TDT4145 Databases and database management systems

Storage, Indexes and Query Processing

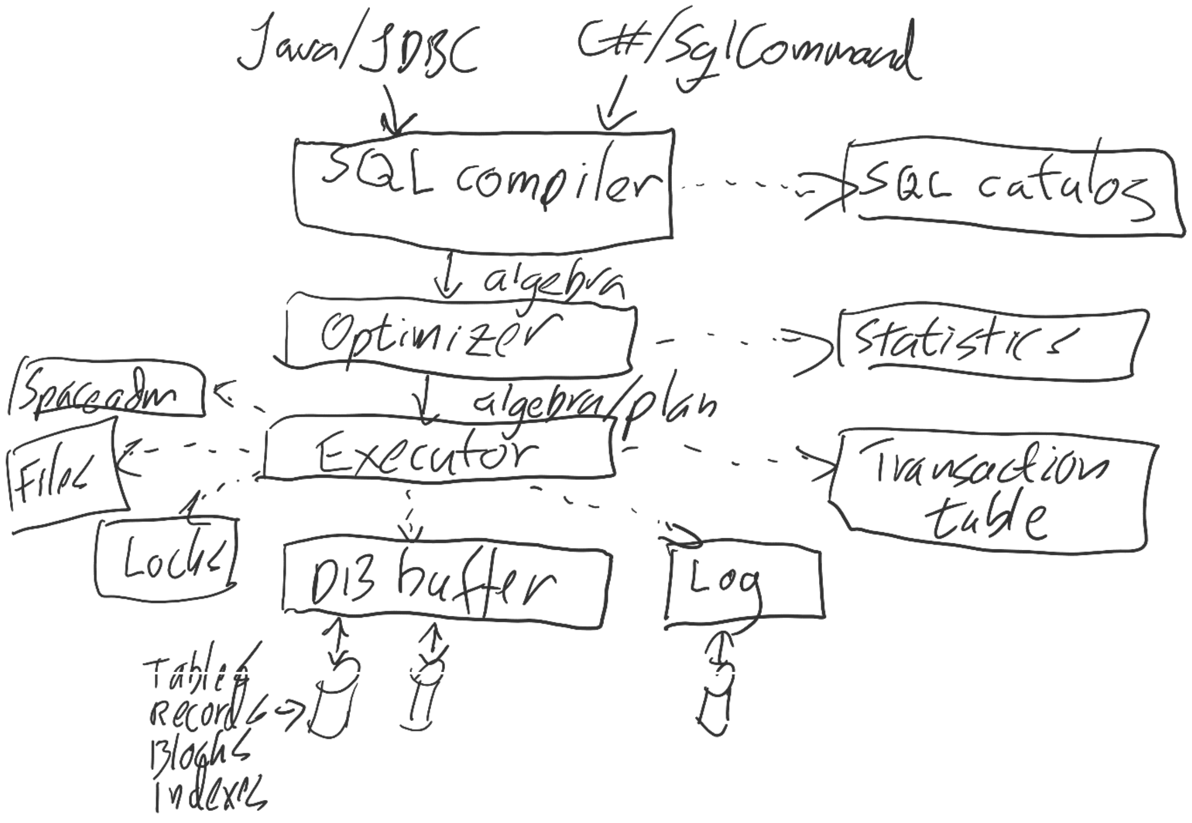
Svein Erik Bratsberg
IDI/NTNU

17. January 2020

This is a compendium made to replace Chapter 16 and 17 and to supplement parts of Chapter 18 in Elmasri & Navathe. Note that Chapter 18 has contents which is compulsory. See the list of compulsory texts (pensumliste).

1. The architecture of a DBMS

A typical software architecture for a SQL DBMS is shown in the figure below:



The SQL compiler takes SQL as input and returns tuples/records to the client. By using the SQL Catalog it can translate the SQL query to an algebra tree. The optimizer transforms the algebra tree to a more optimized algebra tree including a concrete access plan to the data using statistics about the data. The executor interprets the tree/plan and uses many modules in the system to execute the query and access data, logging transactions, locking data, etc. The database buffer keeps data in memory and will periodically write data to disk. The data is read/written as blocks, but the blocks contain data in the form of records being rows in specific tables. The data may also be indexes. The log is responsible for storing log records, which are used to let transactions be atomic and durable. Most of these concepts will be explained in the rest of the course of TDT4145.

2. Database storage

A database needs to be stored permanently, i.e., it resides on a disk, either a HDD (rotating disk) or a SSD. The database may be stored as a file in the file system in the operating system, or it may reside as «raw devices» on special partitions of the disk. The advantage of having special partitions is that the DBMS gets full control of the layout of the data. This is typically combined with the ability to have full control on when data is written to disk. Thus, the DBMS may have full control of what data resides in the buffer (main memory). The DBMS has more knowledge than the operating systems about which data should reside in the memory of the computer. The trend nowadays is to let the file system of the operating system do the file management, and to use operating system support for direct I/O and pinning data in main memory.

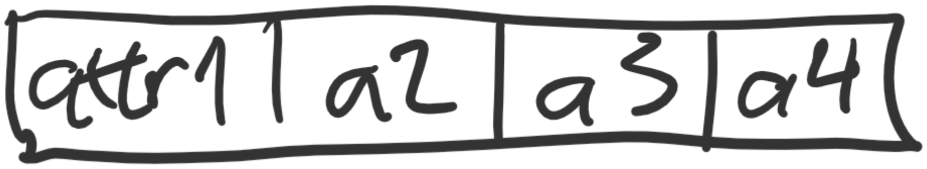
There are multiple tasks related to file management:

1. We need to know which data is related (belonging together) by having files. Thus, data needs to be linked together.
2. The free space of the disk needs to be managed. The DBMS needs to allocate and deallocate data as the database is "living".

