CHAPTER 4

EXAMPLES- CHAPTER 4

4.1 Query Processing

Example 4.1.1. Consider the following query.

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   E.DNO = D.DNUMBER;
```

Assume we have n_E Employee blocks and n_D Department blocks. Moreover, the memory can store n_B blocks. Predict the cost of using a nested-loop join algorithm to execute the query, with `EMPLOYEE` in the outer loop and `DEPARTMENT` in the inner loop.

In memory, we require a block for reading the inner file `D` and a further block to write the join result. The other $n_B - 2$ blocks can be used to read the outer file `E`. This is the chunk size.
When we run the nested-loop algorithm, we load $n_B - 2$ blocks for the outer relation, and then load each block from the inner relation one by one and check whether there are any possible tuples we can output. So, the outer loop runs for

$$\frac{n_E}{n_B - 2},$$

and the inner loop is n_D. The total number of block accesses is therefore

$$n_E + \frac{n_E}{n_B - 2} \cdot n_D.$$

Example 4.1.2. Assume we have the following query.

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   D.MGR_SSN = E.SSN;
```

We have the following primary access paths:

- A B+ Tree on `Mgr_SSN` with level $x_D = 2$.
- A B+ Tree on `SSN` with level $x_E = 4$.

Moreover, the record `EMPLOYEE` has $r_E = 6\,000$ tuples in $n_E = 2\,000$ blocks, and the record `DEPARTMENT` has $r_D = 50$ tuples in $n_D = 10$ blocks. Compute the number of expected block accesses using an index-based nested-loop join algorithm.

- If we use index-based nested-loop on the B+ Tree for Department relation, we are searching the B+ tree to check whether a given `E.SSN` is also `D.Mgr_SSN`. However, since not every employee is

a manager, many of these searches fail. The probability of an employee being a manager is

$$50/6\ 000 = 0.83\%.$$

So, 99.16% of searching using the B+ Tree is meaningless. During the process, we load each employee block, and for each tuple, we traverse the B+ Tree to find whether the SSN corresponds to a manager. This requires

$$n_E + r_E \cdot (x_D + 1) = 2\ 000 + 6\ 000 \cdot (2 + 1) = 20\ 000,$$

block accesses.

- Instead, if we use the B+ tree for Employee relation, then we require
$$n_D + r_D \cdot (x_E + 1) = 10 + 50 \cdot 5 = 260$$
block accesses. The probability of a manager being an employee is 100%, so this approach is much more efficient.

Example 4.1.3. Assume we have the following query.

```
SELECT   *
FROM     EMPLOYEE E, DEPARTMENT D
WHERE    D.MGR_SSN = E.SSN;
```

The record `EMPLOYEE` has $r_E = 6\ 000$ tuples in $n_E = 2\ 000$ blocks, and the record `DEPARTMENT` has $r_D = 50$ tuples in $n_D = 10$ blocks. Compute the number of block accesses we need if we use the sort-merge-join algorithm when the two relations are sorted and when they are not sorted with respect to the joining attribute.

- If the two files are sorted, we load each block from the two relations precisely once, so this requires
$$n_E + n_D = 2010$$
block accesses.
- If both files are not sorted, we need to use external sorting algorithm to sort them. The external sorting process for the `Employee` relation requires
$$2n_E + 2n_E \log_2(n_E/n_B)$$
block accesses- the value n_E/n_B is the number of sub-files initially sorted, where n_B is the number of available blocks in memory. We might have to sort the `Department` relation requires
$$2n_D + 2n_D \log_2(n_D/n_B)$$
block accesses. In total, the sort-merge-join algorithm requires
$(n_E+n_D)+(2n_E+2n_E \log_2(n_E/n_B))+(2n_D+2n_D \log)2(n_D/n_B)) = 38\ 690$
block accesses if both the relations are not sorted.

