Lecture 19 — Query Optimization

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 6, 2022

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!

Query Processing

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

1. Parsing and translation
2. Optimization
3. Evaluation

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.

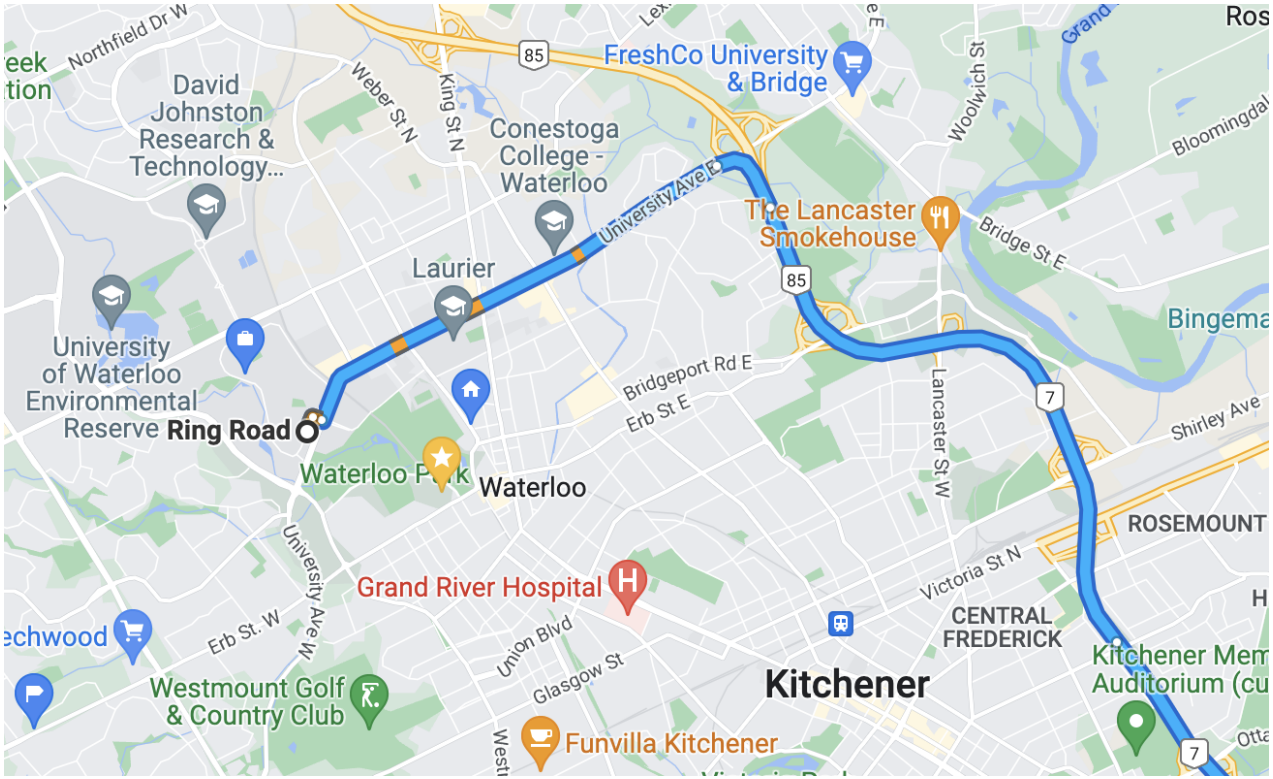
It is not expected that users will write optimal queries.

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?

