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

November 12, 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.

1 Logarithmic Files

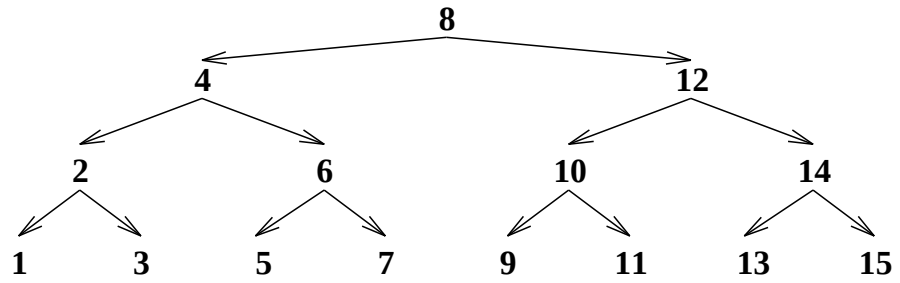
Logarithmic files are structured as trees. This makes it possible to find a given record in a logarithmic number of comparisons, or, more importantly for secondary storage, a logarithmic number of accesses. Not all trees give logarithmic performance, however. We can start with the binary tree, familiar from searches in RAM. Figure 1 shows that a "complete" binary tree behaves ideally (case 1.), but that the same construction process can give a degenerate "tree", whose performance is linear, not logarithmic at all (case 2.).

This perfomance depends on the order in which the data was presented to the construction algorithm. The construction creates the root node with the first record, then compares the second record to this and creates a left or a right descendent, respectively, according to whether the second record is smaller or bigger than the first. It continues in this way, and we can see that with a data order such as `8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15` we will get the ideal (case 1.). But if the data arrives in the order `A, B, C, ..`, or `Z, Y, X, ..`, or a variety of other orders, we have degeneracy (case 2.).

1.1 B-Trees

It would be very nice if we could find a tree structure which satisfies the following rules.

1. Ideal ($h = \log_2 N$)



2. Degenerate ($h = N$)

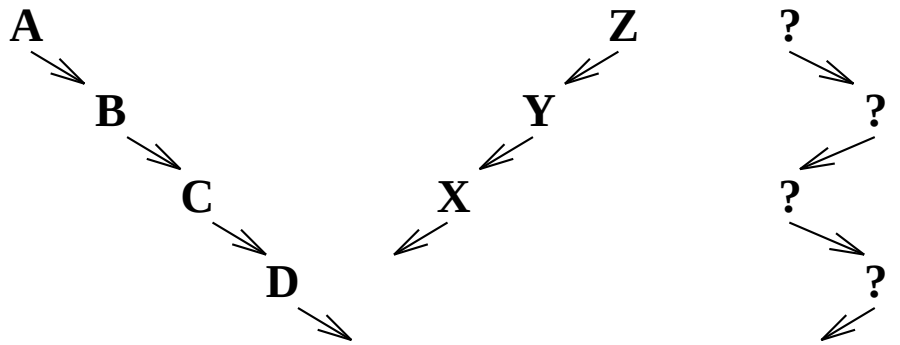


Figure 1: Degeneracy in Binary Trees

1. The root has at least two subtrees (unless it is a leaf).
2. Every node (apart from root or leaf) has s subtrees, $f/2 \leq s \leq f$.
3. All leaves are on the same level.

The third rule guarantees that the tree is never degenerate; rather it is what is termed *balanced*.

The structure that satisfies these rules is called a *B-tree*[BM72]. This is a tree structure expressly designed for secondary storage, in which the cost is measured in numbers of accesses rather than in numbers of comparisons. Each node of a B-tree is a block of data on secondary storage. It contains many records, which may each be compared with the record being sought at relatively little cost, once the block has been found and transferred to a buffer in RAM. In fact, we often make the assumption, untrue but a useful crude approximation, that the comparisons are free, and that the access is the entire cost.

Implicit in the above discussion is that binary trees are *dynamic*: they handle volatile data, at least insofar as that new data may be added at any time. B-trees are also dynamic, and before we look at them we discuss a strategy for dealing with dynamic data, which will be useful for data structures we will come to later.

The strategy is *preserve the access method* under additions and delations. That is, if we decide that, say, B-trees, are to be accessed in a certain way, then they must still be accessed in this same way no matter how much they have grown or shrunk. Preferably, preserving the access method is an automatic part of the insertion and deletion processes, and does not need to follow from reorganizing the entire file.

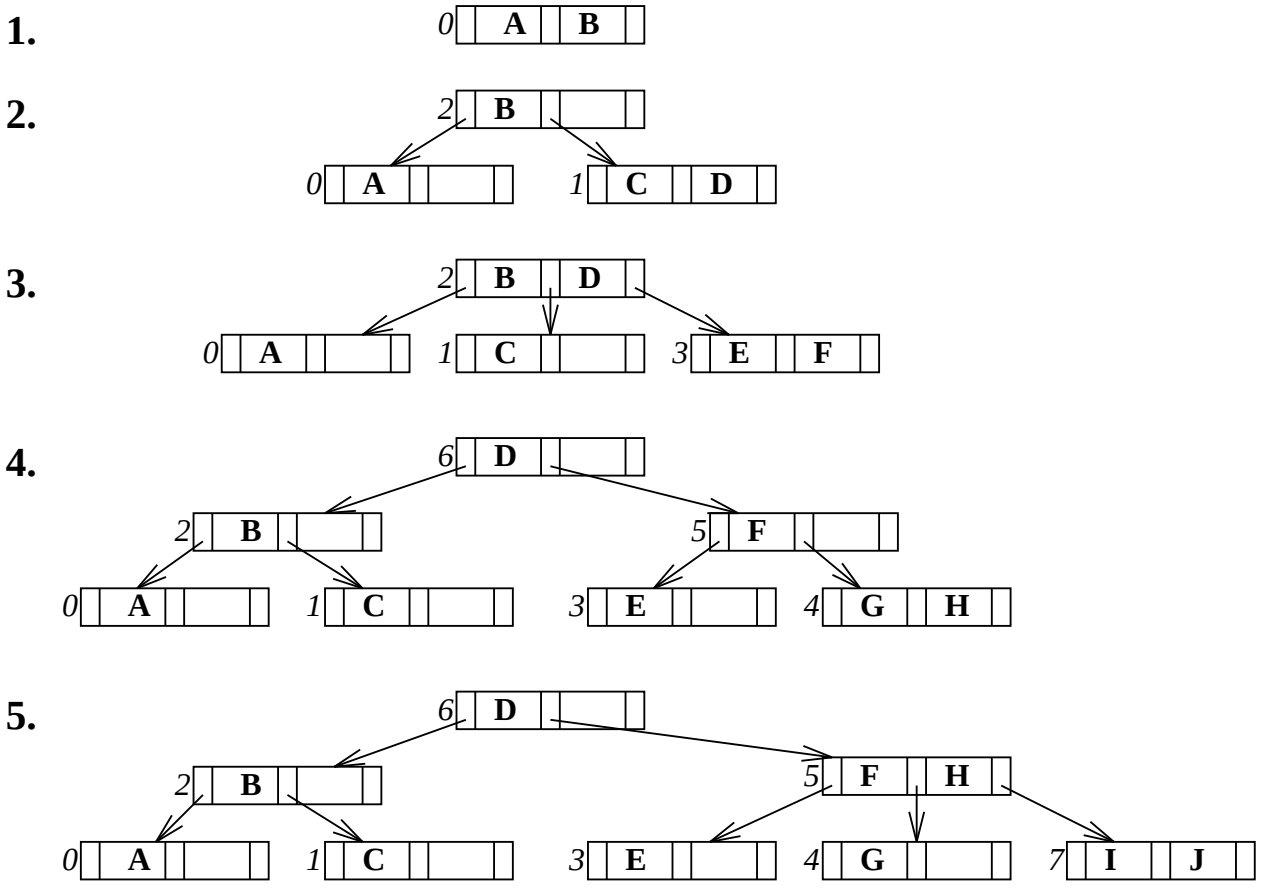


Figure 2: Growing a B-tree with $f = 3$

Associated with this strategy is a tactic which is also generally applicable: *split the blocks on growth; merge them on shrinking.* Let us see how this tactic generates a tree structure which satisfies all three rules above: the B-tree. Figure 1.1 shows the growth of a B-tree with records `A, B, C, D, E, F, G, H, I, J, ..` being added in that order.

The maximum fanout, f, of the B-tree shown is 3, which means that any one node holds at most $f - 1 = 2$ records: if the record being sought is lower than the first record in the node, descend the first pointer; if it is higher, but lower than the next record, descend the second pointer; .. ; finally, if it is higher than all records in the node, descend the last pointer.

Figure 1.1 shows, in five stages, the addition of ten records to the B-tree. First, `A` is added, creating the root node. Second, `B` is added, and there is room in the second slot of the root node. Third, `C` is added, but overflows the root node. So the root node splits into two nodes, one with `A`, the other with `C`, and `B` is moved up to discriminate between the two nodes at a higher level: well, there is no higher level, so a new root node is created, with `B` in it. Fourth, `D` is added, and there is room for it in the second slot of the node containing `C`. This completes the first two stages shown. In stage three, `E` and `F` are added, splitting the rightmost leaf, filling the root node, then filling the new rightmost leaf. In stage four, another level is added to the B-tree, by creating another new root node.

In figure 1.1, a "record" is shown with a single value, such as `A`: of course, the record would generally have many fields, and only one of these fields would be used for the comparisons.

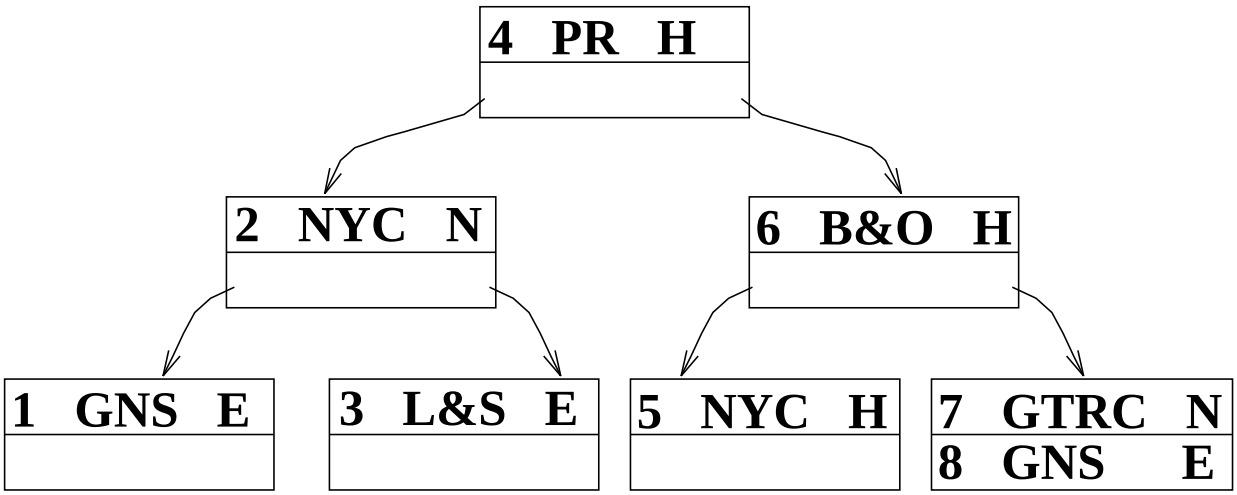


Figure 3: A B-tree Showing Full Records

Algorithm BI: (Insert search key *sk* into B-tree.)