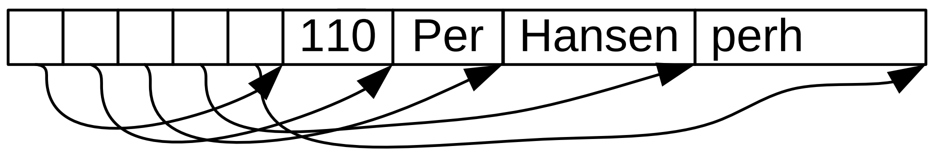
Further, in this text, we will not make any assumption about which solution is chosen, i.e., databases are stored in a device identified by a deviceId. We may think of a deviceId as a file name or as a native deviceId.

3. Record format

A traditional DBMS uses what is called a *row store*, in which every row of a table is stored as a record in the database. There are multiple possible record formats. The SQL catalog describes the format of the table. When the table has fixed length attributes, each attribute resides on a fixed offset within the record. Thus, to decode a record the DBMS knows where in the record each attribute resides. This is illustrated by the figure below: A record with 4 attributes.



When you have variable length fields, e.g., VARCHAR, we need to store the record differently. Below we show the «record vector» format, where each field of the record may be of variable length. The record is almost self-describing by having pointers to each field. Often this is combined with a special field telling the number of attributes, the number of keys, etc. Type information may also be combined with each field pointer. However, this is usually described in the SQL catalog.



Another format to support variable length fields is to use delimiter characters to indicate the end of the field. The delimiter character is a special value used to show that a field is ending. This may also be supported by a special end of record character.

4. Block format

A block is a basic storage unit used by the DBMS to store data in a database. Every time the DBMS reads or writes data to/from the disk, it is in the unit of blocks. Very often the data is stored both on disk and in memory as blocks, such that it is easy to write or read data from the disk. However, higher level DBMS software typically sees data as collections of records, e.g., tables or results of queries.

A block is usually identified by a BlockId, which may consist of a deviceId and an index into the device. A BlockId could be a 4 byte or a 8 byte field. With 4 bytes, one byte is used for the deviceId and 3 bytes to the index within the device. E.g., if each block is 16 KB, each device may store 2^{24} blocks of 16 KB, which makes altogether 256 GB. By using 8 bytes identifiers, you may e.g. use 1 byte for deviceId and 7 bytes for offset, creating the ability to span really large devices.

We may think of a block as a collection of slots, where a record is identified by (BlockId, slot number). This is often named as a RID, Record Identifier. Thus, a RID is a semilogical identifier, which identifies the block on the device, but which is logical within the block, creating the possibility to move a record inside the block using an appropriate block format. Note that a record could be accessed by a key as well, e.g. in B+-trees, where the records are sorted according to the search key of the B+-tree.

There are multiple possible block formats used in DBMSes. In its simplest form the fixed length records are simply aligned after each other, and the DBMS may calculate the positions by simple arithmetic using the RID and the record size found in the SQL catalog.

If the records are of variable length, e.g., using VARCHAR attributes, we need a more advanced scheme. We may use delimiter characters of attributes and records. Delimiter characters are special values to show the end of an attribute and the end of a record.

Another scheme used is to have a format as shown in the figure below. This allows for variable length records, for sorted records, easy delete and insert operations, etc. It is also possible to compact a block easily by «copying records downwards». This is done when a sufficient number of records has been deleted, and there are multiple “holes” between the records. Records (“tuples” in the figure) are aligned from the start of the block while the record pointer vector (“page directory” in the figure) is aligned from the end of the block. In addition to what is shown in the figure below, it is typical to support flip/flop, i.e., an atomic read/write stamp at the start and end of the block. These are numbers which should match when reading a complete block. Thus, the numbers are updated prior to write of the block. There are other header fields of the block as well, used for identification, recovery purposes, and possible checksums.

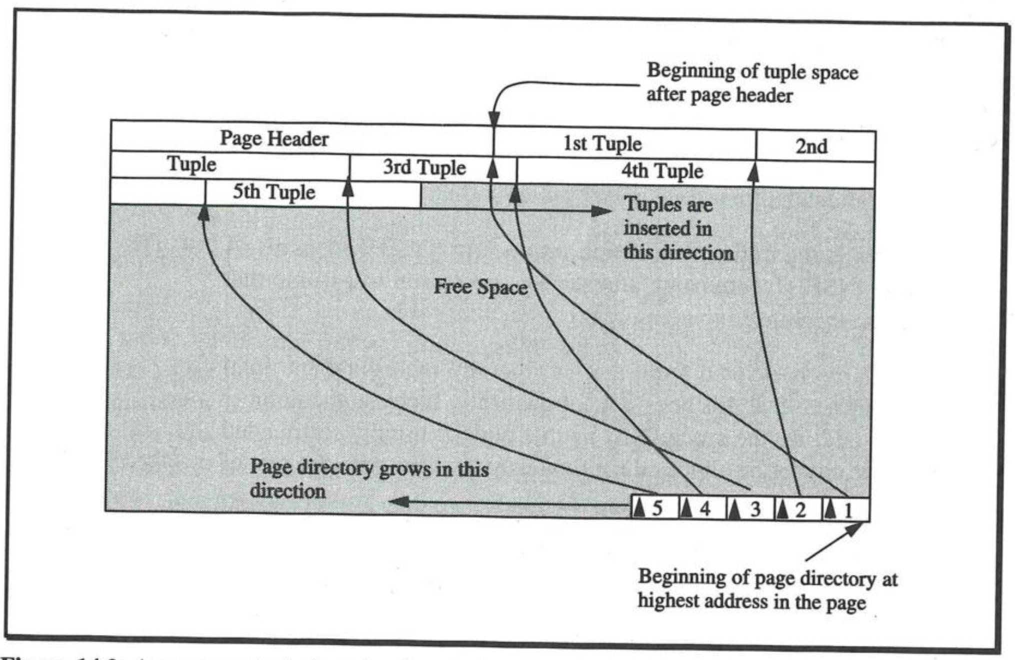

Beginning of tuple space after page header
Page Header
1st Tuple
2nd
Tuple
3rd Tuple
4th Tuple
5th Tuple
Tuples are inserted in this direction
Free Space
Page directory grows in this direction
5
4
3
2
1
Beginning of page directory at highest address in the page

Figure 14.2: A common technique to allocate two dynamic data structures in the

5. Database buffer

Blocks are read from the disk into database buffers to be used the DBMS. The database buffer is usually a large buffer comprising big parts of the main memory of the computer. Blocks are accessed on disk using the BlockId, which identifies the device or the file, and the index into the device or the file. When a block is residing in the database buffer, it is also identified by the BlockId, and there exists a hash index to find the block in the buffer. Blocks belonging to the same hash value may be chained together. The nice thing about this organization is that blocks may be read and written directly into / from the database buffer without reformatting any data.

