Lecture 19 — Query Optimization

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 13, 2024

Imagine you are given an assignment in a course and you are going to do it now.

You will probably:

(1) Figure out what exactly the assignment is asking you to do;

(2) Figure out how you are going to do it; and finally

(3) Do it!

Feeling Lost?

We’re focusing on step two here.

The database is just a concrete example of the idea.

Query Processing

The procedure for the database server to carry out the query are the same:

1. Parsing and translation
2. Optimization
3. Evaluation

Adapt, Improvise, and Overcome

The DB server does not just execute a pre-planned series of steps.

Yes, it's executing the (unchanging) binary code...
But the path taken varies based on factors known only at runtime!

Steps Breakdown

Scan to figure out where the keywords are and what is what.

Check SQL syntax; then the names of attributes and relations.

Make a query graph, which is used to devise the execution strategy.

Follow the plan.

Do Skip Steps!

We will not spend time talking about the scanning, parsing, and verification steps of query processing.

A query with an error is rejected and goes no further through the process.

Start with SQL

Usually a query is expressed in SQL and that must then be translated into an equivalent relational algebra expression.

Complex SQL queries are typically turned into query blocks, which are translatable into relation algebra expressions.

A query block has a single select-from-where expression, as well as related group-by and having clauses; nested queries are a separate query block.

Select Your Choice

A query like `SELECT salary FROM employee WHERE salary > 100000;` consists of one query block.

We can select all tuples where salary is more than 100 000 and then perform a projection of the salary field of that result.

The alternative is to do the projection of salary first and then perform the selection on the cut-down intermediate relation.

Select with Subquery

```
SELECT name, street, city, province, postalCode FROM address
WHERE id IN (SELECT addressID FROM employee WHERE department
= 'Development');.
```

Then there are 2 query blocks, 1 for the subquery and 1 for the outer query.

If there are multiple query blocks, then they do not have to follow the same strategy; they can be optimized separately if desired.

Make the Plan

What we need instead is a query execution plan.

To build that up, each step of the plan needs annotations that specify how to evaluate the operation.

Make the Plan

This includes information such as what algorithm or what index to use.

An algebraic operation with the associated annotations about how to get it done is called an evaluation primitive.

The sequence of these primitives forms the plan, that is, how exactly to execute the query.

Time for Plan B

If there are multiple possible way to carry out the plan, the system will need to make some assessment about which plan is the best.

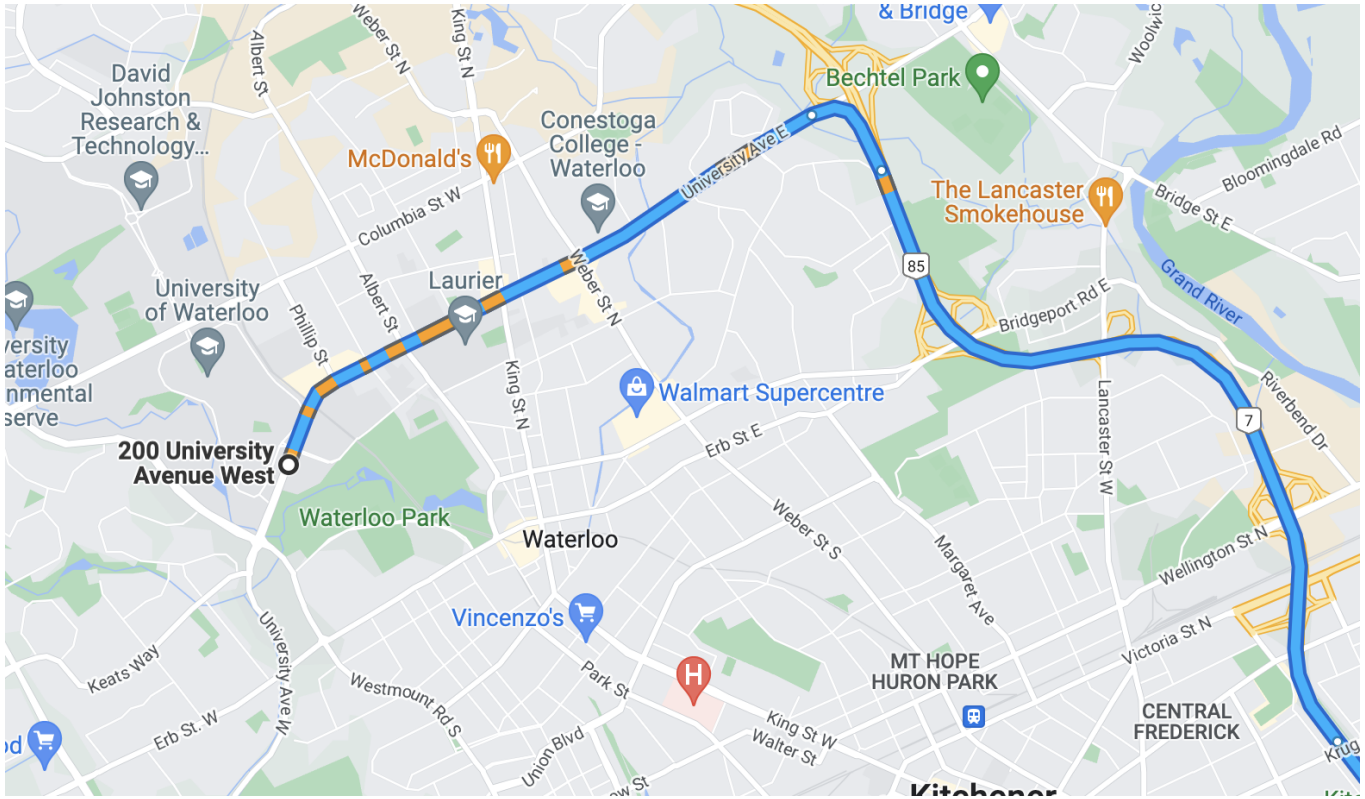
It is not expected that users will write optimal queries.

