Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jan Urbański
Nr albumu: 219721

Join optimisation with Simulated Annealing

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Stencla
Instytut Informatyki

Maj 2010

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Abstract

This paper presents a prototype implementation of a module for optimising the join order in an SQL query. It uses a randomised algorithm based on the simulated annealing method. Combined with the relational database management system PostgreSQL the module can be used to plan arbitrary SQL queries.

Keywords

simulated annealing, join order optimisation, PostgreSQL

Socrates-Erasmus code

11.3 Informatyka

Subject classification

H. Information Systems
H.2. Database Management
H.2.4. Query processing

Title in English

Join optimisation with Simulated Annealing

Contents

List of Figures

List of Tables

Introduction

Part of the process of planning a query is determining the order in which relations are joined and the decision to execute the joins in a particular order can often be critical for the speed of the later execution step. It is more and more frequent that database systems need to deal with queries involving hundreds or even thousands of joined relations. Techniques like join and subquery flattening contribute to increasing the number or relations that are considered simultaneously for a join. Visual report building systems and data mining software frequently generate queries with numerous outer joins.

Finding the optimal join order for a SQL query is in general an NP-hard problem [13]. For small queries searching through the entire solution space is feasible, but in many practical cases it would lead to unacceptable planning times and, even before that, memory exhaustion.

A popular approach to solving that problem is using randomised algorithms. The relational database management system PostgreSQL includes a module called GEQO (GEnetic Query Optimiser) which, as the name suggests, uses a genetic algorithm to plan queries.

In this paper we will present an optimiser module implementation that is an alternative to GEQO and is based on the simulated annealing technique. We will describe how the module integrates with the rest of the PostgreSQL optimiser and compare the results obtained by using our module with the ones produced by GEQO.

The proposed module is called SAIO and the ideas centred around that project have been presented during the 2010 PGCon, an annual conference gathering top contributors, administrators and users of PostgreSQL. It has been deemed interesting by the community and might eventually find its way into the official PostgreSQL codebase.

Chapter 1

Problem description

In this chapter we will give an overview of the problem and the various approaches that can be found in existing literature. We will define what constitutes a solution, what are the ways in which a solution can be transformed into another and how are solutions compared between each other.

1.1. Solution space

The goal of the join order search module is to create a plan of accessing disk data underlying the relations to be joined and ways of constructing relations that are results of joins, eventually obtaining a single relation, which is the result of the entire join. We will therefore consider as solutions structures providing the information on accessing base relations, the order in which they are combined and the ways the combinations are made.

In practise a solution can be represented as a binary tree. The leaves are base relations, usually corresponding to a physical table in the database. The inner nodes are relations that are created by joining the child relations. The root node is the result relation. The nodes are labelled with the specific way of constructing the node, which we will discuss later. In every join node we can distinguish the inner and the outer child, which reflects the asymmetry of the join operation — even if joining A with B yields the same result as joining B with A, the cost is often different.

1.1.1. Left-deep, right-deep solutions and bushy plans

Solutions can be classified based on the shape of the result tree. A special class of solutions are ones that always have a base relation as the outer join node. This effectively means constructing the final relation by first joining two base relations, then joining the result to another base relation, then to another and so on. Trees representing such solutions are called left-deep query processing trees. Previous research has been done focusing specifically on left-deep trees and some existing algorithms only yield such trees as results [17].

By analogy, a right-deep tree is one that always has a base relation as the inner node of a join. Left and right-deep solutions are not very different, but research proves that constraining the solution space to these types only gives unsatisfying results [16].

Another class of solutions are ones that contain intermediate relations as both the inner and outer side of a join. The plans created by these solutions are called bushy plans and allow the highest amount of freedom in creating the join order, as obviously the set of all left and right-deep plans is a strict subset of the set of all bushy plans.

All optimiser modules used in PostgreSQL are capable of yielding bushy plans and the solution proposed in this paper also supports supports such output.

1.1.2. Considered join types

There are many ways to form a join between two relations. While the result of constructing a join is always the same, the algorithm used can have a big influence on the runtime. Moreover the way of constructing a lower level join can influence the cost of constructing higher level joins. A typical situation is when an algorithm applied to construct one join is more expensive compared to another, but produces sorted output, thus sparing the upper nodes in the tree the necessity to resort data before applying procedures that need sorted input. The PostgreSQL optimiser considers the following join types:

- Nested loop join
- Merge join
- Hash join

The nested loop algorithm [3] has been of special interest to researchers and it has been shown that even narrowing the problem to only take nested loop joins into consideration, it remains NP-complete [5].

