Ian Kenny

December 5, 2016

Databases

Lecture 10

In this lecture

- Physical design and performance.

Introduction

Clearly, a database (unless very small) cannot be stored in RAM. It will be too large. RAM is too limited and too expensive to store most databases. The data must also persist, which also makes RAM unsuitable.

Thus, database tables must be stored in *secondary storage* such as hard disks and then small parts of the database can be loaded into RAM. Applications can only directly access RAM, not secondary storage.

RAM is fast and disk storage is (relatively) slow. This means that there are performance issues to consider in the design of a DBMS and in the design of a particular database.

The first aspect of database performance to consider is, therefore, the physical storage and accessing of the database on disk.

Database files

Relational databases systems store tables in files. They are, therefore, reliant on the underlying operating system for file creation, deletion and access, etc.

Operating systems store data on HDDs, etc., as sequences of bytes. They use abstractions such as the *file* and the *directory*, however, to provide a more convenient metaphor for the data stored.

HDD and other controllers are built to access data in *blocks* rather than as individual bytes. A block size might be 4KB. A block is the minimum unit of data for reading from the disk and writing to the disk. Multiple contiguous blocks can be accessed in one access operation. This is more efficient. A file will be stored in one or more blocks.

Physical data proximity can also mean logical data proximity.

Database files

The relations in a database are stored in one or more files. The organisation of the data within the database files is the responsibility of the DBMS.

The data in a database file is also organised into *blocks* but these blocks are not necessarily the same size as the blocks used by the disk controller. It makes sense to set them to be the same size though. Sometimes (and this helps avoid confusion) the blocks inside the database files are called *pages*. So a typical page size is also 4KB.

The DBMS is responsible for allocating data to the pages of a file. The OS is responsible for the allocation of the file's blocks on the disk. If the DBMS sets the page size to be the same as the block size, then clearly this optimises the disk accesses to some extent. If the page size was 5KB then every page read would require two block reads.

Database data

Each relation is generally stored in a file. Relations can be *clustered* in files, meaning that a file (or files) can contain multiple related relations. This can be convenient for small amounts of data, and where the relationships between the relations are clear and unchanging. The advantage of clustering is that it can make joins more efficient, for example, because the frequently-joined tables can be clustered together in the same file. But this is only for joins on specific, well-known conditions.

Within each file the data is organised into pages. A page will contain a subset of the data in a relation.

Within each page, the data is stored as *records*. A record generally contains a single tuple. Therefore, a record in a page is the physical representation of a tuple.

Within a record each attribute value is a *data item*.

Database file organisation

The data in a file can be physically organised in various ways, ranging from unorganised to highly organised. If the file itself is unorganised, a separate structure called an *index* can be used to provide the desired structure (and to bring other benefits, as will be seen).

Heap files

In a heap file, data is stored in no particular order. A particular record could be stored anywhere in the file. Records will generally be added in the order in which they are created, however, but this is not guaranteed. This means each new record will be added at the end of the file.

With such a file organisation method, any required sorting or searching functionality must be implemented elsewhere. The file itself does not assist.

All SQL queries types could be implemented with this type of file organisation but, with no support from the file organisation itself, the implementation of the queries would be expensive. This is not how it works in practice, however. Such file organisations are used in association with other ordering and searching techniques, as will be seen.

Sequential files

With sequential file organisation, the records are stored in some order according to a *key* value. This might be the primary key or some other candidate key of the records, or it might be some other often-used attribute (e.g. employee age rather than employee ID). They key does not need to be unique. The choice of key depends upon the application and the most regularly required ordering.

The records will generally be sorted on one attribute only but they could be sorted on composite keys, i.e. keys with more than one attribute. This is generally less useful, however.

If a composite key contains two attributes, the first one will be the primary ordering attribute and the other one the secondary ordering attribute. There would be no benefit for searching all of the records on the basis of this secondary ordering because you would still possibly have to access all records.

Sequential file organisation

There is clearly an overhead to sequential files since the order must be maintained as insertions and deletions occur. It is advantageous if the records are stored physically in order since then large numbers of them can be accessed in one operation. However, when insertions and deletions occur, potentially either the entire set of records has be moved to maintain the order, or ‘overflow’ pages need to be allocated for the new records (in the case of insertion). If this approach is taken, then the performance could degrade as more and more records are placed into overflow areas and are, hence, not in order.

If, on the other, the records are linked together as a linked list, hence can be physically in any order, insertion and deletion are more efficient since that simply involves adjusting pointers. However, if the records are not now physically in order, this removes the potential efficiency of being able to access multiple related records in one access operation.

