^f).$$

However, a systematic approach reduces this to 2^{f-1}, and this is susceptible to dynamic programming, as figure 19 shows.

We sum the axial distribution from step MP1, looking for multiples of N/f_i. For instance, for the *Toy* axis, with $f_{Toy} = 4$ and $N = 32$, we would be looking for partial sums 8, 16, and 24. If we found these values, we would immediately have our optimal boundaries between segments. Instead, for the *Toy* axis, we have partial sums 3, 6, 12, 21, 25, 27, 29, 31, 32. So the closest we can come to placing boundaries is either side of the closest partial sums: 12, 21, 25. This reduces the $\mathcal{O}(V_i^f)$ problem to 2^{f-1}.

However, we can do much better than that. Figure 19 shows the process stretched out vertically for ease of understanding. Continuing our discussion of the *Toy* axis, we see the four potential segments mapped out vertically, with the candidate boundaries between each positioned horizontally under the appropriate gap in the axial distribution. The algorithm for step MP3 works from top left to bottom right of this graph and back again. On the way



A. *Toy* Axis B. *Maker* Axis

Figure 19: Optimal Partitions of *Toys* and *Maker* Axes

down, it records the costs of choosing any given candidate and how those costs were arrived at. On the return, it uses this information to choose the boundaries that gave the minimum cost.

We break the downward pass into three components to make the discussion easier. First, a number is written on each edge of the graph, its cost, which is the number of overflows resulting if the two ends of the edge had been chosen as boundaries. Thus, the edge that spans the values with 3, 3, 6 records represents a segment that would store 3+3+6=12 records. Now suppose that the page capacity were two, as discussed for this example in step MP2, and the segments have eight records each. Then 8 of the 12 records would fit and 4 would overflow, so the cost is 4 for this edge.

Next, the lowest cumulative cost is recorded for each boundary placement, and a short arrow is placed to indicate how this cost was arrived at. Thus, the edges leading to the second candidate for the boundary between segments S2 and S3 would give it a cost of 0 + 7 or 4 + 1, respectively, and the cheaper is 4 + 1, so 5 is recorded and the arrow points along the edge labelled 1. The next boundary candidate down from this, between S3 and S4, costs 5, which can be arrived at from either direction, so there are two short arrows.

These two considerations produce the cheapest value for the partition, and the algorithm needs only follow the short arrows back again to determine the choice of boundary candidates.

However, if we get a single answer from each axis, we have no flexibility in putting them together in step MP4. The final multipaging is determined and step MP4 is not needed. Since there is no guarantee that combining the best from each axis will give the best overall, we should keep MP4, and MP3 should offer it some choices. Accordingly, figure 19 shows long arrows as well as short, and alternative values for the boundary candidates. These give the next-best solutions for each axis. We will see next that combining the optima does not give the best overall answer in this example.

MP3 does its work for each axis. Figure 19 shows the results for the *Maker* axis as well as for the *Toy* axis.

Step MP4 looks at the data, for the first time in algorithm MP since MP1 when the axial distributions were found. It is limited to one pass of the data, and builds a histogram for each combination of the boundary candidates identified by step MP3. For our example, MP3 aA

2	3	5	1
2	1	1	5
1	1	1	3
1	1	2	2

α	0.67	1.0
π_{opt}	1.13	1.25

aB

2	3	6	0
2	1	2	4
1	1	2	2
1	1	3	1

α	0.67	1.0
π_{opt}	1.13	1.25

aC

5	5	1	0
3	1	1	4
2	1	1	2
2	2	1	1

α	0.67	1.0
π_{opt}	1.16	1.28

bA

1	2	4	0
3	2	2	6
1	1	1	3
1	1	2	2

α	0.67	1.0
π_{opt}	1.13	1.25

bB

1	2	4	0
3	2	4	4
1	1	2	2
1	1	3	1

α	0.67	1.0
π_{opt}	1.09	1.25

bC

3	4	0	0
5	2	2	4
2	1	1	2
2	2	1	1

α	0.67	1.0
π_{opt}	1.13	1.25

cA

1	2	4	0
1	1	1	1
2	1	1	5
2	2	3	5

α	0.67	1.0
π_{opt}	1.16	1.28

cB

1	2	4	0
1	1	2	0
2	1	2	4
2	2	5	3

α	0.67	1.0
π_{opt}	1.13	1.25

cC

3	4	0	0
2	1	1	0
3	1	1	4
4	3	2	3

α	0.67	1.0
π_{opt}	1.09	1.31

Figure 20: Histograms for Candidate Partitionings

returned three possibilities for each axis. We will identify them as A, B, C for the *Toy* axis and a, b, c for the *Maker* axis, as follows.

