1 Introduction

Any programmer knows that optimization is important. It makes code run faster, which is a good thing. It is thus no surprise that optimization is used not only for source code, but also a lot of other things. In databases, for instance, optimization is crucial.

Databases are all about data: we insert new data and modify or delete existing data. But most importantly we query databases to give us answers about questions we have regarding the data it contains. Research has been very active in this area and found solutions to make all these things faster by making use of optimizations. Because, however, queries are generally more often issued than say insertions or modifications, special attention was and still is paid to query optimization.

In this summary we will try to give a gentle introduction to this very broad field. We will mostly focus on relational databases, as most of existing research did as well and because those are still the most used database types.

2 Looking inside Query Processing

Whenever a query is run on a relational database, it goes through a series of steps, at least ideally (actual implementations might not make a clear distinction between these steps). In this summary we assume that queries are given in SQL. Usually, these steps (as seen in figure 1) are the following:

Parser First your query is parsed. This checks if it is well-formed and valid according to the grammar rules of SQL. If not your query will be refused and you receive an error. On the other hand, if everything went well the parser will output an abstract syntax tree that you might remember from your compiler class. In some cases it will also simplify or reformulate elements of the SQL language to a more formal language, called relational algebra.

Optimizer This AST is then forwarded to the optimizer. The optimizer’s job is it to transform the AST it receives into an execution plan that contains details about how the query should be carried out. Normally, there are many such plans for a single query and the optimizer should find a good one (even
better the optimal one). Choosing a good plan is complicated and avoiding very bad plans is sometimes good enough. The plan the optimizer has to find includes details such as which indices to use for best performance or which order of operations can be evaluated most efficiently; more about this later.

**Code Generator/Interpreter** The code generator’s job is to carry out the plan it receives from the optimizer. Most often this is implemented in the form of an interpreter that goes through the plan and calls the appropriate functions in the query processor.

**Query Processor** The query processor is the entity that is doing the hard lifting. It receives instruction from the interpreter and carries them out.

```
Parser  AST  Optimizer  query plan  Code Generator/Interpreter  call sequence  Query Processor
```

Figure 1: Steps of query processing

### 3 The Query Optimizer

The query optimizer can again be split up into several modules (see figure 2). This separation simplifies the discussion about the optimizer, actual implementations might very well deviate from this model.

First of all, the optimizer is comprised of two different optimization steps, the *declarative* optimization step and the *procedural* optimization step.

We first explain what differentiates them: unlike the procedural step, the declarative step does not have access to statistics or costs. This means that it must restrict itself to optimizations that always or very likely bring an improvement. In other words, it is “static” and the optimizations it does resemble a little bit those done by source code compilers. We will now consider both steps in detail.

### 4 Declarative Optimization

As you can see in figure 2 the only declarative module is the rewriter. The query rewriter takes a query (in the form of the AST) and transforms it into an
equivalent query that rewriter thinks will run faster or is better optimizable in a later step.

4.1 Folding

One of the jobs of the rewriter might do is to solve expressions that can be evaluated without accessing data in a table, such as simple arithmetic or logical expressions. We have such a case in the listing below where a constant expression is evaluated to its result.

```
SELECT 1 + 2 + 3  
→ SELECT 10;
```

The rewriter can also eliminate superfluous clauses. Consider the following query:

```
SELECT * FROM table WHERE column < 300 AND column < 500;
```

The second part of the conjunction has no effect, so this could be simplified to

```
SELECT * FROM table WHERE column < 300;
```

Disjunctions can be simplified in a similar fashion,

```
SELECT * FROM table WHERE column < 300 OR column < 500;
```

can be rewritten to

```
SELECT * FROM table WHERE column < 500;
```

Of course, these are only very simple examples. Beyond these kinds of simplifications, sometimes the rewriter is used to bring the AST into a specific form that is expected by later modules. In SQLite for instance BETWEEN expressions are eliminated and transformed into AND expressions like this:

```
expr1 BETWEEN expr2 AND expr3  
→ expr1 >= expr2 AND expr1 <= expr3
```

Also, a series of disjunctions on the same column are replaced with a more efficient IN. I.e.

```
column = expr1 OR column = expr2 OR column = expr3 OR ...  
→ column IN (expr1, expr2, expr3, ...)
```

IN is more efficient because it can exploit (possibly existing) indices.