Indexes

If a file is unordered, or is ordered according to only one key (which a sequential file will be, even if the key is composite), then that will be inconvenient for running general queries on the data. More on this later.

We usually need some way of accessing quickly particular records or ranges of records, for example to satisfy a WHERE condition in an SQL SELECT statement. There could be millions of records in a set of database files. Simply scanning the data files could be a very slow process.

DBMSs use structures called *indexes* to speed up the searching of database files. An *index* is an ancillary data structure that operates a bit like the index in the back of book. A book index contains some *search key* and a page number. The index then allows us to find a complete section of the book but with limited information (key, page number).

Indexes

All other details about the book section found at the given page number are not present in the index. The index offers a truncated form of information about the sections in the book.

In a DBMS, an index is a structure that enables the location of particular records, or multiple records in a range, based on a key. The physical location of a record can be called the *record ID*. At its simplest, an index contains a list of key:record ID pairs. To find a record we simply look up the key and then read the record location (the record ID). Because an index is ordered, we can take advantage of search methods that work well with ordered data. For example, binary search.

Why would we do this? Firstly, it might be possible to load an index fully into RAM, or at least significant parts of it. This will clearly give a huge speed increase to the process of finding a record ID.

We can do this because an index is essentially a 'mini version' of the actual data file. If a data file contains 20 data items per record (i.e. tuple), the index file might need only one data item and one record ID per record, thus about 10% of the data.

Indexes

The underlying file that contains the actual records is not necessarily ordered. An index, on the other hand, will be ordered somehow or provide some other convenient method for accessing the records (we will come to this shortly).

In order to find a particular record in an *unordered heap file* from its key, we need to (in the worst case) search all of the records. With an index, ordered on a key we can search the index using a binary search, for example, to locate the record. This will be much faster.

If a range of records is sought, then the index can be searched for the start record and then searched linearly for the rest of the range.

In an unordered file with no index, this would certainly require every record to be read, not simply in the worst case.

Files/Indexes

The index for a relation could be stored in the same file as the records. This would mean that each index entry could point to the actual data record within the same file. This will clearly be the fastest method.

The index could also be stored in a separate file. This means that the DBMS would have to load the index and then use that to direct further disk accesses. This will be a bit slower but may be necessary if the files involved are large.

Indexed Sequential files

With Indexed Sequential files, the records are stored *sequentially* as with the sequential method, according to some key (perhaps the primary key).

Additionally, at least one index is also stored that uses some other key on the records, and stores the location of the records in the sequential file, sorted by this other key. Different indexes can exist for different orderings. There can be multiple indexes for a file.

This type of storage allows rapid access to particular records (perhaps using binary search) and also sequential access for accessing all records, or all records in a range using different keys. For example, if we had an index for employee ages, we could search by age (and do SQL queries based on that). If we had another index, ordered on salary, we could do searches on ranges of salary values, etc.

Hash files

Files can be organised using a hash function. Records are placed into the file in a position computed by a hash function. The hash function takes a key value (a single attribute or combination of attributes) and computes a location based on the result. This means it is easy to find out where in a file a particular record is - simply compute its hash value and that gives the record location. Records are, therefore, not stored in order.

Hash indexes

Hash functions can also be used to create index files. If a hash function was used to organise the original file of records, a separate hash function can be used to organise the index.

Hash function methods are good for storing and locating individual records quickly. The hash value is simply computed which gives the actual location in the file, or the location in an index file which points to the actual record.

These methods are less helpful if a number of related records are needed since they will not necessarily be stored contiguously. The hash function could produce physically distant locations for two logically related records. They are also not helpful for queries that involve a *range* of values since, again, there is not necessarily a physical proximity to logically related records (i.e. records with similar values for an attribute).

Indexed Sequential Access Method (ISAM)

The ISAM method for creating an index uses a tree structure. The index is created when the data is first added so the tree is generated from the *original* set of data. The data records are stored in the leaves of the tree. The internal nodes of the tree contain a set of keys (each node might hold two keys, for example). The nodes also contain pointers. Each pointer points either to the next level of the tree or to a set of records in a leaf node. The pointer to the left of each key in a node points to a sub-tree which has key values *less than or equal to* that key value. The pointer to the right points to a sub-tree that has key values *greater* than that key value.