Example 4.1.4. Assume we have the query

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   E.SSN = D.MGR_SSN;
```

We have $n_E = 100$ Employee blocks, $r_D = 100$ Department tuples in $n_D = 10$ blocks. In memory, we can store $n_B = 12$ blocks, and we have a 2-level index on `E.SSN`. Find the number of block accesses required using a nested-index join algorithm.

The algorithm runs as follows:

- For each department d:
  - Use the B+ Tree on `E.SSN` to find the Employee `d.MGR_SSN`.
  - Output the tuple `(d, e)`.

  So, we require

$$n_D + r_D(2+1) = 10 + 100 \cdot 3 = 310$$

  block accesses.

Example 4.1.5. Next, assume we have the query

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   E.SSN = D.MGR_SSN;
```

We have $n_E = 100$ Employee blocks, $r_D = 100$ Department tuples in $n_D = 10$ blocks. In memory, we can store $n_B = 12$ blocks, and we have a hash function on `D.MGR_SSN` that maps to 10 buckets. Find the number of block accesses required using a hash-join algorithm.

The algorithm runs as follows.

- Load all the Departments `d`. This fits in memory.
- For each Employee `e`,
  - Use the hash function to find the bucket for Department `d`.
  - Find the tuple `d` in the bucket.
  - Output the tuple `(d, e)`.

  So, we require

$$n_D + n_E = 10 + 100 = 110$$

  block accesses.

Example 4.1.6. Assume that we have the query

```
SELECT  *
FROM    EMPLOYEE E, EMPLOYEE S
WHERE   E.SUPER_SSN = S.SSN:
```

Assume we have $r_E = 10\ 000$ tuples in $n_E = 2\ 000$ blocks. Moreoer, assume that 10% of the employees are supervisors. We have a 5-level B+ Tree on `SSN` and 2-level B+ Tree on `Super_SSN`. Compute the number of expected block accesses using an index-based nested-loop join algorithm for both the B+ trees.

The index-based nested-loop join algorithm for either B+ Tree is the following.

- For each Employee `e`
- If `e.SUPER_SSN` is not null (i.e. the employee is not a supervisor)
  - use the B+ Tree to find Employee `s.SSN`
  - Output the tuple `(e, s)`.

Since 10% of the employees are supervisors, if we use a level-t B+ Tree, then we need to retrieve all the employee blocks, and then $t + 1$ block accesses for 90% of the non-supervisor employees. So, if we use the 5-level B+ Tree on `SSN`, then we require

$$n_E + 0.9 \cdot r_E(5 + 1) = 2000 + 0.9 \cdot 10000 \cdot (5 + 1) = 56000$$

block accesses. Instead, if we use the 2-level B+ Tree on `SUPER_SSN`, then we require

$$n_E + 0.9 \cdot r_E(2 + 1) = 2000 + 0.9 \cdot 10000 \cdot (2 + 1) = 29000$$

block accesses. The B+ Tree on `SUPER_SSN` requires fewer block accesses since it only indexes supervisors, while the B+ Tree on `SSN` indexes both supervisors and non-supervisors.

4.2 Selection selectivity

Example 4.2.1. Consider the following query.

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO = 5 AND SALARY = 40 000;
```

Assume that:

- the number of distinct values of salary NDV(`Salary`) = 100;
- the number of distinct values of department numbers NDV(`DNO`) = 10;
- there are $r = 1000$ employees that are evenly distributed among salaries and departments.

Compute the selection selectivity and the selection cardinality of the query.

The selection selectivity for the `Salary` attribute is

$$sl(\texttt{Salary}) = \frac{1}{100} = 0.01$$

Moreover, the selection selectivity for the DNO attribute is

$$sl(\texttt{DNO}) = \frac{1}{10} = 0.1.$$

So, under the assumption that salary is independent of the department, we find that the selection selectivity of the query Q is

$$sl(Q) = sl(\texttt{Salary}) \cdot sl(\texttt{DNO}) = 0.001.$$

In terms of the selection cardinality, this is 1 tuple.

Example 4.2.2. Consider the following query

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO = 5 OR SALARY = 40 000;
```

Assume that:

- the number of distinct values of salary NDV(`Salary`) = 100;
- the number of distinct values of department numbers NDV(`DNO`) = 10;
- there are $r = 1000$ employees that are evenly distributed among salaries and departments.

