CHAPTER 3

# DESIGN AND QUERY

## 3.1 Physical Design

There is a 3-level storage hierarchy- primary storage, main memory such as RAM and cache; secondary storage, such as hard-drive disks (HDD) and solid-state disks (SSD); and tertiary storage, such as optical drives. As we go down the storage hierarchy, the storage capacity increases, but access speed decreases significantly, and the price decreases.

The fundamental challenge is that a database is too large to fit in the main memory. So, we have to store them in secondary storage (i.e. hard disks). This has the following consequences:

- Since the hard disk is not CPU-accessible, we need to first load the data into main memory from the disk, and then process it in the main memory.
- The speed of data access becomes low- data access from HDD takes 30ms, while it only takes about 30ns in RAM.

So, the main bottleneck in query execution is transferring data from the disk to main memory. We will organise data on disks in such a way that minimises this latency.

Our first challenge is therefore to organise tuples on the disk to minimise I/O access cost. We will first consider how data is represented. A tuple is represented as a sequence of binary digits. Records are grouped together into blocks. A file is composed of many blocks. This is illustrated below.

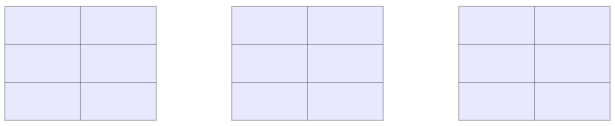


Figure 3.1: A file stored in blocks

Here, the 3 tables form a file; each table represents a block; and a cell in the table is a record.

Records can be of fixed or variable length. This depends on whether the attributes have a fixed size (in bytes), or whether the size of the attributes can vary. We will assume that the attributes have fixed size.

A block is of fixed length. The blocking factor ($bfr$) is the number of records that can be stored in a block. It is given by

$$bfr = \text{floor}(B/R),$$

where a record occupies $R$ bytes and a block can store $B$ bytes. For example, if $bfr = 100$, then we can store up to 100 records in a single block. We must have at least one record per block, so we require $B \geq R$.

Another challenge is how we allocate the blocks of files on the disk. We can use linked allocation. Here, each block $i$ has a pointer to the physical address to the logically next block $i + 1$ anywhere on the disk. So, it is essentially a linked list of blocks. This is illustrated below.

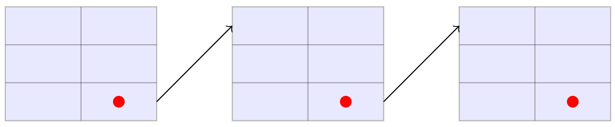


Figure 3.2: A file as a linked list of blocks

Our next challenge is to distribute records within blocks to minimise I/O cost. For this, we will consider 3 representations of data:

- heap (or unordered) file, where we add a new record to the end of the file.
- ordered (or sequential) file, where we keep the records physically sorted with respect to some ordering field.
- hash file, which makes use of a hash function to each record, and uses this as the physical block address.

If we use ordered file, then we need to decide on the ordering field. Similarly, if we use hash file, then we need to choose the hash field. This should be done in a way that minimises the I/O cost.

We define I/O access cost as the cost for:

- retrieving a whole block from the disk to memory to search for a record with respect to some searching field(s) (the search cost); and
- inserting, deleting or updating a record by transferring the whole block from memory to disk (the update cost).

The cost function is the expected number of block accesses (read or write) to search, insert, delete or update a single record. The block is the minimum communication unit. We can only transfer blocks, and not records from disk to memory (and vice versa).

## Heap files

Inserting a new record is efficient for heap files. We just load the last block from disk to memory. The address of the last block is part of the file header. Then, we insert the new record at the end of the block and write it back to the disk. We have 2 block accesses every time, so this is $O(1)$ block accesses.

Searching for a record is inefficient. We have to perform a linear search through all the $b$ file blocks. In particular, we load a block at a time from disk to memory and search for the record. If the searched tuple is not unique, we need to access all the files. Assuming that the searched tuple is unique, then it takes about $b/2$ block accesses to find it. In both cases, this is $O(b)$ block accesses. Since databases store large amounts of data, this is considered inefficient.