1.1.3. Considered access methods

Data that will compose the final result is read from disk. The base relations usually have a one to one correspondence to tables stored on disk (more on situations where base relations are not equivalent to tables will be said in chapter 2). There are different methods in which a database management system can access disk data, and depending on the pattern of access, index presence and usage and order in which they are accessed, the cost of getting the data to be joined can vary. PostgreSQL supports the following access methods:

- Sequential scans
- Index scans
- Bitmap index scans

Given that, we can represented a join tree graphically, putting $r1$, $r2$, $r3$, etc as base relation names, labelling their corresponding nodes with an access method and labelling the upper level nodes with the join type.

1.2. Possible moves in the solution space

Both the current genetic algorithm used by PostgreSQL and the module described in this paper have a notion of transforming one point in the solution space into another. It is important to determine the possible moves that can be taken in each algorithm step.

One class of moves is changing the access method of a base relation. Such moves are always valid, because regardless of the way the relation data is accessed, the result is always the same. The only difference between data obtained via different access methods is the sort

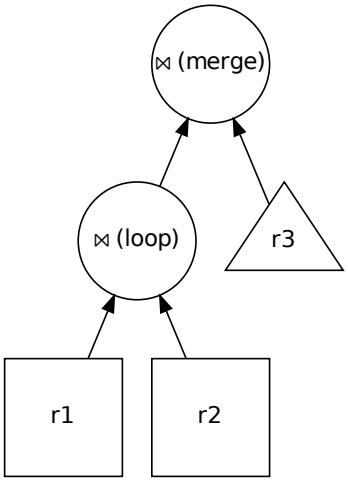


Figure 1.1: An example join tree representing a join between two relations, $r1$ and $r2$, both of which are accessed through a sequential scan. They are joined using the nested loop algorithm, with the result being joined with a hash join to $r3$. $r3$ is accessed using an index.

order, which only means that for some combinations the planner will have to consider the overhead of sorting tuples being read before using them in the join algorithm.

Another class of moves that lead from a valid point in the solution space to another valid point is changing the join algorithm. Regardless of the join algorithm used, the result of performing a join is always the same and again, the only difference is the sort order (or the lack of it) of the output.

If only inner joins are considered, valid moves also include changing the order in which the joins are applied. These types of moves are governed by the following equivalences [16]

- commutativity: $r1 \underset{r1,r2}{\bowtie} r2 = r2 \underset{r1,r2}{\bowtie} r1$
- associativity: $(r1 \underset{r1,r2}{\bowtie} r2) \underset{r2,r3}{\bowtie} r3 = r1 \underset{r1,r2}{\bowtie} (r2 \underset{r2,r3}{\bowtie} r3)$
- join exchange: $(r1 \underset{r1,r2}{\bowtie} r2) \underset{r1,r3}{\bowtie} r3 = (r1 \underset{r1,r3}{\bowtie} r3) \underset{r1,r2}{\bowtie} r2$

where $\underset{r1,r2}{\bowtie}$ is an inner join operation using a predicate on relations $r1$ and $r2$.

This brings us to the set of theoretically valid moves that the algorithm can take when randomly exploring the solution space. Access methods and join algorithms can be changed always, whereas the above join tree manipulations are in general only applicable when considering inner joins. We will explore the issues that arise when outer joins enter the scope of the problem shortly.

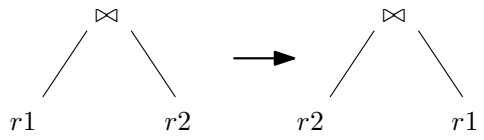


Figure 1.2: The result of joining $r1$ with $r2$ is the same as the result of joining $r2$ with $r1$ if it is an inner join

1.2.1. Join restrictions

A join clause in applied SQL is very often accompanied by a restriction clause, that constrains the rows that the join will output. In standard SQL syntax join restrictions can be expressed with the JOIN ON or JOIN USING syntax. However, many times the application writer writes out all joins in a query using the Cartesian join syntax and then puts constrains on the output rows using the WHERE clause. Indeed, the following two queries are equivalent:

- SELECT c1, c2, c3 FROM r1 JOIN r2 ON (r1.cj = r2.cj)
- SELECT c1, c2, c3 FROM r1, r2 WHERE r1.cj = r2.cj

When lots of relations are being joined together it is very important for the optimiser module to detect the restrictions of each join and be aware of them in the process of deciding the way of executing a query. Consider the following query:

SELECT c1, c2, c3 FROM r1, r2, r3 WHERE r1.cj = r2.cj AND r2.ck = r3.ck

Executing that query as $(r1 \underset{r1,r2}{\bowtie} r2) \underset{r1,r3}{\bowtie} r3$ will allow the executor to handle only the rows that match the restriction clause between $r1$ and $r2$ when executing the join with $r3$. Executing the query as $(r1 \underset{r1,r2}{\bowtie} r3) \underset{r1,r3}{\bowtie} r2$ will create a Cartesian join in the first step, which almost surely will result in bad performance.

1.2.2. Outer joins and join order restrictions

The presence of outer joins changes the space of valid moves in the solution space dramatically. When considering a left join, the commutativity and associativity equivalences given in 1.2 no longer hold. The join exchange equivalence is still valid, though. A simple counterexample depicted in figure 1.3 shows that the left join operator $\underset{P(r1,r2)}{\bowtie}$ is not commutative.

Constructing a counterexample for the associativity equivalence is a bit harder. In fact, the equivalence holds when the join uses a predicate that is never true when any of the operands is not NULL. PostgreSQL calls such predicates *strict* and if a left join is made using a predicate marked as strict, it will consider the join to be associative. In the general case, however, associativity of left joins can be disproven if the predicate can become true for NULL input. An example is the SQL standard `IS NOT DISTINCT` operator. Figure 1.4 illustrates this example. RIGHT joins are subject to the same restrictions and in fact PostgreSQL implements RIGHT joins as LEFT joins, by simply switching the joined relations around.

We can now see that the moves that an algorithm can take in the solution space are limited not only by whether a particular join is an inner or an outer join, but sometimes even by the properties of predicates that are used for joining. The fact that PostgreSQL allows user-defined operators to be used as join predicates complicates the matter even further. Many algorithms proposed by researchers only consider inner joins [7] [9] [11] and cannot

Figure 1.3: Exchanging the inner and outer relation of a left join produces different results.

$r1$	
c1	c2
1	1
2	2
3	3

$r2$	
c1	c3
1	1
2	2
4	4

$r1 \underset{P(r1.c1,r2.c1)}{\bowtie} r2$		
c1	c2	c3
1	1	1
2	2	2
3	3	ω

$r2 \underset{P(r1.c1,r2.c1)}{\bowtie} r1$		
c1	c2	c3
1	1	1
2	2	2
4	ω	4

Figure 1.4: Changing the order of applying Left joins produces different results with a predicate that is not strict, for example `IS NOT DISTINCT`.

$r1$	
c1	c2
1	1
2	2
3	3

$r2$	
c1	c3
1	1
2	2
4	4

$r3$	
c3	c4
1	1
2	2
ω	5

$(r1 \underset{P(r1.c1,r2.c1)}{\bowtie} r2) \underset{P(r2.c3,r3.c3)}{\bowtie} r3$			
c1	c2	c3	c4
1	1	1	1
2	2	2	2
3	3	ω	5

$r1 \underset{P(r1.c1,r2.c1)}{\bowtie} (r2 \underset{P(r2.c3,r3.c3)}{\bowtie} r3)$			
c1	c2	c3	c4
1	1	1	1
2	2	2	2
3	3	ω	ω

be directly applied in a general purpose SQL database system. The PostgreSQL optimiser is careful to track restrictions imposed on the join order by the presence of outer joins and enforce them during planning.