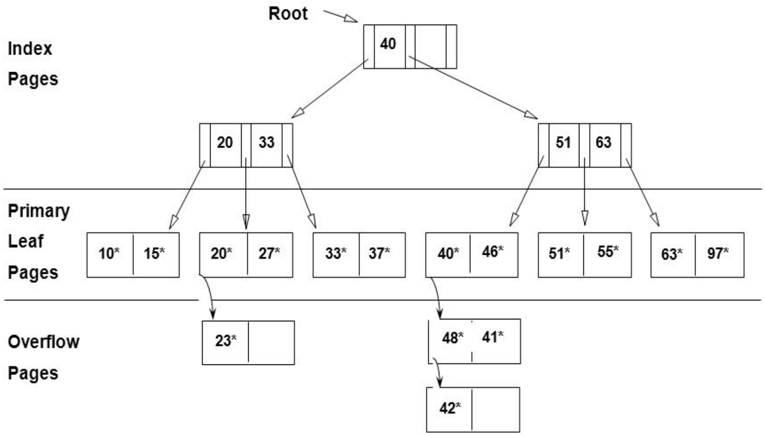
Indexed Sequential Access Method (ISAM)

Since an ISAM index is built when the data is first added, it is a *static* structure. The internal structure of the tree never changes. If a new data record is inserted, the leaf node where it *should* be inserted is traversed to and then the new record is added to an overflow page that is connected to that leaf note.

This means that this structure is best used when not many insertions will be made. If many insertions are made, the benefits of the quick traversal are lost and the search degrades to linear search in the overflow pages.

Consider the image on the next slide.

Indexed Sequential Access Method (ISAM)



(Image: Mike Franklin)

Indexed Sequential Access Method (ISAM)

The *primary leaf pages* are the leaf pages created when the tree is first constructed. The *index pages* are also created at that point. The overflow pages are created as new insertions occur that cannot be accommodated. It could be the case that a leaf page is not full. If that is the case, a new record could be added to a primary page. Generally, however, they will be full and hence overflow pages will be needed. And, when an overflow page is full, the overflow page needs an overflow page, etc.

B+ Tree indexes

The problem with ISAM is that it creates a static index. This can be avoided by using a B+ Tree structure instead. This is the structure used in most DBMSs.

The structure of a B+ Tree, when used in a DBMS, is almost the same as the ISAM structure, but the algorithms allow for arbitrary insertion and deletion in the tree with no degradation of the tree's structure.

B+ Trees

A B+ tree is a slight enhancement of a B-Tree. A B-Tree is a generalised binary search tree in which nodes can have any number of children.

A B-Tree is a tree structure in which each internal (non leaf) node can have more than two children (unlike the binary tree). The number of children that each node can have is specified within a range. Nodes do not have to have the same number of children.

All nodes in a B-Tree have the same depth. If an insertion would result in a node having a different depth, the tree is reorganised so that all the nodes have the same depth.

(The terms B+ Tree and B-Tree (B 'hyphen' Tree) are used synonymously on these slides).

B+ Trees

Each internal node has a list of keys. The keys act as dividers for the node's sub-trees, as with the ISAM method.

For any given key, there are two pointers to sub-trees: one pointing to the left sub-tree and one pointing to the right sub-tree. The sub-tree to the left of the key contains key values that are less than or equal to the key value. The sub-tree to the right contains key values that are greater than the key value.

A parameter d is selected that defines the *minimum* number of keys that each internal node must have. Conventionally, the actual number of keys per internal node varies between d and $2d$.

The minimum *branching factor* of the tree is $d + 1$. This is also sometimes called the *fan-out* of the tree.

Thus, if there are d keys in a node there will be $d + 1$ sub-trees at that node.

B+ Trees

For an internal node x, $n[x]$ is the number of keys at the node.

The key values at a node are ordered. For node x with key values $k_1, k_2, ...k_n$:

$k_1[x] <= k_2[x] <= ... <= k_n[x]$.

Each node has a one more sub-tree than keys. Node x with keys $n[x]$ has $n[x] + 1$ sub-trees.

For node x and any key k_x^i in x, and for any node y in the left sub-tree of k_x^i: $k_y^j <= k_x^i$.

For node x and any key k_x^i in x, and for any node y in the right sub-tree of k_x^i: $k_y^j > k_x^i$.

B+ Trees

One of the key performance benefits of the B+ Tree is obtained by setting *d* has high as possible, i.e. each internal node has as many keys as possible. This has the effect that the tree becomes more shallow (than if *d* is set to a low value), hence fewer 'node accesses' are required to find a key (hence find a record).

Since each node will usually reside in its own page in the file, this also minimises the number of disk accesses required.

The actual records are usually stored in a doubly-linked list which also enhances performance, for example, for range queries. With a range query, the first record in the range can be identified and then the rest of the records accessed sequentially via the doubly-linked list. There is clearly no need to return to the root of the B+ Tree for each record and search for it independently.

Operations on B+ Trees

To **search** for a record using a B+Tree, you simply descend to the node containing the key and then read the record.