Similarly, deleting a file is inefficient. We need to first search the records. Then, we remove it from the block and write the block back to the disk. This leaves unused spaces within blocks, which is quite inefficient. So, this is also $O(b)$ block access. To simulate deletion and avoid this inefficiency, we can use deletion markers. In that case, each record stores an extra bit, where the value 1 means that the record has been deleted. We periodically reorganise the file by gathering the non-deleted records and freeing up blocks with deleted records.

### Sequential files

All the records in a sequential file are physically sorted by an ordering field. We keep the files sorted at all times. Sequential file storage is suitable for queries that require:

- sequential scanning, i.e. data returned in some order (if it is the same as the ordering field);
- order searching, e.g. data similar to given values; and
- range searching, i.e. values between two values (if it is the same as the ordering field).

Retrieving a record using the ordering field is efficient. The block is found using binary search on the ordering field, so we require $O(\log_2 b)$ block accesses. However, retrieving a record using a non-ordering field is not efficient- it is the same as the heap files. So, we require $O(b)$ block accesses. We cannot exploit the sorted nature of the file.

Range queries (with respect to the ordering field) are efficient for sequential files. We just search for the minimum value, and then keep reading the blocks until we find a value bigger than the maximum value. So, the original search is $O(\log_2 b)$ block accesses, and we just read the blocks until the range is exhausted, which is $O(b)$ block accesses. This cannot be minimised since we want to read precisely these blocks; the efficiency comes from the original binary search.

Insertion is not efficient for sequential files. We first locate the block where the record should be inserted- this requires $O(\log_2 b)$. On average, we will need to move half of the records to make room for the new record. This is very expensive for large files. We can alternatively use chain pointers. Here, each record points to the logically next ordered record. If there is free space in the right block, we insert the new there. Otherwise, we insert the new record in an overflow block and use chain pointers. Pointers must be updated; it is a sorted linked-list. Having pointers increases overflow, and we will have to periodically reorganise the blocks.

Deleting a tuple is expensive as well. First, we have to locate whether the record is to be deleted. We can delete the entry, which would add a gap. Also, we can make use of deletion markers. In that case, we update the deletion marker from 0 to 1, and update the pointer not to point to the deleted record. In both cases, we have to periodically reorganise the blocks.

Updating on the ordering field is expensive. The record is deleted from the old position and gets inserted into its new position- it involves both insertion and deletion, both of which are expensive. However, updating on a non-ordering field is efficient. We just find the tuple and update one attribute, and write the block back to the disk. So, this involves $O(\log_2 b) + O(1)$ block accesses.

### Hash files

In hashing, we partition the records into $M$ buckets. Each bucket can have one or more blocks. We then choose a hashing function $y = h(k)$, with output in $\{0, 1, \ldots, M-1\}$, where $k$ is the value of the attribute in the hash field. For this to work efficiently, we require $h$ to uniformly distribute records into the buckets. That is, for each value $k$, a bucket is chosen with equal probability $1/M$. An example of such a hashing function is the modulo function, i.e. $y = h(k) = k \bmod M$.

Mapping a record to a bucket $y = h(k)$ is called external hashing over hash-field $k$. Normally, there is a chance that collisions occur. That is, two or more records are mapped to the same bucket. For example, let $M = 3$ and $h(k) = k \bmod 3$. So, we have 3 buckets. If the attribute values are $k = 1, 11, 2, 4$, then we map them to buckets 0, 2, 2 and 1 respectively. So, there is a collision on bucket 2. This is indirect clustering- we are grouping tuples together with respect to their hashed values $y$, and not with respect to their hash field values $k$.

If we want to retrieve a possible record with respect to the hash field, we do the following:

- we hash the hash attribute $k$ and get the corresponding bucket;
- we use the hash map to get the block address in disk of the relevant bucket;
- we fetch the block from the disk to memory; and
- we search the block in memory linearly to find the record.

So, we require $O(1)$ block access.