The database server should choose the best approach via query optimization.

Although maybe optimization isn't the right word...

At What Cost?

If you are asked to drive a car from point A to point B...



How does google present the time estimate here?

At What Cost?

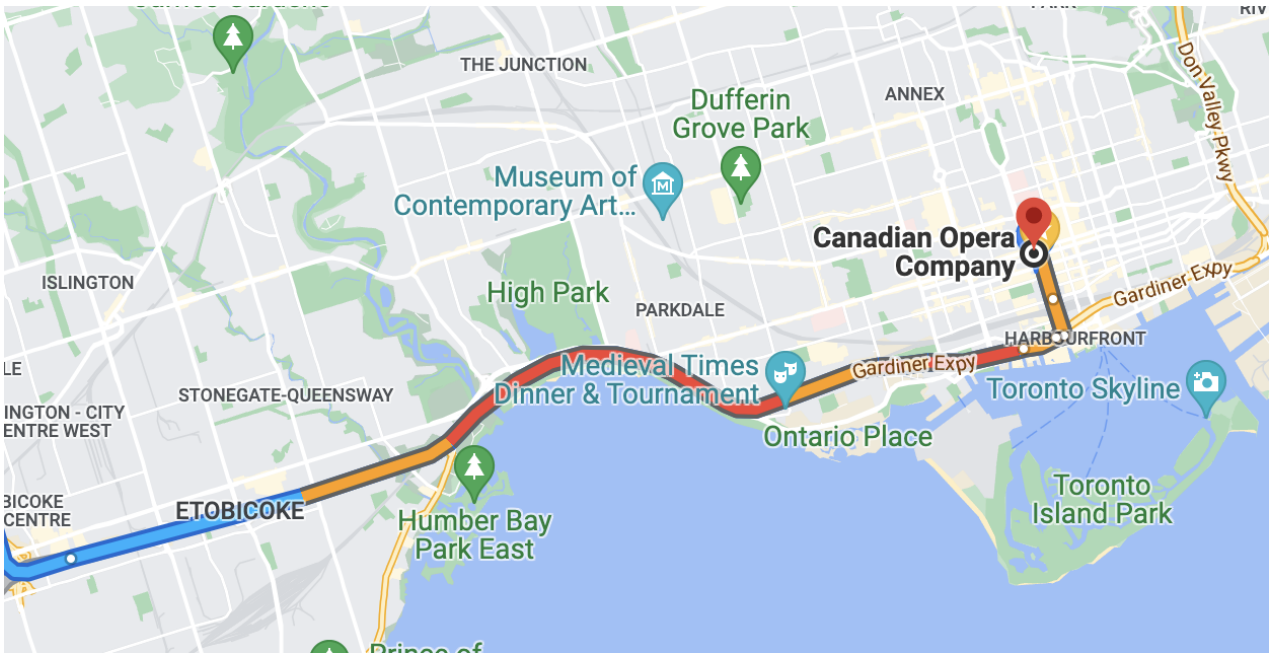
We need to break it down into different sections, such as drive along University Avenue, then get on Highway 85, then merge onto 401...