The DBMS likes to use its own database buffer instead of the operating systems virtual memory. The reason for this is that it gets full control of which blocks reside in memory or not. The DBMS knows more about the semantics of the data and how it is going to be used. Thus, it may use prefetching of data when it sees that a scan of a table is happening, for example.

Another task that a DBMS needs to do is to *force* blocks to disk as a part of transaction processing. This means that it needs to write data blocks which are changed as a part of the processing of user transactions. More often, it needs to force *log blocks* as a part of transaction processing. This is treated in the chapter about recovery in Elmasri & Navathe.

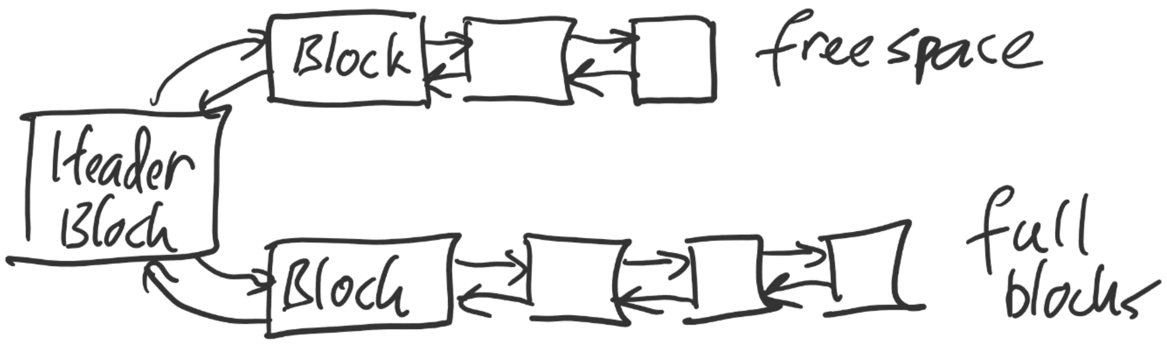
6. Heap Files

The most basic storage of tables is in *heap files.* The data in a heap is not ordered in any particular way. The records are simply stored in the order that they are inserted. A record in the file may be found by iterating through the records of the file. Thus, the operations on the file is to create and delete the file, to read, insert and delete records in the file, and to scan through the records of the file. When there are sufficiently many deletes to a heap file, a reorganization may take place. This will compact the blocks.

There are multiple ways of implementing heap files. Usually, they use a linked list scheme where there are forward and backward pointers between the blocks. The pointers are BlockIds, i.e., logical pointers which may point to blocks in memory or at the disk.

One way is to allocate blocks one at a time when necessary. Another way is to allocate multiple blocks at the time. The reason for this is that disks get more throughput when blocks of a file are allocated in sequence. Where to allocate blocks on disk is a separate theme. Operating systems (file systems) also have good support for smart allocation of blocks onto devices.

Some implementations differ between full blocks and blocks having some free space. They will have two separate linked lists. Thus, it is easy to find blocks that have available space for new records.



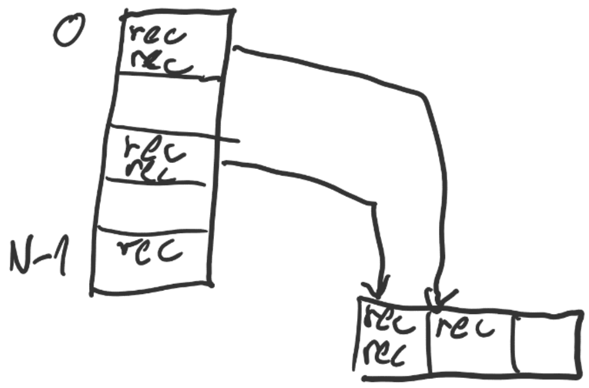
Heap files are mainly used to store tables when there is an index to the table, or it may be used to store records in temporary files during query processing.

7. Static hashing

Static hashing is hashed storage in its basic form. A file is composed of *N* blocks, which store the records. By applying a hash formula to the search key, the record is placed in a block of the file. The hash formula spreads the records evenly onto the blocks. To retrieve a record, the formula is applied to the key, and the appropriate block is found. A simple formula to use is *h(K) = K mod N*, where N is the number of blocks in the hash file.

The essence to good hashing performance is to have a hash function which spreads records evenly. In case of skewed load on the hash file, the hashing is not performing very well. Anyway, it becomes a need to have overflow storage in one or another form:

1. *Open addressing*: Store the record in the first successive block having free space.
2. *Separate overflow*: Special blocks storing overflow records. The overflow blocks may be shared among many blocks, or may be separate for each block needing overflow storage
3. *Multiple hashing*: Use another hash function to calculate the block to store the record. This may be considered as *distributed overflow*.



The figure above shows a simple example of a hashed file with separate overflow. By hashing on the *search key* of the record, the block to store the record is found. When the block becomes full, an overflow block is used to store additional records. In case this becomes full as well, a second overflow block is chained into the list.

The main problem with static hashing is that it is not dynamic. This is a big problem when the number of records is hard to estimate, and it is a dynamic nature of the data. Long overflow chains kill the performance of static hashing.

8. Extendible hashing

Extendible hashing is designed to allow for dynamic hash files, i.e. a hash file where the number of records is not known in advance. When using static hashing the number of records must be estimated at start and the length of the hash file must be decided. When a block in a static hash file becomes full, it is possible to double the length of the file and rehash all the records to fit the new file. However, this requires us to read and write every block of the file. Usually, overflow blocks are used to handle this situation.