Due to collisions, a hash bucket might be full. Using the chain pointers method, we can insert a new record hashed to a full bucket. If there are $O(n)$ overflown blocks per bucket, then we require $O(1) + O(n)$ block accesses. To improve efficiency, we should use a hash function that minimises the number of overflow blocks.

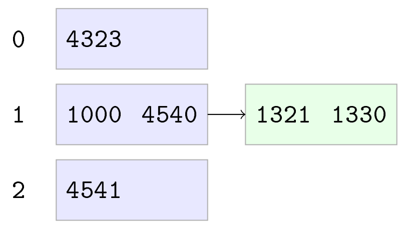
Now, consider deleting a record based on the hash field. We find the record in the main bucket, or an overflown block. So, this takes $O(1) + O(n)$ block accesses. We would need to periodically pack the blocks together to free up the space from deleted records.

When we update a record based on a non-hash field, we locate the record in main or overflow bucket. We load the block into memory, update it and write it back. So, this also takes $O(1) + O(n)$ block accesses. When we update a record on the hash field, we need to delete the record from the old bucket and add it to the new bucket. This also takes $O(1) + O(n)$ block accesses.

Now, assume that $M = 3$, we have 1 block per main bucket, and $bfr = 2$ records per block. The hash attribute is the `SSN` key values:

$$\{1000, 4540, 4541, 4323, 1321, 1330\}.$$

If the hash function is $h(k) = k \bmod 3$, then we can assign the employees into buckets as follows:



The blue blocks are part of the main bucket, while the green ones are the overflow blocks. We require 4 buckets to hash these values. If we were using heap or sequential representation, we would only need 3 blocks. Since the hashing function does not uniformly distribute the blocks, we instead require 4 here.

However, range queries are inefficient in hash representation, even when the range is over the hashed field. For each value in the range, we have to find and retrieve the bucket and select the value from the record. The hash function uniformly distributes the values, so we would not want it to be possible to retrieve all values within the same bucket. So, it takes $O(m) + O(mn)$ bucket accesses where $m$ is the range of the distinct values in the range and $O(n)$ overflow blocks per bucket. This is not efficient. Sequential files are much better in this case.

The distribution of values influences the expected cost. This implies that the expected cost is unpredictable to compute in practice. For instance, assume that we have an age distribution as shown in the table below.

| Age | Count |
|---|---|
| 20 | 2 000 |
| 30 | 900 |
| 40 | 800 |
| 50 | 80 |
| 60 | 23 |
| 70 | 80 |

Moreover, let *bfr* = 40 employees per block. For people aged 20, we need to retrieve

$$\text{ceil}(2000/40) = 50$$

blocks. Instead, if we want to get people aged 60, we need to only retrieve

$$\text{ceil}(23/40) = 1$$

block. So, the number of block accesses depends on the value.

## 3.2 Indexing Methodology

We have looked at physical design of the database. The objective there was that, given a specific file type, we provide a primary access path based on a specific searching field. Now, we will look at index design. In this case, our objective is, given any file type, we provide a secondary access path using more than one searching field.

Adding a secondary access path means that we need to store additional (metadata) files on the disk. Moreover, we need to maintain them so that they are consistent with the actual data. Although the overhead has increased, we significantly improve the searching process by avoiding linear scanning.

When we create an index, we need to follow some principles.

- We create one index over one field. This is called the index field. We will only consider indices built on one attribute.
- An index is another separate file. This is a metadata file, and is the reason the overhead increases.
- All index entries are unique and sorted with respect to the index field. We expect a tuple in the relation to be of the form (`index-value`, `block-pointer`). The block pointer is the location of the block that contains the entry with the given value.
- First, we search within the index file to find the block pointer, and then we access the data block from the data file.

For indexing to be more efficient than linear searching, we require the index files to occupy less blocks than the data file. This is true because the index entries are quite small, with only two attributes. So, we can fit in more index entries in a block than data records, i.e. most of the time,

$$bfr(index\text{-}file) > bfr(data\text{-}file).$$

There are 2 indexing strategies. A dense index has an index entry for every record in the file, as illustrated below.

