CHAPTER 4

QUERY OPTIMISATION

4.1 Query Processing

Almost all SQL queries involve sorting of tuples with respect to sorting requests defined by the user, e.g.

- `CREATE PRIMARY INDEX ON EMPLOYEE(SSN)` means that we sort by `SSN`
- `ORDER BY Name` means that we sort by `Name`
- `SELECT DISTINCT Salary` means that we sort by `Salary` to create clusters, and then identify the distinct values
- `SELECT DNO, COUNT(*) FROM EMPLOYEE GROUP BY DNO` means that we sort by `DNO` to create clusters, and then count the number of values per cluster.

Normally, we cannot store the entire relation into memory for sorting the records. So, to sort the tuples, we use external sorting algorithm.

External Sorting

The external sorting algorithm is a divide and conquer algorithm. We first divide a file of b blocks into L smaller sub-files (so each sub-file has b/L blocks). We require each of the sub-file to fit in memory. We load each small sub-file into memory and sort them (e.g. using quicksort) and then write it back to the disk. At the end of this, the sub-files are sorted.

We then merge sorted sub-files to generate bigger sub-files. We do this by loading the sub-file and combining them (using an algorithm similar to the mergesort merge algorithm). This process continues until we have combined it into the entire file. Note that we do not need to load the entire sub-file in one go to merge. We just load the blocks with similar values to sort a subsection of the sub-files.

The expected cost of external sorting is

$$2b(1 + \log_M L)$$

block accesses, where:

- b is the number of file blocks,
- M is the degree of merging (i.e. the number of sorted blocks merged in each loop), and
- L is the number of the initial sorted sub-files (before entering the merging phase).

This method is expensive as it is linear with respect to the number of blocks. Moreover, as the value of M increases, the number of block access decreases.

Executing queries

We will be considering queries of the form

`SELECT * FROM` *relation* `WHERE` *selection-conditions*

First, assume that selection condition is just a key attribute. An example of such a query is

```
SELECT * FROM EMPLOYEE WHERE SSN = '123';
```

- If we use a linear search, then the expected cost is $b/2$ block accesses, where b is the number of blocks.
- Instead, we can also use a binary search. If the files are sorted with respect to the key attribute, the expected cost is $\log_2 b$ block accesses. Otherwise, we need to sort the files in $2b(1 + \log_M L)$ block accesses, and then find the tuple in $\log_2 b$ block accesses.
- If we have a primary index of level t over the key, then it takes $t+1$ block accesses.
- Moreover, if we have a hash file, then it takes $1 + O(n)$ block accesses, where n is the number of overflown buckets.
- Finally, if the file is not sorted over the attribute and we have a secondary index (B+ Tree of level t), then we require $t + 1$ block accesses.

Next, assume that the selection condition is a range query with respect to a key attribute. An example of such a query is

```
SELECT * FROM DEPARTMENT WHERE DNumber >= 5;
```

If we have a primary index of level t over the key, then it takes $t + O(b)$, where b is the number of blocks in the worst case scenario. Since a primary index relation is sorted, we first find the value `DNumber = 5` and then keep going lower.

Now, assume that we have a clustering index over an ordering, non-key field. An example of such a query is

```
SELECT * FROM EMPLOYEE WHERE DNO = 5;
```

- If the clustering index is of level t, then the expected cost is $t + O(b/n)$, where b is the number of blocks and n is the number of distinct values of the attribute. This is under the assumption that the attribute is uniformly distributed over the tuples.
- If the file is not sorted with respect to the attribute and we have a secondary index (B+ Tree of level t), then we need $t+1+O(b)$ block accesses, where b is the number of block pointers. Here, the B+ Leaf nodes point to a block of pointers to data blocks.

Now, assume we have a disjunctive select statement, i.e. we have an `OR` present in the `WHERE` clause. An example is given below.

```
SELECT * FROM EMPLOYEE
WHERE SALARY > 10000 OR NAME LIKE '%Chris%';
```

The result will be the union of tuples satisfying the two conditions.