By combining all of the segments, you get an estimate of how long that particular route will take.

If you do this for all viable routes, you can see which route is the best.

Every Month is Bad Lane Change Month

If there is a crash on the highway, traffic really sucks and your decision that taking this particular route would be fastest turns out to be wrong.



Short of being able to see into the future, this is more or less inevitable.

Execution Time

Where does the time go in executing a query?

The biggest component is most likely loading blocks from disk, considering how slow the disk operations are.

In reality, CPU time is a nonzero part of query optimization, but we will ignore this for simplicity's sake and use only the disk accesses to assess cost.

Think Disk

The number of block transfers and the number of disk seeks are the important measures of interest here.

To compute the estimate of how long we think it will take to perform an operation, the formula is $b \times t_T + S \times t_s$.

For a hard drive, transfer times are on the order of 0.1 ms and seek times are about 4 ms.

Estimation Strategy Caveats

Usually we imagine the worst case scenario.

Estimates of Work

The estimates calculate only the amount of work that we think it will take to complete the operation.

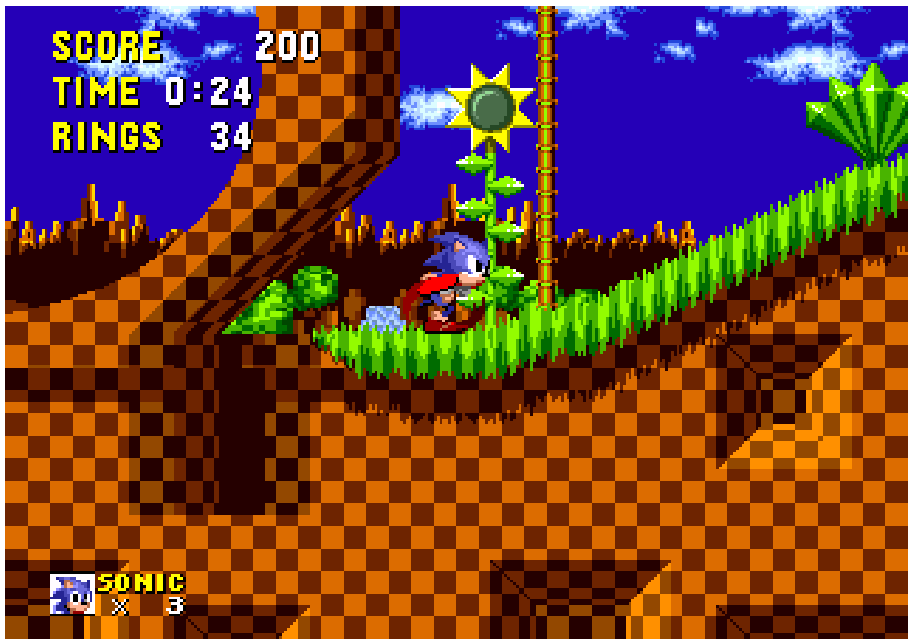
Unfortunately, there are several factors that will potentially affect the actual wall-clock time it takes to carry out the plan.

What do you think they are?

Estimates of Work

- How busy the system is
- What is in the buffer
- Data layout

Remember: the lowest cost approach is not necessarily the fastest!



Equivalents: Alternative Routes

There are many rules to transform a query into an alternative/equivalent.

We aren't going to learn the rules here.

Analogy: in math class you would have learned (at least for $\mathbb{N}, \mathbb{Z}, \mathbb{R} \ldots$)

$3 \times 10 \times 7$ is the same as $7 \times 10 \times 3$;

14×13 is the same as $14 \times 10 + 14 \times 3$.

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.

Bad approach: we will compute the join of employees and addresses, producing 300 results; then select and project on the intermediate result.

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.