Compute the selection selectivity and the selection cardinality of the query.

The selection selectivity for the `Salary` attribute is

$$sl(\texttt{Salary}) = \frac{1}{100} = 0.01$$

Moreover, the selection selectivity for the DNO attribute is

$$sl(\texttt{DNO}) = \frac{1}{10} = 0.1.$$

So, under the assumption that salary is independent of the department, we find that the selection selectivity of the query Q is

$$sl(Q) = sl(\texttt{Salary}) + sl(\texttt{DNO}) - sl(\texttt{Salary}) \cdot sl(\texttt{DNO}) = 0.109.$$

In terms of the selection cardinality, this is 109 tuples.

Example 4.2.3. We are given the following query.

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO = 5 AND SALARY = 30000 AND EXP = 0;
```

Assume that:

- there are $r = 10\ 000$ records in $b = 2000$ blocks.
- the file is sorted with respect to the `Salary` attribute.
- there are 500 distinct salary values, 125 departments, and 2 values of `EXP` (experienced or inexperienced).
- we can fit 100 blocks in memory.

Moreover, we have built the following access paths:

- A clustering index on `Salary`, with 3 levels.
- A B+ Tree on `DNO`, with 2 levels.
- A B+ Tree on `EXP`, with 2 levels.

Find the best way to execute this query.

- We can linearly search all the tuples and return those tuples satisfying the query. This requires 2000 block accesses since none of the searching attributes are key.
- We can make use of the B+ Tree on `DNO`. Here, we first get all the tuples that satisfy `DNO = 5`. For this, we require 2 block accesses to descend the tree, 1 to get the block of pointers, and $s_{\texttt{DNO}}$ blocks to get the actual data blocks (in the worst case). Under the uniformity assumption, we find that the selection cardinality of `DNO` is
$$s_{\texttt{DNO}} = \frac{1}{125} \cdot 10\ 000 = 80.$$
Since we expect 80 tuples to satisfy this condition, we can fit all of them in 16 blocks (as $bfr = 5$). This fits in memory. So, after finding these tuples, we can linearly check whether they satisfy the other two conditions. Overall, we just require 83 block accesses.

- We can make use of the clustering index on `Salary`. For this, we require 3 block accesses to descend the multilevel index and get to the first level of index. Then, it takes

$$\text{ceil}(s_{\texttt{Salary}}/f)$$

block accesses to retrieve all the tuples satisfying `Salary = 30 000`, where $f = 5$ is the blocking factor. Under the uniformity assumption, we find that the selection cardinality of `Salary` is

$$s_{\texttt{Salary}} = \frac{1}{500} \cdot 10\ 000 = 20.$$

In that case,

$$\text{ceil}(s_{\texttt{Salary}}/f) = 4.$$

Since we expect 20 tuples to satisfy this condition, we can fit them in 4 blocks- this fits in memory. After finding these tuples, we can linearly check whether they satisfy the other two conditions. Overall, we just require 7 block accesses.

- We can make use of the B+ Tree on `EXP`. For this, we require 2 block accesses to descend the tree, 1 to get the block of pointers, and $s_{\texttt{EXP}}$ blocks to get the actual data blocks (in the worst case). Under the uniformity assumption, we find that the selection cardinality of `EXP` is

$$s_{\texttt{EXP}} = \frac{1}{2} \cdot 10\ 000 = 5000.$$

The 5000 tuples occupy 1000 blocks, so they cannot all fit in memory. One approach would be to write 900 of the blocks back and keep them for processing later. We would further need to read them at a later point to check that they satisfy the other conditions. This would require 6803 block accesses.

Clearly, the best way to execute this query is by using the clustering index on `Salary`, requiring 7 block accesses.

Example 4.2.4. We are given the following query.

