CHAPTER 3

EXAMPLES- CHAPTER 3

3.1 Physical Design

Example 3.1.1. Assume we have the relation `Employee` with $r = 1103$ tuples, each one corresponding to a fixed-length record. Moreover, each record has the following fields:

- `Name` (30 bytes),
- `SSN` (10 bytes), and
- `ADDRESS` (60 bytes).

Given that the block size $B = 512$ bytes, compute the number of blocks we need to fit all the tuples.

A tuple occupies

$$R = 30 + 10 + 60 = 100$$

bytes. So, the blocking factor

$$bfr = \text{floor}(B/R) = 5.$$

This means that we can store at most 5 tuples per a block. So, to store all $r = 1103$ tuples, we need

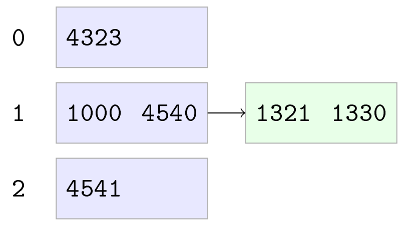
$$b = \text{ceil}(r/bfr) = 221$$

blocks.

Example 3.1.2. Assume we have the relation `Employee` with $r = 6$ tuples with $bfr = 2$. Compute the number of block accesses we need based on different data representations to execute the following query.

```
SELECT   *
FROM     Employee
WHERE    SSN = x;
```

The following is the hash representation of the file.



- First, we consider the hash file representation. We assume that each bucket has equal probability.
  - If the hashed value is 0, then we need to access 1 block in both the best, the average and the worst case.
  - If the hashed value is 1, then we need to access 2 blocks in the worst case and 1 block in the worst case. In the average case, the number of block accesses is
  $$\frac{1+2}{2} = 1.5$$
  - If the hashed value is 2, then we need to access 1 block in both the best, the average and the worst case.

  So, the expected block access in the worst case is
  $$\frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 1 = \frac{4}{3}.$$
  In the best case, the expected block access is
  $$\frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 1 = 1.$$
  The expected block access in the average case is
  $$\frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 1.5 + \frac{1}{3} \cdot 1 = 1.17$$
  block accesses.
- If we use the heap file structure, we need 3 blocks to store the 6 tuples. So, in the worst case, we have 3 block accesses. In the best case, we just need 1 block access. The expected block access for heap representation in the average case is
  $$\frac{1+3}{2} = 2$$
  block accesses.
- In the sequential file structure, we also have 3 blocks. In the average and the worst case, we have
  $$\log_2 3 \approx 1.58$$
  block accesses. In the best case, we just need 1 block access.

Example 3.1.3. Assume that we have hash, heap and sorted file where:

- we have 3 blocks in heap and sorted file;
- we have 4 blocks in the hash file- 3 main blocks and 1 overflown block.

Also, let k be the hash attribute, and let $p\%$ be the proportion of the queries that involve the attribute k. Find the best data structure (in the worst case) to store the values with respect to the ratio p.

- If the file is a heap we would need 3 block accesses in the worst case. The attribute k does not matter because the search is always linear.
- Next, assume that the file is sorted with respect to k. If the search uses the attribute k, then we can use a binary search algorithm to get the result in $O(\log_2 b)$ block accesses. Instead, if it does not use the attribute k, then we have to use the linear search in $O(b)$ block accesses. So, the expected number of block accesses in the worst case is

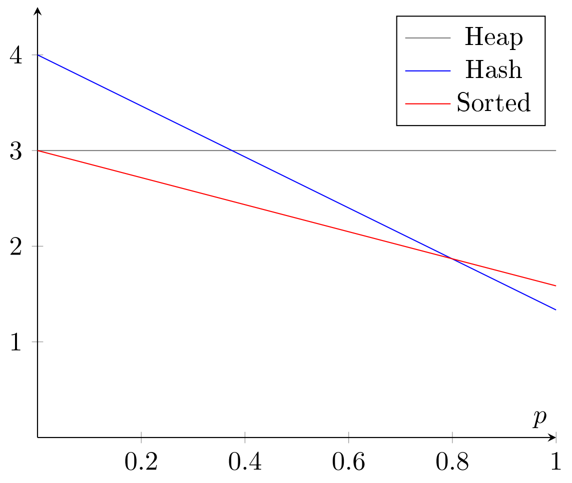
$$p \cdot (\log_2 3) + (1-p) \cdot 3 = 3 + p \cdot (\log_2 3 - 3).$$

- Now, if we use hash representation, we have

$$p \cdot (\tfrac{1}{3} \cdot 1 + \tfrac{1}{3} \cdot 2 + \tfrac{1}{3} \cdot 1) + (1-p) \cdot 3 = 4 - \tfrac{8}{3}p$$

  block accesses in the worst case.