If an access path exists (e.g. B+, hash, primary index) index for all the attributes, we use each of them to retrieve the set of records satisfying each condition. Then, we return the union of the sets to get the final result. Otherwise, we must use a linear search.

On the other hand, assume that we have an conjunctive select statement, i.e. we have an AND present in the WHERE clause. An example is given below.

```
SELECT * FROM EMPLOYEE
WHERE SALARY > 10000 AND NAME LIKE '%Chris%';
```

The result will be the intersection of tuples satisfying the two conditions.

If an access path exists for any of the attribute, we can use that to construct an intermediate result. Then, we can use a linear search to validate the other conditions. If none of the attributes have an attribute path, we need to use linear search. If we have multiple access paths, we should choose the right index to generate the smallest intermediate result. We predict the selectivity (the number of tuples received) for each attribute to do this. This is query optimisation.

Next, assume that we have a join query. Joining is the most resource-consuming operator. We will only be considering two-way equijoin. An example of such a query is

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

There are 5 ways for processing join queries:

- Naive join (no access path)
- Nested-loop join (no access path)
- Index-based nested-loop join (index; B+ Trees)
- Merge-join (sorted relations)
- Hash-join (hashed relations)

Assume we have the following query.

```
SELECT  *
FROM    R, S
WHERE   R.A = S.B;
```

In naive join, we compute the Cartesian product of R and S, and then only filter the ones that satisfy R.A = S.B. This method is inefficient because we get rid of most of the tuples from the Cartesian product most of the time.

In nested-loop join, we have a nested loop algorithm over the two relations to only generate products if they satisfy the join condition. In terms of relations, the algorithm is the following.

```
for each tuple r in R:
    for each tuple s in S:
        if r.A = s.B:
            add(r, s) to the result file;
```

We say R is the outer relation and S is the inner relation. The choice of outer and inner relation affects the computation process. Since we only have access to blocks, the algorithm is the following in terms of blocks.

- Load a set of blocks from the outer relation R.
- Load one block from inner relation S.
- Maintain an output buffer for the matching tuples (r, s) using the algorithm above.
- Join the S block with each R block from the chunk.
- For each matching tuple r in R and s in S, add (r, s) to the output buffer.
- If the output buffer is full, pause and write the current join result to the disk.
- Load the next S block.
- After going through all the R blocks, go to the next set of R blocks.

If we have an index on either A or **B**, we can make use of it in the join. Assume that we have an index I on S.B. Then, the algorithm becomes:

```
for each tuple r in R:
    use index of B to retrieve all tuples s in S
        satisfying s.B = r.A
    for each such tuple s, add (r, s) to the result file;
```

This process is much faster than nested-loop join. If we have two indices, we need to choose the right one to minimise the join processing cost.

Now, assume that both R and S are physically ordered on their joining attribute A and B. Then, we can use the following approach to join them:

- Load a pair (R', S') of sorted blocks into memory.
- Scan the two blocks concurrently over the joining attribute (sort-merge algorithm).
- If matching tuples are found, then store them in a buffer.

In this case, the blocks of each file are scanned only once. But, if R and S are not physically sorted, then we need to use the external sorting method.

Now, if R and S are partioning into M buckets with the same hash function over the join attributes A and B, then we can use a hash-join algorithm. Assuming R is the smallest file and fits into main memory, i.e. M buckets of R are in memory, we start by partioning.

```
for each tuple r in R:
    compute y = h(r.A) // the address of the bucket
    place tuple r into bucket y = h(r.A) in memory
```

Next, we have the probing phase.

```
for each tuple s in S:
    compute y = h(s.B) // using the same hash function
    find the bucket y = h(s.B) in memory
    for each tuple r in R in the bucket y:
        if s.B = r.A:
            add(r, s) to the result file;
```

Assume that we have 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. We will predict the cost of using a nested-loop join 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 $n_E/(n_B - 2)$, and the inner loop is n_D. The total number of block accesses is therefore

$$n_E + (n_E/(n_B - 2))n_D.$$

If $n_E = 2\ 000$ blocks, $n_D = 10$ blocks and $n_B = 7$ blocks, then we require

$$2\ 000 + 10 \cdot \frac{2000}{5} = 6\ 000$$

block accesses. If swap the inner and the outer relations, we require

$$10 + 2\ 000 \cdot \frac{10}{5} = 4\ 010$$

block accesses. So, it is better to have the file with fewer blocks in the outer loop.

