# INFO-H-417: Database Systems Architecture – Lab 2

Lecturer: Stijn Vansummeren
Teaching-Assistant: Stefan Eppe
http://cs.ulb.ac.be/public/teaching/infoh417

Academic Year 2014–2015

---

## Setup

In this second lab, we focus on the physical layout of B-Tree and on performance improvement that can be achieved by using a B-Tree index to access a relation. In this first part we first setup our database prototype. Then, to establish a baseline for comparison, we check the number of blocks that must be read from the disk when querying a relation through a simple scan of all records.

1. In the `IOTest` class, add calls to the `printStatistics` function before and after scanning the `Foo` relation. How many blocks are read when `NUMRECORDS` is 50, 1000, and 10000?

   We see here that to scan a relation (that is, reading each record of each block of the relation), for:

   - 50 records we need to read 1 blocks
   - 1000 records we need to read 19 blocks
   - 10000 records we need to read 184 blocks

   If we want to find out whether a record such that $A = 8$ and $B = 5$ exists, scanning the relation might not be the most efficient way.

## Index Initialization

We will now add a B-Tree Index to our relation in the database, and describe the layout of the B-Tree on disk.

1. Add a B-Tree index (using `BTIndexFactory`) on the `Foo` relation, in the `createDatabase` function. The B-Tree will use $(A, B)$ as its schema (and will hence be dense for the relation).

   The code should read:

   ```
   foo.createIndex(FOO_SCHEMA, new BTIndexFactory());
   ```

2. **Pen and paper exercise** Consider the leaves of the B-Tree. Is there an ordering of the keys inside these leaves? How many keys do you expect to find in the leaves in total, if the number of records in the relation is 10000?

   Since the index is dense, each record of the relation corresponds to a key in the index. Furthermore, all the keys in a BTree are stored at the leaf level. Therefore, there are 10000 keys stored in leaf blocks.

Furthermore, a BTree guarantees a lexicographic ordering of the keys, according to their schema. That is, with our schema $(A_1, B_1)$ the key $k_1 = (a_1, b_1)$ is smaller that the key $k_2 = (a_2, b_2)$ if either $a_1 < a_2$, or $a_1 = a_2 \wedge b_1 < b_2$.

Note that the keys are not only ordered inside a single leaf block, but also accross all the leaf blocks. That is, remember that the leaf blocks of the BTree are chained together as a linked list. If the block $b_1$ occurs before the block $b_2$ in the list, then all keys $k_1$ in the block $b_1$ will be smaller than all the keys $k_2$ in the block $b_2$.

3. **Pen and paper exercise** In our B-Tree, inner nodes have a header of 4 bytes, and each (key, block address) pair is the size of the key plus 4 bytes. The leaf nodes also use 4 bytes for their header, but stores each (key, record address) on the size of the key + 8 bytes. How many leaf blocks do we need to store 10000 records? What is the height of the B-Tree in this circumstance?

   Note that the size of the blocks can be found in the implementation: a block is 512 bytes wide.

   We first calculate the capacity of inner and of leaf blocks (i.e. the number of (key, address) pairs they can contain). In both cases, the space that we can use inside a block is 512 − header size = 508. We also know that each key is 8 bytes wide (the size of two 4-bytes integer values, A and B). Therefore, a single (key, block address) pair inside an inner block will be 12 bytes long, while a single (key, record address) pair inside a leaf block will be 16 bytes long. We can hence conclude that inner blocks have a capacity of $\lfloor 508/12 \rfloor = 42$ slots, while leaf blocks have a capacity of $\lfloor 508/16 \rfloor = 31$ slots.

   Note that, while there are 42 slots in an inner block, the header of an inner block contains an additional pointer. Hence, an inner block is made of 42 keys and 43 pointers to other blocks. That is, each inner block has a *branching factor* of at most 43 inside the tree.

   From the capacity of each kind of block, we can deduce that we need $\lceil 10000/31 \rceil = 323$ leaf blocks to store the keys. Furthermore, we need $\lceil \log_{43}(323) \rceil = 2$ levels of inner blocks to access these leaves. In particular, we will need $\lceil 323/43 \rceil = 8$ inner blocks pointing to leaf blocks, and $\lceil 8/43 \rceil = 1$ block (the root of the BTree) to point to these inner blocks. That is, the height of our BTree is 3.

   **Note**. In our implementation and because of the automatic splitting of full index blocks into two half-sized ones, we will actually end up with leaf blocks with approximatively 15 record references, and therefore with about $\lceil 10000/15 \rceil = 667$ leaf blocks. This also applies to the inner blocks: each inner block will point to approximatively 22 blocks, rather than 43. We will thus have about $\lceil 667/21 \rceil = 32$ inner blocks pointing at the leaf blocks at the intermediate level (this is about $2 \times 2 = 4$ times more than our first estimation). At the next level one could expect these 32 inner block references to be split into two higher level inner blocks. This, however, does not occur. Indeed, the first (root) block never gets full (only 32 out of the 43 available slots are used), therefore it is never split and we will have it pointing to 32 inner blocks. The height of our BTree thus remains 3.

