Lecture 19 — Query Optimization

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 11, 2023

Query Optimization

There are now some clear ways we can transform an input query to determine which of the equivalent representations will have the lowest cost of execution.

It is fair to say that the query optimizer is likely to focus first on join relations since that is potentially the biggest area in which we can make some gains.

Query Optimization

Suppose our query involves a selection and a join.

We want to select the employee number, salary, and address for an employee with an ID of 385.

Suppose number and salary are in the employee table with 300 entries, and the address information is in another table with 12000 entries.

We have a join query, and if we do this badly, we will compute the join of employees and addresses, producing some 300 results.

Then we need to do a selection and a projection on that intermediate relation.

If done efficiently, we will do the selection and projection first, meaning the join needs to match exactly one tuple of employees rather than all 300.

Query Optimization

The query optimizer should systematically generate equivalent expressions.

It is likely that the optimizer does not consider every possibility and will take some “shortcuts” rather than brute force this.

One technique that helps on top of that is to re-use common subexpressions to reduce the amount of space used by representing the expressions during evaluation.

Estimating Statistics

In the previous example I used exact numbers, 300... 1... 12000... etc.,

But for the database server to get those it can either look them up, or it can guess about them.

As mentioned earlier, sometimes certain numbers, like the number of tuples in a relation, are easily available by looking at metadata.

If we want to know, however, how many employees have a salary between \$40 000 and \$50 000, the only way to be sure[1] is to actually do the query.

And we don't want to do the query when estimating the cost...

[1] Other than nuking it from orbit...

Guess we better... guess?

If we cannot measure, then, well, we need to guess. Our guesses are, however, not wild, but instead educated.

Estimates are based on assumptions, and those assumptions are very often wrong.

That is okay. We do not need to be perfect.

All we need is to be better than not optimizing.

And even if we pick the second or third or fifth best option, that is acceptable as long as we are close to the best option.

Cost Centres

There are five major areas where costs for actually performing a query accumulates.

1. **Disk I/O**
2. **Disk Additional Storage**
3. **Computation**
4. **Memory**
5. **Communication**

We will generally proceed on the basis that disk I/O is the largest cost and outweighs everything else.

Metadata

Some items that might be in the metadata:

- n_r: the number of tuples in a relation r
- b_r: The number of blocks containing a relation r
- l_r: the size in bytes of relation r
- f_r: the number of tuples of r that fit into one block
- $V(A, r)$: the number of distinct values in r of attribute A
- $h_{r,i}$: the height of an index i defined on relation r

There can also be metadata about index information as well... which might make it metametadata?

Maintaining Metadata

The more often it is updated, the more effort is spent updating it.

If every insertion or update or deletion resulted in an update, that may mean a nontrivial amount of time is spent updating this data.

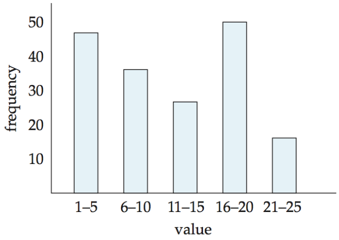
If we only do periodic updates, it likely means that the statistic data will be outdated when we go to retrieve it for use in a query optimization context.

Perhaps some amount of balance is necessary...

Histogram Data

A database may also be interested in keeping some statistical information in a histogram.

The values are divided into ranges and we have some idea of how many tuples are in those ranges.



Estimating Metadata

The above numbers are exact values which we can know and, hopefully, trust.

Although they could be slightly out of date depending on when exactly metadata updates are performed.

The more exact values we have, the better our guesses. But things start to get interesting when we ask something that does not have a category.

Estimating Sizes

As in the previous discussion of costs, we expect select operations and join operations to be the important ones.

They are the ones we will do the most of (probably).

We will start with select operations.

Single Equality Predicate

The base case to consider is a single predicate in a selection with an equality condition (e.g., province = “ON”).

The estimate depends on what we know about the attribute *A* referenced in the selection.

If it is a key field, and therefore unique, we know the select will return 1 result.

Unless, of course, it is not found and we will get zero results.

Single Equality Predicate

If values are evenly distributed, then we can expect that $n_r/V(A,r)$ tuples should be returned.

If there are ten possible values and everything is approximately evenly distributed then a select is expected to produce 10% of the tuples as its result.

The uniform distribution may not be accurate, but if it is a reasonable approximation of the data, it is simple and fast.

If a histogram is available, we can look in the range that contains that search value and guess based on the frequency count for that range instead.

Single Comparison Predicate

The next case is when we have a comparison like the salary greater than or equal to \$150 000.

We can, using the known maximum and minimum values of this attribute, take a guess.

If the requested value v is less than the minimum, we expect no rows returned.

If the requested value is greater than the max also zero.

Otherwise, the formula to estimate, where m is the minimum value and M is the maximum value: $n_r \times \dfrac{v - m}{M - m}$

Of course, if a histogram is available, we would prefer to use that.

Complex Predicates

For conjunction as well as disjunction, we will use a new symbol, s, which represents the selectivity of the a particular selection.

How likely it is that a particular tuple matches a condition.

The probability that a tuple satisfies θ_i is s_i/n_r.

Complex Predicates

For conjunctive selection, we want the probability that a certain tuple meets all of the criteria.

So the formula to estimate is:

$$n_r \times \frac{s_1 \times s_2 \ldots \times s_k}{(n_r)^k}.$$

Complex Predicates

For disjunctive selection it is the probability of matching an one of those conditions.