Toy	3	3	|	6	|	9	|	4	|	2	2	2	1
			A		A		A						
			B		B				B				
					C		C		C				

Maker	1	1	2	3	|	4	|	9	|	6	|	5	1
					a		a		a				
					b				b		b		
							c		c		c		

These combine into nine possibilities for the global partitioning, and figure 20 shows the nine histograms, each labelled by the choice from each axis that produced it. The figure also shows the results calculated for π_{opt} by counting overflows. We supposed that $b = 2$ and $\alpha = 1.0$, and, counting up the excesses over 2 in each histogram, we get the values for π_{opt} shown.

We can easily explore further, however, just by supposing that each page has room for one more record, in this case $b = 3$. Then $\alpha = 0.67$ and counting execesses over 3 in the histograms gives the values for π_{opt} shown for this α. Figure 21 shows plots of $1/\pi_{\text{opt}}$ versus α in each case. The inverse of π_{opt} is used so that the ideal case is the square, $(0..1)\times(0..1)$, with area 1, and any lesser result has an area smaller than 1.0.

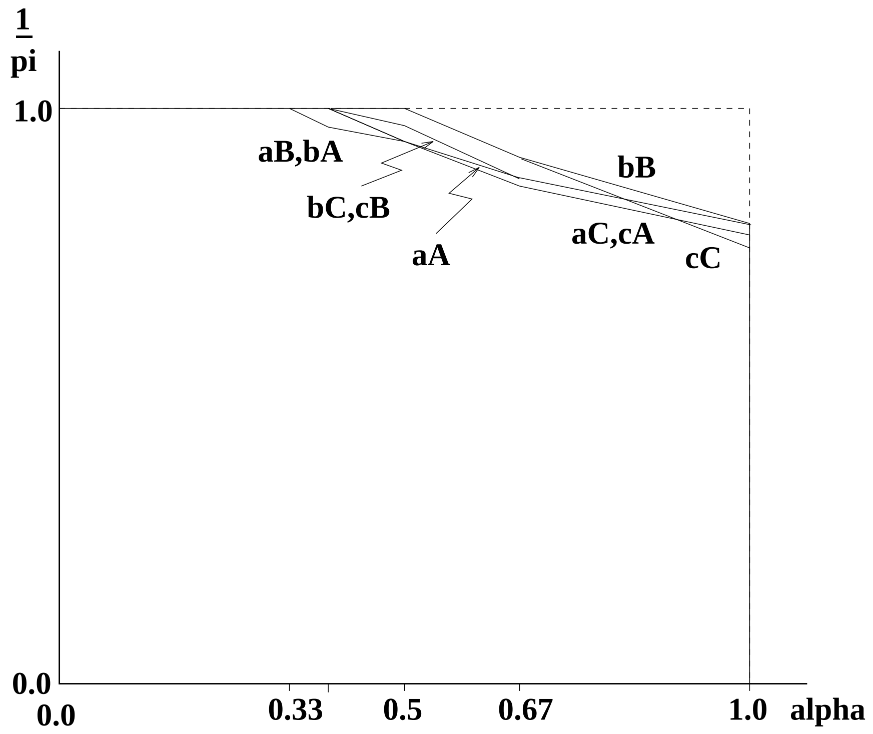


Figure 21: π-α Analysis of the Histograms

From figure 21, we see that the winner is case bB, whereas, for the individual axes, MP3 found A best for *Toy* and a best for *Maker*. What has happened is that shifting two boundaries has repaired a diagonal overflow which is invisible to MP3. It is not guaranteed, though, that even this combination of MP3 and MP4 will produce an optimum result.

There are distributions of records which are pathological and would not be acceptable for multipaging. One such distribution is a diagonal arrangement of the records. Then step MP3 would find the same partitionings for each axis, and the combination would have much too full pages on the diagonal and empty pages everywhere else.