Next, assume we have the query

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

We have a B+ Tree on Mgr_SSN with level $x_D = 2$, and a B+ Tree on SSN with level $x_E = 4$. Moreover, the record E has $r_E = 6\ 000$ tuples in $n_E = 2\ 000$ blocks, and the record D has $r_D = 50$ tuples in $n_D = 10$ blocks.

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.

Next, we consider the sort-merge-join algorithm. We require the two relations to be sorted with respect to the joining attribute. We load each block from the two relations precisely once, so this requires $n_E + n_D = 2\ 010$ 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.

Finally, we consider the hash-join algorithm. We assume that the smaller relation Department fits in memory (i.e. $n_B > n_D + 2$). During the hashing phase, we hash the smaller relation into buckets. During the probing phase, we load a block from the bigger relation, hash it and check within the bucket if there is a match. If we find a match, we add it to the output block. If there are no overflown buckets, this requires $n_E + n_D$ block accesses. Typically, the smaller relation doesn't fit in memory, so we require $3(n_E + n_D)$ block accesses.

Next, assume we have the query

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

Moreover, 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. If we have a 2-level on `E.SSN`, then we can use nested-index join algorithm as follows:

```
for each Department d:
    use B+ Tree (e.SSN) to find 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. If we use a hash function on `D.MGR_SSN`, into 10 buckets, then we can use hash-join algorithm as follows:

```
load all the Departments d
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.

Now, 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, and a 5-level B+ Tree on `SSN` and 2-level B+ Tree on `Super_SSN`. 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: # employee is not supervisor
        use B+ Tree to find Employee s.SSN
        output the tuple (e, s);
```

Assume that 10% of the employees are supervisors. Then, using a level-t B+ Tree, 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

There are two fundamental concepts in optimisation- join and selection selectivity. Selection selectivity is the fraction of tuples satisfying a condition. Join selectivity is the fraction of matching tuples in the Cartesian product.

Before running a query, we would like to predict the selection and the join selectivity, and hence the number of blocks we expect to retrieve. Then, using the actual block accesses, we aim to refine the expected cost. The expected cost is expressed as a function of selectivity. Using this result, we choose the optimal strategy to run a query.

In query optimisation, we are given a query as an input. The output is the optimal execution plan. There are two types of query optimisations:

- Heuristic optimisation- we transform a SQL query into an equivalent and efficient query using Relational Algebra.
- Cost-based optimisation- we provide many execution plans and estimate their costs, and choose the plan with the minimum cost.

The cost function is what we want to optimise. It has parameters- the number of block accesses, the memory requirements, the CPU computational cost, the network bandwith, etc. We will consider the number of block accesses and memory requirements.

We will use statistical information available to estimate the execution of a query. To do this, we store the following information for each relation:

- the number of records r, and the (average) size of each record R.
- the number of blocks b, and the blocking factor f.
- the primary file organisation (heap, hash or sequential).
- the available indices- primary, clustering, secondary or B$+$ Trees.

For each attribute, we further store the following:

- the number of distinct values (NDV) n of the attribute.
- the domain range- the minimum and the maximum value of the attribute (among the tuples).
- the type of attribute- continuous or discrete, key or non-key.
- level t of Index of the attribute, if present.
- the probability distribution function $P(A = x)$, which indicates the frequency of each value x of the attribute A in the relation.

A good approximation of the distribution is a histogram.

Selection selectivity of an attribute A, denoted by $\text{sl}(A)$, is the fraction of tuples that satisfy a given condition. Therefore, its value is between 0 and 1. If $\text{sl}(A) = 0$, then none of the records satisfy this condition with respect to the attribute A. On the other hand, if $\text{sl}(A) = 1$, then all the records satisfy the condition with respect to the attribute A. As a probability, the selection selectivity tells us how likely is it for a tuple to satisfy the given condition with respect to an attribute.

The selection cardinality s is the number of tuples we expect to satisfy a query with respect to a given attribute. In other words, the selection cardinality

$$s = \mathrm{sl}(A) \cdot r,$$

where r is the number of tuples present. So, the value is between 0 and r. We predict this value without scanning any block. For example, if $r = 1\ 000$ and $\mathrm{sl}(A) = 0.3$, then the selection cardinality $s = 300$.

To predict selectivity, we can approximate the distribution of attribute values using a histogram. This allows us to maintain an accurate selectivity estimate. However, we must maintain this histogram whenever there is a change to the attribute for any of the tuples. In this case, we assume that

$$\mathrm{sl}(A = x) \approx P(A = x).$$

That is, the selection selectivity is approximately equal to the probability that the value of the attribute is that value.

Another possible approximation is that all values are uniformly distributed. This means that we do not need to maintain a histogram- all the values are of equal probability, so we barely need to store one value. This means that we do not need to maintain a histogram. However, we provide a less accurate prediction for the selection selectivity. In this case, we assume that

$$\mathrm{sl}(A = x) \approx \frac{1}{n},$$

where n is the number of distinct values of A. Note that the selectivity is a constant value independent of x.

If A is a key attribute, then

$$\mathrm{sl}(A = x) \approx \frac{1}{r}$$

is a good estimate. There is only one tuple that satisfies the condition, so the selection cardinality $s = 1$. So, the uniformity assumption is valid in this case. We do not need to build a histogram.

Now, if A is a non-key attribute, with $n = NDV(A)$, the number of distinct values of A present. Then, the estimate

$$\mathrm{sl}(A = x) \approx \frac{1}{n}$$

is not a good estimate. This is under the assumption that all the records are uniformly distributed across the n distinct values. This is not true in general.

Next, consider the following range selection selectivity.