4.2 Semantic Query Rewriting

A variant of this is called semantic query rewriting ([6]). The idea behind it is rather simple: modern DBMS allow it to define the precise domain of the values in a column (e.g. that all values in a column are smaller than 100) or to enforce custom checks and constraints. In semantic query rewriting this information is used to rewrite queries. Again an example says more than thousand words. Assume we impose the following constraint (in pseudo-SQL as syntax varies from DBMS to DBMS) that states that senior programmers per definition earn more than 100k:

```
assert sal−constraint on emp:  
sal > 100K where job = "Sr..Programmer";
```
Further, assume we want to run the following query:

```sql
SELECT name, floor
FROM emp, dept
WHERE emp.dno = dept.dno AND job = "Sr. Programmer";
```

Now, something interesting happens, the query rewriter adds an additional clause and produces the following.

```sql
SELECT name, floor
FROM emp, dept
WHERE emp.dno = dept.dno AND job = "Sr. Programmer" AND sal > 100K;
```

You might wonder why an additional clause is an optimization: so see this, consider the case where we do not have any indices (so none on the `job` column) but one on the `sal` column. The optimizer can now take advantage of this situation, by exploiting the `sal`-index in e.g. a nested loop join.

Semantic rewriting is seldom used, so are constraints and checks in general. Also, with the trend of using Object Relation Mappers constraints are defined on a higher level and the optimizer cannot access this information.

### 4.3 Query Flattening

A more advanced and widely deployed form of rewriting is *query decorrelation* and query *query flattening* or *unnesting*. Both methods have to do with subqueries. SQL allows subqueries to appear at different positions in a query.

We first consider the case where a subquery appears in the `FROM` clause like in the listing below:

```sql
SELECT * FROM
  (SELECT * FROM data WHERE a = 10) AS s
WHERE s.b = 20;
```

Thinking about this you might see that this subquery can very well be flattened into it’s hosting main query. This is also what most modern DBMS will do. The result will then look like this:

```sql
SELECT * FROM data WHERE a = 10 AND b = 20
```

The benefits of this are twofold: first it gets rid of the subquery and thus avoids the creation of an intermediate table. Secondly, the resulting query is nice and flat which allows later optimization steps to improve things further.

You might ask if this kind of rewriting is always possible. Unfortunately the answer is no. The details of when it is possible and when not are rather complicated and are not covered in this summary. What I can tell you is that the flattening of `FROM`-subqueries is for instance not possible if they contain advanced SQL elements (or specific combinations of them) such as `OFFSET`, `LIMIT`, `DISTINCT` etc. because then often there exists no equivalent flat query for the nested query; then the DBMS for better or for worse has to create a temporary table, which hurts performance, because such tables lack any indices. SQLite has a detailed list of conditions that allow flattening (see [3]), in case you want to know the details.
4.4 Query Decorrelation and Unnesting

As we have seen above subqueries in \texttt{FROM} clauses can cause the creation of intermediate tables, which of course slows down query execution. Much worse, however, are subqueries in \texttt{WHERE} clauses if they are correlated. Correlated means that the subquery makes references to the outer query. You can see such a query in the listing below:

\begin{verbatim}
SELECT C. cust_last_name, C. country_id
FROM customers C
WHERE EXISTS (SELECT 1 FROM sales S
  WHERE S.quantity_sold > 1000 AND
  S.cust_id = C.cust_id);
\end{verbatim}

An \texttt{EXISTS} with a subquery is a pattern that is often used. The problem with this query though is, as already stated, that it is correlated because the inner query (i.e. the subquery) refers to \texttt{C.cust.id} from the outer query. Without any further optimization, the DBMS would need to run the inner query for every tuple of the outer query. That is, of course, terribly inefficient. Fortunately, however, modern DBMS implement a technique called \textit{query decorrelation}. What it does, as the name suggests, is that it transforms the inner query from a correlated subquery into an uncorrelated one. There are rules for doing that for most common correlated subqueries. We will only show the \texttt{EXISTS} case, in which the rewriter would replace the correlated \texttt{EXISTS} with a uncorrelated \texttt{IN} as follows:

\begin{verbatim}
SELECT C. cust_last_name, C. country_id
FROM customers C
WHERE C.cust_id IN (SELECT S.cust_id FROM sales S
  WHERE S.quantity_sold > 1000);
\end{verbatim}