```
SELECT  *
FROM    EMPLOYEE
WHERE   DNO = 5 OR (SALARY >= 500 AND EXP = 0);
```

Assume that:

- there are $r = 10\ 000$ records in $b = 2000$ blocks.
- the file is sorted with respect to the `Salary` attribute.
- there are 500 distinct salary values, 125 departments, and 2 values of `EXP` (experienced or inexperienced).
- we can fit 1100 blocks in memory.

- the minimum salary value is 100 and the maximum is 10 000.

Moreover, we have built the following access paths:

- A clustering index on Salary, with 3 levels.
- A B+ Tree on `DNO`, with 2 levels.
- A B+ Tree on `EXP`, with 2 levels.

Find the best way to execute this query.

- We can linearly search all the tuples and return those tuples satisfies the query. This requires 2 000 block accesses.
- We can make use of the access paths. First, we get all the tuples that satisfy `DNO = 5` using the B+ tree on `DNO`. For this, we require 2 block accesses to descend the tree, 1 to get the block of pointers, and $s_{\texttt{DNO}}$ blocks to get the actual data blocks (in the worst case). Under the uniformity assumption, we find that the selection cardinality of `DNO` is

  $$s_{\texttt{DNO}} = \frac{1}{125} \cdot 10\ 000 = 80.$$

  Since we expect 80 tuples to satisfy this condition, we can fit all of them in 16 blocks (as $bfr = 5$). This fits in memory. So, after finding these tuples, we can linearly check whether they satisfy the other two conditions. Overall, this require 83 block accesses.

  Next, we consider how we find all the tuples that satisfy `SALARY >= 500 AND EXP = 0`.

  - We can use the B+ Tree on `EXP`. Under the uniformity assumption, we find that the selection cardinality of `EXP` is

    $$s_{\texttt{EXP}} = \frac{1}{2} \cdot 10\ 000 = 5000.$$

    Since the tree has 2 levels, it takes 5003 block accesses to find the 5000 tuples that satisfy the condition. This fits in 1000 blocks, so we can store it in memory. Overall, this takes 5086 block accesses.
  - We can also use the B+ Tree on `Salary`. Under the uniformity assumption, we find that the selection cardinality of `Salary` is

    $$s_{\texttt{Salary}} = \frac{1}{500} \cdot 1000 = 20.$$

    It takes 1921 block access to find the 1918 blocks that satisfy the condition. This does not fit in memory. We can hold at most 1084 blocks in memory and then write the other 834 blocks back to the disc. After filtering blocks in the main memory, we free up some space in the memory. In the first iteration, we discard the blocks that do not satisfy `EXP = 0`.

Since there are 2 distinct values of `EXP`, we discard half of them. This gives us 542 free blocks. Now, we load 542 blocks from the disc and filter them in the same way. At this point, we have 292 blocks left to be filtered. Next, we get 271 free blocks. During this iteration, we have 21 further blocks to be filtered. In the next iteration, we can fully filter the blocks. In total, we require

$$1921 + 834 + 542 + 271 + 21 = 3589$$

block accesses.

Therefore, the best choice here is linear searching.

4.3 Join selectivity

Example 4.3.1. Assume we have the following query.

