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*

Key query

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.

Range query

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.

Non-key query

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.

Disjunctive query

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.

Conjunctive query

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.

Join queries

Next, assume that we have a join query. Joining is the most resource-consuming operator. We will only be considering two-way equijoin. These queries are of the following form.

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

There are 5 ways for processing join queries- naive join, nested-loop join, index-based nested-loop join, merge-join and hash-join.

Naive join

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.

Nested loop join

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.

It is more efficient to have the file with fewer blocks in the outer loop.

Index-based nested-loop join

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.

Merge-join

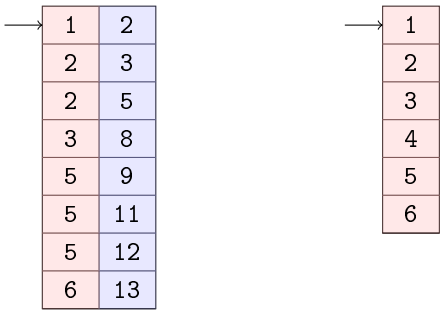
Now, assume that both `R` and `S` are physically ordered on their joining attribute `A` and `B`. Then, we can use the following approach to join them:

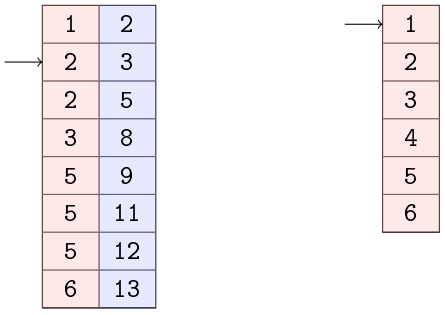
- Load a pair (R', S') of sorted blocks into memory.

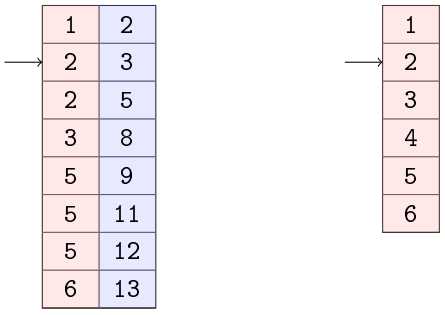
- Scan the two blocks concurrently over the joining attribute (sort-merge algorithm).

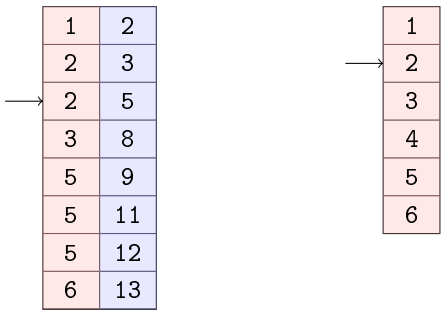
- If matching tuples are found, then store them in a buffer.

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.









Hash join

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;
```

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 $sl(A)$, is the fraction of tuples that satisfy a given condition. Therefore, its value is between 0 and 1. If $sl(A) = 0$, then none of the records satisfy this condition with respect to the attribute A. On the other hand, if $sl(A) = 1$, 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

$$s = sl(A) \cdot r,$$

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 $sl(A) = 0.3$, 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

$$sl(A = x) \approx P(A = x).$$

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

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

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

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

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

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

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 $sl(A \geq x) = 0$. Otherwise,

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

under the uniformity assumption.

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

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

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

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

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

$$sl(Q) = sl(A) + sl(B) - sl(A)sl(B).$$

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

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

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

    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 $s = r \cdot sl(A)$ tuples. So, we require $\text{ceil}(s/f)$ block accesses, where f is the blocking factor. So, the expected cost is

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

    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 $s = r \cdot sl(A)$ 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

$$t + 1 + s = t + 1 + r \cdot sl(A).$$

    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 $s = r \cdot sl(A \geq x)$. So, we further access $\text{ceil}(s/f)$ blocks. In total, we have

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

  block accesses. This is a function of the selection selectivity.

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

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

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

When executing a join query, this is the number of block accesses corresponding to store the result back into memory. This can be used for further processing of the data when it doesn't fit in memory, for example.