BI1. (Search, using **Algorithm BS**, returning pointer r.) If *sk* found, terminate successfully. (If the search key is not a key, i.e., does not uniquely identify a record, the match must occur on sufficient fields to constitute a key, or all matching records must be found.)

BI2. (Subtree pointers.) Set $p, q \leftarrow$ null.

BI3. (Empty.) If $r =$ null, create root node *q, sk, p*. Terminate.

BI4. (Full?) If node r not full, insert entry *sk, p*. Terminate.

BI5. (Split.) Split node r, including *sk*. Set $p \leftarrow$ new node, $q \leftarrow r, r \leftarrow$ parent of r, *sk* $\leftarrow k_{\lceil f/2 \rceil}$. Goto BI3.

Figure 4: Inserting a Record into a B-tree

Figure 3 shows full records of three fields, with the comparisons to be done on the first field (values `1..8`).

We do not give **Algorithm BS** to search for a single record in a B-tree, but figure 4 is the algorithm to insert a new record.

Step BI5 specifies the split procedure: the full node is split in two, with the middle key, $k_{\lfloor f/2 \rfloor}$, moving up into the parent node and the entries after it moving into a new node (figure 5). If the parent node is also full, the process repeats until, if necessary, a new root node is created, in which case the height of the tree increases by one.

Figure 5 shows a single node (dashed box) which has been overfilled: it has $f+1$ pointers, p_i, and f records, k_i. (This condition could happen in the RAM buffer, but not on secondary storage.) So it is just about to split. It splits in the middle (left and right solid boxes), by moving $k_{\lceil f/2 \rceil}$ up to the parent node, as in step BI5. The resulting two nodes are each half full.

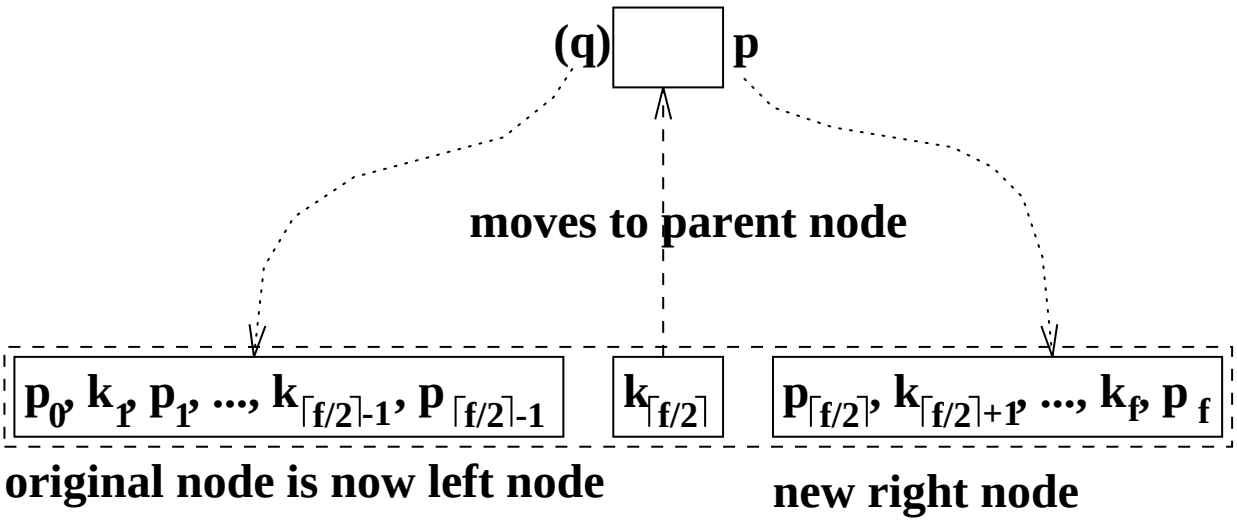


Figure 5: Splitting a Node of a B-tree

A problem with the B-tree so far is that it can be as little as half-full: half the space can be wasted. This is a direct consequence of the fact that one node becomes two each time we do a split. So if we could split two nodes into three instead, the tree would not drop below 2/3 full. This is the basis for the *B^*-tree* ([Knu73], Section 6.2.4): nodes are paired and excess records in one are "rotated" through the parent node into the other, and when both are full, they are split into three, according to figure 6.

Figure 7 shows an example where $f = 3$. Notice that we are required to do 1-for-2 splits whenever there is only one descendent node on a level, so the code is more complicated than before and we cannot completely guarantee the lower limit of 2/3.

A more fruitful approach is to tackle the problem of decreasing the tree height by increasing the fanout. Since we suppose that blocksize in bytes (node size) is dictated by other considerations, it is part of this problem that we must increase the fanout without changing the number of bytes in a block.

The size of pointers can also be assumed to be fixed. So the only way to put more pointers into a node (bigger fanout, f), and hence more entries ($f - 1$), is to make the data entries smaller. The obvious way to do that is to store not whole records but just the keys. The whole records are put only in the leaves. This leads to a B-tree with two kinds of nodes: whole records in the leaves, pointers and keys only in the internal nodes. It is thus called an *inhomogenous* B-tree (also known as a B^+-tree). Figure 8 shows an example with 32-byte nodes and $f = 5$. The internal nodes have f four-byte pointers and $f - 1$ three-byte keys. The leaves have three ten-byte whole records. Keys and records are distinguished by using lower and upper case, respectively, e.g., `A` for the record, `a` for its key.

B-trees were designed for high *volatility*. Clearly, from the foregoing, they can grow indefinitely without degradation. By the inverse process o merging nodes, they can also shrink without degradation. Arbitrary changes to data stored in a B-tree either do not involve the key field on which the B-tree is organized, in which case no rearrangements are needed and the change is easy, or else they do. If the key field is changed in any record, this is tantamount to deleting the record from its present postion and adding it in some new position.