Extendible hashing uses a directory to point into the hash file. The directory points to the logical structure of the hash file, while the blocks in the hash file might be differently allocated.

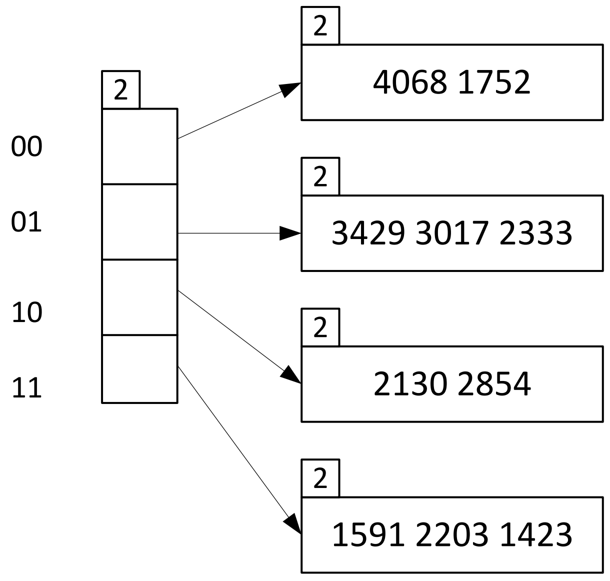
We describe the hash structure by showing an example. Assume a hash function *h(K)* = K MOD 16.

Insert the following keys (K), where we have shown the key, the hash of the key in decimal and binary:

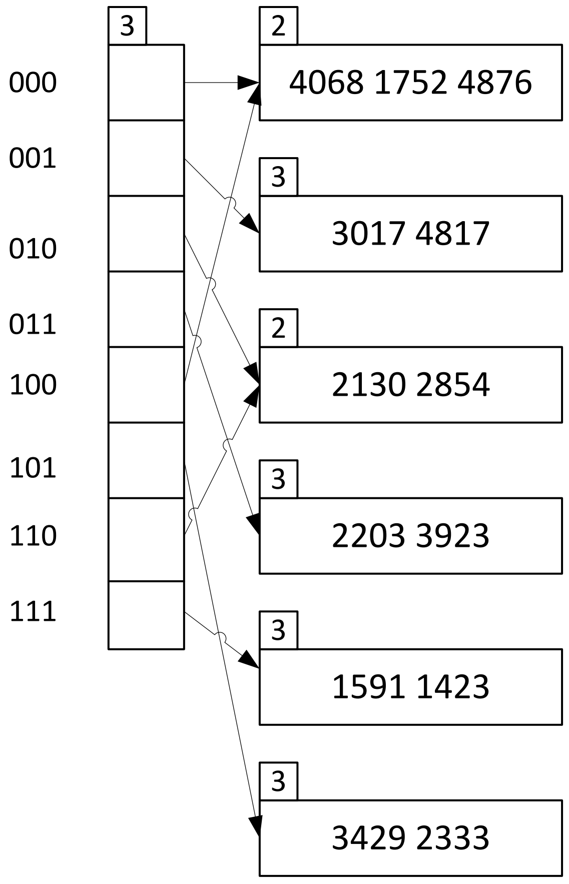
K	**h(K)**	**binary**
4068	4	0100
1752	8	1000
3429	5	0101
2130	2	0010
2854	6	0110
1591	7	0111

2203	11	1011
1423	15	1111
3017	9	1001
2333	13	1101
3923	3	0011
4817	1	0001
4876	12	1100

We start by having a directory with 4 slots and 4 data blocks in the hash file. Each block has space for three keys. The key 4068 has hash value 4 (decimal) or 0100 (binary). The directory is indexed by the two last digits of the binary number. Thus, the key 4068 is hashed to block 0. After inserting the first 10 records we get the following hash file:



The number 2 attached to both the directory (global depth) and the data blocks (local depth) tells that we have used two bits in the hashing. We always start by having *local depth==global depth*. When inserting the 11th record (3923) we get a *directory doubling*. After inserting all 13 records we get this structure:



After the directory doubling we use three bits for the hashing when inserting new records and when splitting existing blocks. We do the splitting by using the bits from the right (less significant bits), meaning the three last bits.

The insert of the key 3923 results in a block split. Thus, the keys in this block (11) are rehashed into two blocks (011 and 111). Thus, keys 1591 and 1423 are moved to 111 (7 in decimal), while the new key 3923 ends in 011 together with the existing key 2203.

Similarly, for the insert of key 4817, it results in a split of block 01 to 001 and 101. Note that blocks that are not split retain their attached number, i.e. the *local depth*. The number attached to the directory is named *global depth*.

When a block is full and we try to insert a new key and the local depth for the block equals the global depth, we get a directory doubling.

Extendible hashing is regarded as a good solution to dynamic hash files. However, for lookups the directory result in two accesses to the hash file instead of one as in the normal case in static hashing. For many years the existence of a directory was considered as a problem, since it could become large compared to the size of main memory. Fortunately, hardware has evolved since that time.

9. Indexing

This chapter is intended as a definition of central indexing and storage concepts used within the course TDT4145 at NTNU.

There are two reasons for doing indexing.

1. Improve retrieval speed when accessing through the indexed field.
2. Ensure uniqueness for the indexed field.

The disadvantage for doing indexing is that you need to update the index when doing inserts or updates of the indexed records.

Index field: Field/attribute of the record/row that is indexed.