## Querying the Index

In this second part, we will use the index to answer a few queries. We will also improve the implementation by chaining leaf blocks, to enable range queries (e.g. finding the records such that $A = 8$ and $B >= 5$). When statistics are to be computed, we will only consider the situation where 10000 blocks are loaded.

1. Using the index, fetch the record for which $A = 5$ and $B = 2$. Note that since keys inside the B-Tree are represented as `Record` objects, the `createRecord` function can be used to create the query key for the index.

   Check the number of disk reads and of pin requests for this operation. Explain this result by means of the result from exercise 3 of part I (index initialization).

   Compare with the disk reads and of pin requests when scanning the relation.

We get 4 blocks read from the disk, and 5 pin requests for looking up the record from the B-Tree. In comparaison we needed to read 184 blocks from the disk (and a similar number of pin requests) to scan the whole relation.

These 4 blocks are the following:

- 1 block keeps track of the root of the BTree. The rationale is that when many keys are inserted into the index, at some point each leaf block and each inner block (and in particular the root itself) will be filled to its capacity. The next key insertion will result in the root being split into two blocks, and a new inner block will be created to act as the root. However, to load the index from disk, we need to be able to refer to a block address that remains constant. Therefore, we use one block that acts as a descriptor of the BTree; in particular, this block keeps track of the root of the BTree.
- 2 inner blocks of the BTree.
- 1 leaf block of the BTree.

The additional pin request is due to some implementation details: when the iterator is returned, it pins the leaf of the BTree from which the iteration starts, to ensure that it will stay available.

2. Describe how the `BTIndexIterator` class uses the chaining of leaf blocks to provide ranged queries.

   The iterator returns each key from a block in turn. When it has returned the last key of its currently loaded block, the iterator loads the next block from the leaves chain and uses the first key of that block the next time it returns a key.

3. Looking at the code of the `BTIndexBlockLeaf` class, determine which operations need to be aware of block chaining. Update the code accordingly.

   Two accessors are provided to get/set the next block address. These need to be implemented by accessing the `header` of the leaf block. These accessors are called when a leaf block is split (in the `split` function) to maintain the chain of leaf blocks:

   - Set the next pointer of the newly created block to our current next pointer:

     ```
     sibling.setNextPtr(this.getNextPtr());
     ```

   - Set the next pointer of our block to the address of the newly created block.

     ```
     this.setNextPtr(sibling.getBlockAddress());
     ```

4. Using the index, fetch the records for which $A = 8$ and $B >= 5$. How does this compare with scanning the relation?

   This time, we need to lookup the key $(A = 8, B = 5)$ in the index. This gives us an iterator that we can use to scan over all the records that are greater or equal than the lookup key. Therefore, we can print all the keys up to the first keys for which the $A$-value is greater than 8.

   Since the B-values range from 0 to 999, and since (in our implementation) a leaf block only contains 15 keys on average, we can see that we will need to access about $\lceil 994/15 \rceil = 67$ blocks to perform this operation. This is less than the cost of scanning the whole relation.

## Querying the index (continued)

In this section, we try a variety of query operations to understand the limits of the B-Tree index. For the purpose of this exercise, you can assume that the $B$-values range from 0 to 1000, while the $A$-values range from 0 to 9

1. Explain how you can perform the following queries on the B-Tree.

   - Find the records such that $A = 2$ and $B >= 4$.

     Similar to the last exercise of the previous section.

   - Find the records such that $A < 3$.

     We can start from a lower bound of the first record, i.e. $(A = 0, B = 0)$, and scan all the records from the BTree until $A = 3$. As a rough estimation, we will need to access a $3 * 999/15 \approx 200$ blocks, which is still cheaper that reading the whole relation.

   - Find the records such that $A = 3$ and $B < 4$.

     We can start from a lower bound of the $A = 3$ records, i.e. $(A = 3, B = 0)$, and scan all the records from the BTree until $B = 4$. As a rough estimation, we will need to access 1 block.

   - Find the records such that $B = 3$.

     Here, we essentially need to start scanning from the key $(A = 0, B = 3)$, and we can stop at $(A = 9, B = 3)$. Therefore, we need to read almost all the leafs of the BTree (667 blocks) mostly to retrieve spurious results that we will need to discard (e.g. all keys where $4 \leq B \leq 999$).

     Since this BTree will not help us processing the query, we can try to find alternative ways of processing this query efficiently. As we have seen in the first part of the lab, scanning the whole relation is more efficient (less disk blocks need to be read). Another BTree over the schema $(B, A)$ would also be more appropriate (why?).

   - Find the records such that $B < 4$.

     This is essentially the same situation as for $B = 3$.

   - Find the records such that $A > 3$ and $B > 7$.

     For this query, most in the relation will be part of the result: since for each value of $A$, we have all values of $B$ from 0 to 1000 in our database, almost 7/10 percent of the relation must be returned. While our index can easily help identifying all records such that $A > 3$, we will need to read approximately $667 * 7/10 = 467$ leaf blocks of the BTree. Therefore, it will be more advantageous to read the 184 blocks of the relation directly.

     Fundamentally, this is due to the fact that our BTree maintains more bookkeeping information per record than the relation itself. For instance, in the BTree, each key is associated with its position in the original relation.

2. **Supplemental** Write the code implementing the previous queries, and compare the performance with scanning the relation.