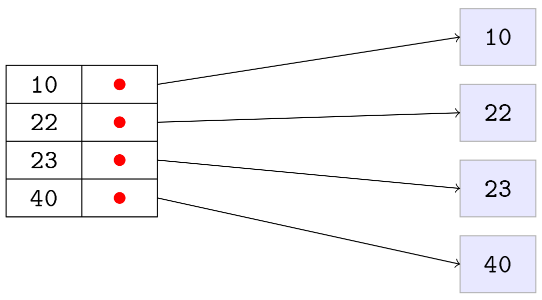


Figure 3.3: A dense index

So, the index file has the same number of entries as the data file. A sparse index has an index entry for only some of the records, as illustrated below.

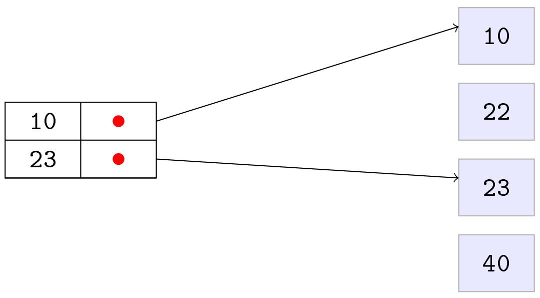


Figure 3.4: A sparse index

This can be used when the indexing field is how the data files are sorted. So, if the value is 22, we look at the block containing 10- the next block starts with 23, so if 22 is present in the data file, it must lie within this block.

Another expectation we have is than searching over index is faster than over the file. Since indexing file is an ordered file, we can adopt a binary/tree-based approach to find the pointer to the actual data block. This makes the procedure much faster than linear search.

There are 3 types of index types- primary, clustering and secondary index. Primary index is where the index field is an ordering, key field of a sequential file (e.g. `SSN`, where the file is sorted by `SSN`). Clustering index is where the index field is an ordering, non-key field of a sequential file (e.g. `DNO`, where the file is sorted by `DNO`). Secondary index is where the index field is non-ordering, i.e. the files are not sorted with respect to this attribute. The index field may be key or non-key.

## Primary Index

A primary index is built over a sequential file, where the indexing field is the field the files are sorted by. Since the data file is sorted, we can use sparse indexing to index the data files, as illustrated below.

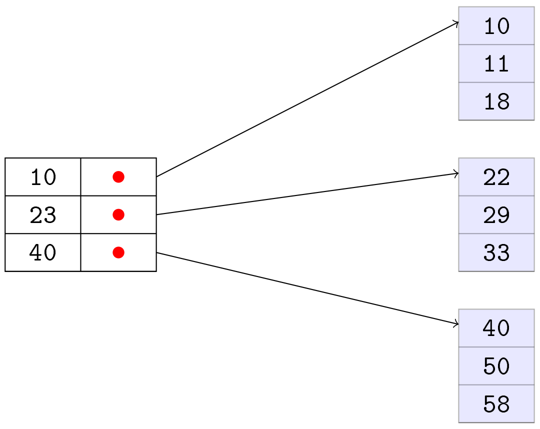


Figure 3.5: Primary Indexing

So, we have one entry in the index file per data block. The $i$-th index entry $(k_i, p_i)$ refers to the $i$-th data block. The value $k_i$ is the field value of the first record in block $i$. The first data-record in block $i$ with value $k_i$ is called the anchor of block $i$.

To find a particular tuple based on the index entry, we first search in the index file the block where we expect this entry to be. The index entries are sorted, so we use a binary search. This gives us the anchor entry of the block. We then load this block and search within the block for the tuple.

If the anchor record is deleted/updated, we need to update the sequential data file- this is a very costly computation. Moreover, after the update, we need to propagate the update to the index file. The index file is also a sequential file, so this too is a costly computation.

If a non-anchor record is updated, we need to update the sequential data file. This need not be propagated to the index file if the order of the blocks isn't affected. If the order of the blocks is affected, or the non-anchor record is deleted, we need to update the data file and the index file like we did previously.