```
SELECT   *
FROM     EMPLOYEE E, DEPARTMENT D
WHERE    E.DNO = D.DNUMBER;
```

We have the following context:

- there are $r_E = 10\ 000$ employee tuples that fit in $b_E = 2000$ blocks;
- there are $r_D = 125$ department tuples that fit in $b_D = 13$ blocks;
- the blocking factor of the joined tuple is $f_{RS} = 4$;
- we can fit $n_B = 10$ blocks in memory.

Moreover, we have built the following primary access paths:

- Primary Index on DNUMBER $x_{Dnumber} = 1$ level.
- B+ Tree Index on DNO $x_{Dno} = 2$ levels.

Using this information, find the best approach to execute this query.

The selection selectivity of the `DNO` attribute in the `DEPARTMENT` relation is given by

$$sl(\texttt{DNO}) = 1/125 = 0.008,$$

under the uniformity assumption. An employee works in a department- this is the joining condition. Assuming that the department an employee works in is unique, the join selectivity is

$$js = 1/\max(125, 125) = 0.008.$$

This means that the join cardinality is

$$jc = js \cdot r_E \cdot r_D = 10\ 000$$

tuples. These fit in 2500 blocks.

- First, we consider a nested-loop join algorithm. If the department is in the outer loop, the following is the algorithm:
  - load $n_B - 2$ `DEPARTMENT` blocks.
  - load all the `EMPLOYEE` blocks, one at a time.
  - check whether any tuple in the `EMPLOYEE` block can be joined with a tuple in one of the `DEPARTMENT` blocks.
  - Restart until all the `DEPARTMENT` blocks have been loaded.

  Here, we are only storing one block during the execution to store the output (i.e. if the block is full, we write it back to memory). This is why we can use $n_B - 1$ blocks in memory to store the

`EMPLOYEE` and the `DEPARTMENT` blocks. So, the total number of blocks that we read is

$$b_D + (\text{ceil}(b_D/n_B - 2)) \cdot b_E = 4013.$$

We know that the join cardinality is 10 000 tuples, which fit in 2500 blocks. So, the total cost of executing this query (read and write is)

$$4013 + 2500 = 6513$$

block accesses.

- Next, we consider an index-based nested-loop join algorithm with employee as the outer relation. The following is the algorithm:
  - Load an `EMPLOYEE` block.
  - Use the primary index on `DNUMBER` to find the relevant block (and then the tuple) to join.
  - Output the result.
  - Continue for all the `EMPLOYEE` blocks.

  So, the total number of blocks that we read is

  $$b_E + r_E(x_{\texttt{DNumber}+1}) = 22\ 000.$$

  If we write the result back into the disc, this requires 2500 further block accesses. So, we have

  $$22\ 000 + 2500 = 24\ 500$$

  block accesses in total.

- Now, we consider an index-based nested-loop join with department as the outer relation, we need

  $$b_D + r_D(x_{DNO} + s_{DNO} + 1) + (js \cdot r_E \cdot r_D)/f_{RS} = 12\ 288$$

  - Load a `DEPARTMENT` block.
  - Use the B+ Tree on `DNO` to find the s relevant blocks (and then the tuples) to join, where s is the selection cardinality of `DNO`.
  - Output the result.
  - Continue for all the `DEPARTMENT` blocks.

  Under the uniformity assumption, we find that the selection cardinality of `DNO` is

  $$s_{\texttt{DNO}} = 0.008 \cdot 10\ 000 = 80.$$

  So, the total number of blocks that we read is

  $$b_D + r_D(x_{\texttt{DNO}} + s_{\texttt{DNO}} + 1) = 9\ 788.$$

If we write the result back into the disc, this requires 2500 further block accesses. So, we have

$$22\ 000 + 2500 = 12\ 288$$

block accesses in total.

- Using a hash-join, we need

$$3(b_D + b_E) + (js \cdot r_E \cdot r_D)/f_{RS} = 8539$$

in total block accesses to read and write the blocks.

- Since we have a B+ Tree built on `DNO`, we cannot use the sort-merge algorithm. The attribute must be non-ordering.

So, the best approach to join them is a nested-loop join.

Example 4.3.2. We want to execute the following SQL query.

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D, DEPENDENT T
WHERE   T.E_SSN = E.SSN AND E.SSN = D.MGR_SSN;
```

We have

- $r_T = 50$ dependents in $b_T = 3$ blocks;
- $r_D = 125$ departments in $b_D = 13$ blocks;
- $r_E = 10\ 000$ employees in $b_E = 2000$ blocks;
- the blocking factor $f = 10$, and for the result block $f_{RS} = 2$;
- maximum $n_B = 100$ blocks in memory.