We can plot these lines and see which option is better depending on the value of p.



So, if $p < 0.8$, we should use the sorted representation. Instead, if $p > 0.8$, we should use the hash representation.

3.2 Indexing Methodology

Example 3.2.1. Assume we have the `EMPLOYEE` table with key attribute `SSN`.

- We have $r = 300\ 000$ records.
- A record takes $R = 100$ bytes.
- The block size is $B = 4\ 096$ bytes.
- The attribute `SSN` occupies 9 bytes.
- A pointer occupies 6 bytes.
- The data files are sorted with respect to `SSN`.

We want to execute the query.

```
SELECT   *
FROM     EMPLOYEE
WHERE    SSN = k;
```

Compute the number of block accesses we need to execute this query using a linear search, binary search and a primary index.

The blocking factor is

$$bfr = \text{floor}(B/R) = \text{floor}(4096/100) = 40.$$

So, the number of data blocks we need is

$$b = \text{ceil}(r/bfr) = 7500.$$

- A linear search takes on average

  $$b/2 = 3750$$

  block accesses.
- A binary search takes on average

  $$\log_2 b \approx 11.9$$

  block accesses.
- The index entry has components: `(SSN, Pointer)`. We know that we need 9 bytes for SSN and 6 for the pointer, so we need 15 bytes for each index. Therefore, the indexing blocking factor is

  $$ibfr = \text{floor}(4096/15) = 273.$$

  We have 7 500 data blocks, so 7 500 index tuples. So, we need

  $$ib = \text{ceil}(7500/273) = 28$$

  blocks to store all the index files.

  Next, we compute the number of block accesses with the primary index.

- We first search for the SSN in the index. Since the primary index is sorted with respect to the index, it takes about

$$\log_2 28 \approx 4.8$$

block accesses to find it.
- After finding the index block, we need to load the data block. This takes 1 block access.

So, we have 5.8 block accesses to find the relevant tuple.

Example 3.2.2. Assume we have the `EMPLOYEE` table with non-key attribute `DNO`.

- We have $r = 300\ 000$ records.
- A record takes $R = 100$ bytes.
- The block size is $B = 4\ 096$ bytes.
- The attribute `DNO` occupies 9 bytes.
- A pointer occupies 6 bytes.
- The data files are sorted with respect to `DNO`.
- There are 10 departments.
- The `DNO` values are uniformly distributed over the tuples, i.e. there are 30 000 employees in each department.

We want to execute the query

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO = x;
```

Compute the number of block accesses we need to execute this query using a linear search and a clustering index.

The blocking factor is

$$bfr = \text{floor}(4096/100) = 40.$$

So, the number of blocks we need is

$$b = \text{ceil}(r/bfr) = 7\ 500.$$

- We first consider a linear scan with an exiting feature- we stop as soon as we have found all the tuples. Based on the uniformity assumption, each department has 750 blocks. If we have `DNO = 1`, then we retrieve the first 750 blocks and then stop. If `DNO = 2`, then we retrieve the first two 750 blocks and then stop. Assuming that it is equally likely for the `DNO` to take all these values and that the value `x` is a valid `DNO`, the expected block accesses is:

$$\frac{1}{10} \cdot 750 + \frac{1}{10}(750 \cdot 2) + \cdots + \frac{1}{10}(750 \cdot 10) = 4125.$$

- Next, we consider a clustering index on DNO. The indexing blocking factor is
$$\mathit{ibfr} = \text{floor}(4096/15) = 273.$$
We have 10 data blocks, so we need
$$\mathit{ib} = \text{ceil}(10/273) = 1$$
block to store all the index files. When executing the query, we find the block pointer corresponding to `DNO = x` in 1 block access. Then, we load the 750 contiguous blocks to find all the relevant tuples. This requires 751 block accesses.

Example 3.2.3. Assume we have the `EMPLOYEE` table with non-ordering key attribute `SSN`.

- We have $r = 300\ 000$ records.
- A record takes $R = 100$ bytes.
- The block size is $B = 4\ 096$ bytes.
- The attribute `SSN` occupies 9 bytes.
- A pointer occupies 6 bytes.

We want to run the query