## Clustering index

A clustering index is used to index a sequential file on an ordering, non-key field. The indexing file is a set of cluster of blocks, with a cluster per distinct value. The block pointer points at the first block of the cluster. The other blocks of the same cluster are contiguous and accessed via chain pointers. The clustering index is sparse because we are only using some of the index entries to index the entire data file.

An illustration of clustering index is given below.

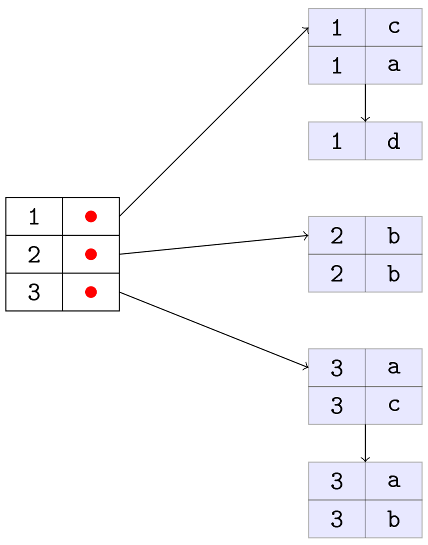


Figure 3.6: Clustering Index

The number of index entries is the same as the number of the clusters. Each pointer in the index entry points to the first entry of the cluster. Each block is pointing to the next contiguous blocks using a linked list data structure.

## Secondary Index

Now, we want to index a file on a non-ordering field. The file might be undordered, hashed, or ordered, but not ordered with respect to the indexing field. There are 2 cases here, where the secondary index is unique or not.

If we want a secondary index on a non-ordering, key field, we need one index entry per data record, i.e. a dense index. This is because the entries are scattered around the data file with respect to this index, and so we cannot use anchors.

An illustration of secondary non-ordering key index is given below:

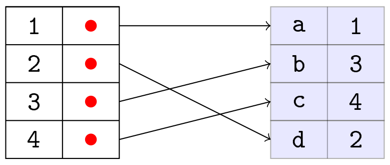


Figure 3.7: Secondary Index on a non-ordering, key index. Note that we are linking to tuples in the figure for simplicity, and not to blocks.

For each index entry, we store the block pointer where the tuple can be found.

The second type of secondary index is on a non-ordering, non-key field. So, some records are having the same indexing value. Moreover, the data files are not sorted with respect to the index attribute. So, we need to store all the block pointers for a given index in yet another file.

This is illustrated in the figure below.

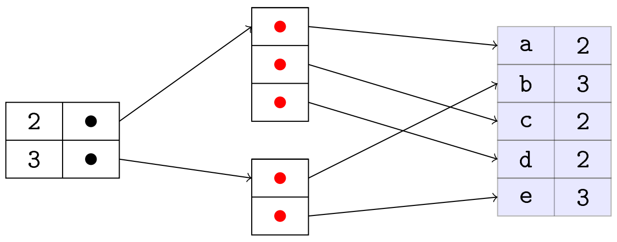


Figure 3.8: Secondary index on a non-ordering, non-key attribute. Note that we are linking to tuples in the figure for simplicity, and not to blocks.

In the figure above, we have 2 possible values for the index- 2 and 3. Within the indexing file, we point to a secondary indexing file. In the secondary index file, we have block pointers for all the data records that have the matching index value (a cluster/group). Indexing at the first level is sparse since there are many tuples in the data field with the index value since this value is not unique.

So, there are 2 levels of indirection here. At the first level, we have a block of block pointers of a cluster. At the second level, a block pointer points to the data block that has records within this distinct index value.

## Multi-level index

All the index files (primary, clustering and secondary) can be used to build multi-level index files. They are all ordered with respect to the indexing field. Moreover, the indexing field have unique values, and each index entry is of fixed length. This means we can create an primary index on each of these index files. We could even build an index over that index, and so on. This is multi-level indexing.