1.3. Cost model

The join order in PostgreSQL is determined purely at the planning stage, before reaching the executor. This means that any algorithm will only have access to statistical data gathered during the runtime of the database system. This also means that the quality of a plan is measured by the *estimated* cost of executing the query. If a plan with a superior estimate yields inferior execution performance, it has to be considered a failure of the executor submodule. The planner only operates on estimates and relies completely on the statistics subsystem to get correct data. A relational database management system uses primarily three resources of the host operating system: the disk, the main memory and the CPU. A widespread assumption is that the cost of disk input/output operations dwarfs the cost of memory and CPU usage, and some authors only consider disk I/O as a contributing factor to query execution cost [16]. On the other hand, small databases running on systems with large main memory often

fit completely in RAM. In such databases disk I/O stops being the bottleneck of query execution and predicted CPU consumption starts deciding on the quality of a plan. Finally, there are subtleties concerning the memory access patterns of databases that fit in RAM, where reference locality starts playing an important role [10]. Data access patterns even more important with data on rotating mediums (such as traditional hard drives), where reading disk blocks sequentially is often much cheaper than random access. PostgreSQL takes the road of combining disk I/O and CPU utilisation in a single cost model. The cost of executing a query is expressed as a unitless floating point number. User-defined operators and functions can make cost estimation more difficult, as the database system cannot know the real cost of executing them. This problem is specific to PostgreSQL, as the extensibility of the system allows users to define operators that perform arbitrary database queries or even execute arbitrary application code.

Chapter 2

The PostgreSQL optimiser

This chapter focuses on the internals and specifics of the PostgreSQL optimiser. As a general purpose database, PostgreSQL needs to handle a wide variety of SQL constructs and its extensibility means that it has to be able to efficiently plan complex joins, both outer and inner, while dealing with user-defined functions and operators.

2.1. Query planner algorithm

PostgreSQL represents both base relations and relations created as join results as structures called `RelOptInfo`. Each `RelOptInfo` contains a list of `Path` structures, that represent the way a relation is constructed. For each `RelOptInfo` various `Path`s are considered, if they can provide output in different sort order. It is important, because some join types require sorted input, which means that a `Path` that imposes an order on the tuples might be a better choice than a `Path` that is cheaper, but guarantees no ordering, because it would have to be sorted anyway before executing the join.

The join planning module starts by creating a `RelOptInfo` structure for each physical table in the query. If the query includes subqueries they might be planned separately, by invoking the planner recursively on them. A `RelOptInfo` created from a physical table has several `Paths`, corresponding to the available access methods on the table. These include a sequential scan and various index scans if the table has indices. Next the join order is determined. A join always happens between two `RelOptInfos` and the result of a join is also a `RelOptInfo`. `Path` structures in `RelOptInfos` that are join relations correspond to the join algorithm used, and so there can also be multiple of them present. In each `RelOptInfo` PostgreSQL only keeps a single `Path` (the cheapest one) per possible sort order. More expensive methods of obtaining the same result in the same order are clearly inferior and are discarded immediately.

Each `RelOptInfo` contains two special `Paths`, the one that will produce the output fastest and the one that will start outputting data earlier. They are called, respectively, the cheapest total path and the cheapest startup path.

`RelOptInfo` structures are labelled with sets of numbers. The ones created from physical tables or from recursive planner invocation are given one-element sets as labels, typically 1, 2, 3... After that `RelOptInfos` created by joining two lower level structures are labelled with the set union of the child labels. And so the `RelOptInfo` created as a join of 1 and 2 would be labelled $(1, 2)$. Because a `RelOptInfo` contains all possible `Paths` that lead to a certain result, using unordered sets as labels is possible. Two `RelOptInfos` with the same label *always* represent the same result — the result of joining base relations in any order that respects rules outlined in 1.2.

2.1.1. Query flattening

An SQL query is parsed by PostgreSQL into a tree structure. The FROM list is an important node in that tree, as it contains the names of relations appearing in the FROM clause, which are joined together. Explicit JOIN clauses are separate nodes in the parse tree and so relations joined explicitly appear at a different level in the parse tree than the ones joined implicitly. The planner flattens the query tree by transforming explicit JOIN clauses into FROM entries and by saving the join condition in a `RestrictInfo` structure. Similarly, it scans through the WHERE clauses to find ones that involve relations that are joined together and creates `RestrictInfo` structures from them as well.

