1 Physical Database Design

The logical level of DB systems must be **implemented** in the physical level. The implementation should perform well even with huge amounts of data.

2 Storage Media

From the **storage device hierarchy** (cache, memory, harddisk), we know that we have fast access to memory and slow access to disk, but memory is limited due to the cost and is volatile (erased when system failure). Our algorithms for accessing data must try to **minimize disk access**, i.e. access as few blocks as possible, and preferably not move the disk-arm back and forth too much. This can be accomplished by **scheduling** a set of accessed blocks with the elevator algorithm (if the arm has passed the block we want to access, we must wait until the arm reverses direction), **organize files sequentially** on the disk.

3 Storing Records on Disk

A **field** is an attribute value, as stored on disk. A **record** is a list of fields on disk, corresponding to a tuple. A **file** is a set of records stored on disk. Databases are stored on block-access devices (HDs), and a **block** is the smallest addresable unit of data on such a device, e.g. 512 bytes.

If the maximum size of a record is known, data should be stored as **fixed-length records**. The record size should preferably be **aligned with block-size**, possibly with padding, to avoid 2 block accesses per record. A fixed-length record file should have a **header** with a pointer to the first free record in the file. Each free record should have an extra field with a pointer to the next free record (**linked list**), or NULL, if it is the last free record. When a record is deleted, the record pointer is added to the linked list.

Storage of relations with **variable-length records** is necessary when using variable-length attributes (e.g. Binary Large OBjects (BLOBs)), multi-valued attributes (sets of values), or storing multiple record types in the same file (to reduce file count).

The **byte-string representation** method is similar to C strings, storing records as strings, terminating each record with a special **sentinel** symbol. This method initially saves disk space, but after deleting and inserting records, the file will be fragmented. If a record grows longer, it must be moved, which can be expensive. The **slotted-page structure** is similar to a self-defragmenting filesystem with a header, followed by free space, and the records stored contigously (without space in between) at the end of the file. The header consists of the following fields:

- The number of record entries in the header ($\approx$ inode count)
- A pointer to the end of the free space
- An array containing the location and size of each record ($\approx$ inode table)

Records are inserted at the end of the free space. Each time a record is deleted, the data is 'defragmented'.

The **fixed-length representation** method uses one or more fixed-length records to represent a single variable-length record. If the records have a **known maximum length**, we can place a sentinel in the field after the last of the record to signify the end of the record. Another possibility is a **linked list representation**, where we add a pointer field at the end of each record, pointing to the next record in a list of records, where the single-valued fields are omitted in all but the first in the list. A NULL pointer is used on the last record in the list. The downside to this approach wasting the space of the single-valued fields in all the 'overflow' blocks.

The **order** of records in files is significant for the time complexity of the search and insert operations. If files are organized as **heaps** (no ordering), inserting is constant time ($O(1)$), but searching is linear ($O(n)$) in the number of records. If files are organized **sequentially** on some attribute set, the insert operation is potentially slow, as we may have to reorganize the file. This can be somewhat remedied by using a linked-list representation. Searching on values from the attribute set on which the file is ordered, however, can be done in logarithmic time ($O(log_2\ n)$). A **clustering file organization** is accomplished by splitting a relation into files, based on the value of a particular attribute, e.g. store all the records where city='Aalborg' in one file, and the records where city='Aarhus' records in another file.

4 Indexes

Indexes are used for **optimizing queries**.

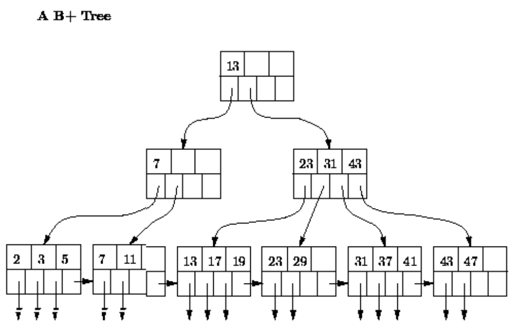
A **primary index** is an index on the attribute set, on which the data file is ordered. A primary index can be **dense**, having an index entry for each value of the key, or **sparse**, having index entries for only a subset of the key values. When using a sparse index, we find the correct range of key values in the index, and then use linear scan to find the correct record.

To speed up searching in a large index, we can make a sparse index on the primary index. This is called a multilevel index.

A **secondary index** is an index on another attribute than the one on which the file is ordered. This must be dense.

4.1 B$^+$-tree Indexes

A **B$^+$-tree index** is an effecient multi-level index, based on a search tree. A **search tree** is a tree data structure, in which each node has a set of **keys**, dividing a range of the key value into subranges. To search the tree, we compare to each value in a node, and if the key value is not in the node, we select the child node whose values are between 2 key values in the parent, by following a corresponding pointer.



A **B$^+$-tree** is a **balanced** search tree, i.e. all leaves have the same depth (the height of the tree). The **maximum number of pointers** per node is decided by the designer of the implementation - a sensible number is enough pointers for the node to fill an entire disk block. The **minimum number of pointers** is half the maximum number, rounded up. The leaves have data pointers corresponding to keys, pointing to the actual records, and each leaf has a pointer to the next leaf, enabling linked list-traversal of the leaves.

Searching a B$^+$-tree index can be performed using only logarithmic ($O(log_t\ n)$) disk accesses, where t is the number of pointers per node.

Updates to a B$^+$-tree index may not violate the bounds on the pointers per node, so updates to the tree structure must be performed in some cases.

Insertions may result putting too many keys in a node. To prevent this from ever happening, whenever we encounter a node that is 'full', meaning that it has the max number of keys, we split the node into 2 new nodes, and move the median (middle) key up to the parent node. If the root node has the max number of keys, we split it and make a new root with the median value.

There are two problems with deleting elements: first, the element in an internal node may be acting as a separator for its child nodes, and second, deleting an element may put it under the minimum number of elements and children. Each of the problems must be dealt with, recursively up through the tree. The deletion, though complicated, is still performed using only logarithmic disk accesses, as a function of pointers per node.

4.2 Hash Index

A **hash index** is yet another index type, using a hash function from search key to a **bucket** pointer. A bucket is just an unordered record file, which must then be linear scanned. It is important that the hash function distributes the keys uniformly in the buckets, to avoid too much linear scan. It is only usable for equality (=) queries, results from range queries will be scattered in different buckets.