Under the uniformity assumption, there are

$$50/10 = 5$$

dependents per employee. Moreover, we have the following primary access paths:

- A clustering index on `T.E_SSN` of $x_{\text{E\_SSN}} = 2$ levels with 10 distinct `SSN` values.
- A primary index on `D.MGR_SSN` of $x_{\text{MGR\_SSN}} = 1$ level.
- A B+ Tree on `E.SSN` of $x_{\text{SSN}} = 4$ levels.

Find the number of block accesses require to compute the result using the following approaches:

1. First joining `EMPLOYEE` and `DEPENDENT`, and then joining the result with `DEPARTMENT`.
2. First joining `EMPLOYEE` and `DEPARTMENT`, and then joining the result with `DEPENDENT`.

1. In the first case, we first join `EMPLOYEE` and `DEPENDENT`. The join selectivity in this case is

$$js = 1/\max(10, 10\ 000) = 0.01\%.$$

So, we expect

$$jc = js \cdot |E| \cdot |D| = 50$$

tuples to be selected in this process.

For each dependent, we get their employee using the B+ Tree on `E.SSN`. This takes

$$b_T + r_T \cdot (x_{SSN} + 1) + jc/f_{RS} = 278$$

block accesses to read and write the tuples into memory. We can also use the clustering index on `T.E_SSN`. For each dependent, we get the dependent using this index. This takes

$$b_E + r_E \cdot (x_{\text{E\_SSN}} + \text{ceil}(s_{\texttt{E\_SSN}}/f)) + jc/f_{RS} = 32\ 025$$

block accesses to read and write the tuples into memory. Clearly, the better option is to use the B+ Tree.

In this case, the intermediate result has $r_{ET} = 50$ tuples, which fit in 25 blocks. These blocks can fit in memory. Next, we take the intermediate result and join it with the `DEPARTMENT` relation. So, we take a tuple from the intermediate result in memory and check whether it is a manager. The join selectivity for `EMPLOYEE` and `MANAGER` is

$$js = 1/\max(10\ 000, 125) = 0.01\%.$$

So, we expect

$$jc = js \cdot r_{ET} \cdot r_D = 0.625$$

tuples to be selected in the process. This will fit in 1 block.

We can use the primary index on `D.MGR_SSN` to filter out the tuples. We take a tuple from the intermediate result and use the primary index to retrieve the manager details. This requires

$$r_{ET} \cdot (x_{\text{MGR\_SSN}} + 1) = 100$$

blocks to be read from memory. We do not need to load the intermediate blocks since they are already in memory. Moreover, we do not write the final outcome into disks. In total, this plan requires

$$278 + 100 = 378$$

block accesses.

2. In the other plan, we first join `EMPLOYEE` and `DEPARTMENT`. Here, the join selectivity is

$$js = 1/\max(10\ 000, 125) = 0.01\%.$$

So, the join cardinality is

$$jc = js \cdot r_E \cdot r_D = 125$$

managers. Since the `DEPARTMENT` relation is sorted with respect to `MGR_SSN`, we can find all the managers in

$$\text{ceil}(jc/f_{RS}) = 63$$

blocks.

For each department, we can get its manager using the B+ Tree on `E.SSN`. This takes

$$b_D + r_D(x_{SSN} + 1) + jc/f_{RS} = 701$$

block accesses. Also, for each employee, we can get the department they manage (if any) using the primary index on `D.MGR_SSN`. This takes

$$b_E + r_E(x_{\text{MGR\_SSN}} + 1) + jc/f_{RS} = 22\ 063$$

block accesses. The better option here is to use the B+ Tree.