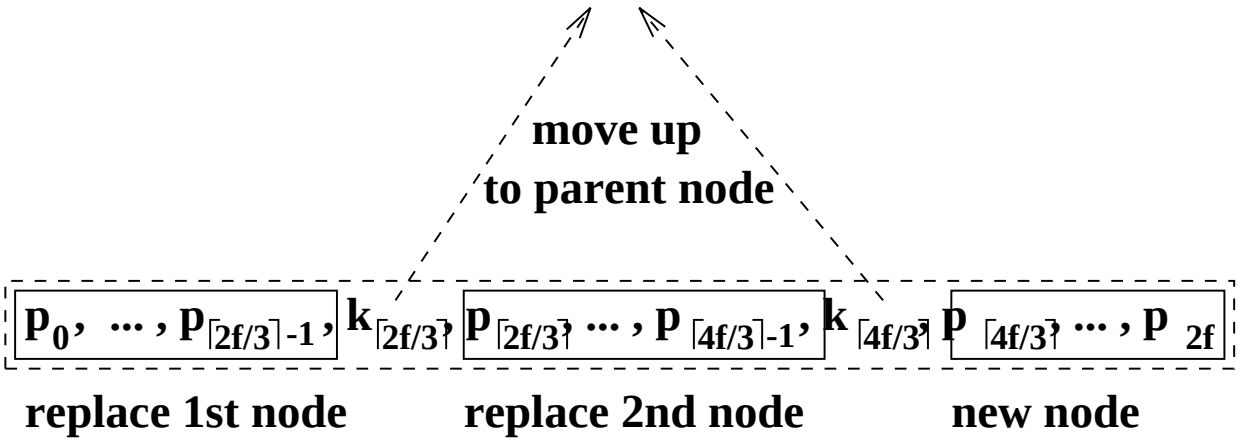


Figure 6: Splitting a Node of a B*-tree

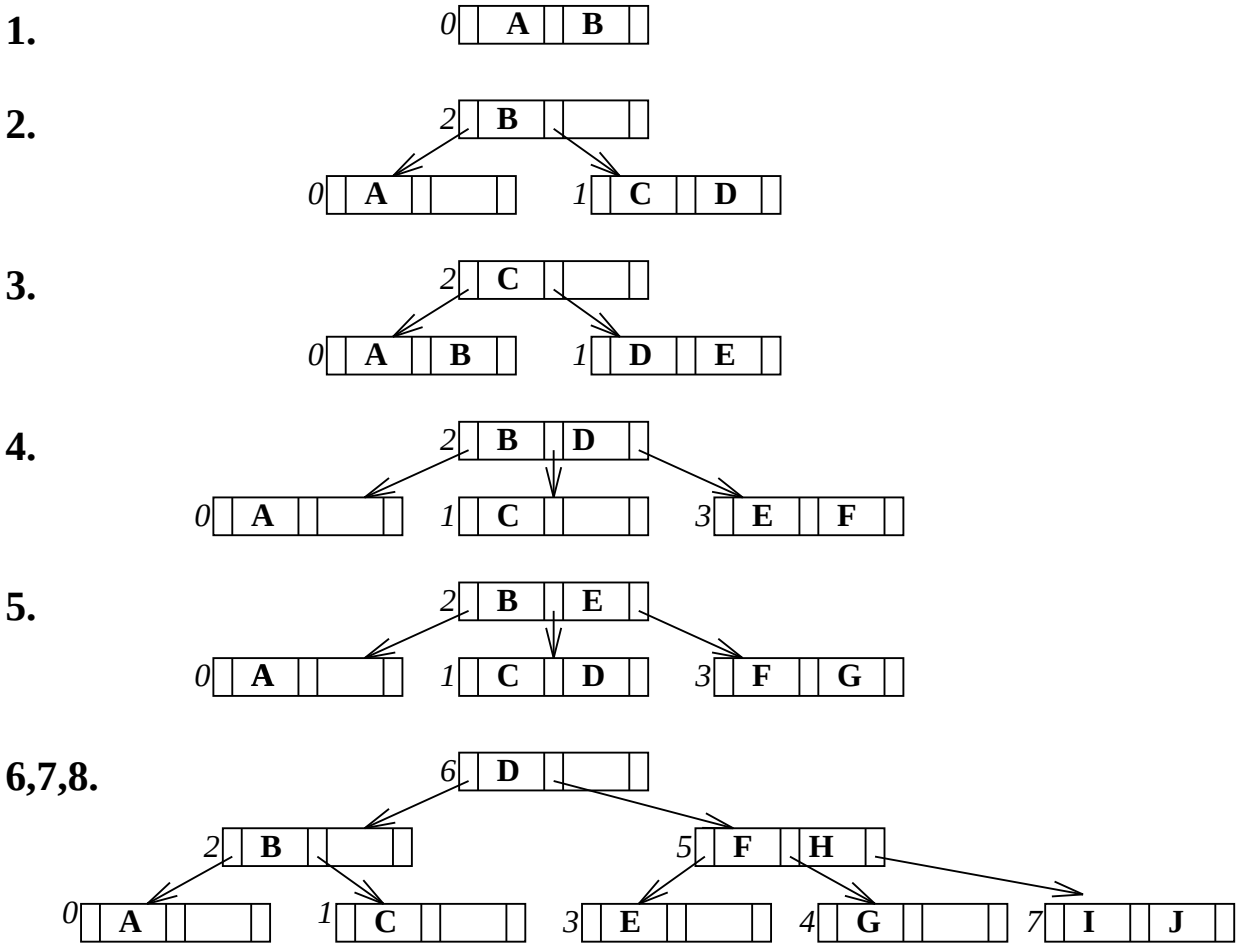


Figure 7: Growing a B*-tree, $f = 3$

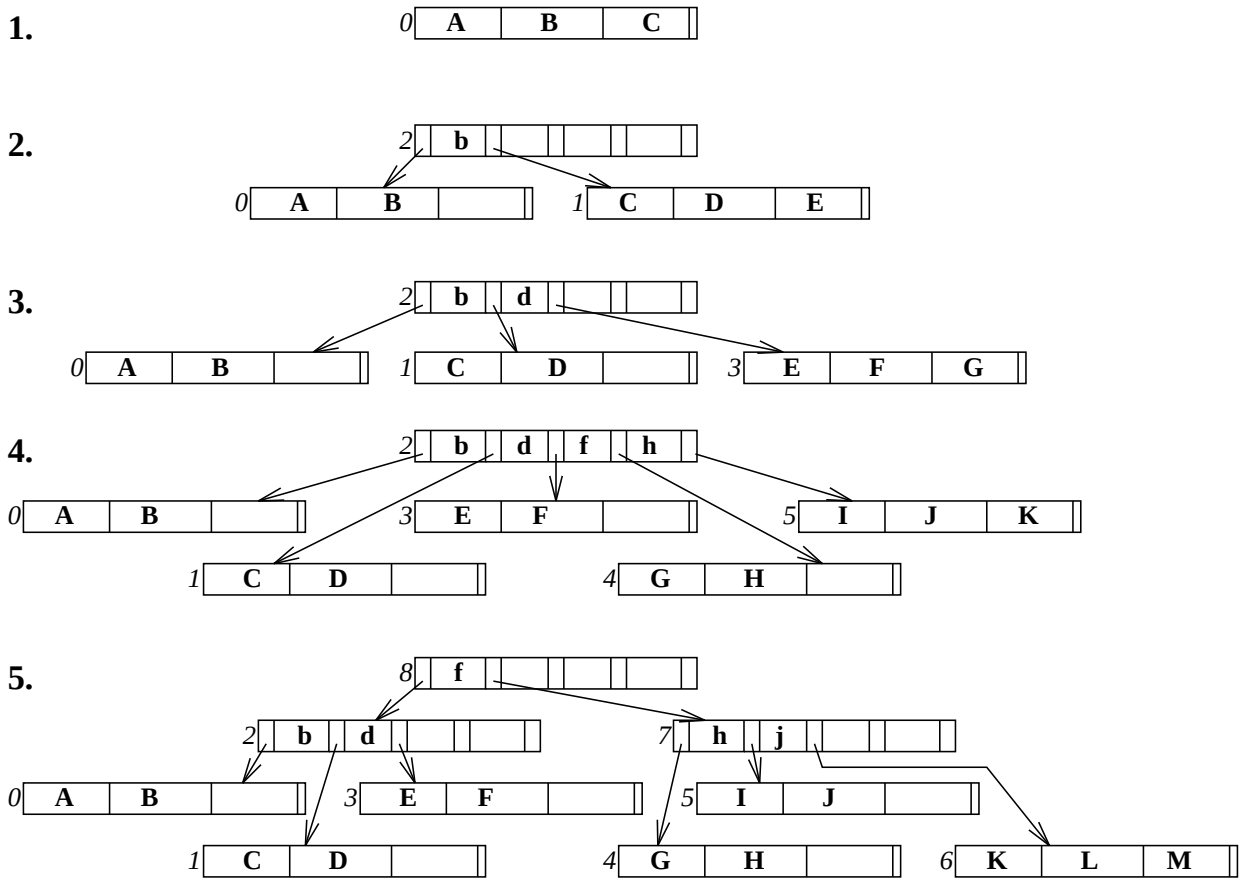


Figure 8: Inhomogenous B-tree (B^+-tree)

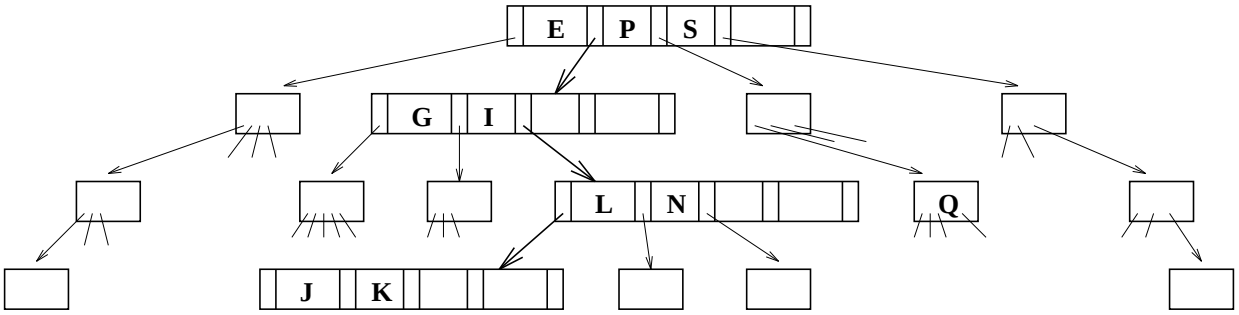


Figure 9: Activity: Range Search in a B-tree

So B-trees are good for high volatility of all kinds. There is clearly no problem with low volatility either.

B-trees are organized according to one, favoured, field, or possibly according to a predetermined combination of more than one field. Searches using other fields cannot be better than linear in cost, while searches on the key field(s) are logarithmic in cost. It follows that B-trees are not good for high *symmetry*. They are of course good for low symmetry.

For low *activity*, the number of accesses is the height of the B-tree, at worst. (Homogenous B-trees can find a record in one access, if the record is in the root node.) For high activity, we could simply read all the nodes of the file, following whatever order the blocks are stored in. But there is a better way if the search is a *range search*, from one value of the key to another. We use the B-tree structure to find the first value in a logarithmic number of accesses, then we traverse the nodes of the tree in in-order until the high value has been reached. This way, we can complete the search without reading all the blocks. So B-trees are good for both low and high activity, and we notice that *order-preserving* data structures are particularly good for range searches of arbitrary activity.

In figure 9, we see how such a search might find all records from `K` to `Q`. If $|\text{RAM}| = \mathcal{O}(\log N)$ then each needed page is retrieved only once in the traversal, even though there is no provision for back-pointers. In the example, four nodes are assumed to fit into RAM.

1.2 Tries

A *trie* is a tree structure in which the nodes are empty and the data is stored at the edges. It was originally known as a *digital tree* [dlB59], but the shorter name, "trie" (pronounced "try", but derived from "information re*trie*val") [Fre60], is mainly used. Tries provided the first sublinear substring-finding algorithm, *PATRICIA* ("Practical Algorithm To Retrieve Information Coded In Alphanumeric") [Mor68], which was subsequently adopted by the Oxford English Dictionary when digitizing the 500-megabyte New O.E.D. [GBYS92].

Here, we shall explore tries for data consisting of records of fixed-length fields. At the end of the section, we shall mention some of the advantages of tries for both this kind of "structured" data and "non-structured" text data. It is significant that the same basic trie idea serves for both, and may provide a common implementation for all kinds of data, including record data, text, spatial data, ...: a common foundation for multimedia, for example.

The term "digital tree" gives away the central idea. Each node has edges branching out from it, corresponding to all the possible digits, or characters, or bits, that might be next in the data. When the maximum fanout is 2, the trie is a *binary trie*, and each level of the

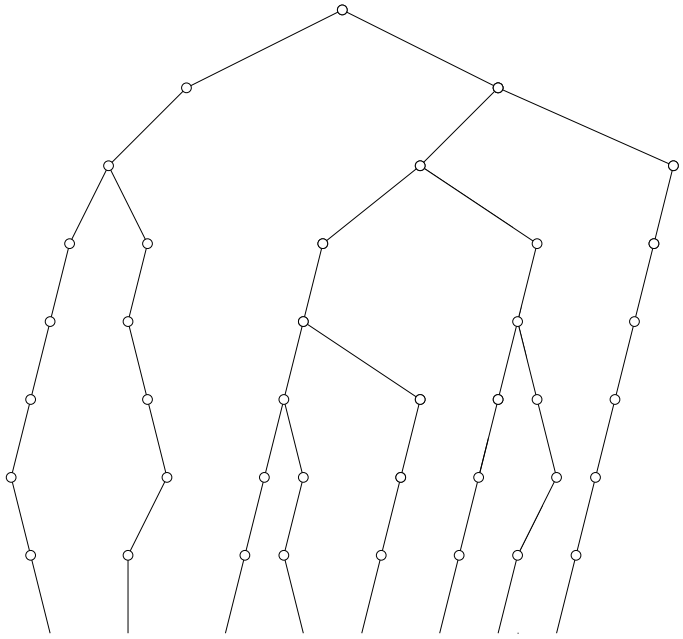


Figure 10: A Trie of Eight Records

trie corresponds to a bit in the representation of the data. Only left and right branches are possible from each node. If the bit is `0`, the branch is to the left; if the bit is `1`, the branch is to the right.

So the datum, `00000011`, would be represented by an eight-level trie, with branches *left, left, left, left, left, left, right, right*. With this single datum, each node in the trie would have only one edge leaving it, either left or right. If we then added `10000000`, the root node would have two edges leaving it, both left and right, and the trie would consist of this combined node with two subtries (to the left, the last seven edges above; to the right, seven `left` edges).

We show the trie for the eight data values

```
00000011
00101100
10000000
10000101
10001000
10100000
10101100
11000000
```

in figure 10.

Notice that the trie holds all the information contained in the original data, yet, unlike the trees we have considered hitherto, nothing need be stored in the nodes. Notice also that *common prefixes* are stored only once. A common prefix is a sequence of bits at the beginning of more than one data field, such as `1010`. The first `0` bit of every datum starting in `0` is stored in the edge going left from the root, and the first `1` bit is stored in the right edge. Thus there is tremendous potential for compression of data stored as tries, especially if we can avoid storing pointers to subtries as well.

1.2.1 Pointerless Representation

We look at a simple way of storing tries which needs only two bits per node. There are four possibilities: a node has two descendents; a left descendent only; a right descendent only; and no descendents. (In the latter case, it is a leaf.) We use the two bits with a bit corresponding to each of the possible descendents: `1` if there is a descendent, otherwise `0`. Here is the result for the example trie. (We have aligned each level to the left, because this makes the picture similar to the data structure we will soon be looking at. We also have left gaps in each level, so that it can be related to figure 10; but these gaps are not needed by the algorithms. We do not show the leaves, because they are all on the ninth level, and they are the only nodes on the ninth level.)

```
11
10    11
11    11              10
10 10 10        10    10
10 01 11        11    10
10 01 11    10 10 01 10
01 10 10 10 10 10 10 10
01 10 10 01 10 10 10 10
```

You should trace each of the eight data fields down through this trie: `00000011` will follow the leftmost column, while `11000000` will keep as far right as possible.

