Curs 3: Indexing -- Part 1

Many queries refer to a small set of the database records. For example, if we want to sum up the number of products a certain client has purchased, given his id, it will be extremely inefficient to search all the relation *purchase_product,* and to count only the products ordered by a single client.

An index is an additional structure meant to optimize queries. Intuitively it works in the same way a textbook index works. Searching the index instead of searching the entire relation (table) reduces the processing time of the request just as finding a key word at the end of the book and jump exactly to the desired page or pages reduces the effort of reding the entire book.

When creating indexes, we should consider the memory we will allocate for additional data structures and the cost of updating those structures to reflect all insert/update/delete operations performed on the database tables, see figure 1. Suppose we index the lines in table *products* by the search key *product_name*. And we change the name of a product (example "*phone"* => "*smart_phone"*). Then the entry in the index corresponding to the old value ("*phone"*) of the column *product_name* must be deleted and the new value ("*smart_phone*") of the search key *product_name* must be inserted. This will cause the DML update operation to be slower. Similar synchronization must be made in case of DML insert or delete operations.

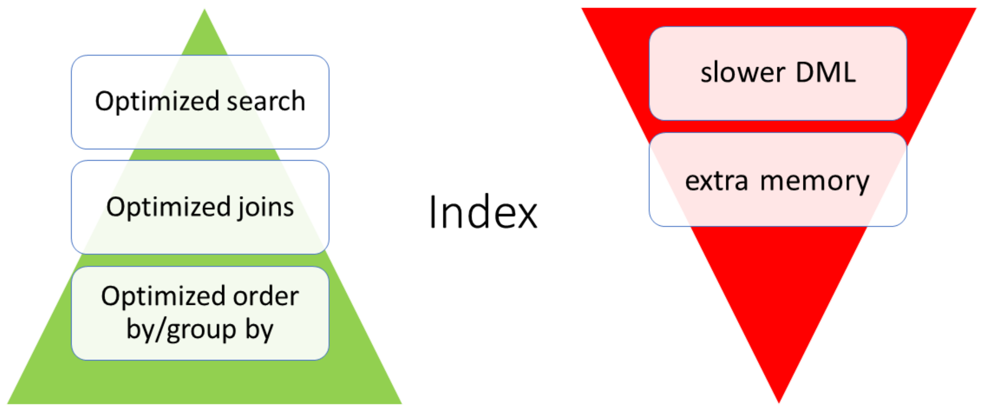


Figure 1: Advantages and cost of adding an index.

Ordered indexes.

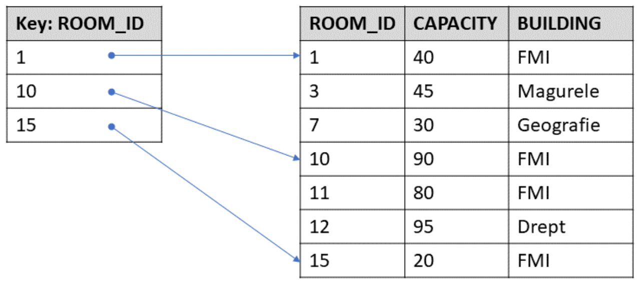
An ordered index is associated with a search key. A sorted index stores the values of the search key in order and maps search key values to database records. A ***cluster index***, also called ***primary index*** defines the sequential order of the lines in the database files. We may store several ordered indexes for a table but only a single index, called *clustered index*, may define the order in which we store the lines of the table. That is because the order in which the lines are memorized is fixed. Indexes defined on other key, different form the key the primary index is defined, are called secondary indexes. For example, we may define a cluster index on the table *products*, based on the search key *product_id*, and we may also store two secondary indexes for search keys: *product_name*, *price*.

Sparse or dense indexes.

An index entry consists in *a value of the search key* and *pointers* to one or more records with the same value for the search key. A pointer locates the record in the block of data on the disk, indicating the block and the offset in the block.

A ***dense index*** has entries for all values of the search key. A *cluster dense index* stores a pointer to the first record in the block with a given value of the search key. If the search key is also the primary key, there will be only one record with a specific value of the search key. If the clustered index is defined on a non-primary search key, then all the records with a given value of the search key will be memorized sequentially, so it's sufficient to find the first record in the block with a given value.