If the meanings of the fields on the axes were such that rotating the coordinates were acceptable, this pathological distribution could be multipaged by rotating the space and dropping one of the dimensions. Another pathological distribution which cannot be treated in this way is a ring of records.

2.2 Symmetry and Volatility: Dynamic Multipaging

Since tidying and multipaging are designed to cope with both low and high activity and low and high symmetry, high volatility is the next target. By remembering to split pages within a strategy of preserving the access method, we can attain this capability too. Because the access method requires addressing a whole slab of pages from the axis, we cannot split individual pages without also splitting all pages in the slab orthogonal to one of the axes. Figure 22 shows one possible sequence of such splits, with the newly created slabs of pages outlined in dashed lines.

Before we pursue further the construction of such a variant of multipaging, we should

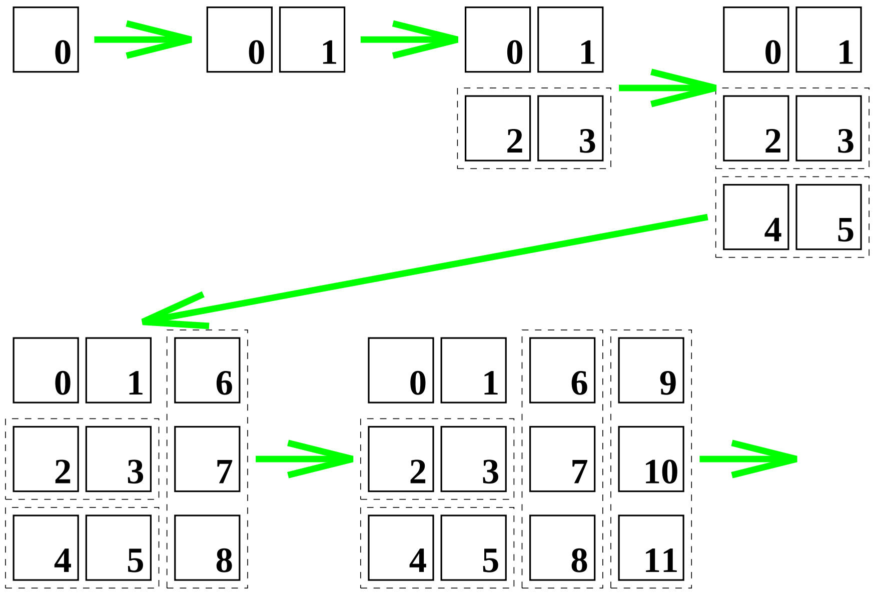


Figure 22: Splitting Pages for Dynamic Multipaging

check that there is an access method. The problem, as we see from figure 22, is that the pages are no longer numbered in simple row-major or column-major order, as in a matrix. Nonetheless, the pages are systematically ordered, in the sequence of their creation. Since a whole slab is created at a time, this history can be captured by entries in the axial indexes. In fact, it suffices to record there the number of the first page in the slab orthogonal to the axis at that position. Figure 23 shows these entries in two two-dimensional cases, each reflecting a slightly different splitting history.

If we want to find the number of the page with coordinates (i, j), we can see that the slab that page is in starts with the page numbered $\max(p_x(i), p_y(j))$, where p_x and p_y are the page numbers recorded in the two axial indexes. Where in the slab page (i, j) is found is just the value of the other coordinate, the one that did not give the maximum. The figure shows the results for the address $a(2, 1)$ for both splitting orders. For the left-hand example, the calculation is

$$\begin{aligned} a(i,j) &= \max(p_x(i), p_y(j)) + \text{ the other one} \\ a(2,1) &= \max(p_x(2), p_y(1)) + \text{ the other one} \\ &= \max(6,2) + \text{ the other one} \\ &= 6 + j = 6 + 1 = 7 \end{aligned}$$

For higher dimensions, the same process finds the start of the slab as the maximum, for all coordinates, of the base pages from the axial indexes. The slab is a $(d-1)$-dimensional array, whose addresses are calculated using the usual array indexing formula in $d-1$ dimensions.

We have left out something important. Our dynamic multipage array grows at its faces, but we should be able to split pages in the middle of the space. When an inner page is split, the new slab created is considered to be a new face, and the axial index that has a new entry for it is simply re-ordered. (There is no harm in having "pointers" to slabs of pages, or even to individual pages; it is pointers to individual *records* that are inefficient.)