```
SELECT   *
FROM     EMPLOYEE
WHERE    SSN = x;
```

Compute the number of block accesses we need using a linear scan and a secondary index on a non-ordering key attribute `SSN`.

The blocking factor is

$$\mathit{bfr} = \text{floor}(B/R) = 40$$

records per block. So, we need

$$b = \text{ceil}(r/\mathit{bfr}) = 7500$$

blocks to store all the data.

- First, we consider a linear search. Since the attribute `SSN` is key, this would take on average
$$b/2 = 3750$$
block accesses.

- Next, we consider using the secondary index. An index entry takes 15 bytes. So, the index blocking factor is

  $$\mathit{ibfr} = \text{floor}(4096/15) = 273.$$

  Since we are using a dense index, we have 300 000 index entries. So, we need

  $$\mathit{ib} = \text{ceil}(300\ 000/273) = 1099$$

  blocks.

  To search for a single employee, we first search within the index. Since the index file is sorted, we can find it in

  $$\log_2(\mathit{ib}) \approx 10.1$$

  block accesses. Next, we load the unique block pointed by the index entry. So, we require 11.1 block accesses to find the relevant tuple.

Example 3.2.4. Now, assume that we have the `EMPLOYEE` table on the non-ordering key attribute `SSN`.

- We have $r = 300\ 000$ records.
- A record takes $R = 100$ bytes.
- The block size is $B = 4\ 096$ bytes.
- The attribute `SSN` occupies 9 bytes.
- A pointer occupies 6 bytes.

What level of multilevel index should we build? Use this value to predict the number of block accesses when using the index to find a tuple given the `SSN` value.

The blocking factor is

$$\mathit{bfr} = \text{floor}(B/R) = 40$$

records per block. So, we need

$$b = \text{ceil}(r/\mathit{bfr}) = 7500$$

blocks to store all the data.
An index entry has tuples with attributes `SSN` and a pointer. So, this takes 15 bytes. In that case, the index blocking factor is

$$m = \text{floor}(4096/15) = 273.$$

Therefore, at L1 index, we have

$$b_1 = \text{ceil}(300\ 000/m) = 1099$$

index blocks, since the index is dense. So, at L2, we have 1 099 index entries, and we require

$$b_2 = \text{ceil}(b_1/m) = 5$$

blocks to store them. Furthermore, at level-3, we have 5 index entries, and we just require

$$b_3 = \text{ceil}(b_2/m) = 1$$

block for it. Since we just need 1 block here, we build a 3-level index. Now, to find the unique employee we are searching for, we just need 4 block accesses- 3 going from one index level to a lower level, and the last one to get the data block.

3.3 B and B+ Trees

Example 3.3.1. Assume that we have created a 3-level B tree of order $p = 23$ that is 69% full. Compute the number of data and tree pointers we can store in the tree. Moreover, assume the following.

- The block size is $B = 512$ bytes.
- A data pointer takes $Q = 7$ bytes.
- A tree pointer takes $P = 6$ bytes.
- A key value takes $V = 9$ bytes.

Using this information, compute the number of blocks we have in total.

In each block, we store

$$69\% \times 23 \approx 16$$

tree pointers, and 15 data pointers. In that case, the root block contains 16 tree pointers and 15 data pointers. At level 1, we have 16 blocks, each of which contains 16 tree pointers and 15 data pointers. At level 2, we have 16^2 blocks, each of which contains 16 tree pointers and 15 data pointers. At level 3, we have 16^3 blocks, each of which contains no tree pointers and 15 data pointers. This is summarised in the table below.

level	data pointers	tree pointers
0	15	16
1	$16 \cdot 15$	16^2
2	$16^2 \cdot 15$	16^3
3	$16^3 \cdot 15$	0

In total, we can store

$$15 + 16 \cdot 15 + 16^2 \cdot 15 + 16^3 \cdot 15 = 65\ 535$$

data pointers and

$$16 + 16^2 + 16^3 = 4368$$

tree pointers.
A block has size

$$15 \cdot (7 + 9) + 16 \cdot 6 = 336 \text{ bytes.}$$

So, the blocking factor

$$\mathit{bfr} = \text{floor}(512/336) = 1.$$

In that case, we need

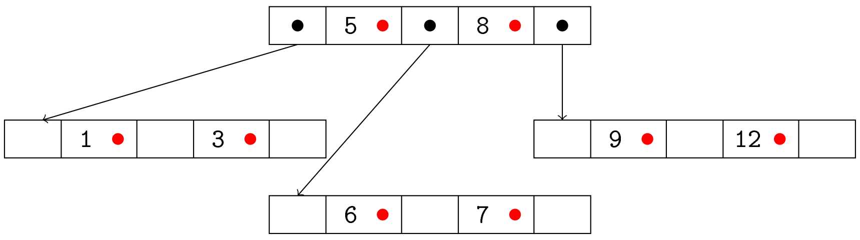
$$1 + 16 + 16^2 + 16^3 = 4369$$

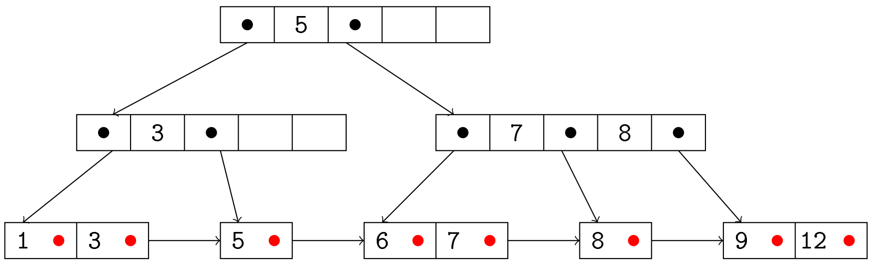
blocks to store the 3-level B tree.

Example 3.3.2. Assume we have the following query.