Once a query has been decorrelated, it is tried to get rid of the subquery for good and replace it with a join, this is called \textit{query unnesting}. In this special case we can indeed do that, as you can see in the listing below.

\begin{verbatim}
SELECT C. cust_last_name, C. country_id
FROM customers C, sales S
WHERE S.quantity_sold >= 1000
  AND C.cust_id SEMI JOIN S.cast_id;
\end{verbatim}

Instead of using a simple join we use a semi join. The semantics of a semi join are very similar to a classical inner join, except for one thing: once a tuple from the outer relation has been successfully joined with a tuple from the inner relation there is a skip to the next tuple in the outer relation: in short, an outer tuple is joined at most with one inner tuple, namely the first one that matches. This speeds things up and is exactly what we want: if no corresponding inner tuple (\texttt{sale}) is found, we don’t want the outer tuple (\texttt{customer}) in the output relation. On the other hand we do want it in the output relation, if it can be joined with at least one tuple from the inner relation. A transformation into a join is not always possible. Then the DBMS is constrained to create an intermediate table from the subquery.
5 Procedural Optimization

Now we come to the procedural part of the optimizer. The main part of it is the planner. The planner is given the rewritten query by the rewriter (mostly in form of an AST). It’s job is it to find a good query plan for the given query. A query plan is a detailed description of how the database engine is supposed to carry out the query: which indices should it consult, or should better do a linear scan?, which join methods should it use?, which order should joins be evaluated in?, these are some questions that a query plan should give answers to.

Unfortunately, the number of possible query plans even for a single and simple query is huge; in fact finding the best query plan has long known to be a NP-hard problem.

The challenge that the planner is facing is thus to find a good query plan from all the possible combinations of query plans. This can be seen as a classical search problem. As you can see in figure 2 there are several other modules in the procedural optimization steps which all are interacting with the planner in one way or another.

First, we have the algebraic space and the method-structure space, both of which form the planner’s search space, as their name might suggest. The algebraic space consists of all possible ways the query can be expressed by means of algebraic transformations: for example for specific operations (such as joins) the operands can be reordered or expressed in terms of other operations. These transformations can alter the performance characteristics of a query.

Secondly, we have the method-structure space, which includes all possible ways to execute the query by e.g. using different algorithms or methods for carrying out a specific operation. An example of this would be different strategies to access an attribute of a relation (e.g. index vs. sequential scan).

Finally, we have two more modules. The cost model tries to estimate the cost of a specific query plan. This part is obviously crucial: without the cost model the planner has no idea if one plan is better than the other. To assess the cost of a plan, the cost model considers various factors. The two most important are I/O costs (e.g. sequential vs. random access) and the size and value-distribution of relations that appear in a query plan (the larger a relation the higher the cost, of course). Estimating relation sizes and distribution is usually delegated to a proper module, called the size-distribution estimator about which we will hear more later.

5.1 System-R style Planners

One of the first DBMS that employed advanced query optimization techniques was System R by IBM which also was one of the first systems to implement SQL. The principles of the planning algorithm used by System R are still found in many modern DBMS. In this section we will look at simplified version of it that builds on the so-called select-project-join model. In this model possible queries are comprised of the following three operations: select (σ), project (π) and join (⋈).

Remember that select filters tuples for a specified condition, project reduces the “width” of tuples, taking only specified attributes. Finally, join joins two relations based on a given join condition.
As already stated above, SQL queries are parsed and turned into an abstract syntax tree in a previous step. In our simplified setting these trees will look like what you see in figure 3, which shows three semantically equivalent representations of the query in the listing below. The following figures and examples are all taken from [5].