We can return to the issue of construction, which is now just a matter of deciding when

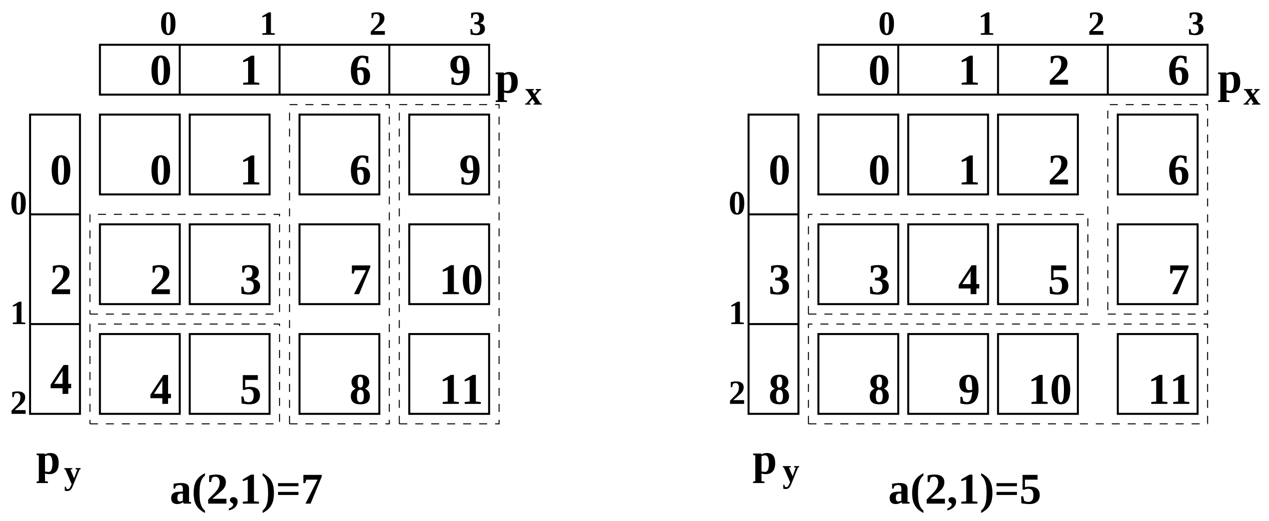


Figure 23: Addressing Dynamic Arrays

and which way to split. Here are six criteria, which we can mix and match into a family of algorithms.

Split Criteria

1. (π) If $\pi > \pi_0$ then split.

2. (α) Split unless this makes $\alpha < \alpha_0$.

Direction criteria

3. (Shape) Increase f_i for the axis, i, for which V_i/f_i is largest (in order to equalize all V_i/f_i, as far as possible).

4. (π) Split in the direction so that π is minimized. (N.B. Keep a log of overflows in the axial index for each row, column, ..)

5. (α) Increase f_i for the axis, i, for which f_i is largest (to create least number, n/f_i, of new pages and so decrease α the least).

Shift criterion

6. (π) If $\pi > \pi_0$ and shifting a boundary will make $\pi \leq \pi_0$, shift in the direction so that π is minimized. (See (4).)

Note that shape and α criteria are easy to calculate. The π criterion in (1) must be tested by doing or simulating the shifts or splits, and so is much more expensive. However, the π criteria deal directly with what is usually the important consideration, namely keeping the probe factor down. For determining the direction of the split, criterion (3) is probably more important than (4), in order to keep the address space the same shape as the data space.

Algorithm MPI (Multipage Insert)

A collection of alternative algorithms:

• 6, 1, 3, 4, 5	emphasizes π
• 6, 1, 3, 5, 4	π, then α
• 6, 2, 3, 5, 4	split emphasizes α
• 6, 2, 3, 4, 5	split using α, then π

References

[Knu73] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume III. Addison-Wesley Publishing Co., Reading, Mass., 1973. 1st edition.

[Lit80] W. Litwin. Linear hashing: a new tool for file and table addressing. In F. H. Lochovsky and R. W. Taylor, editors, *Proc. 6th Internat. Conf. on Very Large Data Bases*, pages 212–23, October 1980.