You should also note the number of bits of the trie needed to store the 64 bits of the eight original data values: 39 2-bit nodes, or 78 bits. This is less than the 2×64 we would expect from using two bits per bit. The sharing of common prefixes has compressed the data. For such a small file, the "compression" ratio is $78/64 = 1.2$: for larger files, we will see that the ratio can be 0.1 (that is, compressed by 90%) or lower.

In general, how do we navigate a trie represented in this way, with no pointers and only two bits per node? We count `1`-bits. Suppose we wanted to find if `10001000` is present. The first `1` matches the `11` of the first node, (specifically, the second `1`): so we look for the second node on the next level. The second bit, `0`, matches this node, which is also `11`, (specifically, the first `1` of the node). So we have crossed two `1`-bits on the second level, which means that on the third level we will again look for the second node. This is again a `11`, and our scan of this level can stop on the first of these `1`s (since the third bit of `10001000` is `0`), having crossed three `1`-bits. So, on the fourth level, we will need to check the third node. And so on.

Note that this means we do not need the columns aligned the way they are shown above, and we can just write each level as a sequence from left to right.

```
11
10 11
11 11 10
10 10 10 10 10
10 01 11 11 10
10 01 11 10 10 01 10
01 10 10 10 10 10 10 10
01 10 10 01 10 10 10 10
```

Furthermore, we do not even need to distinguish the levels, but just write the whole thing as a sequence of bit pairs. This is because we can also determine the total numbers of nodes

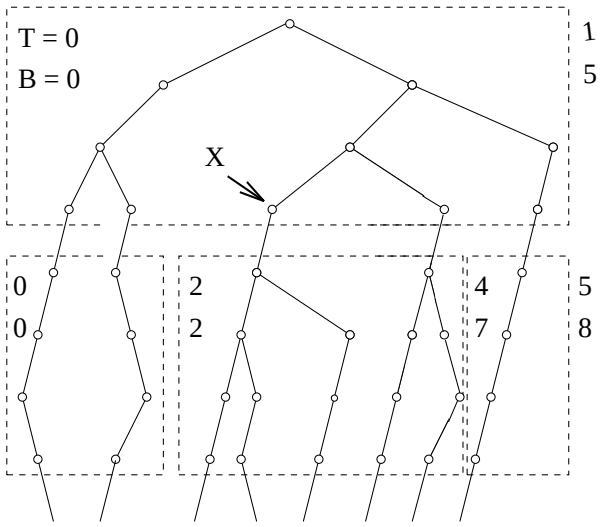


Figure 11: Paged Trie

in each level by counting *all* the 1-bits in the level before: 11 in the top level means two nodes in the next; 10 11 in the next means three nodes below it, and so on.

However, this is all a little too clever. How are we searching the trie? Sequentially. We must look at all the 1-bits on each level up to the node to be compared (which we can call *next* for this level) in order to find *next* for the next level; and we must look at *all* 1-bits on the level in order to find what we will call the *last* node for that level. This means that we must look at every bit of every level in the trie (except we may not need to search the last level beyond its *next* node).

We do not want to do a sequential search. Our search should not be more expensive than logarithmic. How do we fix this? By doing most of the counting in advance, and storing the result as part of the trie. How should we do the advance counting? We are contemplating tries for secondary storage, and the logarithmic cost should be measured in numbers of accesses to disk or other secondary storage medium. So we should break the trie into pages and record counts at the level of each page.

Here is the idea, due to Orenstein [Ore83]. We divide the trie into *levels of pages* with a certain number of levels of *nodes* in each page level. Within the page level, pages divide the trie so that trie edges cross only the top and bottom boundaries of the page, never the sides. (We might call these two dividing operations *slice* and *chop*: slice (horizontally) the trie into page levels, and chop (vertically) each level into pages.) For each page, there are two counts: T counts the number of trie edges crossing the *top* boundary *of all the pages before it on the same level*; B counts the number of trie edges crossing the *bottom* boundary of all the pages before it on the same level. If these counts are stored correctly, they enable us to calculate which page of the next level a given trie edge will cross to from the page we are currently on in the present level.

For simplicity, we will assume that the counts are stored separately, and the page addresses are stored with them. If we assume the whole set of counts can be initially loaded into RAM, this restricts our file size to n pages, where $3n$ integers can fit into RAM: not a severe restriction, and one that could be overcome by storing the counts on the pages.

In the example of figure 11, the address (page) of the (left) descendent of the node marked "X" can be computed as follows. B for the page "X" is on is 0, which means that no links

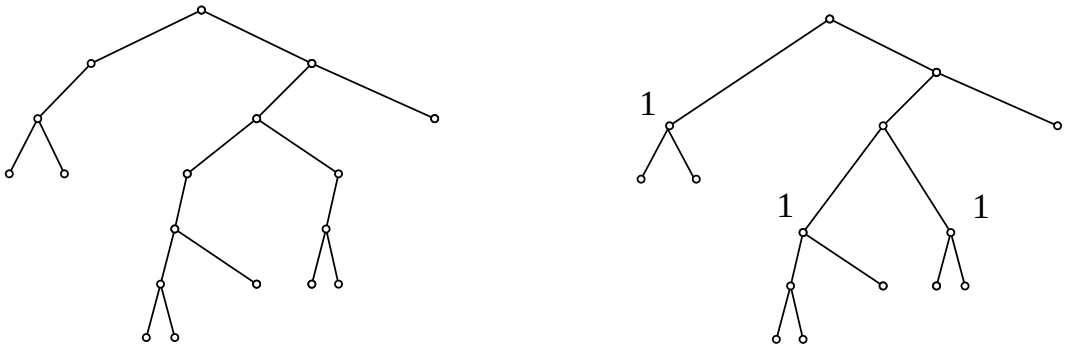


Figure 12: Truncated Tries

(a) Truncated Trie (b) Patricia Trie

have already descended from earlier pages (to the left) on that level. The descendent of "X" is the third link in its page. So in the next level, we must look for a T which does not exceed 0+3=3. Candidates on the level below are $T = 0$, $T = 2$, $T = 4$ and $T = 5$, and 2 is closer than the other value (0) less than 3, so we know that the descendent of "X" that we are looking for is on the second page of the next level.

Here is the trie paged with four node levels per page level, and with a search for `10001000`, using "`x`" this time to indicate which "`1`" is hit by the search at each node level.

```
T 0  1x              T 1
B 0  10 x1           B 5
     11 x1 10
     10 10 x0 10 10

T 0  10 01     T 2  1x 11            T 4  10   T 5
B 0  10 01     B 2  11 x0 10 01      B 7  10   B 8
     01 10          10 10 x0 10 10         10
     01 10          10 01 x0 10 10         10
```

1.2.2 Truncated Tries

The trie discussed so far is rather "leggy": it has a number of paths which, after some branching, just form a single leg down to the leaf. The bit pair representation requires two bits for each node, but the nodes in the legs each have one `1` and one `0`. These nodes could each be represented with a single bit, namely the corresponding bit in the data value that generated the leg.

These thoughts lead to the *truncated trie*, in which paths descending from the root are truncated after the last bifurcation. Figure 12 (a) shows the previous trie truncated in this way. There are now only 18 nodes, instead of 39.

The bit pair representation of the truncated trie is similar to that for the original trie, except that it also includes (using one bit per bit instead of two) the data that has been removed by truncation. So the representation is hybrid, sometimes using two bits per node and sometimes one, and a distinction must be made between the two kinds of representation. We use `00` for the lowest node in a path, and associate with it a bitstring representing the remainder of the path.

As long as the data values are fixed in length, we do not need any special delimiters to distinguish node data from non-node data: we can calculate the position and size in each page of each item. The nodes, and T and B, tell us, by the methods we used above, how many nodes there are in each level. For each `00` node, there will be a bitstring giving the rest of the data value, of length $h - l$, where h is the length of the data value (height of the trie), and l is the node level where the `00` node was found. One way to store the page is to divide each node level into node and non-node data. We do not elaborate here.

A fancier truncated trie removes *all* nodes that are not branches (`11`) and replaces them by a count of how many nodes have been skipped. Figure 12 (b) shows an example, which now has only 15 nodes because three of the 18 were nodes that did not bifurcate, yet were not below the lowest bifurcation in their path. Note that an integer is associated with each node below nodes that have been skipped, *skip*, which tells how many nodes were skipped. This trie is called a *Patricia trie* or *patrie*, after the algorithm, PATRICIA, that first used it for text retrieval.

Note that the Patricia trie loses information (unless we store the missing bits), and so a reported success must be checked against the data stored elsewhere. Thus, the trie serves as an index to data rather than a representation of it. This is not useful for most record-based data, unless the data values are very long, but it is extremely useful for text, the context in which Patricia tries were first proposed. A text may be regarded as a set of sequences of bits (or characters), each running from a given position in the text all the way to the end. These "semi-infinite strings", or *sistrings* are very long, and the trie representing them must be truncated as much as possible. It also must be compressed as much as possible, because the index can be even bigger than the data. (Consider an index which gives access to every character string of a text: it might have a pointer to every byte of the text, and the pointers could be 4-byte integers. Already the index is four times the size of the text. Patricia tries have been built that are about three times the size of a text they index every byte of [MS93].)

Storing the Patricia trie differs a little from the full and truncated tries because the Patricia trie has only two kinds of node, so we use only one bit per node: `1` for a node with two descendents, and `0` for a node with none. Associated with each `1` node, *skip* must be stored. If the Patricia trie is not an index, the missing bits must also be stored, both for nonzero *skip*s, and for truncated paths. If the data values are fixed in length, the missing data may be stored without delimiters and found by calculation, but for a Patricia trie index to variable-length data such as text, pointers to the data must be stored for each `0` node. (We can save bits in such pointers by pointing only to a data *page*, not to the individual byte: for instance, pointers to 1K pages are ten bits shorter than pointers to each byte. We would use a linear search, such as `grep`, within the page.)

1.2.3 Dynamic Tries

Although it is faster to build a trie with the data previously sorted, as we have done in section A.2, inserting data into or deleting data from an existing trie is not difficult, particularly if the counters and addresses are stored separately from the bits representing the nodes.

Figure 13 shows the simplest kind of addition—to a page which has room for the entire addition. The B values for all the pages to the right most be incremented. If we suppose that all the counters and pointers can fit into RAM, the changes can be made at no cost (in accesses to secondary storage).

