When data is stored on disk based storage devices, it is stored as blocks of data. These blocks are accessed in their entirety, making them the atomic disk access operation. Disk blocks are structured in much the same way as linked lists; both contain a section for data, a pointer to the location of the next node (or block), and both need not be stored contiguously.

Due to the fact that a number of records can only be sorted on one field, we can state that searching on a field that isn’t sorted requires a Linear Search which requires N/2 block accesses (on average), where N is the number of blocks that the table spans. If that field is a non-key field (i.e. doesn’t contain unique entries) then the entire table space must be searched at N block accesses.

Whereas with a sorted field, a Binary Search may be used, this has log2 N block accesses. Also since the data is sorted given a non-key field, the rest of the table doesn’t need to be searched for duplicate values, once a higher value is found. Thus the performance increase is substantial.

What is indexing?

Indexing is a way of sorting a number of records on multiple fields. Creating an index on a field in a table creates another data structure which holds the field value, and pointer to the record it relates to. This index structure is then sorted, allowing Binary Searches to be performed on it.

The downside to indexing is that these indexes require additional space on the disk, since the indexes are stored together in a table using the MyISAM engine, this file can quickly reach the size limits of the underlying file system if many fields within the same table are indexed.

How does it work?

Firstly, let’s outline a sample database table schema;

Field name Data type Size on disk

id (Primary key) Unsigned INT 4 bytes

firstName Char(50) 50 bytes

lastName Char(50) 50 bytes

emailAddress Char(100) 100 bytes

Note: char was used in place of varchar to allow for an accurate size on disk value. This sample database contains five million rows, and is unindexed. The performance of several queries will now be analyzed. These are a query using the id (a sorted key field) and one using the firstName (a non-key unsorted field).

Example 1

Given our sample database of r = 5,000,000 records of a fixed size giving a record length of R = 204 bytes and they are stored in a table using the MyISAM engine which is using the default block size B = 1,024 bytes. The blocking factor of the table would be bfr = (B/R) = 1024/204 = 5 records per disk block. The total number of blocks required to hold the table is N = (r/bfr) = 5000000/5 = 1,000,000 blocks.

A linear search on the id field would require an average of N/2 = 500,000 block accesses to find a value given that the id field is a key field. But since the id field is also sorted a binary search can be conducted requiring an average of log2 1000000 = 19.93 = 20 block accesses. Instantly we can see this is a drastic improvement.

Now the firstName field is neither sorted nor a key field, so a binary search is impossible, nor are the values unique, and thus the table will require searching to the end for an exact N = 1,000,000 block accesses. It is this situation that indexing aims to correct.

Given that an index record contains only the indexed field and a pointer to the original record, it stands to reason that it will be smaller than the multi-field record that it points to. So the index itself requires fewer disk blocks than the original table, which therefore requires fewer block accesses to iterate through. The schema for an index on the firstName field is outlined below;

Field name Data type Size on disk

firstName Char(50) 50 bytes

(record pointer) Special 4 bytes

Note: Pointers in MySQL are 2, 3, 4 or 5 bytes in length depending on the size of the table.

Example 2

Given our sample database of r = 5,000,000 records with an index record length of R = 54 bytes and using the default block size B = 1,024 bytes. The blocking factor of the index would be bfr = (B/R) = 1024/54 = 18 records per disk block. The total number of blocks required to hold the table is N = (r/bfr) = 5000000/18 = 277,778 blocks.

Now a search using the firstName field can utilise the index to increase performance. This allows for a binary search of the index with an average of log2 277778 = 18.08 = 19 block accesses. To find the address of the actual record, which requires a further block access to read, bringing the total to 19 + 1 = 20 block accesses, a far cry from the 277,778 block accesses required by the non-indexed table.

When should it be used?

Given that creating an index requires additional disk space (277,778 blocks extra from the above example), and that too many indexes can cause issues arising from the file systems size limits, careful thought must be used to select the correct fields to index.

Since indexes are only used to speed up the searching for a matching field within the records, it stands to reason that indexing fields used only for output would be simply a waste of disk space and processing time when doing an insert or delete operation, and thus should be avoided. Also given the nature of a binary search, the cardinality or uniqueness of the data is important. Indexing on a field with a cardinality of 2 would split the data in half, whereas a cardinality of 1,000 would return approximately 1,000 records. With such a low cardinality the effectiveness is reduced to a linear sort, and the query optimizer will avoid using the index if the cardinality is less than 30% of the record number, effectively making the index a waste of space.

if you write into a table (UPDATE or INSERT) with one index, you have actually two writing operations in the file system. One for the table data and another one for the index data (and the resorting of it (and - if clustered - the resorting of the table data)). If table and index are located on the same hard disk this costs more time. Thus a table without an index (a heap) , would allow for quicker write operations. (if you had two indexes you would end up with three write operations, and so on)

However, defining two different locations on two different hard disks for index data and table data can decrease/eliminate the problem of increased cost of time. This requires definition of additional file groups with according files on the desired hard disks and definition of table/index location as desired.

Another problem with indexes is their fragmentation over time as data is inserted. REORGANIZE helps, you must write routines to have it done.

In certain scenarios a heap is more helpful than a table with indexes,

e.g:- If you have lots of rivalling writes but only one nightly read outside business hours for reporting.

Also, a differentiation between clustered and non-clustered indexes is rather important.