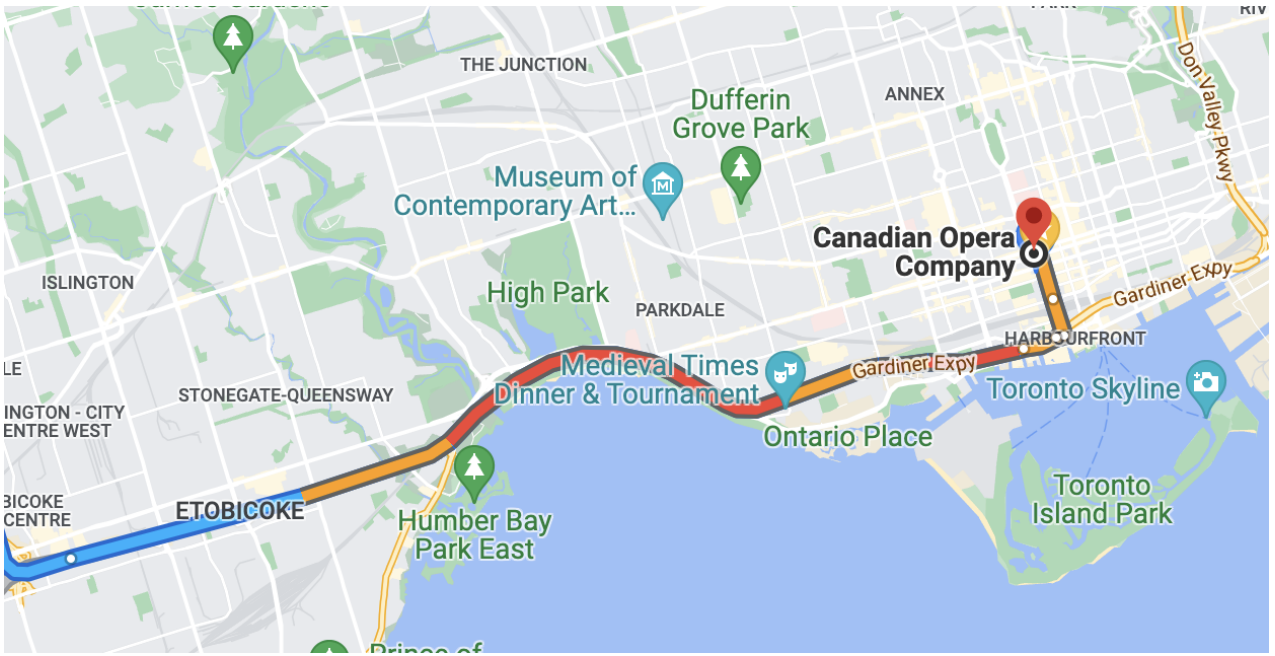
We need to break it down into different sections, such as drive along University Avenue, then get on Highway 85, then merge onto 401...

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.

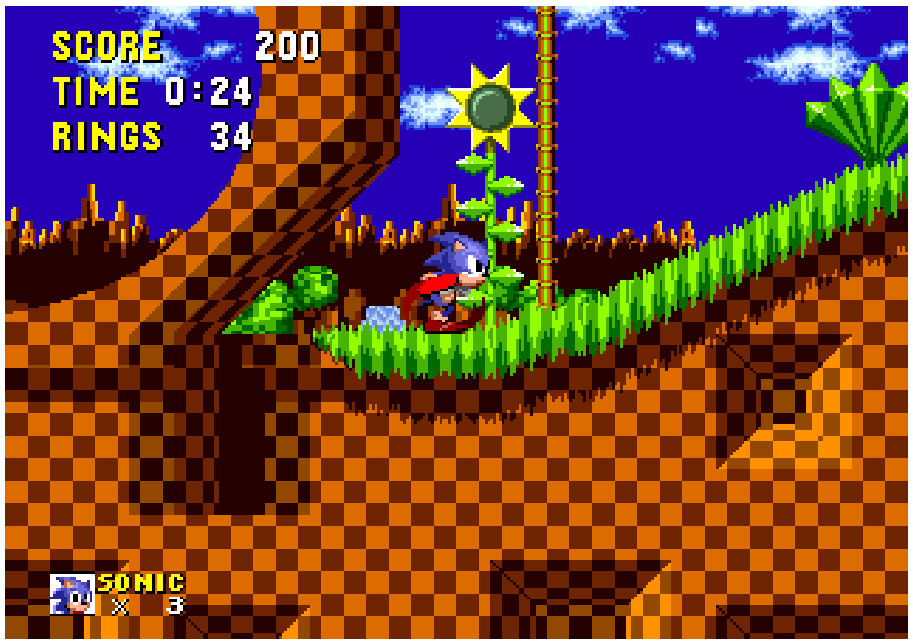
Unfortunately, there are several factors that will potentially affect the actual wall-clock time it takes to carry out the plan.

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!



Query Optimization

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.

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.

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 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.

Estimates are based on assumptions; 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.