The original index file is called the base or Level-1 index file. Indexing this index file gives us a Level-2 index file. We can continue this on to get a Level-$t$ index file for any positive integer $t$. Now, we aim to find the best level-$t$ of a multi-level index to expedite the search process trading off speed-up with overhead.

The following is an example of a level-2 index.

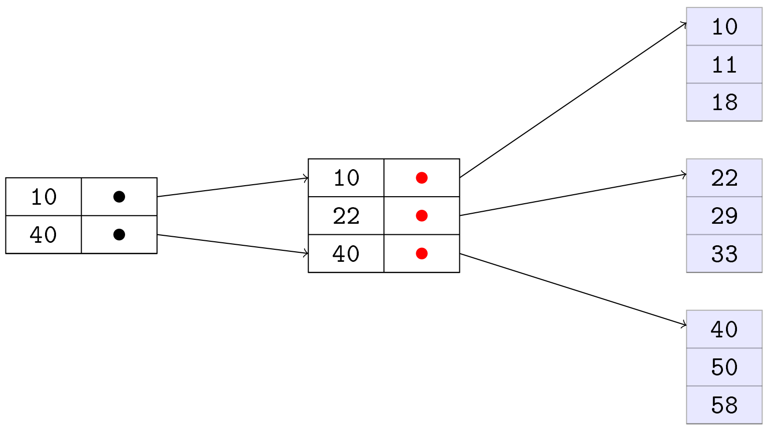


Figure 3.9: A 2-level index

The first index is a primary index, so it is using anchors. Also, the second level is a primary index, so it points to the anchors within the level-1 index file. The level-2 index has 1 block (containing 2 records), so we should not create level-3 index on it.

In general, we should stop not create level-$t + 1$ index if level-$t$ where the top-level index has 1 block. So, given a level-1 index with blocking factor $m$ entries/block, the multi-level index is of maximum level

$$t = \log_m b.$$

The value $m$ is known as fan-out.
If we want to find a tuple from a level-$t$ index, we first load the $t$-level index block. Using this block, we can load a single $t - 1$-level block, and so on. We continue this until we load a L1 block. So, we need precisely $t$ block accesses to load the L1 data block. Finally, we load the actual data file block using the L1 index. If it takes $n$ block accesses to retrieve the data block from the L1 block, it will take us

$$t + n = \log_m b + n$$

block accesses to retrieve the relevant tuple(s).

## 3.3 B and B+ Trees

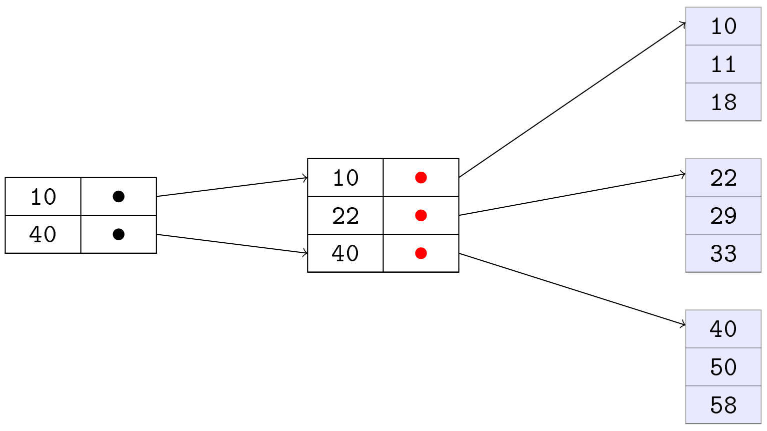
For multi-level indices, we search for a record over a $t$-level index with $t+1$ block accesses. However, insertions, deletions and updates over the data file are reflected over the multi-level indices. This makes the operations costly. This is because all encapsulated indexes are physically ordered files. So, all updates should be reflected to all levels.