```sql
SELECT  *
FROM    EMPLOYEE
WHERE   SSN >= 3 AND SSN <= 10;
```

We are also given the following B and B+ Trees.





Compute the number of block accesses we need in order to execute this query for both a B Tree and a B+ Tree.

- First, we use the B Tree. We access the root block and get the values `SSN = 5` and `SSN = 8`. Since they are both in the range, we need to access all the 3 branches.

  So, we need 4 block accesses within the B Tree, and 6 data block accesses.

- Next, we use the B+ Tree. We access the root block and get the value `SSN = 5`. We want to find the tuple satisfying `SSN = 3`, so we just need to take the left branch. Now, we take the left branch in level 1. Then, we have found the first block that we require. We can make use of the sorted structure of the leaf nodes to find all the other tuples that satisfy the given condition.

  So, we need 2 block accesses within the internal nodes of the B+ Tree, 5 block accesses within the leaf nodes of the B+ Tree, and 6 data block accesses.

Example 3.3.3. Construct a 3-level B+ Tree of order $p = 34$ and $p_L = 31$ that is 69% full. Also, compute the number of blocks we have in total.

We can fit

$$69\% \times 34 = 23$$

tree pointers in an internal node and

$$69\% \times 31 = 21$$

data pointers in a leaf node. The table below summarises the number of tree pointers and key values we can hold in the first 2 levels.

level	key values	tree pointers
0	22	23
1	$23 \cdot 22$	23^2
2	$23^2 \cdot 22$	23^3

So, we can fit 12 720 tree pointers. At level 3, we can fit $23^3 \cdot 21 = 255\ 507$ key values and data pointers. Assuming that one B+ block fits one node, we require

$$1 + 23 + 23^2 + 23^3 = 12\ 720$$

blocks.

Example 3.3.4. Assume that we are given the following query.