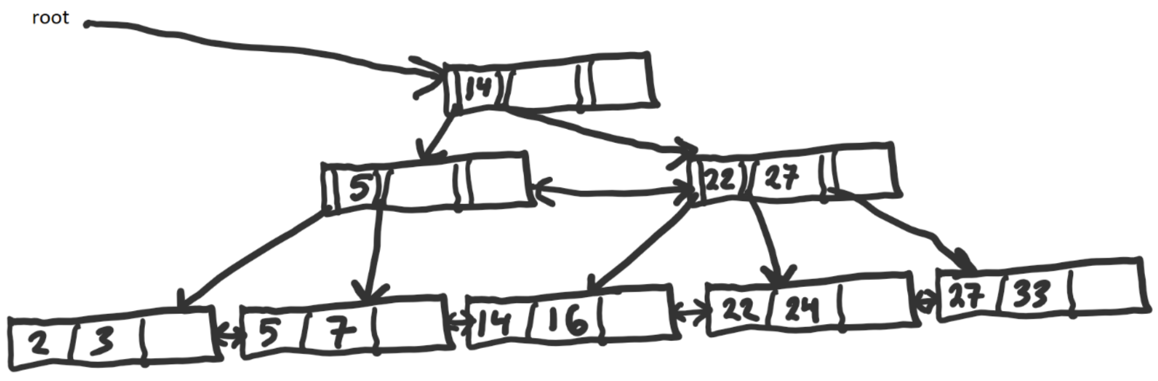
Primary index: An index structure which indexes the primary key of the records/rows.

Clustered index: Index on table where the records are stored physically in the same order as the index records. In practice this is a B+-tree or hash index where the records of the table are stored within the index itself.

Secondary index: An index which is not a primary index.

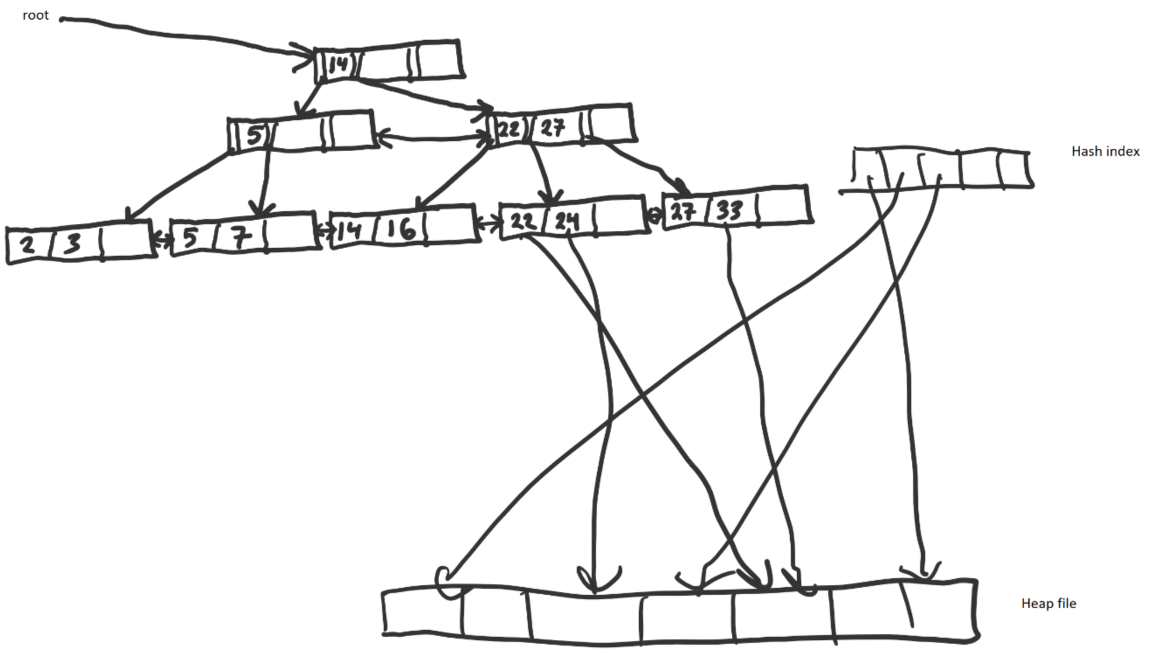
Here we provide a list of typical storage and indexing alternatives when storing and indexing a table:

Clustered B+-tree/clustered index. There is a B+-tree index on the primary key. The leaf level in the B+-tree stores the actual records/rows of the table. Thus, it is a clustered index.



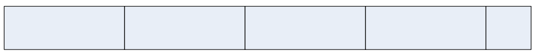
This alternative is used within MySQL's InnoDB storage engine. It is also the storage alternative used in Microsoft's SQL server when a primary key is defined for a table. There, it is called a *clustered index* or simply a *clustered table*.

Heap file and B+-tree. The table is stored in a heap file. There is a B+-tree index on the primary key. There may be a hash index on some fields of the records.



Heap file with a B+-tree index is used in MySQL's MyISAM storage engine. In MicroSoft’s SQL Server this is called a non-clustered index. The pointer from the index to the heap file is called a Record-pointer/RecordId or row locator in Microsoft SQL Server.

- *Heap file*.

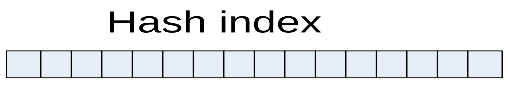
  

  In MicroSoft's SQL Server you get a heap file when no primary key is defined for a table. A heap file is useful when you never do a direct access for a key or search for specific records, except for things like scan, join or aggregation.

- *LSM trees (Log-Structured Merge trees).* This is a modern access method which focuses on write speed over read performance. It keeps the latest written data in a main-memory storage area called MemStore. As new data is inserted, the current MemStore is copied into a disk-oriented structure SSTable. SSTables may be organized in multiple levels, where data is merged downwards in levels over time. This method is useful when you need good write speed and you mainly read the newest data. The LSM trees are regarded as having a superior write speed, due to low “write amplification” (how many disk writes an insert may cause) and due to better compression of data due to larger units, e.g. a SSTable may be 2 MB, while a B+-tree block may be e.g. 8 KB. LSM trees are used in Google’s BigTable, Adobe Hbase, and are present in SQL DBMSes, e.g. through Facebook’s MyRocks which is used as a storage engine in MySQL throughout Facebook. This is regarded as the data structure of “big data”.

- *Column stores*. Traditional SQL databases use row-oriented storage, i.e., each row in the table is stored as a record. Data warehousing requires a lot of aggregation of single attributes and takes advantage of a column-oriented storage of tables. This gives less data to retrieve and makes it much easier to compress the data. Several SQL databases have built in column-oriented storage in their servers.