A ***sparse index*** has entries only for several values of the search key. The procedure of finding a key in a sparse index finds the entry with the largest search key value that is less than or equal to the key value that we search. Then, it sequentially inspects the key following in order until it locates the target key, see figure 2. If we analyse the procedure of finding a key in a sparse index, it is clear that only clustering indexes can be *sparse indexes*. In a non-cluster sparse index we won't be able to find all the keys.



Key: ROOM_ID
1
10
15

ROOM_ID	CAPACITY	BUILDING
1	40	FMI
3	45	Magurele
7	30	Geografie
10	90	FMI
11	80	FMI
12	95	Drept
15	20	FMI

Figure 2: If a sparse index is created on the search key ROOM_ID, the procedure find(12) first locates the block where room_id = 10 is stored, then it searches blocks with lines storing room_id = 11 until it finds room_id = 12.

Sparse indexes are conceptually the idea behind the implementation that we find for indices in the majority of relational SGBD, as tree structures. To optimize indices, we actually build indices of indices, or in other words multi-level indices in a tree-like structure, see the following section B-trees.

B^+-trees

B^+-trees are balanced trees, i.e. every path from the root to the leaf nodes has the same length. Every non-leaf node has between $n/2$ and n children. Only the root may have less then $n/2$ children, but at least 2. For a particular tree, the value n is fixed.

Leaf nodes.

A leaf nodes store $n-1$ keys and n pointers, see figure 3. Each pointer$_i$ for $1 \leq i \leq n-1$ is the address of the record holding the key$_i$, for instance in Oracle this is the row ***rowid***. The last pointer, pointer$_n$ is a link to the next leaf node storing the keys following in order after key$_n$. Chaining the leaf nodes allow efficient sequential processing of the search-key values placed in linear order, to speed up searching values in a given interval.

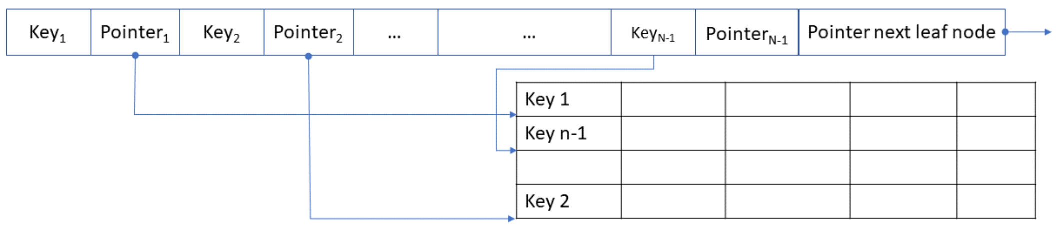


Figure 3: The structure of a leaf node in a B$^+$-Tree.

Non-leaf nodes

Non-leaf nodes on each level are similar to a sparse index. Each node-leaf node stores values delimitating intervals for search-key values, and pointers to other non-leaf or leaf nodes. Just like leaf nodes, an internal nodes stores $m-1$ keys and m pointers, with m between $n/2$ and n.
The first pointer points to the subtree storing keys with value less than key$_1$.
Each pointer$_i$ for $2 \leq i \leq m-1$ points to the subtree containing the keys with values greater than or equal to key$_{i-1}$ and less than key$_i$. The last pointer points to the values greater than or equal to key$_{m-1}$.

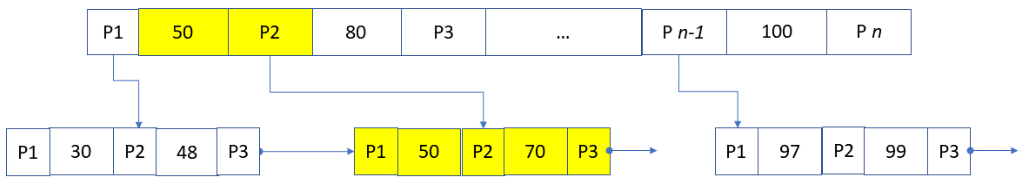


Figure 4: The structure of a non-leaf node in a B$^+$-Tree. P1 points to values less than 50. P2 on the internal node points to the interval [50, 80). Pn points to values greater than 100.