We will try to define a dynamic multi-level indices. This means that the data structure should:

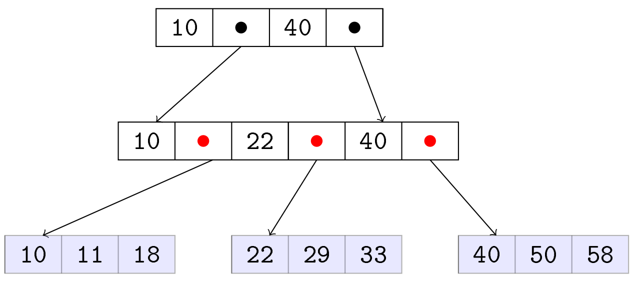
- adjust to deletions and insertions of records;
- expand and shrink following the distribution of the index values; and
- be self-balancing (subtrees should be of the same depth).

We will do this using B and B+ Trees.

We can envisage a multi-level index as a tree. This can be done by turning it clockwise by 90 degrees. For example, consider the following diagram.



We can rotate it clockwise by 90 degrees to get the following tree.



Here, the root is the level-2 index. The children of the root are level-1 index blocks. The leaves are the actual data blocks.

There is a limitation to this. The tree needs to be balanced. This is because it does not adjust to keys' distribution. That is, the leaf nodes are at different levels. The number of block accesses in a multi-level index depends on the depth of the tree, so we would like the tree to be as balanced as possible. In the worst case, we would end up with a linked list of nodes.

Our first challenge is to ensure that the tree is balanced by minimising the tree depth $t$. This challenge has two parts- how do we deal with insertion into a full node, and deletion of a node. We can split a node into two if the node is full, and merge two nodes if a node is to be deleted. This increases the overhead, but is required to keep the tree balanced.

## B Tree

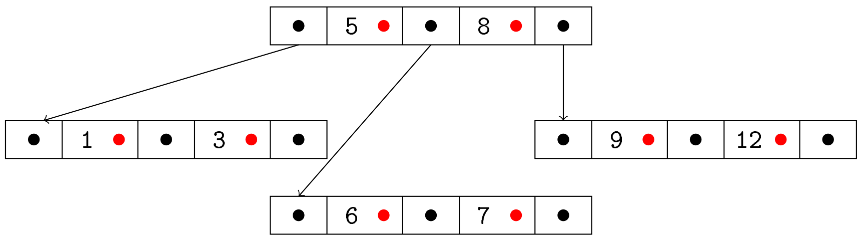
B Trees are used for secondary indices with a key, non-ordering attribute. A B tree node of order $p$ splits the searching space up to $p$ subspaces. We assume $p > 2$. A node is of the form

$$\{P_1, (K_1, Q_1), P_2, (K_2, Q_2), \dots, P_{p-1}, (K_{p-1}, Q_{p-1}), P_p\},$$

where

- $K_i$ is a key value,
- $P_i$ is a tree pointer for those tuples with $X < K_i$,
- $P_{i+1}$ is a tree pointer for those tuples with $X > K_i$,
- $Q_i$ is a block pointer for $K_i$.

A pictorial representation of this is given below.



The advantage of B trees is that every time we descend a level, we split the searching space into $p$ subtrees, which exponentially lowers the searching space. If $p = 2$, this is equivalent to binary search. We typically have $p > 2$. Like in the case above, a tree of order $p$ can store up to $p$ tree pointers (shown in black) and up to $p-1$ key values and their block pointers (shown in red). In the case above, we have $p = 3$- 3 tree pointers, 2 key values and data pointers in each node.

When searching for a tuple of a value using a B tree, we start from the root and descend in the right branches until we get to a key value corresponding to the tuple. Then, we access the corresponding data block and return the matching tuple.

Normally, a data block contains all the elements of the tree. So, each tree node corresponds to a block. We now consider the number of block accesses we need to find the right tuple with the B tree given above:

- If we search for 8, we first access the root block. Since this node contains the value 8, we can access the data block containing 8 with another block access. In total, we need 2 block accesses.
- If we search for 7, we first access the root block. This node does not contain 7, so we descend the middle branch. This block does contain the value 7, so we can access the data block containing 7 with another block access. In total, we need 3 block accesses.
- If we search for 31, we first access the root block. This node does not contain 7, so we descend the root branch. This block also doesn't contain 31, but we know there is no right tree pointer here. This means that there is no record corresponding to the value 31. So, we just need 2 block accesses.