- *Clustered hash index*. Hash index on the primary key. Clustered index storing the actual records/rows of the table, i.e. a clustered hash index. It is called a *hash cluster* in Oracle.

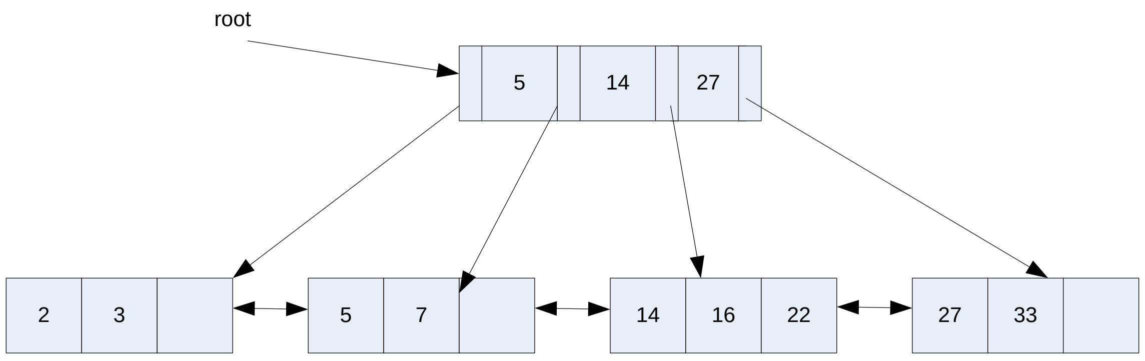


10. B+-trees

This document is intended as a short description of B+-trees to be used within the course TDT4145 at NTNU. Its intention is to replace the text book on B+-trees.

B+-trees are the most widely used access structures within databases. A B+-tree is a generalization of an ISAM structure where you have multiple levels. The advantages of B+-trees are that they are good for many situations, like sequential scans, range scans and they are good at direct access by search key as well. The main properties of B+-trees are that they organize the indexed keys in sorted order and that the tree is always balanced. Thus, they are good for inserts as well. The main disadvantage of B+-trees is that recent inserts tend to spread over the blocks of the tree, making large volumes of write load to the disk.

Below we see an example of a B+-tree with two levels. At the leaf level (level 0) we see *the actual keys* which are indexed by the tree. At the level above (level 1) which in this example is the root level, we see keys which *help us to navigate* to the keys at the level below. E.g. the key 14 has one pointer on the left side. This leads to a block where the keys are less than 14. The pointer on the right side of 14 directs the search to a block where the keys are 14 or greater. The next key at the root block is 27, meaning that the pointer in-between 14 and 27 directs to a block where the keys are 14 or greater, but less than 27. Note that pointers here are *BlockIds,* and they point to the block, not individual records within the block



The keys at the leaf level of the tree may be records storing complete rows in a table, or they may be indexed keys with a field pointing to a record in another (heap) file or tree.

To illustrate some searches in this tree, we first search for key 3. We always start with the current root block. In this case 3 is less than the least key 5. We follow the link to the first block at the level below, the leaf level in this case. To search within a block, a binary search is usually used since the keys are sorted. In this case we find the key as the second key within the block. Note that a binary search first makes sense when the number of records is larger than the “toy” example used in this case.

The second example is searching for the key 15. We start with the root block and find the pointer in between 14 and 27. When searching the leaf level block here, we do not find the key 15. Such a search may be used to return the position for inserting a new record with key 15.

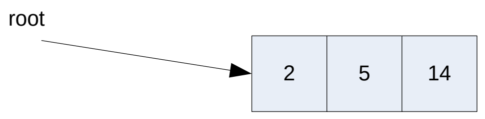
The next example searches for all keys greater or equal to 14. First, we do a binary search at the root block, and find the pointer to the right of key 14. This will point to the block with keys 14 and larger. We scan the leaf level sideways starting with the block containing 14. This is one of the beauties with B+-trees, sideways traversals. Note that the pointers go in both directions, meaning that both forwards and backwards scans are possible. Thus, it is easy to evaluate queries including both “where attribute < 14” and “attribute > 14”. In addition, it is easy to deliver sorted keys when this is requested.

Another strong point of B+-trees is that records may be inserted anywhere in the tree without any noticeable loss in efficiency. If there is space in a block when inserting a record, the record is simply inserted in that block. If it is not enough space, the existing records are divided in-between the existing block and a newly created block. This is called *block split*. Normally, half of the records are moved to the right in the newly created block in the split.

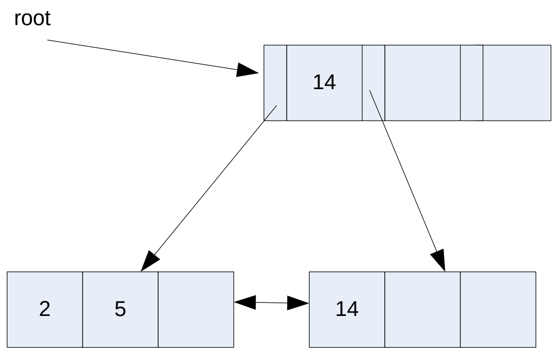
Insertions into B+-trees

We will show a "toy" example of insertions of keys into an empty B+-tree. An ordinary B+-tree would have many records/keys per block, e.g. 20 to 200. However, in our "toy" tree we have just a few. The following keys are to be inserted in the given order: 2, 5, 14, 22, 27, 33, 3, 7, 16 and 24. We assume blocks with space for 3 keys, and with 4 pointers in the blocks above level 0.