Figure 14 shows an addition which forces a lower page to split. All pages must have a capacity of at least $2^t - 1$ bitpairs, where t is the number of node levels per page. This means that a complete subtrie of height t can fit on a page, or two or more partially complete subtries. If a page has only one subtrie, it need not be split during an addition. If it has

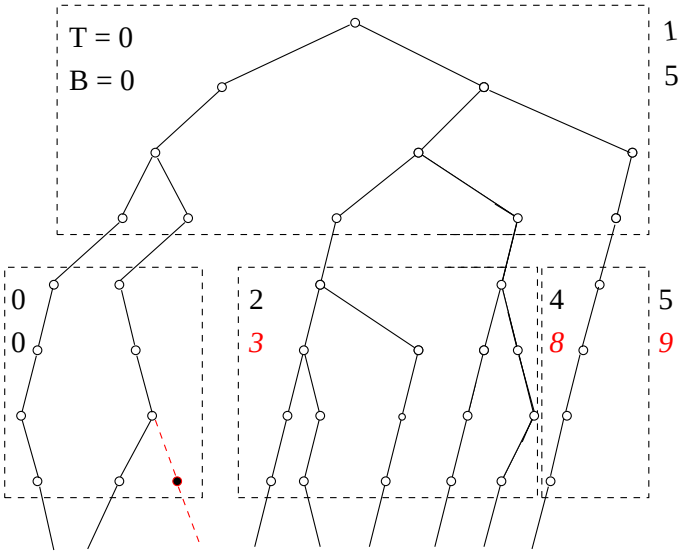


Figure 13: Adding `00101111`

two or more, a split can be arranged which has the most nearly equal number of nodes on both sides, of all splits that do not result in trie edges crossing a vertical page boundary. The emptiest such new page will have t nodes, leaving the other side with some $2^t - t$ nodes: possibly rotation techniques with neighbouring pages can even this out.

Figure 15 shows an addition which affects the T counters as well as the B counters, by propagating from a higher level of pages.

Finally, figure 16 shows a deletion which results in the fusion of two pages.

Splits and fusions can in principle be dealt with in either direction (top-down or bottom-up), and they do not affect the height or the balance of the trie.

An algorithm for insertions and deletions in a Patricia trie [Sha95] in connection with text (see section 1.2.5) must contend with the changes in the pointers to the text, and with the changes to the sistrings themselves (which, we recall, run to the end of the text, so insertions and deletions will alter *every* sistring beginning before the change).

1.2.4 Multidimensional Tries

Bentley's kd-tree [Ben75] stores multidimensional data by cycling among the dimensions as it builds or searches the tree. The field that is used to make the comparisons at any level of the tree is called the *discriminator* for that level. Figure 17 shows eight two-dimensional points and the order in which they have been inserted into a kd-tree. It shows the kd-tree, and it shows how the two-dimensional space is divided by the kd-tree. (By caveat, if a new entry matches an existing entry on the discriminator, the right subtree is searched. This could just as well have been the left subtree, as long as the rule chosen is followed consistently.

If anther order of insertion were chosen, the kd-tree, and the splitting of the space, would have an entirely different shape. (What ordering of this data would result in a completely degenerate tree? What would the space look like?)

The corresponding *kd-trie* does the same thing, but for each bit (or digit, or character) of each key value. With a trie, this has the effect of *interleaving* the bits (digits, characters) of all the searchable data fields.

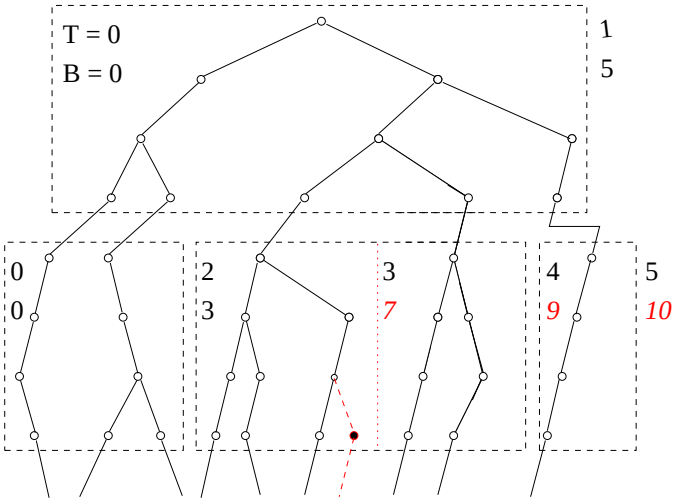


Figure 14: Adding 10001010

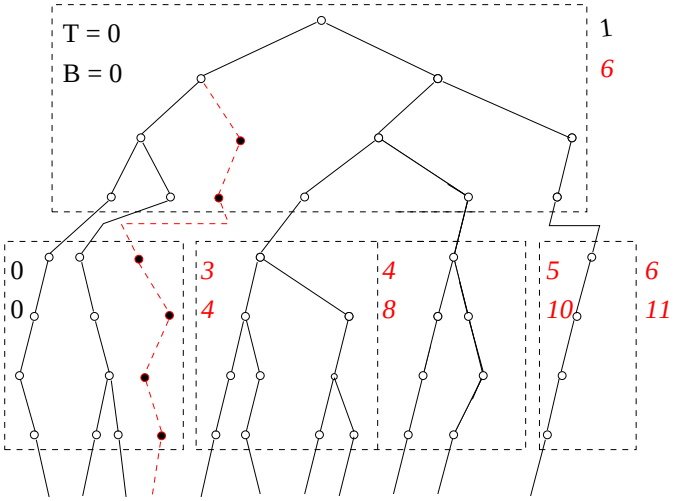


Figure 15: Adding 01011010

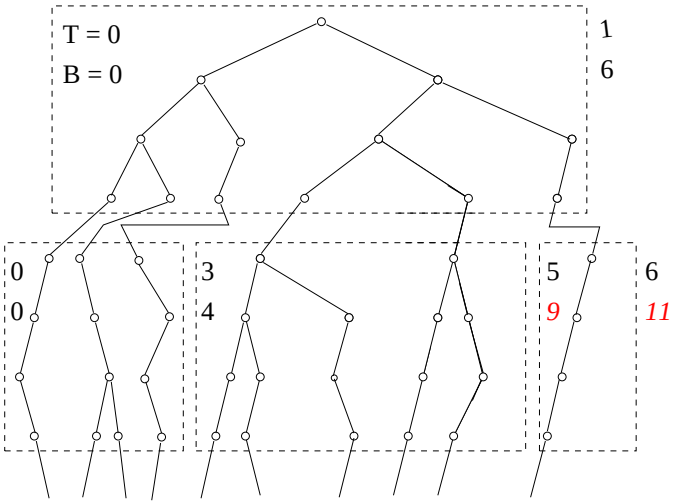


Figure 16: Deleting 10001000

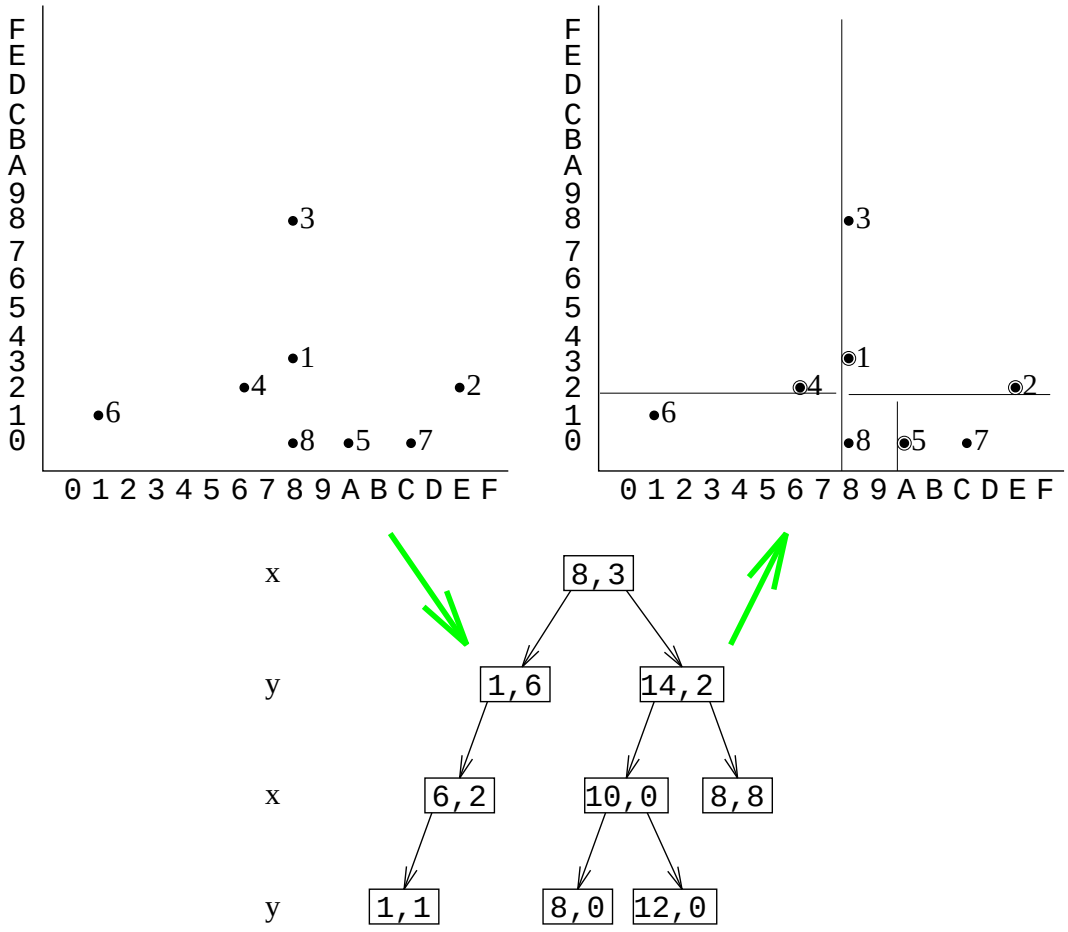


Figure 17: The kd-Tree

Consider the two-dimensional value, `(8, 3)`, or `(1000, 0011)`. To store this in a kd-trie (a 2d-trie, to be precise), we take the first bit of `8`, which is `1`, so we branch right. Then we take the first bit of the next key, `3`, which is `0`, so we branch left. Then back to `8`, whose second bit says left; the second bit of `3` says left; and so on: left, right, left, right, or `10000101`.

The resulting trie is just the one-dimensional trie constructed from values which are the components with their bits interleaved. Thus, the two-dimensional values

```
(1, 1)= (0001, 0001)⇒ (00000011)
(6, 2)= (0110, 0010)⇒ (00101100)
(8, 0)= (1000, 0000)⇒ (10000000)
(8, 3)= (1000, 0011)⇒ (10000101)
(10, 0)=(1010, 0000)⇒ (10001000)
(12, 0)=(1100, 0000)⇒ (10100000)
(14, 2)=(1110, 0010)⇒ (10101100)
(8, 8)= (1000, 1000)⇒ (11000000)
```

give the trie of figure 10. Figure 18 shows how the data, *independently of the order of insertion*, leads to the kd-trie, and how the kd-trie splits the two-dimensional space. Note that, while a tree splits the *data*, a trie splits the *space*.

An *exact match* search, such as for `(8, 3)`, in which we know all the values for the searchable fields, proceeds exactly as the one-dimensional search for `10000101`, after having interleaved the bits of `(8, 3)` to produce `10000101`.

A *partial match* search, say for `(8, ?)`, can return more than one value from different parts of the trie. However, only part of the trie needs to be searched, so the cost is substantially less than a linear search of the whole file. The partial match search can make decisions only at every other node of the trie, and it must (figuratively) follow both branches from any node in between. So, the search for `(8, ?)` will eliminate branches at each even-numbered node (starting from the root at 0), but will follow both descendents of any odd-numbered node. (It would typically do this by a depth-first search.)