### B+ Trees

A B Tree gives rise to a big metadata. We need to find another data structure so that it has the advantages of B tree while minimising storage. A B tree node stores a lot of data: tree pointers, data pointers and key values. We need tree pointers and key values to navigate the B tree.

We want to be storage efficient by freeing up space from the nodes. Also, we want to be search efficient by maximising the fan out (the splitting factor) of a node. We can increase the order of the node to increase fan out. To do this, we get rid of data pointers from the nodes. We still need to store data pointers, but we can remove them from non-leaf nodes.

In a B+ Tree, we have 2 types of nodes. An internal node guides the search process, and only have tree pointers and key values. A leaf node points to actual data blocks. To maximise fan out, internal nodes have no data pointers. However, we need to go to the leaf to access any data pointers. Nonetheless, since we have removed data pointers from the nodes, this process is quite fast.

Moreover, the leaf nodes hold all the key values sorted and their corresponding data pointers. Also, some key values are replicated in the internal nodes to guide and expedite the search process. This corresponds to medians of key values in sub-trees.

A B+ internal node of order $p$ is of the form

$$\{P_1, K_1, P_2, K_2, \dots, P_{p-1}, K_{p-1}, P_p\},$$

where

- $K_i$ is a key value,
- $P_i$ is a tree pointer for those tuples with $X < K_i$,
- $P_{i+1}$ is a tree pointer for those tuples with $X > K_i$.

A B+ tree of order $p$ can store up to $p$ tree pointers and up to $p-1$ key values.

A B+ internal node of order $p_L$ is of the form

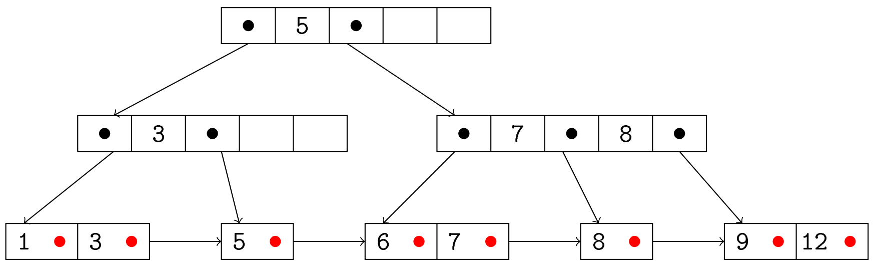
$$\{(K_1, Q_1), (K_2, Q_2), \dots, (K_{p_L}, Q_{p_L}), P_{\text{next}}\},$$

where

- $K_i$ is a key value,
- $Q_i$ is a data pointer for $K_i$,
- $P_{\text{next}}$ points to the next block of leaf nodes.

We store a linked list of leaf nodes. All the leaf nodes are at the same level, meaning that the tree is balanced.

An example of a B+ tree is given below.



Here, the internal nodes have order $p = 3$ and leaf nodes have order $p_L = 2$.

The leaf nodes are linked and sorted by key. All keys of the file appear at the leaf nodes. The leaf nodes contain data pointers only, to expedite navigation. Leaf nodes are balanced for constant I/O cost. Some selected keys are replicated in the internal nodes for efficiency.

To find the block containing a given value, we must always descend to the leaf nodes. So, it will always take 3 block accesses to find the relevant tuple, assuming it is present.

For a B tree, we saw that we can store 65 535 key values and data pointers in the tree in a very similar scenario. Using a B+ tree, we have almost multiplied the capacity by 4. However, we require about 3 times more blocks for a B tree. Nonetheless, adding a new tuple to the B tree can be costly (e.g. if we add a 65 536th tuple). This is not the case for B+ trees- we might not even have to add the new entry to an internal block. Moreover, adding to the leaf nodes is simple since it is a linked list, even if we need to keep it sorted.