After inserting the first three keys we get the following "tree":

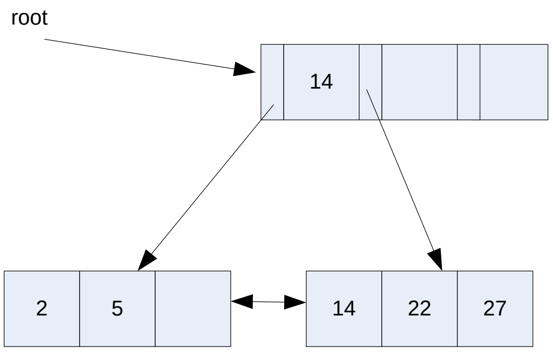


The root is pointing to a single level 0 block. We try to insert key 22, but there is no space for this. Thus, we need to split the leaf level block. After splitting this block, we get the following tree:

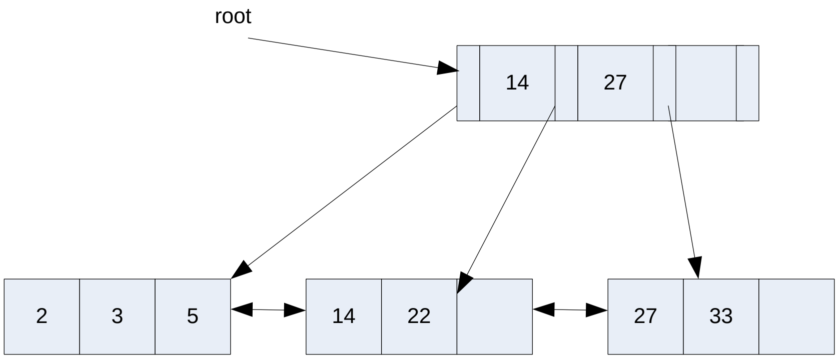


At a block split we move half of the keys to the new (right) block, before we insert the new key. In this example it means that just one record is moved to the block to the right here (key 14). *We always do the block split before inserting the new key.* A new root block is allocated as well. In this root block we insert the *split key*, i.e. the leftmost key in the newly allocated block.

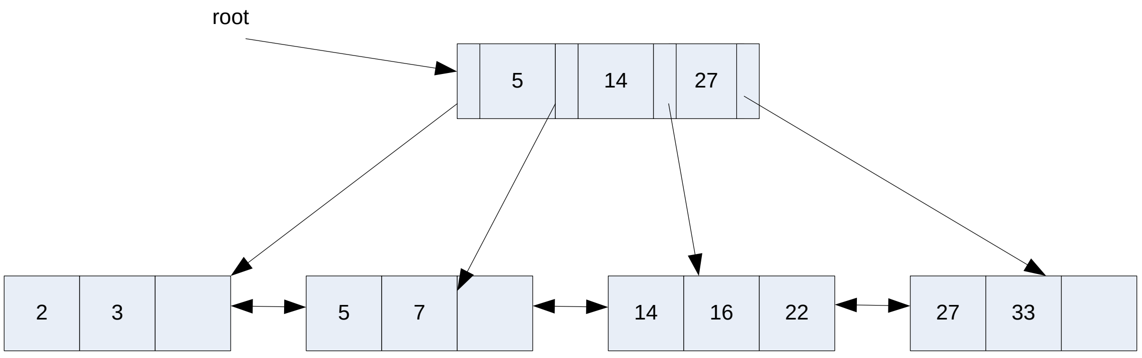
After inserting the two keys: 22 and 27, we get the following tree:



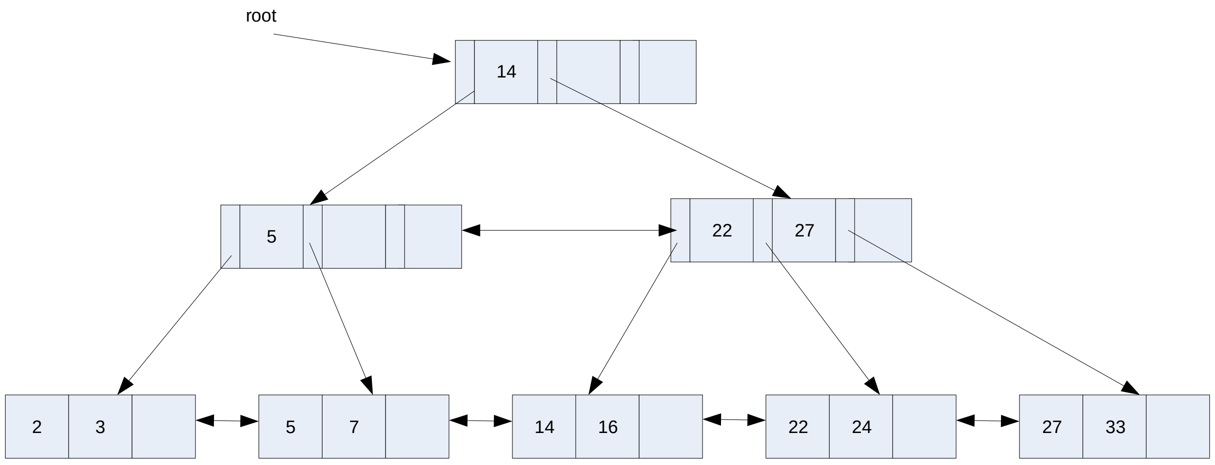
Now, the root points to a block at level 1. After inserting another two keys we get the following tree:



We got another block split at the leaf level. After inserting another two keys, we get the following tree:



We got yet another block split at the leaf level. Note that when splitting a block at the leaf level, we "copy" the split key to the level above. After inserting the last keys, we get a block split at level 1:



Now, something interesting has happened. When splitting a block at a level above level 0, we "move" the split key to the level above. In this case it means that key 14 from level 1 has been "moved" to level 2. The reason for this is that level 1, 2 and above are merely providing directions to level 0, where the "real" indexed keys exist. Thus, we cannot move away a real indexed key from level 0 since it is a part of the user data in the database.

We do not show deletions in B+-trees. Usually, this is done by compacting levels in the tree by a management thread and then deleting the empty blocks. However, some research papers have shown that *delete-at-empty-block* suffices, meaning that you do not need a separate management thread doing this, simply deleting a block when it becomes empty is enough.