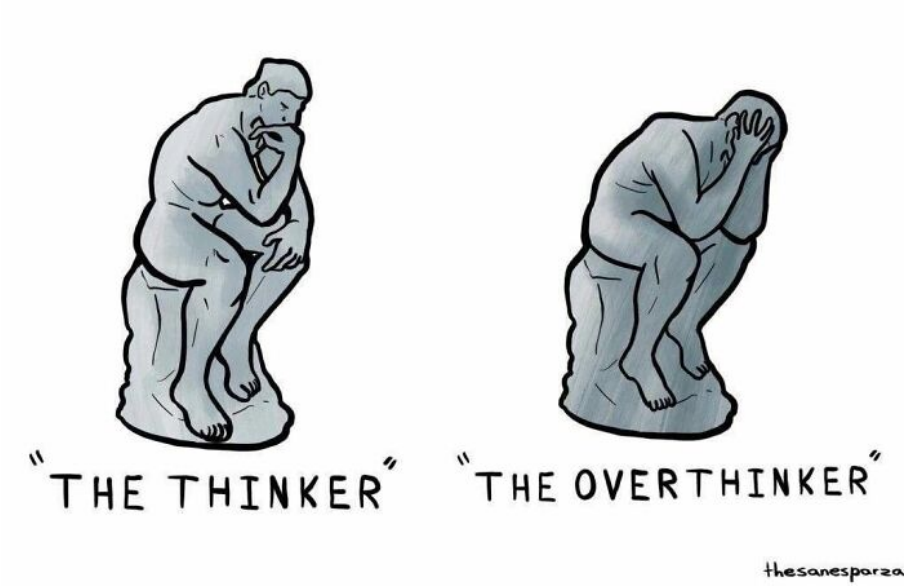
Query Optimization

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.

Idea: re-use common subexpressions to reduce the amount of space used by representing the expressions during evaluation.

Evaluation Plan Selection



These operations are not free in terms of CPU usage or time and it is possible to waste more time on analysis than choosing a better algorithm would save.

Evaluation Plan Selection—Join Focus

A simplified approach, then, focuses just on what order in which join operations are done and then how those joins are carried out.

The theory is that the join operations are likely to be the slowest and take the longest, so any optimization here is going to have the most potential benefit.

Focus on the Join

The order of joins in a statement like $r_1 \bowtie r_2 \bowtie r_3$ is something the optimizer can choose.

In this case there are 3 relations and there are 12 different join orderings.

For n relations there are $\frac{(2(n-1))!}{(n-1)!}$ possible orderings (some symmetric).

Why Are We Doing This?

Why joining so many tables? Is this a design issue?

It cannot examine all (non-symmetric) approaches and choose the optimal one. That would take too long.

Remember Your History

We can create an algorithm that can “remember” subsets of the choices.

Request $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4 \bowtie r_5$? Compute the best order for $(r_1 \bowtie r_2 \bowtie r_3)$.

Then re-use that repeatedly for any further joins with r_4 and r_5.

This “saved” result can be re-used repeatedly.

Make the DB Server Sad

The resultant approach may not be globally optimal (just locally optimal).

If $r_1 \bowtie r_4$ produces very few tuples, it may be maximally efficient to do that join computation first.

That will never be tried in an algorithm where r_1, r_2, and r_3 are combined to a subexpression for evaluation.

It’s Not Hard Data

Remember though, this is as estimating process.

May You Live In Interesting Times

The sort order in which tuples are generated is important if the result will be used in another join.

A sort order is called interesting if it is useful in a later operation.

May You Live In Interesting Times

Suppose r_1 and r_2 are being computed for a join with r_3.

It is advantageous if the combined result $r_1 \bowtie r_2$ is sorted on attributes that match to r_3 to make that join more efficient.

If it is sorted by an attribute not in r_3, an additional sort will be necessary.

Generalizations Are Always Wrong

The best plan for computing a particular subset of the join query is not necessarily the best plan overall.

That extra sort may cost more than was saved by doing the join itself faster.

This increases the complexity, obviously, of deciding what is optimal.

Fortunately there are, usually anyway, not too many interesting sort orders...

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

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

Guess we better... guess?

If we cannot measure, then, well, we need to guess.

No need to be perfect: all we need is to be better than not optimizing!

Cost Centres

Areas where costs for performing a query accumulate:

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

Assumption: disk is the largest one.

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.