The calculation will be simplified by trying to compute 1 minus the probability it satisfies none of the conditions:

$$1 - (1 - \frac{s_1}{n_r}) \times (1 - \frac{s_2}{n_r}) \times ... \times (1 - \frac{s_k}{n_r})$$

Complex Predicates

Negation is pretty simple.

If we have a predicate $\neg\theta$ then the number of tuples likely to meet that is the number of tuples in the relation subtract those that do match θ.

Estimating Join Size

If we are computing a join where the relations have no attributes in common, then the “join” is really the cartesian product.

The number of tuples in the output will be the number of tuples in r_1 multiplied by the number of tuples in r_2.

That’s a lot.

Estimating Join Size

If the intersection of r_1 and r_2 is a key (unique value) for r_1 then we know that a tuple in r_2 can match with at most one in r_1.

So the maximum number of tuples that could occur is the number of tuples in r_2.

If the intersection is also defined as a foreign key on r_2 then it is exactly the number of tuples in r_2.

This also applies symmetrically (so you can swap r_1 and r_2 in the previous statements).

Estimating Join Size: No Keys

The difficult case occurs if the intersection of two relations is not a key in either of those two relations.

In that case, looking at the attributes in the relation, which we will call A, then the number of tuples in the join relation are:

$$\frac{n_{r_2}}{V(A, r_2)}.$$

Since we expect that they match with r_1 then we estimate that there will be:

$$\frac{n_{r_1} \times n_{r_2}}{V(A, r_2)}.$$

Estimating Join Size: No Keys

The denominator here depends only on r_2...

If you switched the positions of r_1 and r_2 we could get a different answer if $V(A, r_1)$ is not identical to $V(A, r_2)$.

That is unusual but not impossible.

If that is the case, we would choose the smaller of the two estimates.

Projection

Projection on an attribute A will result in either n_r tuples if duplicates are allowed, or $V(A, r)$ if they are not.

The relational algebra definition does not permit duplicates, mind you.

Aggregation

Suppose we seek an aggregation on some attribute B grouping by A.

Imagine in an example that B is salary and A is job title.

Then there will be one tuple in the output for each distinct value of A.

Set Operations

If a set operation has both operands as the same relation, it can be rewritten as a compound predicate and that would give us our estimates.

Those sorts of transformations are not especially difficult to imagine:

$\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$ is easily rewritten as $\sigma_{\theta_1 \vee \theta 2}(r)$.

Set Operations

If the set operation is on different relations then we could actually estimate it directly.

We will typically assume worst case scenario, so:

(1) union will be the size of the two relations,

(2) intersection will be the size of the smaller of the two relations, and

(3) in set difference we will assume in $r_1 - r_2$ that the result is the size of r_1 (i.e. no tuples removed).

Outer Join

Outer joins are fairly simple.

For a left outer join: $r_1 ⟕ r_2$ the estimated size is $r_1 \bowtie r_2$ plus the size of r_1.

Right outer join $r_1 ⟖ r_2$ is estimated as $r_1 \bowtie r_2$ plus the size of r_2.

The full outer join ⟗ is just the combined size of r_1 plus r_2.

Distinct Values

If we are interested in knowing how many distinct values of an attribute A there are, we have a few shortcuts based on the query.

The simplest possible case is if the query specifies that A equals some value a in which case there will be at most 1 distinct value of A.

If the predicate is complex and there are several allowed values $a_1, a_2, a_3 ... a_n$ then we can be certain the maximum number of possible values of A is n.

The last easy case is if there is a comparison, in which case the estimate is $V(A, r) \times s$ where s is the selectivity of the selection.

Distinct Values

Otherwise we will assume that the distribution of values of A has no relation to the distribution of values in the selection predicate.

The correct way to guess is through the use of probability theory.

We could also do a quick estimate of the smallest value of $V(A, r)$ (the most restrictive selection condition).

This same logic holds for estimating how many unique values will be returned in a join: the smallest condition will limit how many rows are in the relation.

Advanced Query Optimization

There are three more advanced techniques we will mention but not discuss deeply:

- Top-K
- Update Ordering
- Multiquery Optimization

Top-K

Consider a query that asks you to find the top *k* tuples based on some attribute, such as finding the top ten customers based on annual revenue.

We could potentially save some work if we know that no further results are required.

For something like top customers by revenue, we would probably need to find all the tuples that match, aggregate them, and then sort by income...

Update Ordering

Update queries often have a bunch of conditions restricting them to one tuple.

If that is the case, update optimization is not really an issue.

But if the update affects a number of rows, then there is a possible problem called the Halloween problem.

An updated tuple could be inserted into the relation ahead of where the scan is, meaning the tuple could be updated again incorrectly.

Update Ordering

The problem can be solved in a few ways.

One is to determine the tuples to be modified first before any updates take place.

Another has to do with splitting up the updates to be done into batches so that we don’t have to wait for the whole selection to be done.

Multiquery Optimization

The first way we can optimize multiple queries at once is to avoid repeating any operation that is used in more than one query.

That is to say, if query 1 and query 2 both have some part in common, the result of that subquery should be reused rather than recomputed a second time.

That would, of course, require that query 1 does not modify the data, which could cause a problem.

Multiquery Optimization

The second way to optimize when we have multiple queries is to do some things in parallel.

Suppose query 3 scans over a particular relation *r* looking for some attributes, and query 4 scans over the same relation *r* looking for something different

We could note the tuples matching queries 3 and 4 in one single pass rather than having to scan it twice.