```
SELECT * FROM RELATION WHERE A>= x
```

The domain range is defined to be $\max(A) - \min(A)$, and the query range in this case is $\max(A) - x$. If $x > \max(A)$, then $\mathrm{sl}(A \geq x) = 0$. Otherwise,

$$\mathrm{sl}(A \geq x) = \frac{\max(A) - x}{\max(A) - \min(A)},$$

under the uniformity assumption.

In particular, if we have $r = 1\ 000$ employees, the attribute value ranges from 100 to 10 000 and $x = 1000$, then the selection selectivity is

$$\mathrm{sl}(A \geq 1000) = \frac{\max(A) - x}{\max(A) - \min(A)} = \frac{10\ 000 - 1000}{10\ 000 - 100} = 0.909$$

In that case, the selection cardinality is 909 tuples.

Now, consider the conjunctive selectivity, i.e. a query satisfying 2 conditions. An example is given below.

```
SELECT * FROM RELATION WHERE (A = x) AND (B = y)
```

If the two attributes A and B are statistically independent, then the selectivity of the conjunction query q is

$$\mathrm{sl}(q) = \mathrm{sl}(A = x) \cdot \mathrm{sl}(B = y).$$

Next, 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.

The selection selectivity for the Salary attribute is

$$\mathrm{sl}(40\ 000) = \frac{1}{100} = 0.01$$

Moreover, the selection selectivity for the DNO attribute is

$$\mathrm{sl}(5) = \frac{1}{10} = 0.1.$$

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

$$\mathrm{sl}(Q) = \mathrm{sl}(40\ 000) \cdot \mathrm{sl}(5) = 0.001.$$

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

Now, we consider a disjunctive selectivity condition, i.e. a query satisfying one of 2 conditions. An example is given below.