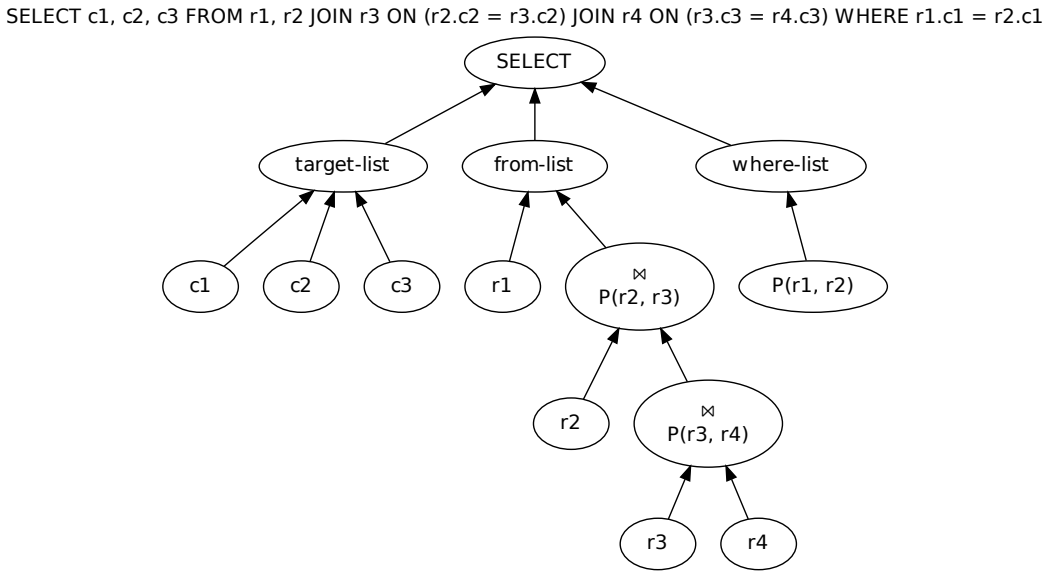


Figure 2.1: The parse tree representing a query. Explicit JOINs are constructed as separate nodes and relations appearing in the FROM list are put in a separate list.

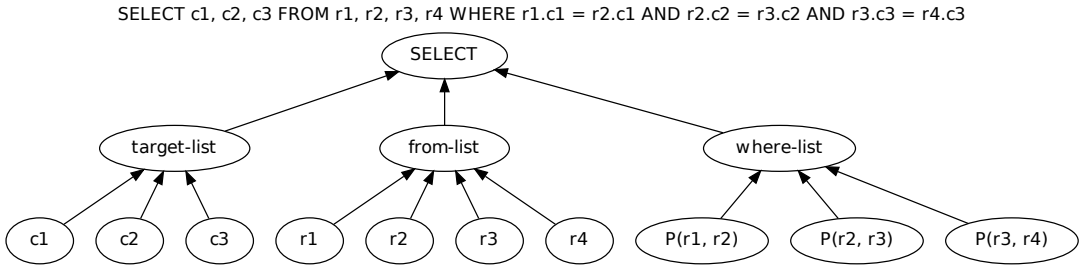


Figure 2.2: The parse tree is flattened, explicit JOINs are converted to restriction qualifiers and the relations being joined are put in the FROM list.

Another axis along which flattening is done are subqueries. If a subquery appears in the FROM list it is pulled up in the parse tree and transformed into a JOIN clause. These operations are done to make sure the planner has as much liberty as possible when determining

join order for base relations, which means making the list of joined relation as big as possible and avoiding recursive planner invocation. On the other hand, as we mentioned earlier join planning is an NP-hard problem and so PostgreSQL tries to prevent excessive growth of the join list to avoid an uncontrollable increase in planning time. There are user-definable parameters that tell the planner what is the maximum number of subqueries that will get pulled up and transformed into JOINs and similarly how many explicit JOIN clauses will be flattened into a single JOIN list before deciding that the particular JOIN needs to be solved recursively as a separate subproblem.

