9. Physical Access

Access methods

Data may be stored on disk in different formats to provide efficient query execution.
The **Access Method Manager** transforms an access plan generated by the optimizer into a sequence of physical access requests to (database) disk pages.
It exploits **Access Methods**, software modules, specialized for single physical data structures and provides primitives for reading/writing data.
An **access method** selects the appropriate blocks of a file to be loaded in memory by requesting them to the **Buffer Manager**.

Sequential structures

Tuples are stored in a given sequential order.
There are several sequential structures.

Heap file

Tuples are sequenced in *insertion order*.
All space of a block is exploited before starting a new one.
Additionally, *delete* and *update* may fragment the block, leading to wasted/unused space.
Sequential reading/writing is very efficient.
Usually, in DBMS,it's coupled with unclustered indices to **support** search and sort operations since **heap file** alone doesn't support sorting etc.

Ordered sequential structure

Tuples are written according to the value of a given **sort key**.
It supports **sort**(and thus group-by),**search** and **join** operations
However, problems arise if we want to preserve the sort order when inserting new records.
This problem is solved by leaving a percentage of free space in each block

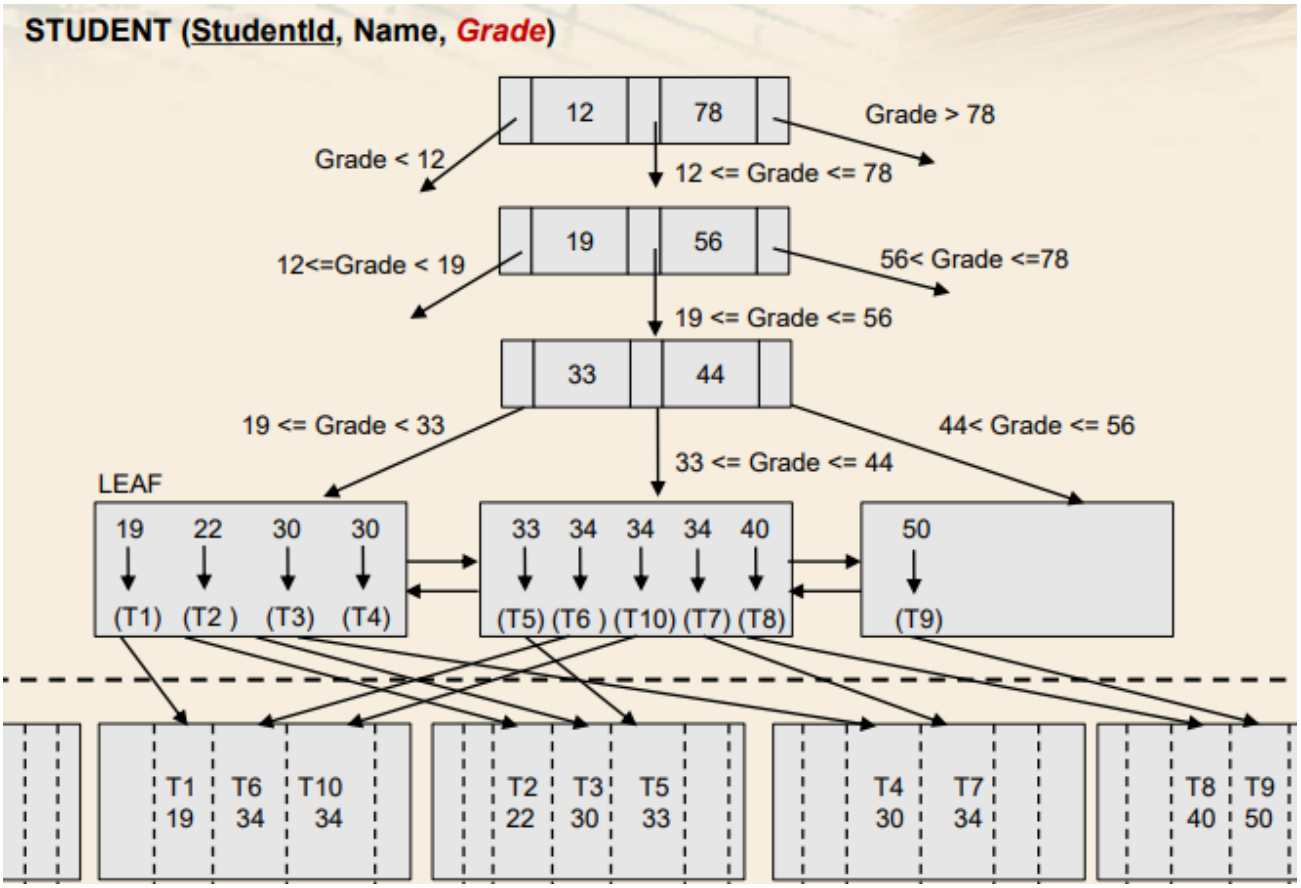
Another solution is the using of an **overflow file:** an additional block created for the sole purpose of storing new tuples that don't fit in the correct block