```
SELECT * FROM RELATION WHERE (A = x) AND (B = y)
```

Then, the selection selectivity of this query is

$$\mathrm{sl}(Q) = \mathrm{sl}(A) + \mathrm{sl}(B) - \mathrm{sl}(A)\,\mathrm{sl}(B).$$

Here as well, we assume that the two attributes A and B are statistically independent.

Consider the following query

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

We find that the selectivity of this query is

$$\text{sl}(Q) = \text{sl}(40\ 000) + \text{sl}(5) - \text{sl}(40\ 000) \cdot \text{sl}(5) = 0.109.$$

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

Now, assume that we have r tuples, and the attribute A has n number of distinct values. Under the uniform assumption of the attribute, the expected number of block accesses is

$$\text{ceil}(s/f) = \text{ceil}(r/f \cdot n),$$

where s is the selection cardinality and f is the blocking factor. The graph below summarises this.

Now, we apply this formula to refine the selection cost. Assume that we have the query

```
SELECT * FROM RELATION WHERE A = x;
```

We have b blocks; the blocking factor is f; we have r records; and the number of distinct values of the attribute A is n. We propose different approaches and the expected number of block accesses in each case.

- Assuming that the tuples are sorted with respect to the attribute A, we can use a binary search algorithm.
  - If A is a key attribute, then the expected cost is $\log_2 b$ block accesses. It is independent of the selection selectivity.
  - If A is not a key attibute, then it takes $\log_2 b$ block accesses to find the first block with record $A = x$. We then access all the contiguous blocks whose records satisfy $A = x$. This takes $\text{ceil}(s/f) - 1$ further block accesses. In total, the expected cost is

    $$\log_2 b + \text{ceil}(s/f) - 1 = \log_2 + \text{ceil}(r \cdot \text{sl}(A)/f) - 1.$$

    This is a function of the selection selectivity.
- Now, assume that we have a level-t multilevel primary/clustering index on the attribute.
  - If A is a key attribute, then the expected cost is $t+1$ block accesses. It is independent of the selection selectivity.
  - If A is a non-key attribute, then it takes t block accesses to descend the tree. The selection cardinality is $s = r \cdot \text{sl}(A)$ tuples. So, we require $\text{ceil}(s/f)$ block accesses, where f is the blocking factor. So, the expected cost is

    $$t + \text{ceil}(s/f) = t + \text{ceil}(r \cdot \text{sl}(A)/f).$$

    This is a function of the selection selectivity.

- Assume next that we have a level-t B+ tree on the attribute.
  - If A is a key attribute, then the expected cost is $t+1$ block accesses. It is independent of the selection selectivity.
  - If A is a non-key attribute, then it takes t block accesses to descend the tree. At this point, we have access to the block pointer to all the blocks containing a tuple satisfying the condition. The selection cardinality is $s = r \cdot \mathrm{sl}(A)$ tuples. Since the tuples are not sorted with respected to A, we assume that we must access s blocks to retrieve all the tuples. In that case, the expected number of block accesses is
$$t + 1 + s = t + 1 + r \cdot \mathrm{sl}(A).$$
    This is a function of the selection selectivity.
- Finally, assume that we are using a hash file structure.
  - If A is a key attribute, then the expected cost is just 1 block access (if there are no overflown buckets). This is independent of the selection selectivity.
  -

Next, assume that we have the query