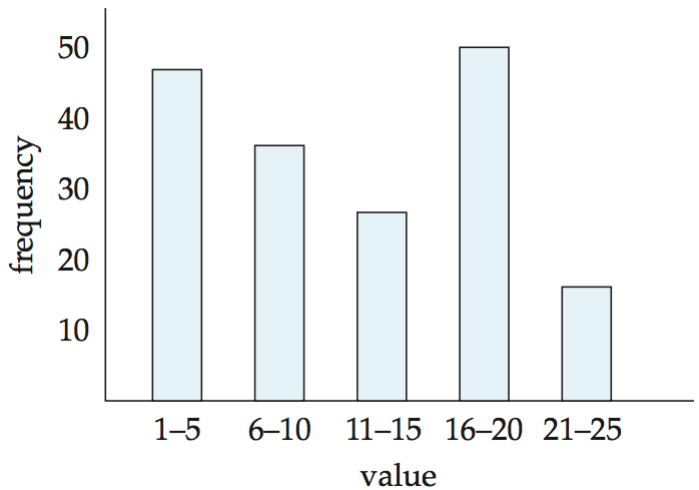
If we only do periodic updates, it likely means that the statistic data will be outdated when we go to retrieve it for use in a query optimization context.

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.

Let us focus now on Join Elimination.

So much of the previous examination has focused on the cost of the join and that has highlighted in a real way just how expensive it is to perform a join.

For this reason, good optimizer routines will 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.

We will shortly see how that is accomplished.

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?

That would be nice but would you also like a pony?

Join Elimination

Developers make mistakes, as you know, or perhaps some legacy code cannot be changed for some reason.

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

If there is a more efficient route, then it's worth taking from the point of view of the database server.

If you ask for some operation that the compiler knows it can replace with an equivalent but faster operation, why wouldn't you want that?

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

We will examine some real SQL queries to see how we can get rid of a total unnecessary join.

This join can only be removed if the database server can prove that the join is not needed.

Therefore the removal of this operation has no impact on the outcome.

Join Elimination Example

Consider a statement that looks like this: `SELECT c.* FROM customer AS c JOIN address AS a ON c.address_id = a.address_id;`

This gets customer information and joins with those where there are addresses on file.

This is an inner join and as presented simply we cannot do anything with this information.

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.

This means we could in fact replace that query with `select * from customer;` with no need for any references to the join table at all.

That would be much, much faster since it is a simple select with no conditions.

More Elimination

The foreign key and not null constraints on the address ID field of the customer make it possible for the optimization of the join elimination to occur.

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 associated address.

Or Don’t...?

Suppose the foreign key constraint is removed.

Does that change anything? No – a unique constraint on the address would be sufficient.

Therefore it can once again be replaced with the simple `select * from customer;`.

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.

Eliminate a Wilder Join

When an outer join occurs with distinct keyword.

In a many-to-many relationship (the example in the source material is about actors and films) then outer join would produce duplicate tuples.

The query asks for a listing of actor names.

It is an outer join query (which makes no sense...)

Because it’s an outer join you will even return the actors who appear in no films.

Bad Example is Bad?

Why would you query all actors whether or not they had been in a film by referencing films?

If you don’t care whether they had been in a film, why do you even look at the films table...

Anyway, this sort of thing could happen in an application where the SQL statement is composed by some if-statement logic.

E.g., checkboxes like “appears in a film” and “does not appear in film” and two conditions are added (like “incoming = true OR incoming = false”).

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.

In the inner join example, the foreign key and not null constraints, for example, are beneficial.

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

Join Elimination Analogy

You are asked to search through the library to find all copies of the book “Harry Potter and the pthread House Elves”.

That is a plausible task.

But, suppose that you know as well there is a rule that this library will keep only one copy of that book ever.

If that is the case, as soon as you have found the single copy of that book, you can stop looking (no need to check more “just in case”).

This sort of optimization is very similar in that the rules let us avoid doing unnecessary work and that is a big part of the optimization routine.

Evaluation Plan Selection

It was perhaps oversimplifying to have said earlier that choosing a plan was just as simple as picking the one with the lowest cost.

There is a little bit more to it than that.

There about choosing the one with the lowest cost is correct (generally) but the difficulty is in devising and calculating all possible evaluation plans.

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

We already know that 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.

In fact, for n relations there are $\frac{(2(n-1))!}{(n-1)!}$ possible orderings.

Some of them, are obviously symmetric which reduces the number that we have to calculate, since $r_1 \bowtie r_2$ is not different from $r_2 \bowtie r_1$.

In any case, even if we can cut down the symmetrical cases the problem grows out of hand very quickly when n gets larger.

Why Are We Doing This?