Here are the nodes reached by the search for `(8, ?)`, with the affected `1`-bits replaced by `x`.

```
T 0  1x            T 1
B 0  10 xx         B 5
     11 x1 x0
     10 10 x0 10 x0

T 0  10 01     T 2  x1 11           T 4  x0   T 5
B 0  10 01     B 2  xx 10 10 01     B 7  x0   B 8
     01 10          x0 x0 10 10 10       x0
     01 10          x0 0x 10 10 10       x0
```

We see that three paths are followed down to the bottom level, resulting in retrievals of `(8, 0)`, `(8, 3)` and `(8, 8)`. In this tiny example, three of four pages are accessed, but in general only pages on the paths leading to valid results will be read from disk.

Note that the order of the eight two-dimensional values, above, looks a little strange. It is the ascending order of the values of the interleaved bits. If we plot this order in two dimensions for all possible values, it gives the fractal known as a Peano curve, or, in this context, *Z-order* [OM84].

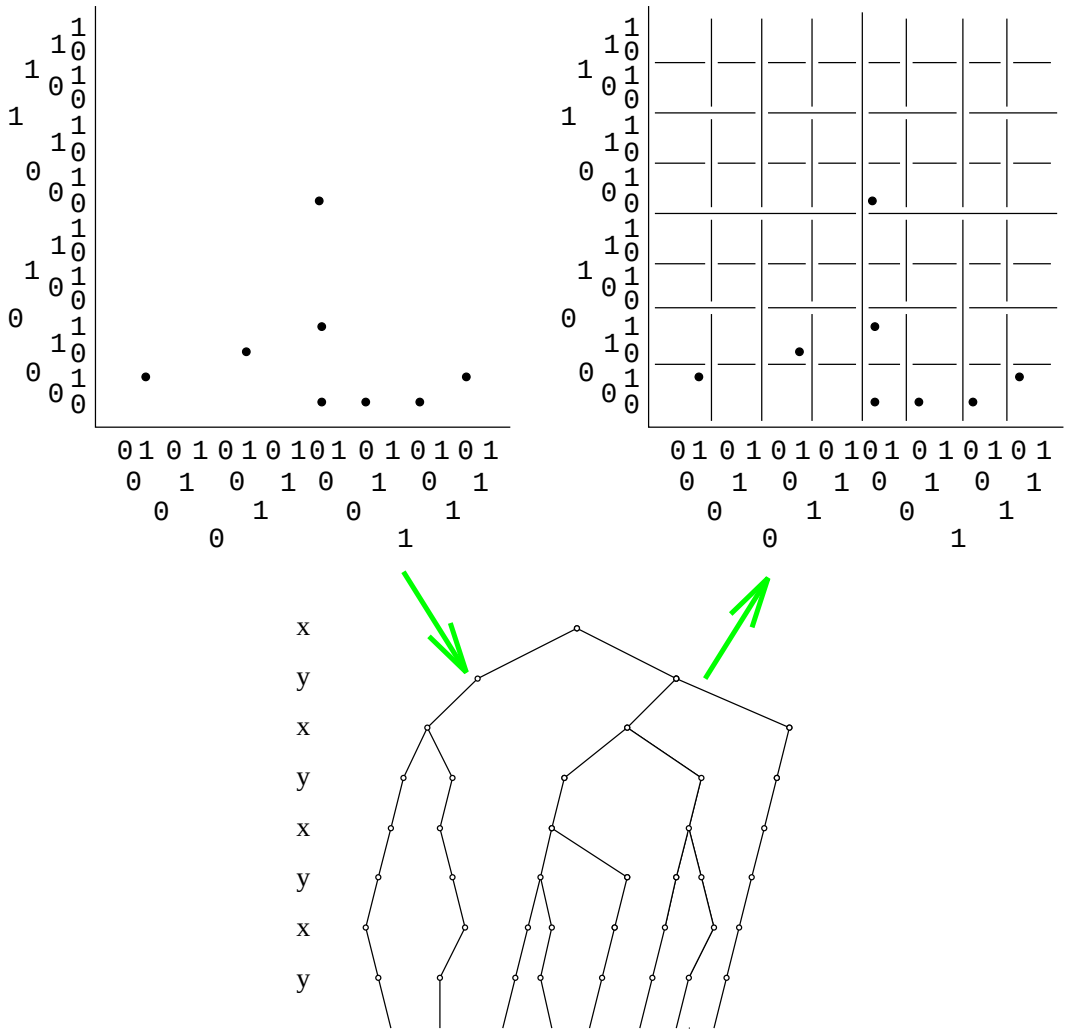


Figure 18: The kd-Trie

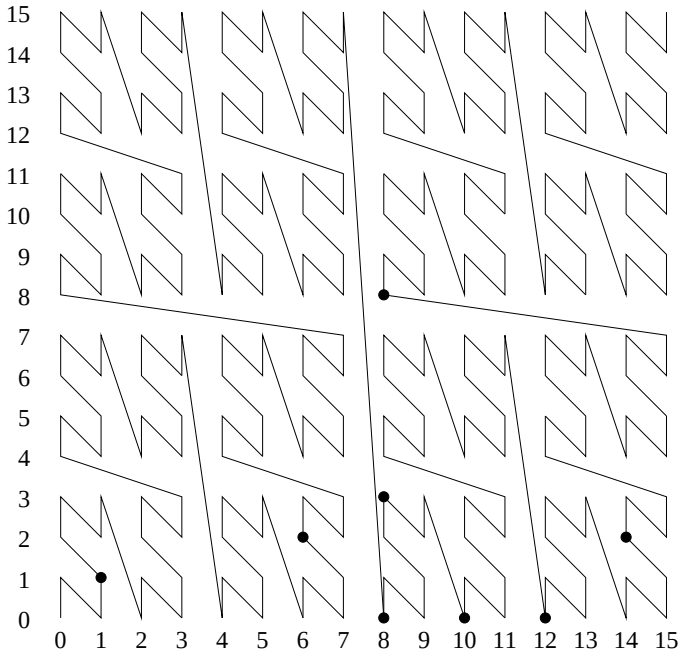


Figure 19: Z-order in Two Dimensions, Showing Eight Data Points

Figure 19 shows this curve for all possible values of one byte (eight interleaved bits). The eight data points discussed are shown. Note that the order in which they are met, as we follow the line from the origin, is the order they are given in, above.

Since bit interleaving is possible in any number of dimensions, so is Z-order. It is an exercise in visualization to imagine the shape of a Z-order curve in three (or more) dimensions.

Z-order is a spinoff from tries. It provides a linear order for multidimensional data with the property that two points close in the multidimensional space tend to be close in Z-order, and vice-versa. More precisely, for two dimensions, half the nearest neighbours in Z-order are nearest neighbours in 2-space; three quarters of the remainder are second-nearest neighbours in 2-space; of the remaining eighth, three quarters are seventh-nearest neighbours; and so on: an exponentially diminishing probability that nearest neighbours in Z-order are far from each other in 2-space. This ordering can be used, without constructing a trie, to map multidimensional data into data structures, such as B-trees, which work for only one dimension.

Note that bit interleaving, and hence Z-order, are easier to work with if the components have the same number of bits. Of course, if one component has, say, twice as many bits as the others, it can be visited twice in succession one of the times its turn comes up in the cycle; similar adaptations can be made for other differences in bit lengths.

A very general form of multidimensional query is the *orthogonal range query*. This is the conjunction of *range queries*, such as $(1.7 < x \leq 34.2) \wedge$ (`"joe"` $\leq y \leq$ `"sue"`). Figure 20 shows a two-dimensional orthogonal range query (a) and two special cases: in (b) and (d), the vertical range is the entire attribute; (d), moreover, has a horizontal range of only one value, a "partial match" query; and (c) has only one value in both ranges, the "exact match" or single-record query we have discussed before.

Kd-tries and Z-order can, in principle, process orthogonal range queries without reading any irrelevant records, i.e., records outside the range. They can do this by *probing* (logarithmic access) to the start of a range or subrange, then *scanning* (sequential access) the range

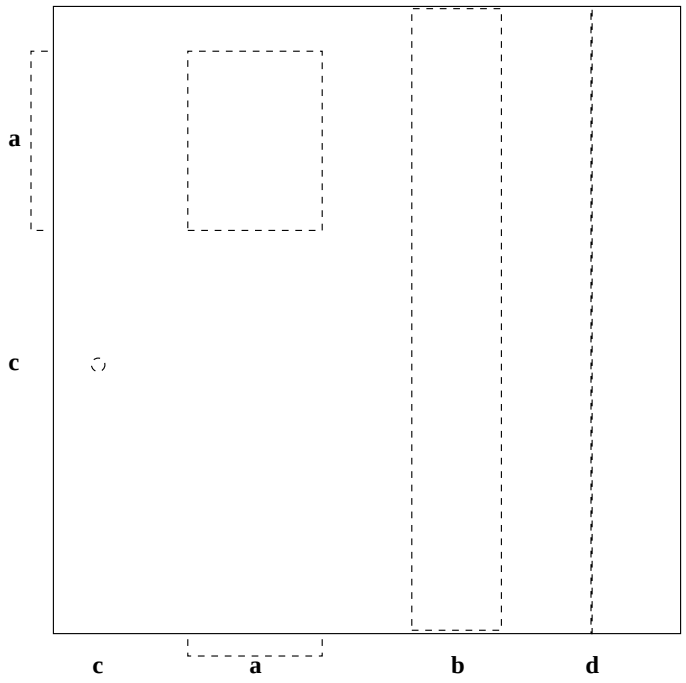


Figure 20: Orthogonal Range Queries, with special cases b) partial range, c) exact match, d) partial match

or subrange.

In practice, there is a trade-off between the costs of probes and scans, which depends on implementation parameters such as page size and even track and cylinder size. It may be cheaper to scan some extraneous data than to do another probe. Figure 21 shows four such queries in a Z-order space, and the numbers of probes and scans needed to avoid any extraneous reads. However, at the cost of a longer scan which goes outside the range, (b) and (c) can be answered in 1 probe and 1 scan each. (d) could be simplified to fewer than 4 probes and scans, with 2 probes and scans a possible good compromise (judging only from the picture, not on the basis of implementation-dependent cost analysis).

1.2.5 Results of some Research on Tries

We review two properties of tries and their benefits for data processing on secondary storage. The first is the ability of tries to compress data, which reduces the transfer times and increases the amount of data that can fit in RAM. The second is that data is stored in the trie edges, not in the nodes. If the most significant bits are near the root (the usual convention) this supports queries that require *variable resolution*. This is useful when an approximate answer to a query is sufficent, or when a series of improving approximations helps eliminate unwanted results.

The first two parts of this section are about compression and variable resolution, respectively. The third part reviews some of the capabilities tries offer for text retrieval.