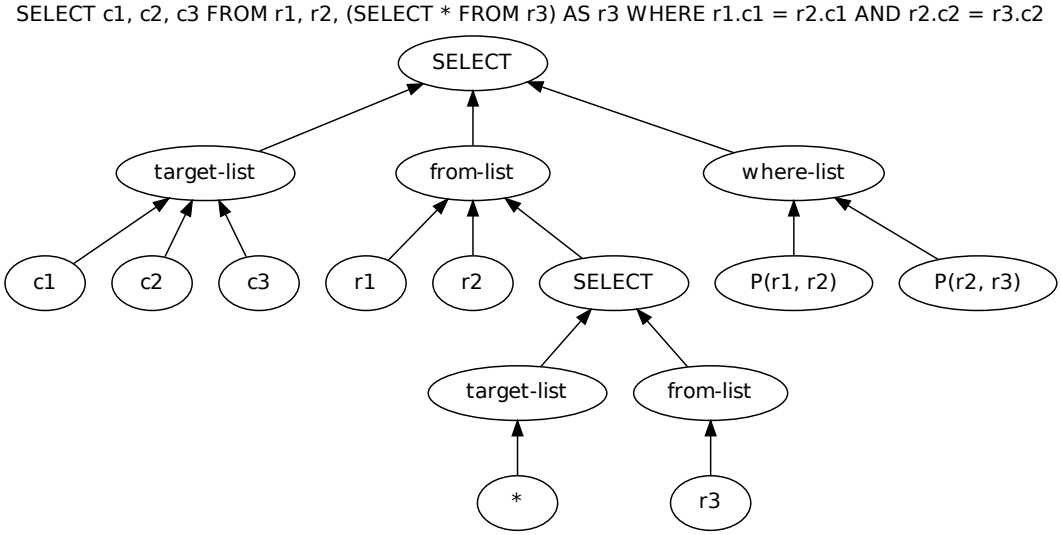


Figure 2.3: Subqueries have their own parse trees and can be planned separately if the are not pulled up into the parent join.

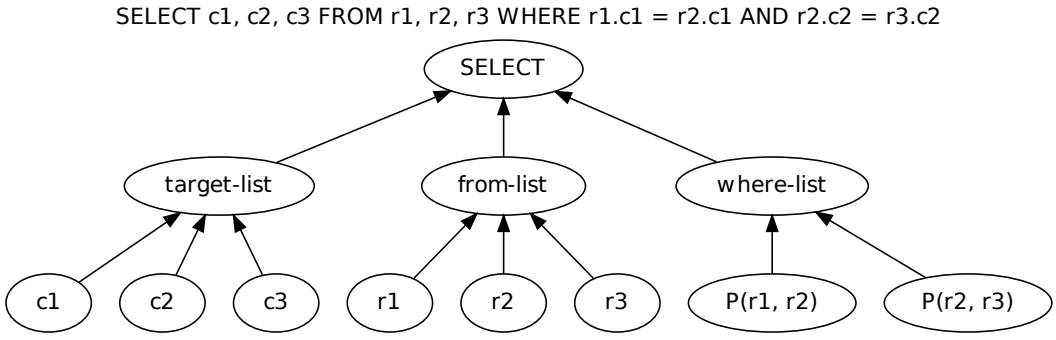


Figure 2.4: The subquery is pulled up and merged with the other relations in the FROM list.

Other considerations apart from user defined limits can prevent the planner from flattening joins and pulling up subqueries. If a subquery contains function execution the planner cannot pull it up, because it could change the number of times a function is executed. Since in

PostgreSQL functions can perform arbitrary operations, including modifying database tables or executing application code, the planner cannot do any optimisations that could result in a different number of executions. Users are provided with a syntax to mark functions guaranteed to never change the database state on multiple executions and such functions do not prevent subquery pulling. Subqueries with GROUP BY or LIMIT/OFFSET clauses are also never pulled up, because that would obviously change the result of the whole query. Also joins sometimes cannot be flattened; for instance PostgreSQL will never flatten a FULL OUTER JOIN, so putting one in a query forces the planner to recursively plan the two sides of the join and execute it only after both sides are computed.

2.2. Query tree transformations

The parse tree undergoes some additional transformations before it is handed over to the join search module. They are aimed at producing higher quality plans either by narrowing down the solution space or by allowing more relations to take part in the join search algorithm.