**SELECT** C. cust_last_name, C. country_id
**FROM** customers C
**WHERE** C. cust_id IN (SELECT S. cust_id FROM sales S
                      **WHERE** S. quantity_sold > 1000);

Query 1

As said earlier, the problem we are having is the huge search space. One reason for this being that operations have properties that allow for different tree representations. Commutativity and associativity are two of these properties that hold for joins. Whenever we have to join two relations $R_1$ and $R_2$, we can either use $R_1$ as outer relation and do $R_1 \bowtie R_2$ or vice-versa and do $R_2 \bowtie R_1$. Depending on method-structure properties, one might be much more efficient than the other. Similarly, associativity allows us to arbitrarily choose the order in which we want to carry out joins. I.e. we have $(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3)$.

### 5.1.1 Search-Space Pruning

A classical way of coping with a large search space that is pruning. If we know that certain plans are not optimal we can remove them upfront. System R optimizers do exactly that. There are three “restrictions” that are applied to the search space. Restriction 1 and 2 will never remove the optimal plan. Restriction 3, on the other hand, is a heuristic and very well might remove the optimal plan and is sometimes not applied.

**Restriction 1** considers selections and projections. First, plans in which selections appear above joins are removed from the search space. The reason for this should be obvious: if we do the selection before the join, the input relations for the join become smaller and thus increasing the overall performance of a query. Secondly, selections are often not executed as part of a near join operation and not carried out separately. For instance in figure 3 the tree T2 is removed because the selection is above the join. We also remove plans that
have unnecessary projections. This, for instance, removes T3 from figure 3. In the left branch of the tree we see two selections done on the EMP relation, the first being pointless. After restriction 1 the search space is still $\Omega(n!)$, where $n$ is the number of relations involved in the query.

**Restriction 2** is rather simple. It will remove all plans that contain a cross join, unless explicitly asked in the query. A cross join can always be avoided by using ordinary inner joins.

**Restriction 3** Restriction 3, a heuristic, removes all plans where an intermediate results appears as an inner operand in a join. Figure 4 shows different ways to associate multiple joins. T1 is called *left deep*, T2 *right deep* and T3 is called *bushy*. What speaks for R3 is that often the optimal left deep plan is not much slower than the overall optimal plan. Further, left deep tree have a series of nice properties: first, since the inner relation is not intermediate, the join algorithm can exploit all the indices that might exist on the inner relation; second it allows pipelining of the outer relation. Restriction 3 greatly reduces the search space to $O(2^n)$.

### 5.1.2 Dynamic Programming Search

After pruning we have no other choice but to search the remaining search space. The algorithm used for this in System R is a dynamic programming algorithm, that is sketched below. Before we come to it though, we need to define what an *interesting order* is. An example explains it best: assume that at some point we need to join two relations. We consider using a merge join to do the job which means that we need to sort the input relations. The input relations might themselves be the result of an operation. If such an operation happens to produce an input relation already sorted in such a way that the merge join does not have to resort it we speak of an *interesting order*, an order on a relation that could be useful for a later operation. The point is that it might be better to use a suboptimal method to produce the input relations if it yields an interesting order instead of using the optimal one which would require an additional sorting step and thus all in all be slower.
Back to the dynamic programming algorithm: **First**, the algorithm will find all possible ways to access each relation involved in the query considering sequential scan or access via an index if adequate. Next it will ask the estimation module to determine a cost for these access strategies. Once done that it will keep for each relation only the lowest cost access strategy and any strategy that produces an *interesting order*.

**Then** for each pair of input relations $R_1$ and $R_2$ that are to be joined it searches the best way to evaluate them; this might start a recursion. Once found we determine the best way to join $R_1$ and $R_2$ considering join methods and access costs or prune the plan if appropriate. The cost for joining $R_1$ and $R_2$ is cached, so it does not have to be recalculated again.

This algorithm is guaranteed to find the best plan in the (pruned) search space. The runtime of this algorithm, however, is exponential. This seems problematic, but since the number of relations is usually rather small this normally is not an issue, at least up to around 10 joins.

### 5.2 SQLite’s Planner

An interesting take at query optimization is found in SQLite. It models the problem as a graph problem (see figure 5).