```
SELECT * FROM RELATION WHERE A >= x;
```

- Assume further that we have a level t primary index and A is a key attribute. We traverse the tree in t block access, finding the data block for $A = x$. Then, the range selection cardinality is $s = r \cdot \mathrm{sl}(A \geq x)$. So, we further access ceil(s/f) blocks. In total, we have
$$t + \mathrm{ceil}(s/f) = t + \mathrm{ceil}(r \cdot \mathrm{sl}(A)/f)$$
  block accesses. This is a function of the selection selectivity.

Now, we consider the best way of executing the following query.

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

There are 10 000 records, and the blocking factor is 5. So, there are 2 000 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.

We consider different strategies to execute the query.

- We can linearly search all the tuples and return those tuples satisfying the query. This requires 2 000 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_{DNO} blocks to get the actual data blocks (in the worst case). Under the uniformity assumption, we find that
$$s_{\text{DNO}} = \frac{1}{125} \cdot 10000 = 80.$$
Since we expect 80 tuples to satisfy this condition, we can fit all of them in 16 blocks (we know that $bfr = 5$). 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
$$\text{ceil}(s_{\text{Salary}}/f)$$
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
$$s_{\text{Salary}} = \frac{1}{500} \cdot 10000 = 20.$$
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_{EXP} blocks to get the actual data blocks (in the worst case). Under the uniformity assumption, we find that
$$s_{\text{EXP}} = \frac{1}{2} \cdot 10000 = 5000.$$
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. So, this requires 6803 block accesses.

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

Next, we consider the best way of executing the following query.

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

We have the same case as above. Also, now we can fit 1100 blocks in memory. Moreover, the minimum Salary value is 100 and maximum Salary is 10 000.

The conjunction `SALARY >= 500 AND EXP = 0` has selection selectivity

$$slc = sl(\text{Salary}) \cdot sl(\text{EXP}) = \frac{10\ 000 - 5000}{10\ 000 - 100} \cdot \frac{1}{2} = 0.4795.$$

In that case, the disjunction `DNO = 5 OR (SALARY >= 500 AND EXP = 0)` has selection selectivity

$$\text{sl(DNO)} + slc - \text{sl(DNO)} \cdot slc = \frac{1}{125} + 0.4795 - \frac{1}{125} \cdot 0.4795 = 0.4837.$$

So, we expect 4837 tuples to satisfy this condition. In that case, the least number of blocks we need to access would be

$$\text{ceil}(4837/5) = 968$$

blocks. We consider different strategies to execute the 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 acccess 83 blocks to get all the tuples that satisfy `DNO = 5`. Now, we consider how we find all the tuples that satisfy `SALARY >= 500 AND EXP = 0`.
  - First, we use the B+ Tree on `EXP`. 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.
  - Next, we use the B+ Tree on `Salary`. It takes 1921 block access to find the 1918 blocks that satisfy the condition. This does not fit in memory.

Therefore, the best choice here is linear searching. Moreover, even if we could fit infinite blocks fit in memory, using the B+ Tree on `Salary` would need 2003 block accesses.

4.3 Join selectivity

We measure join selectivity with respect to the cartesian product of the two relations. For example, consider the following tables. Their Cartesian product has $5 \cdot 3 = 15$ entries. However, the following is the joined table with respect to `DNO` and `DNumber`. There are only 5 entries here, so the join cardinality is 5. Moreover, the join selectivity is $5/15 = 1/3$.

If we are joining two relations R and S, then the join selectivity is given by

$$js = |R \bowtie S|/|R \times S|.$$

It is a value between 0 and 1. Moreover, the join cardinality is given by

$$jc = js \cdot |R||S|.$$

We can predict the join selectivity/cardinality using the join selectivity theorem. Given $n = \text{NDV}(A, R)$ and $m = \text{NDV}(B, S)$, then

$$js = 1/\max(n, m).$$

This implies that

$$jc = |R||S|/\max(n, m).$$

We illustrate the this with an example. Assume that we have a relation `E` of employees and `D` of dependents. An employee can have multiple dependents. Their common attribute is `SSN` of the employee. We have

- $n = \text{NDV(SSN, E)} = 2000$ and $|E| = 2000$;
- $m = \text{NDV(SSN, P)} = 3$ and $|P| = 5$.

Therefore, the join selectivity is

$$1/\max(n, m) = 1/2000 = 0.0005.$$

This implies that there is a 0.05% probability of getting a matching tuple from the cartesian space.

Now, assume we have relations R and S with b_R and b_S blocks. They can be joined using `R.A = S.B`. In memory, we have n_B blocks, and there are n distinct values of `R.A` and m distinct values of `S.B`. The blocking factor is f_{RS} matching tuples per block. The size of the maching tuple (r, s) is the sum of the size of the tuple r and s. We wrtie every full result block to disk. We expect the result to have

$$jc = js \cdot |R||S| = |R||S|/\max(n, m).$$

So, the number of result blocks is

$$k = (js \cdot |R| \cdot |S|)/f_{RS}.$$

If we use a nested-loop join, then the expected cost to read the blocks is

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