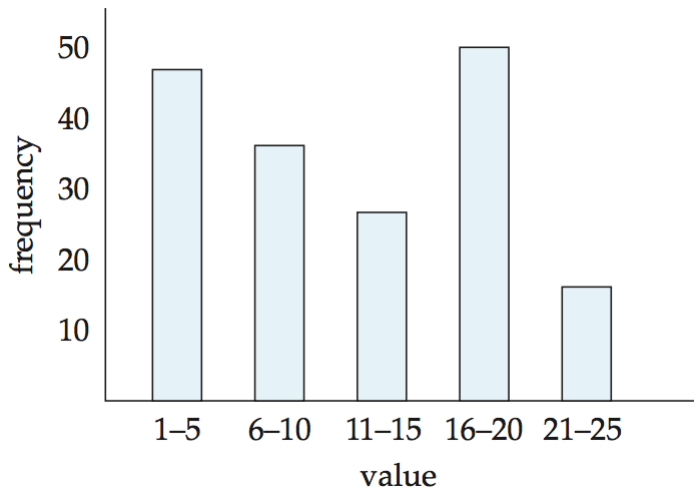
If we only do periodic updates, it likely means that the statistic data will be outdated when we go to retrieve it for use in a query optimization context.

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.

Join Elimination

Slight digression on Join Elimination.

Attempt to eliminate the join altogether if it can be skipped.

The optimizer can only do this if there is certainty that the outcome will not be affected by not doing the join.

Join Elimination

You may ask, of course, why should the optimizer do this work at all?

Why not simply count on the developers who wrote the SQL in the first place to refactor/change it so that it is no longer so inefficient?

Grind leetcode (please no) and git gud (seriously wtf).

Regardless, SQL is a language in which you specify the result that you want, not specifically how to get it.

Compilers don’t admonish the user for writing code that it has to transform into a faster equivalent, they just do that transparently.

Join Elimination Example

Query: `SELECT c.* FROM customer AS c JOIN address AS a ON c.address_id = a.address_id;`

Inner join; as presented, cannot eliminate.

We need to make sure that the customer data has a matching row.

Eliminate the Join

Suppose that we have a foreign key defined from customer's `address_id` to the address `id` field.

If nulls are not permitted, then we know for sure that every customer has exactly one record in the address table.

Therefore the join condition is not useful and may be eliminated.

New query: `select * from customer;` faster (select without conditions).

More Elimination

The foreign key and not null constraints on the address ID field of the customer enable elimination.

An outer join constraint can be removed as well.

Imagine the query said this: `SELECT c.* FROM customer AS c LEFT OUTER JOIN address AS a ON c.address_id = a.address_id;`.

All tuples are fetched from customer whether or not there is an address.

Or Don’t...?

If, however, both constraints are removed, and we cannot be sure that there is at most one address corresponding to a customer…

Then we have to do the join.

Harder to Read

Obviously, the more complex the query, the harder it is to determine whether or not a particular join may be eliminated.

The same queries written on a database in which the constraints have not been added would not be eligible for the join elimination optimization.

This reveals a second purpose why constraints are valuable in the database.

Join Elimination Analogy

Task: find all copies of the book “Harry Potter and the pthread House Elves”.

Rule: this library will keep only one copy of that book ever.

As soon as you have found the single copy of that book, you can stop looking.

Shortcuts

Now we will talk about some heuristic rules (guidelines, really) that we have definitely mentioned earlier.

Perform selection early

No surprises here: the sooner we do a selection, the fewer tuples are going to result and the fewer tuples are input to any subsequent operations.

There are exceptions, however: if we throw away too much, it forces us to do a full table scan to complete a join.

Perform projection early

Analogous to the idea of doing selection early, performing projection early is good because it tosses away information we do not need.

Just like selection, however, it is possible the projection throws away an attribute that will be useful.

Set Limits

Another strategy for making sure we choose something appropriate within a reasonable amount of time is to set a time limit.

Optimization has a certain cost and once this cost is exceeded, the process of trying to find something better stops.

But how much time to we decide to allocate?

Plan Caching

In any busy system, common queries may be repeated over and over again with slightly different parameters.

A student wishes to query what courses they are enrolled in.

If one student does this query with a particular value for student ID number, we can re-use that same evaluation plan in the future.

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.

Applicability

If we're not implementing a database is this still useful?

Yes—the database is just an example!

Real lesson: how to programmatically generate, evaluate, and choose amongst alternatives.