```sql
SELECT   *
FROM     EMPLOYEE
WHERE    SSN = k;
```

We are told the following:

- the size of a block $B = 512$ bytes;
- the number of tuples $r = 100\ 000$;
- the blocking factor $bfr = 10$ records/block;
- a pointer takes up 10 bytes;
- the indexing key takes 10 bytes.

Moreover, we have the following primary access paths:

- A secondary index over `SSN`.
- A B+ tree of order 25 over `SSN`.

Compute the number of block accesses we require to execute the query using a linear search, secondary index and the B+ Tree.

- First, we consider a linear search. Since there are 100 000 records and $bfr = 10$, we have 10 000 data blocks. On average, the linear search will require 5000 block accesses since `SSN` is a key attribute.
- Next, we consider a secondary index. Since the index is secondary, we need a dense index- every data record appears in the index, with a data pointer (10 bytes) and the key value (10 bytes). So, each record takes up 20 bytes. We know a block can store 512

bytes, so each block can hold

$$\text{floor}(512/20) = 25$$

tuples. In that case, we require

$$\text{ceil}(100\ 000/25) = 4\ 000$$

extra blocks to store the secondary index. Moreover, using the secondary index, we require

$$\log_2 4000 \approx 12$$

block accesses to find the right tuple on the secondary index file, and then a further data block access to retrieve the tuple. In total, the search requires 13 block accesses.

- Finally, we consider a B+ tree index of order 25. Assuming the nodes are completely full, each internal node can hold 24 key values and 25 tree pointers, and a leaf block can hold 24 key values and data pointers. A pointer and an indexing key take 10 bytes, so an internal node takes up

$$25 \cdot 10 + 24 \cdot 10 = 490$$

bytes. Moreover, a leaf node takes up

$$24 \cdot 10 + 24 \cdot 10 + 10 = 490$$

bytes. So, we can fit 1 internal/leaf node per block. Next, we compute the level of B+ Tree we need to store the 100 000 tuples:

level	key values	tree pointers
0	24	25
1	$25 \cdot 24$	25^2
2	$25^2 \cdot 24$	25^3

So, at level 3, we have

$$25^3 \cdot 24 = 375\ 000$$

data pointers/key values. Since we only have 100 000 tuples, we are only using up about 30% of the leaf nodes. Moreover, we need

$$1 + 25 + 25^2 + 25^3 = 16\ 276$$

extra blocks to store the B+ tree.

Nonetheless, we can find a tuple using a B+ tree with just 5 block accesses:

1. we access the root internal node,
2. we descend to the relevant level 1 internal node,
3. we descend to the relevant level 2 internal node,
4. we descend to the relevant level 3 leaf node, and
5. we retrieve the relevant data block.

Example 3.3.5. Assume that we have the query

```sql
SELECT   AVG(SALARY)
FROM     EMPLOYEE
WHERE    SSN >= L AND SSN <= U;
```

Here,

- there are $b = 1\ 250$ blocks,
- there are $n = 1\ 250$ employees,
- the blocking factor $\mathit{bfr} = 1$ employee/block,
- `SSN` takes up 10 bytes,
- a pointer takes up $P = 10$ bytes,
- the B+ tree is of order $p = 5$ and $p_L = 10$,
- the leaf nodes are full,
- the SSN values range from 1 to 1250.

Find the maximum ratio

$$\alpha = \frac{U - L}{1250}$$

of `SSN` values selected by the query such that searching using the B+ Tree is more efficient than a linear scan.

We know that an internal node can hold 4 key values and 5 data pointers. So, we need a 3-level B+ tree so that the leaf nodes can hold the 1 250 records. The table below shows how many entries are present in the internal nodes.

level	key values	tree pointers
0	4	5
1	$5 \cdot 4$	5^2
2	$5^2 \cdot 4$	5^3

So, at level 3, we precisely have

$$5^3 \cdot 10 = 1\ 250$$

key values/data pointers.
When we execute this range query over the B+ tree, we need 3 block accesses to descend from the root to the leaf node containing the data pointer for `SSN = L`. Each leaf node contains 10 blocks, so we approximately need to access

$$\frac{U - L}{10} = \frac{1250\alpha}{10} = 125\alpha$$

leaf blocks. Moreover, we need to access all $U - L = 1250\alpha$ data blocks. Overall, we need to access

$$3 + 125\alpha + 1250\alpha = 3 + 1375\alpha$$

blocks.
Using a linear search, we need to access all the 1250 blocks since the tuples are not sorted with respect to `SSN`. Therefore, for B+ trees to be useful, we require

$$3 + 1375\alpha < 1250 \implies \alpha < \frac{1247}{1375} \approx 0.906.$$

So, if the ratio is less than 90.6%, then we should use B+ tree. Otherwise, we should use linear searching.