This structure is often supported by a clustered **B+-tree** and is used by DBMS to store intermediate result.

(B-)Tree Structures

Provide a "direct" access to data based on the value of a **key** field(s).
It doesn't costrain the physical position of tuples and it's the most used in DBMS.
It's structured as follows:

- **one** root node
- **many** intermediate nodes
- **many** children for each node
- **leaf** nodes provice access to actual data
  Leaf nodes can either have direct access to the data(**clustered**) or having pointers to their actual location inside the main memory(**unclustered**).
  In a **B+-tree**, leaf nodes are double-linked together with their successors.
  If the index covers all the information of a given query, it's called *covering index*.
  Only 1 **clustered** index can exist at a time and it's usually created on the **primary key**.
  They(clustered index) may cause additional overhead during deletion/insertion.

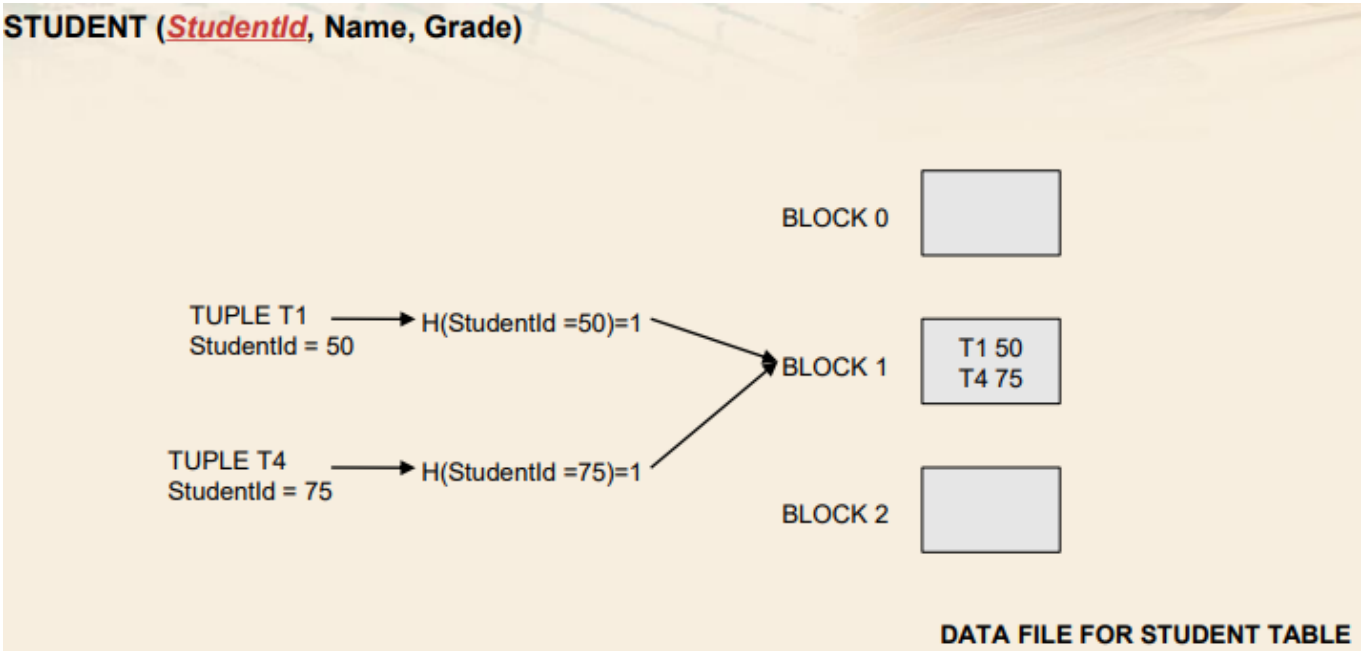


Example of an unclustered B+tree index on the grade attribute.