Search in B$^+$-trees

Intuitively we search a record with a given value of the search key *v*, starting from the root of the index. The procedures search from the root to the leaf where the searched key *v* is located, following the pointers of the multilevel index. Each time we go down one level we search in a smaller interval containing *v*.

We will consider the simplified case where three are no duplicates for the search key. If the search key contains duplicates the B+-trees implementing the index is build on a combinations of keys, for instance *search key* + *primary key*.

If at a given step the procedure inspects the value in node A, then it follows a pointer of node A.

If *i* is the smallest number such that $v < \text{key}_i$ we follow $\textbf{pointer}_{\boldsymbol{i}}$

If there is an number *i* such that $v = \text{key}_i$ we follow $\textbf{pointer}_{\boldsymbol{i+1}}$

If for all numbers *i* $v > \text{key}_i$ we follow the last pointer register in node A.

An example is illustrated in figure 5.

It is possible that when reaching a leaf node, the searched key value v is not found. In this case the procedure end returning no records from the database.

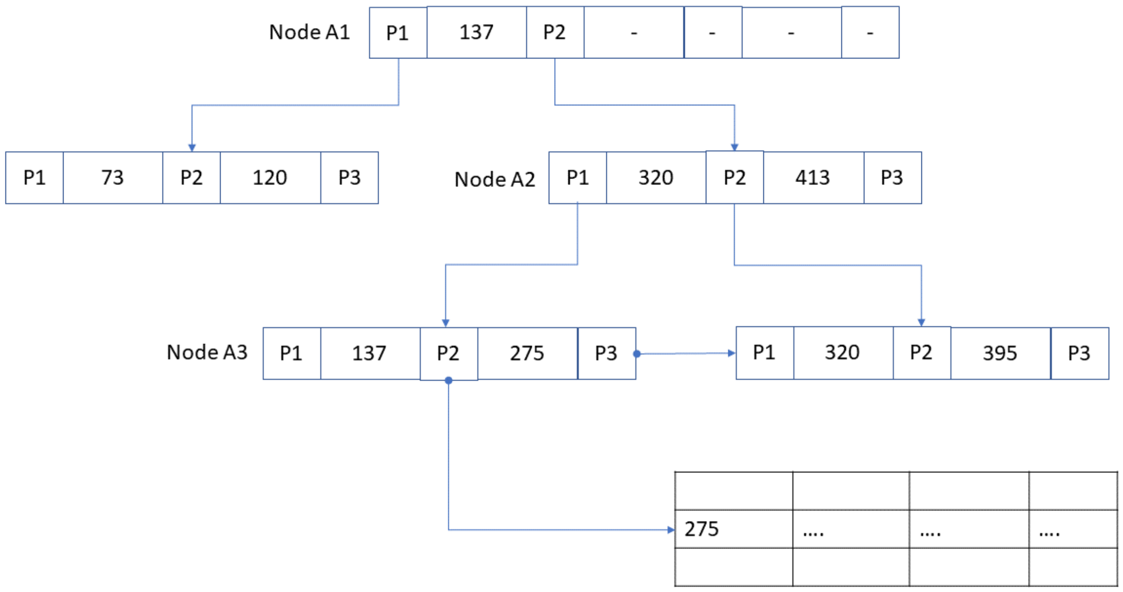


Figure 5: Search key 275. 275 is greater than 137 so we move from root following P2 of node A1(root). 275 is less then 320 so we move from node A2 to a leaf node following P1 from node A2. Finally on the leaf node A3 we find 275 and locate its address using the pointer P2. If we search for values greater than 275 we should follow pointer P3 from leaf node A3. Notice that only two pointers are stored in the root. In this example n is 4, so an internal node may store between 1 and 3 values, thus having between 2 and 4 children.

Assignment: (first 5 solutions will be graded) Give an example of a search procedure for an interval using a B$^+$-trees with n=4**.**

Bibliography

[1] https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16

[2] https://docs.aws.amazon.com/dms/latest/oracle-to-aurora-postgresql-migration-playbook/chap-oracle-aurora-pg.tables.iot.html

[3] Abraham Silberschatz, Henry Korth and S. Sudarshan - Database System Concepts chapter 7.