Compression

Figure 22 shows how tries actually compress data at the same time as providing a search mechanism. Truncated tries were used for this measurement, and the records are 4 bytes long (so the files tested range up to 0.4 gigabytes). We see that under about 100 records, the

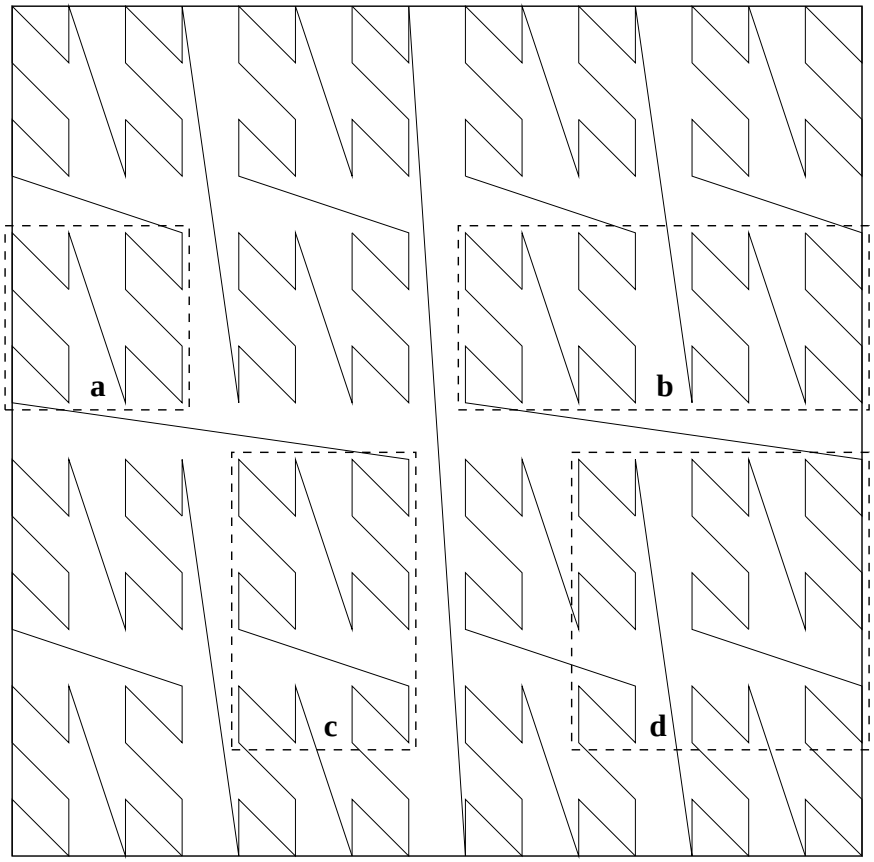


Figure 21: Orthogonal Range Queries for Z-Order a)1 probe, 1 scan; b,c) 2 probes, 2 scans; d) 4 probes, 4 scans

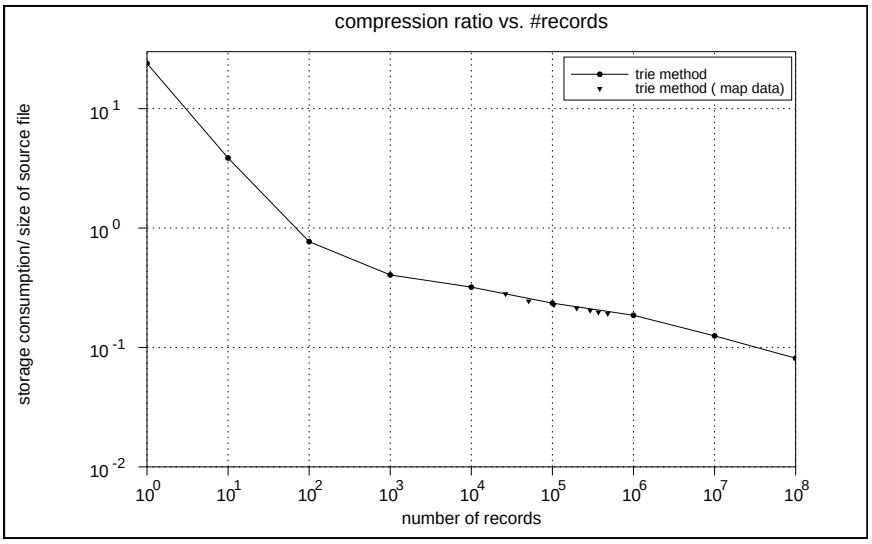


Figure 22: Trie Compression vs. File Size

trie is bigger than the original data, but beyond that, it compresses up to more than 90%. The more values stored, the more sharing there is of common prefixes. On the other hand, the more bits per value, the less the likelihood of common prefixes for a given number of records, and the compression will be less.

The plot also shows separate points measured on tries made from data from a contour map, in order to show how accurately the generated data reflects the behaviour of at least one type of natural data.

A direct consequence of this reduction in size is a reduction in access cost. Figure 23 shows trie performance compared with other file structures, in the case of 20% activity. Tries can be many times better than anything else.

Variable Resolution

Wherever data can be usefully approximated by ignoring the less significant bits, digits, or characters, trie storage permits us to discriminate among many data values by reading only the top portion of the trie, if the most significant digits are stored near the root.

Maps provide a good illustration. Suppose we have a gigabyte of map data, say, covering Canada at 1:50000 scale. If our display can show a megabyte at a time, we can use various data structures, including tries, to fetch 1/1000 of our map (activity 0.1%), say, Prince Edward Island, and display it to full resolution. Tries alone allow us to display an overview of all of Canada for the same 0.1% query cost. Furthermore, tries permit us to combine these two operations and show a larger portion, but less than all, say, of Saskatchewan, to the resolution permitted by the display. Other techniques would require us to read all the data and process it so that small features would disappear, say by replacing small wiggles by overall trends; or they would require us to store the map, preprocessed, several times, and present only the version that comes closest to the required resolution.

Figure 24 shows how a vector diagram is represented at various resolutions by the different levels of a trie. (The levels displayed descend in steps of four because each edge in the two-dimensional diagram is represented by its four ccordinates, with the bits interleaved in a four-dimensional kd-trie.)

Measurements [MS94] on queries performed on such variable-resolution tries show search times linear in the number of nodes retrieved (going up to 800 seconds for 12 million nodes,

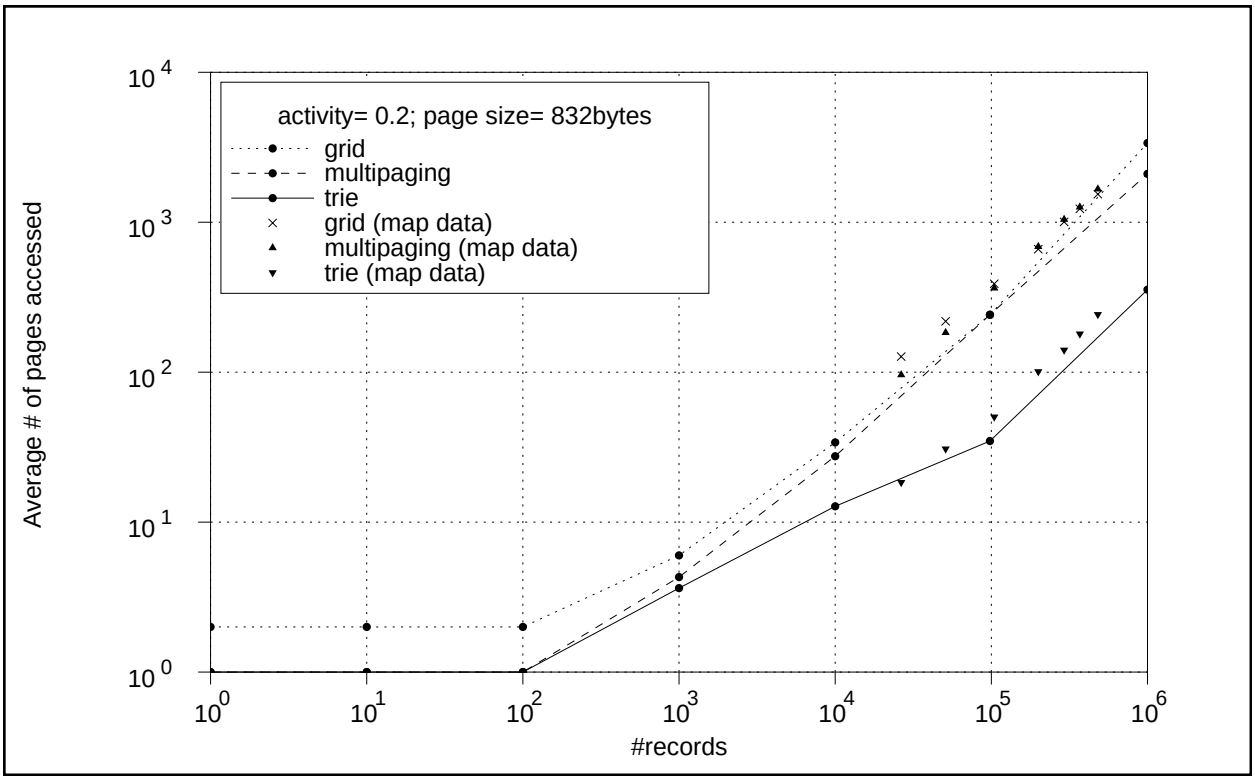


Figure 23: Access Cost vs. File Size

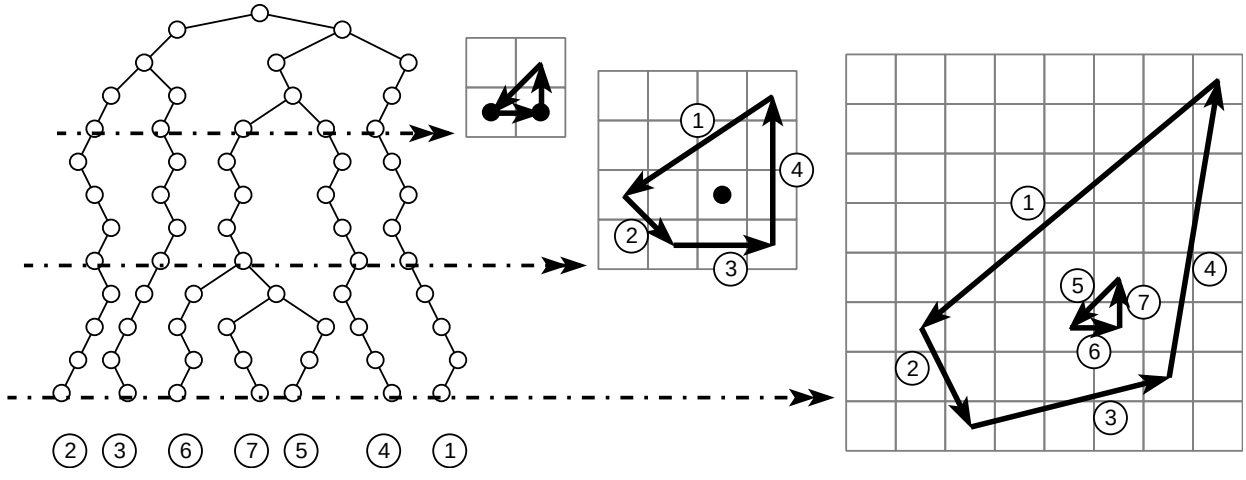


Figure 24: Variable Resolution Trie

on a 25MHz workstation). The trie in this measurement is 64 node levels in height, so the map can be retrieved at 16 different resolutions. This performance is optimal.