Hash indexes

It guarantees direct and efficient access to data based on the value of a key field.
Supposing the hash structure has **B** blocks, the **hash function**, applied to the key field value of a record, returns a value between 0 and **B-1.**
It's more efficient than a Btree index in case of an equality predicate. It's not true for a range

predicate as it will have to apply the hash function and access the block for every possible value in the range.
It can be either clustered or unclustered(see above).



Example of clustered hash index.

Bitmap indexes

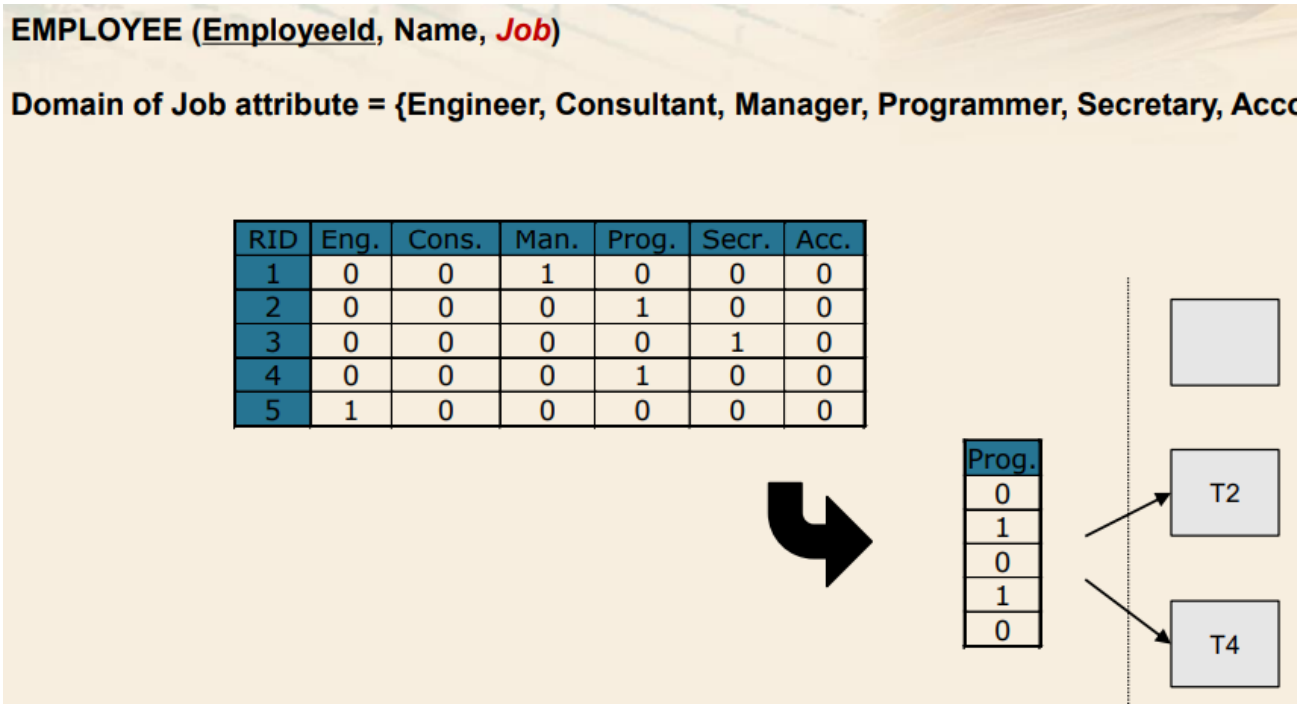
It guarantees direct and efficient access to data based on the value of a key field and it's based on a *bit matrix*. It can support *group by* operations.

Bit matrix

There are as many rows as the number of records.
There are as many columns as the number of values the indexed attribute can assume.
Position(i, j) is:

- 1 if tuple i takes value j

- 0 otherwise.



Example of bitmap index on attribute Job. The columns represent the values the attribute Job can assume. E.g. Tuple with RID=1 has Man. set to 1 as in that tuple, job assumes the value of Manager.

This kind of index is very efficient for boolean expressions of predicates.
It can't be used for continous attributes(floating numbers for example).