To **insert** a record, there are various cases, illustrated on the forthcoming slides.

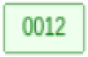
To **delete** a record, there are various cases, illustrated on the forthcoming slides.

B+ Tree: insertion

Consider a B+ Tree with $d = 1$, hence the maximum number of keys per internal node is 2. We will now look at the state of the tree after several insertions.

B+ Tree: insertion

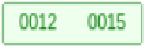
Insertion of key '12'.



https://www.cs.usfca.edu/~galles/visualization/BTree.html

B+ Tree: insertion

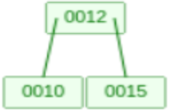
Insertion of key ‘15’.



https://www.cs.usfca.edu/~galles/visualization/BTree.html

B+ Tree: insertion

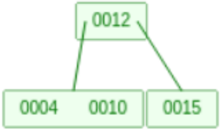
Insertion of key ‘10’. Since adding ‘10’ to the existing node would have exceeded the maximum number of keys, the node is split and two further nodes are created. The middle key value remains in the root node and the others are placed into the left (less than and equal to) tree and the right (greater than) tree.



https://www.cs.usfca.edu/~galles/visualization/BTree.html

B+ Tree: insertion

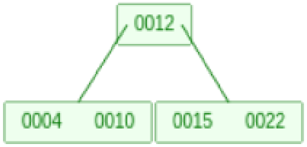
Insertion of key ‘4’. This can be accommodated in an existing node.



https://www.cs.usfca.edu/˜galles/visualization/BTree.html

B+ Tree: insertion

Insertion of key '22'. This can be accommodated in an existing node.

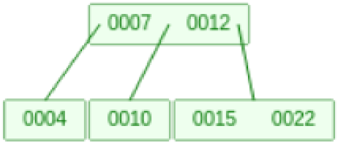


https://www.cs.usfca.edu/~galles/visualization/BTree.html

What will happen now if we insert '7'?

B+ Tree: insertion

7 is less than 12 and the root node still has capacity. 7 can be added to the root node, meaning that another sub-tree had to be created.

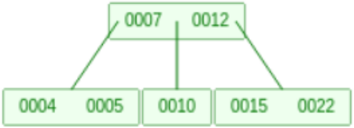


https://www.cs.usfca.edu/˜galles/visualization/BTree.html

What if we now add ‘5’?

B+ Tree: insertion

5 can be accommodated in the <7 sub-tree.



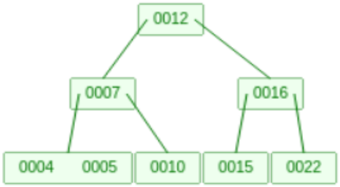
https://www.cs.usfca.edu/˜galles/visualization/BTree.html

We can see that the node containing 10 still has capacity too.

What if we now add '16'?

B+ Tree: insertion

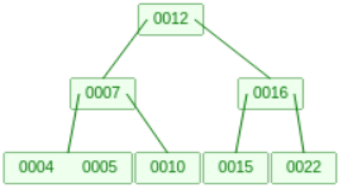
To add 16 we need to split the node containing 15 and 22 and create a new sub-tree with 16 as the root of that sub-tree. However, this would mean that right hand part of the tree now had a greater depth than the left hand part. The tree must be reorganised so that this is not the case. This involves 'bringing down' 7 from the root and creating a new sub-tree rooted at 7.



https://www.cs.usfca.edu/~galles/visualization/BTree.html

B+ Tree: deletion

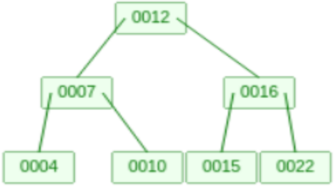
Deletion involves the ‘reverse’ process. Consider the same tree on the previous slide but now with the deletion of 5 (shown on the next slide).



https://www.cs.usfca.edu/˜galles/visualization/BTree.html

B+ Tree: deletion

Deletion of '5'.

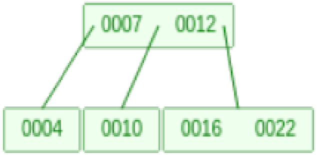


https://www.cs.usfca.edu/~galles/visualization/BTree.html

Note that now, all of the nodes 'minimal occupancy'. Any deletion now is going to trigger a reorganisation of the tree to maintain the uniform depth.

B+ Tree: deletion

Deletion of '15'.



https://www.cs.usfca.edu/~galles/visualization/BTree.html

Comparison of file and index methods

We will now consider each of the methods outlined in terms of the following common operations.

- Scanning all records.
- Searching on an equality condition.
- Searching on a range condition.
- Inserting a record.
- Deleting a record.