where D is the outer relation and E the inner relation. The expected cost of reading and writing the results to memory is

$$b_D + \text{ceil}(b_D/(n_B - 2)) \cdot b_E + (js \cdot |E||D|)/f_{RS}.$$

This is a function of the join selectivity.

If we use an index-based nested-loop join using a primary index on `D` with x_D levels, then the cost to read the blocks is

$$b_E + |E| \cdot (x_D + 1),$$

where b_E is the number of blocks in `E`. Then, the expected cost of reading and writing is

$$b_E + |E| \cdot (x_D + 1) + (js \cdot |E||D|)/f_{RS}.$$

If we have a clustering index over `E` with x_E levels with selection cardinality s_E and blocking factor f_E, then the cost to read the blocks is

$$b_D + |D| \cdot (x_E + \text{ceil}(s_E/f_E)).$$

Then, the expected cost of reading and writing is

$$b_D + |D| \cdot (x_E + \text{ceil}(s_E/f_E)) + (js \cdot |E||D|)/f_{RS}.$$

This is a function of the selection selectivity and the join selectivity.

If we have a B+ Tree over the non-key attribute `E` with x_E levels with selection cardinality s_E and blocking factor f_E, then the cost to read the blocks is

$$b_D + |D| \cdot (x_E + 1 + s_E).$$

Then, the expected cost of reading and writing is

$$b_D + |D| \cdot (x_E + 1 + s_E) + (js \cdot |E||D|)/f_{RS}.$$

If we use a sort-merge algorithm, then the cost to read and write is

$$b_R + b_S + (js \cdot |R| \cdot |S|)/f_{RS}.$$

If we use a hash-join algorithm, the cost to read and write is

$$3(b_R + b_S) + (js \cdot |R| \cdot |S|)/f_{RS}.$$

Next, we apply these equations to find the best algorithm to use when joining the relations `EMPLOYEE` and `DEPARTMENT`, where

- 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.

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

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

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. 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, we consider the number of block accesses we require for each algorithm.

- For nested loop join, we need

  $$b_D + (\text{ceil}(b_D/n_B \cdot 2)) \cdot b_E + (js \cdot r_E \cdot r_D)/f_{RS} = 6513$$

  block accesses.
- Using an index-based nested-loop join with employee as the outer relation, we need

  $$b_E + r_E(x_{DNumber} + 1) + (js \cdot r_E \cdot r_D)/f_{RS} = 24\ 500$$

  block accesses.
- Using an index-based nested-loop join with department as the outer relation, we need

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

  block accesses.
- Using a hash-join, we need

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

  block accesses.
- Assuming that 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.

3-way join optimisation

Now, we will consider joining 3 relations. Assume 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.

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.

Under the uniformity assumption, there are

$$50/10 = 5$$

dependents per employee.

There are 2 possible ways to join the 3 relations:

- We find the employees which have at least one dependent, and then checking whether they are managers.
- We find the employees that are managers, then check if these managers have dependents.

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

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

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

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

to 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.

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

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

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

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

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

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

block accesses. In total, this plan requires

$$701 + 375 = 1076$$

block accesses. So, the first plan is better.

Join and select optimisation

Now, we will consider the best way of executing a query involving both a join condition and a selection condition. The query is the following.

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

We have the following:

- 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.

We will consider different plans to execute this query.

- We can first find the employees that have `Salary = 1000` and then join.
- We can first join `EMPLOYEE` and `DEPARTMENT` and then filter the employees that have `Salary = 1000`.

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

$$\mathit{sl}(\text{Salary}) = 1/500 = 0.002.$$

So, the selection cardinality is

$$s_{\text{Salary}} = 0.002 \cdot 10\ 000 = 20.$$

That is, there are $r_{FE} = 20$ employees that satisfy this condition. Using the clustering index, we require

$$x_{\text{Salary}} + \text{ceil}(s_{\text{Salary}}/f) = 5$$

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

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

So, the join cardinality is

$$\mathit{jc} = \mathit{js} \cdot r_{FE} \cdot r_D = 0.25.$$

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.

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

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

Using the B+ Tree, we require

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

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. Clearly, it is better to use the filter-then-join approach.