Some summarized concepts about B+-trees:

- The indexed records are stored at the leaf level (level=0). The blocks above the leaf level are merely used to navigate to the leaf level.
- The records/keys at one level are sorted.
- The block should at least be 50% filled. On average a block will be 2/3 or 67% filled when inserting keys randomly. Some books operate with 69% fill degree (ln(2) = 0.69)
- The blocks have a size that fits the disk, typically, 4KB, 8KB, 16KB or 32KB.
- Blocks are disk-oriented and are traditionally read into memory in disk format. However, it is also possible to convert the disk format to e.g. Java objects when reading in a block, and opposite when writing the block.
- Search is performed from the root to the leaf level.
- In comparison to LSM trees, B+-trees are regarded to have superior read performance, but LSM trees have superior write performance.

11. Queries and storage structures

This document is intended to complement the text book from Elmasri & Navathe on how simple queries are executed using the indexing and storage concepts presented previously in this document. The intention is to complement Chapter 18 of the text book.

We will use the following storage alternatives for a table.

1. *Heap file*
2. *Clustered B+-tree*
3. *Heap file and unclustered B+-tree*.
4. *Clustered hash index*

Example

To explain the concepts, we will use an example. Assume a table of employees:

Employee (<u>empno</u>, name, age, depno, salary).

We assume there to be 100 000 employees in the table.

1. *Heap file*:

Each block in the heap file may contain 100 employees, so that a heap file will contain 1000 blocks. Note that this is an example and the heap file is a useful starting point.

2. *Clustered B+-tree*

When inserting the same records into a clustered B+-tree there will be 67 % *filldegree*, meaning 67 records will fit in each block in average (when using random inserts). The leaf level (level=0) of the B+-tree will then contain 1500 blocks. How many levels will there be in the B+-tree? Usually, a B+-tree has 3 or 4 levels. In this case an index record in the B+-tree will be on the format *(empno, BlockId)*. If we assume such a record to need 20% of the storage space of an Employee record, level 1 of the clustered B+-tree will contain 1500 records because there is one record pointing to each block at the leaf level. Each block may contain 67 * 5 = 335 records, because the size of each record is 20 % of the Employee records. Thus, there will be 5 blocks on level 1 because 5 * 335 = 1675, being more than 1500. There will be 1 block on level 2 since this level contains just 5 records. The clustered B+-tree has 3 levels.

3. *Heap file and unclustered B+-tree*

The heap file will be the same as in the original with 1000 blocks. How big will the unclustered B+-tree be? In this case the records at the leaf level will be on the format (empno, RecordId). The RecordId may be assumed to be *(BlockId, index within block)*. E.g. 4 + 4 bytes. If we say *(empno, RecordId)* needs 25% of the space of an employee record, the leaf level will contain 25% * 1500 blocks = 375 blocks. Level 1 will contain 375 records *(empno, BlockId*). This will probably be contained within 1 block. There will be space for 335 records in average per block, as calculated in the clustered B+-tree example. But, the split usually appears when the block is full. That is when 335 + 50% = 503 records are inserted. Thus, this B+-tree will have 2 levels.

4. *Clustered hash file*

A hash file typically has 80 % fill degree. Thus, a clustered hash file of the example table will contain 1000/0.8 = 1250 blocks.

Simple SELECT queries

There are multiple types of queries. We consider some simple ones.

SELECT * FROM table;

This is to select all records from the table.

1. (*heap file*) We need to scan all 1000 blocks.
2. We may scan the leaf level of the B+-tree, but we also need to traverse down to the leaf level. Thus, 2 + 1500 blocks.
3. Here, we may scan the heap file. 1000 blocks.
4. We need to read the whole hash file, 1250 blocks.

SELECT attributes FROM table WHERE key=constant;

We assume the key to be the indexed attribute.

1. In average, we need to scan half of the heap file. Thus, 500 blocks. We assume they key to be unique here.
2. We need to traverse down the B+-tree. Thus, 3 blocks.
3. We need to traverse down the B+-tree plus accessing the heap file using a RecordId. Thus, 2 + 1 blocks.
4. In a hash file the average blocks to be accessed is 1.2. This is due to some overflow blocks.

SELECT attribute FROM table WHERE key > constant;

1. The heap file needs to be scanned entirely. Thus, 1000 blocks.
2. Here, we need to traverse down the B+-tree, and scan forwards at the leaf level. If we assume that 20 % of the keys match, we traverse 2 blocks down and 0.2*1500 sideways. 302 blocks.
3. If we choose to use the B+-tree index, we need to traverse 1 block downwards and 0.2*375 = 75 blocks sideways. In addition, we need to follow all 0.2*100 000 = 20 000 pointers (RecordIds) to the data records in the heap file. Thus, 20 000 + 1 + 75 = 20 076 blocks. Then, it is better to scan the heap file, 1000 blocks.
4. We need to scan the hash file. 1250 blocks.

12.JOIN queries

We will treat nested loop join. For join queries indexes are of no help, unless they are indexing the join key.

If we add another table to our example

Department(<u>dno</u>,dname,manager,location)

we could consider a query like:

SELECT name, salary, location FROM Employee, Department WHERE Employee.depno=Department.dno AND dname='Sales';

To illustrate this, we assume there to be 500 departments stored in 5 blocks in a heap file. We assume there to be 5 blocks in the buffer. Thus, in a nested loop implementation, we may use 3 buffer spaces for one table, 1 for the other table and 1 for holding a result block.

Heap files.

In a plain nested loop, we could read 3 blocks into the buffer from Department and then read 1000 blocks (one and one) from Employee. Then we read the last 2 blocks from Department and then scan the 1000 blocks of Employee again. In total we get 5 (Department) + 2*1000 (Employee) = 2005 blocks read. We do not say anything about the result, because we do not know the size of the result (i.e. how many Employees that work in the Sales department). If we switch the order of the tables, by reading the employee table into the 3 buffer slots, we get 1000 + (1000/3)*5 blocks. Thus, 1000 + 334*5 = 2670 blocks. It is considered as a general rule to have the smallest table in the outer loop of the nesting.

If we consider the query in detail, we note that there probably will be few departments which are used in the join, possibly just one (dname='Sales'). Thus, we may read this record from department first. This takes either 1, 2 or 5 blocks depending on the organization. This record / these records will certainly be contained within one block in buffer. Thus, we only need to scan the 1000 blocks in the Employee table. In total, e.g. 1001 blocks are read.