The ‘costly’ component of any of these operations is disk access time. This will depend on the number of records hence the number of pages in the file(s) accessed. The fewer disk accesses the better.

On average, an index file is about 10% of the size of a record file. This means that loading indexes is a better option than loading record files, where possible.

Heap files

To **scan** a heap file clearly requires examining all records in the file.

To search on an **equality** condition for a single item requires, on average, searching half of the records.

To search on an **equality** condition for a multiple items require searching all of the records.

To search on an **range** condition requires examining all records in the file.

Inserting a record simply involves adding the record to the end of the file (but see *delete* below).

If no compacting occurs, **deleting** a record simply means marking the location in the file as available. The insert method can be adapted to find the first available location rather than always adding at the end. If compacting occurs then all records after the deleted record will need to be moved.

Sequential files

To **scan** a sequential file requires examining all records in the file.

To search on an **equality** condition for a single item where the condition depends on the key, binary search can be used to find the record in log_2 steps.

To search for a single item on an equality based on a different attribute to the key requires, on average, half of the records to be examined.

To search on an **equality** condition for a multiple items where the condition depends on the ordering key, binary search can be used to find the first record in log_2 steps, and then a linear search obtains the rest of the records with the same key value.

To search for multiple items based on a different attribute to the key requires all of the records to be examined.

To search on an **range** condition, binary search can be used to find the first record in the range in log_2 steps, and then linear sequential search can locate the rest of the records in the range.

Sequential files

Inserting a record involves locating the entry position, inserting the record and then either shifting all of the subsequent records, in the case of contiguous storage, or by fixing pointers, in the case of linked list storage.

Deleting a record involves locating the record to be removed, removing the record and then either shifting all of the subsequent records, in the case of contiguous storage, or by fixing pointers, in the case of linked list storage.

If the overhead of insertion and deletion are undesirable, then new records can be added to overflow pages instead (which does negate the benefits of sequential organisation somewhat), and deletions could involve simply marking a location as available.

Heap file with a B+ Tree index

To **scan** a heap file requires examining all records in the file. With the B+Tree this involves reading all of the leaf pages.

To search on an **equality** condition for a single item requires only a tree search for the required key. This applies to multiple items too. The rest of the items will generally be found in the same leaf page.

To search on a **range** condition for a single item requires only a tree search for the key of the first record, and then a linear search for the rest of the records in the range.

Heap file with a B+ Tree index

Inserting a record involves adding the record to the heap file and then adding the record to the B+ Tree index. This has a worst case performance when no node in the index has capacity and a new one must be created, with the key having to 'rise' up the tree. If the capacities at the nodes are large then this operation should not often result in the worst case.

Deleting a record involves the operations as discussed under *heap* files, plus the reorganisation of the tree, as discussed above.

Heap file with a Hash index

To **scan** a heap file requires examining all records in the file.

To search on an **equality** condition for a single item requires only computing the hash value from the key and then directly accessing the record.

To search on an **range** condition with a hash index degrades to a full scan. The hash method does not help us in this case.

Inserting a record involves adding the record to the file and then computing its hash and storing that in the index.

Deleting a record involves deleting the record from the file (as discussed above) and then also deleting it from the hash index.

Performance

Indexes are used to improve the performance of DBMSs in executing queries. Files without indexes, even sequential files, can lead to performance issues.

A hash index is useful if the queries to be executed on a relation are generally of the form *'WHERE attr = val'*. The DBMS simply computes the hash value for attr (if a hash index has been created for that attribute) and is led directly to the record ID for the record. A hash index is no use for range queries because records are stored independently of each other with this method. Related records will more than likely not be stored contiguously in the file.

A B+ Tree index is useful for all queries but is generally slower for straightforward equality conditions.

Creating indexes

SQL does not have commands for creating indexes. However, all DBMSs have commands to create them. In *Postgres* the basic syntax is as follows

```
CREATE INDEX indexname
ON tablename (attr);
```

By default, *Postgres* creates B+Tree indexes.

The buffer pool

Each DBMS maintains a *buffer pool* which is a series of blocks of RAM that it uses for query processing, etc. The more blocks of RAM that are allocated to the DBMS the better for the performance of the DBMS.

Any query that generates large intermediate results might be slow because there may be insufficient blocks of RAM available to store it. In this case, the result will be computed in blocks that *will* fit in the available RAM blocks, and/or intermediate results will have to be stored on the disk. Again, this will be slower.

In addition, therefore, to minimising the number of disk accesses required to locate a record or series of records, the *size* of the intermediate results should also be minimised so that they fit in the available buffer space. We will return to this point in the next session.