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 Query Optimisation