Text

Tries offer the best known method for searching texts in sublinear time, so for large texts, such as the 600 MB of the Oxford English Dictionary, they are indispensible. It seems inevitable, however, that the tradeoff is a large index. Before looking at tries, consider what must be the simplest approach to indexing a text of N bytes and n sistrings (see section 1.2.2), namely an array of pointers ordered according to the alphabetical order of the sistrings they point to, and using binary search [GBYS92]. Since each pointer must be at least lg N bits in size, the index will be

$$(n \lg N)/8 = 3.4n \text{bytes}$$

for $N = 100$ MB: if the text is indexed on every character, $n = N$, and the index is 3.4 times the size of the text.

This array contains nothing but pointers to the text, which we must have in an index, and so would seem to be as small as we can get. But tries can give a smaller and faster index: because the trie holds information about the data, as well as pointers to it, it avoids comparing the search key with the text, which the binary search must do repeatedly. This is faster, and pointers to the text may point only to pages, not to individual bytes, as we mentioned in section 1.2.2.

To show that tries are better, and Patricia tries in particular, requires quantitative details. Tries are hard to analyze mathematically, and we must do experiments. Although tries built from random and independently distributed data can be analyzed probabilistically [Dev82], text is neither random (`e` is more frequent than `f`) nor independent (`th` is more frequent than `tz`). Measurements [MS93] show that a trie built from text such as Shakespeare, the Bible or UNIX manual pages can be at least ten times as deep as a random trie. They also show that Patricia tries are noticeably smaller than truncated tries, despite the need to store skip data. Experiments with pointerless Patricia tries on 1MB texts (drawn from the above sources, Webster's *Ninth New Collegiate Dictionary*, and a collection of source programs in *C*) show storage requirements varying from 2 to 2.5 times the text size, for indexes to every byte of the text. Calculations [MS93] also suggest that 100MB texts, with the same distribution parameters, require indexes from 2.7 to 3.1 times the text size, better than the pointer array.

The tries save 12 bits per pointer by pointing to 4Kbyte pages, and this more than accommodates the trie storage of all the different bits of each data value. Despite the large index size, we have successfully exploited the great compression tries allow.

The search times for single sistrings on the 1MB files take just over 1/10 second. Calculations again suggest that 100MB texts, with the same distribution parameters, require only about 1.4 times longer to search than the 1MB texts: the search times are effectively independent of the size of the text.

Not only do trie indexes to text support searching for any substring. They also permit a variety of other searches, such as matching a regular expression (e.g., `a.*=`, find all assignments of variables beginning in `a`), proximity search (e.g., finding adjacent occurences of `t` and `h`), and most common substring (e.g., find the most common word in a text). Tries can also find approximations to a search string, which differ by only a certain number of operations such as substitution of one letter for another, transposition of two letters, insertion, or omission of letters [SM96]. Another kind of approximation is the *Soundex*, an encoding of words which captures their phonetics: the trie search, being digital, can encode the text data on the fly, and compare with the Soundex code for the search string [Sha95].

A Appendix

A.1 Searching the Trie

In the above searches, we have been able to mark an `x` at each level of the trie. Since we got to the bottom of the trie, the search was a success.

Now, how do we write a program for this?

Let's look at the search process on a single page. The bit pairs are just found in a sequence with no indicator for the end of a level. Here is the root page.

```
11 10 11 11 11 10 10 10 10 10 10
```

So we maintain a counter, *size*, which tells us how many pairs are on each level, and we can use this to increment a "cursor", *last*, which gives the last node on the present level. Here they are for the root page

size= 1; *last*= 0;

- Each time we see "11", we increment *size*.
- When we come to *last*, we set the next value for *last* by adding *size* to it.

Try it for the root page (the bit pairs are counted from position *pos*=0).

pos	*bitpair*	*size*	*last*
		1	0
0	11	2	2
1	10		
2	11	3	5
3	11	4	
4	11	5	
5	10		10
6	10		
7	10		
8	10		
9	10		
10	10		15

So the last bit pair in each level is at positions 0, 2, 5, 10, respectively. (We don't need the 15 because we are at the end of the page.)

But we are not just reading through the trie: we are searching. In each level, we need a bit pair to compare with the current input bit. Call the position of this bit pair *next*. We need to count the "1" bits in each bit pair up to the bit pair at *next*, in order to find *next* for the next level. We use a counter, *srch*, to do this.

- Initially, for the root page, *next*= 0; *srch*= 0.
- Before *next*, increment *srch* for every "1" bit.
- At *next*, increment *srch* for the "1" bits including the one we marked "`x`", above, but not past it.

- At *last*, reset *next= last + srch*, and reset *srch= 0.*

Meanwhile, maintain *size* and *last* as above. Here it is for the first four bits of our example search key, `10001..`

pos	*bitpair*	*input*	*srch*	*next*	*size*	*last*
		1	0	0	1	0
0	11	0	2,0	2	2	2
1	10		1			
2	11	0	2,0	4	3	5
3	11		2		4	
4	11	0	3,0	8	5	
5	10				10	
6	10		1			
7	10		2			
8	10	1	3	13		
9	10					
10	10					15

So the bit pair to be checked in each level against the input bit is at positions 0, 2, 4, 8, respectively. (If the next level of nodes were also stored on this page, we can see that the next bit pair to be checked would be at position 13, and the end of the level would be at position 15.) Note that input bit `0` matches bit pairs `10` and `11`, and input bit `1` matches bit pairs `01` and `11`.

Now we must consider changing pages. We might have to sequence through all the bit pairs of the trie, but T and B save us from looking at any but the relevant pages on each level. We need to use the index holding T, B and A (the address, not shown) to do two things: 1) find A for the next page, and 2) recalculate *next* and *last* so they work correctly for the page instead of for the whole trie.

Continuing the example, we can see how to do each of these. 1) *srch*, counting the `1` bits, also counts the descendent subtries from the current level. In this case, it has value 3, which means that we need to go to the third subtrie from the beginning of the next level. T for that level tells us which page this is on: we scan the T index until an entry ≥ 3, then pick the page before. (In general, when descending from a page other than a leftmost page of its level, such as the root page, we scan until $\geq B'+3$, where B' is the B value for the page we are just leaving.)

2) The value we need for *next* is $B'+srch-T-1$, where T is the T-value for the descendent page we have now selected, and *srch* gets reset to 0. In the example, we have *next*$= 0 + 3-2-1= 0$; *srch*$= 0$.

For *last*, we reset *size* to $T'' - T$, where T'' is the T-value for the page following the selected page on the same level (or the end-of-level value if the selected page is the last on the level). Then *last*$=$ *size* $- 1$. For the example, *size*$= 4 - 2 = 2$; *last*$= 1$.

The final consideration is, if we have found the key in the trie, how do we find the corresponding record in the data? The data was loaded in the same order as the keys, and so the remainder of each record occupies a consecutive, fixed-length position in the data file. A calculation almost the same as the one that found *next* for the next level of trie pages will give the position of the data record. In the example, our succesful search concluded at the third bit of the last level of nodes on the second page of the last level of pages. Since B for this page is 2, the record we want is the fifth, which is the one at position 4 of the data file, if we count from 0. The position is thus $B + srch - 1$

The overall logic for the search algorithm should be three nested levels of loops, the outer loop over all page levels, the middle loop over all node levels within the page, and the inner loop a combination of four steps: loop until *next*; code for *next*; loop until *last*; code for *last*.

```
for (int pgLev=0; pgLev<h; pgLev++)
  if (pgLev > 0) { find and read next page; reset size, last, next, srch }
  for (int ndLev= 0; ndLev<t; ndLev++)
    while (...isBefore(next) ) { increment size, srch }
    fail if no match, or succeed if last input bit, or get next input bit
       and increment size, srch
    while (...isBefore(last) ) { increment size }
    if (ndLev < t - 1) { reset next, srch, last }
```

A.2 Building the Trie

We build the trie by inserting the data values in ascending lexicographic order. Initially, there is one page for each page level, and the top page level will never have more than one page, the root page. When we enter a new value, we change the node where it differed from the value before it from `10` to `11`. (Why will it always be `10` before the addition?) Below this point, a column of new nodes is added to the right of the existing nodes.

As we enter the data, we keep track of the top count, T, and the bottom count, B, including the present page (T and B up to but not including the present page will not be changed by additions to this page, but are also kept for the index.) We also keep space for N, the number of nodes in the page (this is redundant, but handy), and A, the future address of the page on disk.

When a page is full, we write it to disk and record its address. We usually must anticipate that a page will be made to overflow if we add a column of new nodes: since trie edges must not cross page boundaries within the page level — that is, from pages to their siblings — we may never add only part of a column to a page. Furthermore, the change of a node to `11` on a page, and the subsequent addition of new nodes below it, may overflow the page, in which case the rightmost subtrie on the page must be split off, the rest of the page written to disk, and a new page started including this subtrie.

Here is the start of building the above trie, representing eight levels of nodes by two levels of pages. `showPage` displays T and B, the counts up to the page, and T_c and B_c, the "cumulative" counts *including* the page; and, of course, N, the number of nodes. The page address is not shown — it is not known until we write to disk — but the level of the page (0 or 1) is shown in parentheses after `showPage`.

key `00000011`
`showPage`(0): (T, B, T_c, B_c, N)= (0, 0, 1, 1, 4)
`10`
`10`
`10`
`10`
`showPage`(1): (T, B, T_c, B_c, N)= (0, 0, 1, 1, 4)
`10`
`10`
`01`
`01`

```
key 00101100
showPage(0): (T, B, T_c, B_c, N)= (0, 0, 1, 2, 5)
10
10
11
10 10
showPage(1): (T, B, T_c, B_c, N)= (0, 0, 2, 2, 8)
10 01
10 01
01 10
01 10
```

References

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–17, September 1975.

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–89, 1972.

[Dev82] L. Devroye. A note on the average depth of tries. *Computing*, 28:367–371, 1982.

[dlB59] R. de la Briandais. File searching using variable-length keys. In *Proc. Western Joint Computer Conf.*, pages 295–8, San Francisco, March 1959.

[Fre60] E. H. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–9, Sept. 1960.

[GBYS92] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. *Information Retrieval: Data Structures and Algorithms*, pages 66–82, 1992.

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

[Mor68] D. R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–34, 1968.

[MS93] T. H. Merrett and H. Shang. Trie methods for representing text. In D. B. Lomet, editor, *Foundations of Data Organization and Algorithms*, pages 130–145, Chicago, Illinois, October 13–15 1993.

[MS94] T. H. Merrett and H. Shang. Zoom tries: A file structure to support spatial zooming. In T. C. Waugh and R. G. Healey, editors, *Advances in GIS Research Proceedings, Volume 2*, pages 702–804, Edinburgh, Scotland, September 5–9 1994. Taylor & Francis.

[OM84] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. Third ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, pages 181–90, April 1984.

[Ore83] J.A. Orenstein. Blocking mechanism used by multidimensional tries. Unpublished Letter, February 1983.

[Sha95] H. Shang. Trie methods for text and spatial data on secondary storage. Ph.D. Dissertation, School of Computer Science, McGill University, January 1995.

[SM96] H. Shang and T. H. Merrett. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):540–7, August 1996.