Once more than three relations are affected by a join query it may be an opportunity to stop and think very hard about what is going on here.

This is quite unusual if the database design is good.

The database server may want to ask why do you have a join query that goes across six or eight or twelve relations.

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

Remember Your History

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

If we have, for example, $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4 \bowtie r_5$, we can break that down a bit.

We could compute the best order for a subpart, say $(r_1 \bowtie r_2 \bowtie r_3)$.

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

This “saved” result can be re-used repeatedly turning our problem from five relations into two three-relation problems.

Make the DB Server Sad

This is a really big improvement, actually, considering how quickly the factorial term scales up.

The trade-off for this approach is that the resultant approach may not be globally optimal (but instead 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.

The previous statement that said $r_1 \bowtie r_4$ produces very few tuples as if it is a fact.

The optimizer does not know that for sure and must rely on estimates where available.

Dynamic Programming Join Optimization

A simple pseudocode algorithm for using dynamic programming to optimize join orders is below.

In this, imagine that there exists a structure `result` that contains both a plan and a cost element.

This result is stored in some array or other data structure for future retrieval.

This recursive algorithm has $O(3^n)$ behaviour which is... well... it's not going to win algorithm of the year.

Dynamic Programming Join Optimization

```
procedure find_plan( subquery S )
  if current subquery S result has already been computed
    return previously computed result for S
  end if

  declare variable result

  if current subquery S contains no joins
    set result.plan for S to best way of accessing this relation
    set result.cost for S this relation based on plan
  else
    for each non empty subset S1 of current relation S that is not equal to S
      variable r1 = find_plan( S1 )
      variable r2 = find_plan( S - S1 )
      A = best algorithm for joining r1 and r2
      cost = r1.cost + r2.cost + cost of A
      if cost less than current best plan for S
        result.plan = execute r1, execute r2, join using A
        result.cost = cost
      end if
    end for
   end if
return result
```

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.

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 some attribute not in r_3 that means an additional sort will be necessary.

Generalizations Are Always Wrong

With this in mind it means that 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...

Generating Alternatives

Join order optimization is a big piece of the puzzle but it's not the only thing we can do in query evaluation.

Let's briefly revisit the subject of how equivalent queries are formed.

We already decided it is too expensive to try out all equivalent queries, but perhaps we are determined to try to at least generate lots of alternatives.

To Generate Alternatives

1. A way of storing expressions that reduces duplication and therefore keeps down the amount of space needed to store the various queries.
2. A way of detecting duplicate derivations of the same expression.
3. A way of storing optimal subexpression evaluation plans so that we don't have to recompute them.
4. An algorithm that will terminate early the evaluation of a particular plan if it is already worse than the cheapest plan so far found.

Nested Subqueries

If possible, nested subqueries will be transformed into an alternative representation: a join query.

To summarize the rather long story, if evaluated the “slow” way the subquery needs to be run a lot of times.

Thus, to make it faster, we would prefer to turn it into a join (which we already know how to handle).

If really necessary we can run the subquery once and use that temporary relation in a join (where exists or “in” predicates may fall into this category).

Shortcuts

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

We talked about, for example, how to perform a selection.

Now we can actually discuss them more formally.

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.

Performing the selection is almost always an improvement.

Chances are we get a lot of benefit out of selection: it can cut a relation down from a very large number of tuples to relatively few (or even one).

Perform selection early

There are exceptions, however.

Suppose the query is $\sigma_\theta(r \bowtie s)$ where θ refers only to attributes in s.

If we do the selection first and:

(1) r is small compared to s and

(2) there is an index on the join attributes of s but not on those used by θ

...then the selection is not so nice.

It would throw away some useful information and force a scan on s.

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.

Left-deep join orders

Some query optimizers do not bother doing all the fanciful join optimization routines to solve which joins are best.

Instead they will consider join orders where each of the right operands of the join is always one of the initial relations r_k from the query $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$.

The reasoning behind this is it takes "only" $O(n!)$ time to consider all left-deep orders rather than all possible join orders.

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.

Another student will repeat the exact same query with her student ID number instead.

The results will be different and this query may be more expensive on the second run.

That is expected, all we really needed was an estimate.

To wrap up the topic of query optimization we'll have a video.

The talk is entitled "How Modern SQL Databases Come up with Algorithms that You Would Have Never Dreamed Of" by Lukas Eder:

`https://www.youtube.com/watch?v=wTPGW1PNy_Y`