In the graph each vertex represents a relation. There is an arc between two vertices $u$ and $v$ if the corresponding relations are to be joined. The labels on the arcs represent the logarithmic cost of the join. Now the problem of finding an optimal plan can be stated as finding a minimum cost path through the graph such that each vertex is visited exactly once. SQLite does not optimally solve this problem, but uses a simple heuristic: the graph is traversed only once, always picking the lowest cost edge. This way SQLite rarely finds the best plan, but it seems good enough to avoid very bad plans. Moreover, SQLite has usually no problem with queries that contain more than 10 joins.

### 5.3 Randomized Planners

Because performance of the dynamic programming algorithm is bad if a large number of joins are involved in a query, research has been done to approach the problem from different directions. This includes a wide number of classical ways to solve optimization problems in computer science. One such method is *random hill climbing*, where starting from an arbitrary query plan, small changes are made to it. Then for every such change it is tested whether it represents an improvement. If so it is made the new starting point for the next iteration. The *random* in its name means that with a certain (small) probability, the algorithm will also select plans which do
not bring an improvement or even are a little worse than a previous plan. This is done to not get stuck in a possible local optima.

A prominent variation of this that has also been used for query optimization is simulated annealing. It is quite similar to random hill climbing except that the probability for picking a non-improving plan is determined by a cooling process. Initially, the “temperature” is hot and the algorithm is very keen to pick plans that do not improve its situation. As the temperature cools down it becomes more and more conservative and tends to only take improving plans. The idea behind this is that initially the algorithm should easily escape local optima but eventually concentrate on finding the best possible solution.

Finally, there also exist genetic algorithm planners. For instance PostgreSQL can be configured to use such a planner. Here we simulate an evolutionary process. In a pool we keep a specific number of plans. Iteratively pairs of plans are picked from the pool and combined (by taking properties from both plans) and possibly mutated (randomly changing properties) to form a new plan that is put into the pool. Then only the top $n$ (the fittest) plans are kept in the pool, while all the others are removed. At this point the algorithm repeats, until a satisfying solution has been found.

For all these algorithms countless variations exists. When evaluate the performance of different methods, despite the many factors that can influence the result, research has found that as a rule of thumb we can say that the use of randomized algorithms it not recommended for queries with less than 10 joins.

### 5.3.1 PostgreSQL’s Genetic Planner

Interestingly, the genetic programming query optimizer that comes with PostgreSQL uses a very similar problem encoding as the planner in SQLite (i.e. as a TSP problem). But instead, the problem is, of course, solved with a genetic algorithm.

The algorithm used is based on the genitor algorithm\[7\] and essentially is a genetic algorithm for TSP approximation. The GA is steady state which means that instead of replacing the parent generation with its children, the parents and the generated children compete with each other to make it into the next generation, i.e. the fittest parents or children will survive. Further, the algorithm does not make use of mutation. For crossover, the edge recombination crossover algorithm is used. Edge recombination crossover tries to minimize the the number of edges in a new child that do not appear in any of the parents, which avoids adding arbitrary edges to the encoded TSP solution.

### 5.4 Distribution Estimation

We already talked about the distribution estimation module before. Here are some more notes about it. The purpose of this module is the estimation of the size and distribution of relations. This is important as the cost estimator needs this information to properly calculate the cost of a plan, which in turn is required by the planner to pick a good plan from the search space.

Classically, most DBMS still use histograms for this. The principle idea is very simple: the domain of an attribute (of a relation) is split into buckets, each covering a domain range. With each bucket we associate a counter, if a value
falls into a specific bucket, we update the counter. There are a few problems that DBMS implementors have to solve: the first one is their maintenance, i.e. updating the histogram whenever the underlying data changes that much, that the histogram gives a wrong picture. Secondly, to strike the balance between accuracy of the histogram versus the cost of maintaining it.

A big problem of size and distribution estimation is the so-called *attribute value independence assumption*. This assumption is made by DBMS to greatly simplify estimation. It states that there is a statistical independence between columns, which is hardly ever true. Consider a table of cities with two columns, one for the city’s name and one for the city’s zip code, obviously these two columns are not at all independent but the estimator will assume they are and greatly underestimate probabilities, which in turn might cause the planner to pick the wrong plan. This pitfall can be avoided by keeping so-called cross column statistics, whose efficient implementation is still subject to research.

References