The intermediate result takes 63 blocks, which can fit in memory. We will now join this result with the `DEPENDENT` relation. The join selectivity for `EMPLOYEE` and `DEPENDENT` is

$$js = 1/\max(10\ 000, 10) = 0.01\%.$$

So, we expect

$$jc = js \cdot r_{ED} \cdot r_T = 0.625$$

tuples to be selected in the process. This will fit in 1 block.

We can take one intermediate tuple from memory, and check whether the manager has a dependent, using the clustering index on `T.E_SSN`. This requires

$$r_{ED}(x_{\text{E\_SSN}} + \text{ceil}(s_{\texttt{E\_SSN}}/f)) = 375$$

block accesses. In total, this plan requires

$$701 + 375 = 1076$$

block accesses.

Example 4.3.3. We want to execute the following query:

```
SELECT  *
FROM    EMPLOYEE E, DEPARTMENT D
WHERE   E.Salary = 1000 AND E.SSN = D.MGR_SSN;
```

There are $r_E = 10\ 000$ employee tuples in $b_E = 2000$ blocks. There are $r_D = 125$ department tuples in $b_D = 13$ blocks.
We have built the following primary access paths on the two relations:

- A clustering index over `E.Salary` of $x_{\text{Salary}} = 3$ levels. There are 500 distinct salary values.
- A B+ tree over `E.SSN` of $x_{\text{SSN}} = 4$ levels.
- A primary index over `D.MGR_SSN` of $x_{\text{MGR\_SSN}} = 1$ level.

The blocking factor of `E` and `D` is $f = 10$, and the joined tuple is $f_{RS} = 2$. We can fit $n_B = 100$ blocks in memory.
Find the number of block accesses required to compute the result using the following approaches:

1. First filtering and then joining.
2. First joining and then filtering.

1. We will first consider the filter-then-join approach. We can use the clustering index on `E.Salary` to find all the employees that satisfy `Salary = 1000`. Under the uniformity assumption, the selection selectivity is
$$sl(\text{Salary}) = 1/500 = 0.002.$$
So, the selection cardinality is
$$s_{\texttt{Salary}} = 0.002 \cdot 10\ 000 = 20.$$
That is, there are $r_{FE} = 20$ employees that satisfy this condition. Using the clustering index, we require
$$x_{\text{Salary}} + \text{ceil}(s_{\texttt{Salary}}/f) = 5$$
block accesses.
Next, we join the intermediate result with the relation `DEPARTMENT`. Since the intermediate result occupies 2 blocks, it can fit in memory. The join selectivity is
$$js = 1/\max(10\ 000, 125) = 0.01\%.$$
So, the join cardinality is
$$jc = js \cdot r_{FE} \cdot r_D = 0.25.$$
This fits in 1 block. For each tuple in the intermediate result, we use the primary index over `D.MGR_SSN` to get the relevant department tuple, if any. This requires
$$r_{FE} \cdot (x_{\text{MGR\_SSN}} + 1) = 40$$
block accesses. In total, we require
$$5 + 40 = 45$$
block accesses.

2. Next, we consider the join-then-filter approach. We will use the B+ Tree over `E.SSN` to join `EMPLOYEE` and `DEPARTMENT`. The selection selectivity is

$$js = 1/\max(10\ 000, 125) = 0.01\%.$$

Using the B+ Tree, we require

$$b_D + r_D \cdot (x_{\text{SSN}} + 1) + \text{ceil}(js/f_{RS}) = 701$$

block accesses. The intermediate result is 125 tuples (one for each department). This takes

$$\text{ceil}(125/f_{RS}) = 2$$

blocks. This can fit in memory.

Next, we filter the tuples from the intermediate result that satisfy `Salary = 1000`. This can be done by linearly scanning the entries in memory. It needs no further block access since the salary is already present in the intermediate result. So, it takes 701 block accesses in total.