2.2.1. Join removal

The PostgreSQL planner tries to identify joins that can be removed without influencing the result. Obviously, when a join can be discarded while maintaining the same result, it constitutes a better plan than executing that join. There are several conditions that have to be met for a join to be removed:

- it has to be a left join
- a unique constraint has to exist on the join column of the table on the nullable side of the join
- the operator used in the join condition is never true on NULL input
- none of the columns from the nullable side of the join can appear in the query outside of the join condition

If all these conditions are proven to be true, join is not executed and is removed from the parse tree. An example of join removal would be transforming the following query:

SELECT r1.c1, r1.c2, r1.r2_id FROM r1 LEFT JOIN r2 ON (r1.r2_id = r2.id)

into:

SELECT r1.c1, r1.c2, r1.r2_id FROM r1

assuming that *id* is the primary key of the $r2$ table. Such situation ofter arises in automatically generated queries, where reporting software puts useless joins in the query and only some columns are actually used by the client.

Join removal can only take place in case of a left join, because in inner joins the number of rows emitted by the join can be smaller than the number of rows that would be emitted if only the left-hand side of the join was scanned. The presence of a unique index on the join column ensures that the number of rows in the result will not be fall compared to the number of rows the join would produce. The requirement for the operator to be strict (cf. 1.2.2) is there to prevent the situation where less rows are emitted after removing the join, because the nullable side of the join contained multiple NULL values in the join column — the requirement for a unique constraint is not strong enough to prevent that, as NULLs are never equal to each other.

2.2.2. Semi and antijoins

There are some kinds of SQL constructs that the planner will transform into joins, even though they are not syntactically joins. The PostgreSQL planner is able to transform EXISTS and NOT EXISTS expressions into semijoins and antijoins, respectively. This means that queries such as

SELECT c1, c2, c3 FROM r1 WHERE EXISTS
(SELECT 1 FROM r2 WHERE r1.c1 = r2.c2)

will be transformed by the planner into a semijoin between $r1$ and $r2$. Similarly, NOT EXISTS constructs are transformed into antijoins. Of course semi and antijoins also impose join order restrictions and cannot be freely rearranged with inner joins. Each time the planner decides to transform an EXIST clause into a join, it stores additional information in the parse tree to track which join orders are prohibited.

2.3. The standard join order planner module

The standard join order module implements a dynamic algorithm that explores the entire possible solution space.In each step the algorithm fills the *nth* level of the accumulator with relations consisting of $n + 1$ joined based relations. It is an improved version of the original system R optimiser algorithm [14].

The procedure for building relations maintains a global cache to prevent building the same `RelOptInfo` multiple times. One needs to remember that `RelOptInfos` are labelled with sets of identifiers of base relations. If the function creating a new `RelOptInfo` finds a relation with the label that is the union of the labels of its arguments, it simply returns that cached relation instead of constructing a new `RelOptInfo` and adding all possible `Paths` to it. This makes the planner code simpler, as it can keep trying to join relations even if the particular join has already been created while adding no performance penalty. This can however become an issue for join search modules that want to repeatedly plan the same query and each time need all `RelOptInfos` and their `Paths` to be recreated. As we will see in section 2.4 the standard way of dealing with this problem is truncating the global cache after each planning cycle.

2.3.1. Algorithm outline

First we define a helper function that gets passed a relation and a list of join candidates, iterates over them and forms a new joinrel if their labels do not overlap, it has not yet been constructed in the current level and there is a join restriction between the two relations. The optional boolean argument can be used to force the creation of a join relation even if there is no condition that prevents the join from becoming a Cartesian product.

```
1  function make_rels(rel, other_rels, current_level, force=false):
2      for r in other_rels:
3          if label(r) ∩ label(rel) ≠ ∅:
4              continue
5          if label(r) ∪ label(rel) ∈ current_level:
6              continue
7          if has_join_restriction(r, rel) and not force:
8              continue
9          new_rel = make_join_rel(rel, r)
10         if not new_rel:
```

```
11              continue
12          label(new_rel) = label(r) ∪ label(rel)
13          append(current_level, new_rel)
```

The algorithm first finds all base relations, that is ones corresponding to physical tables or subqueries. An accumulator array is initialised with the initial relations at the zeroth level of the algorithm.

```
1  initial_rels = get_base_rels()
2  accumulator = array()
3  accumulator[0] = initial_rels
```

In order to fill level n the algorithm loops over relations from level $n-1$ and tries to join then with every relation from the zeroth level — that is every base relation. In the special case of level 1 an optimisation is done to prevent duplicate work. Because the relation building routine is symmetrical, when processing relation r only relations that are *after* r in the base relation list are considered. We introduce the `A after x` as all elements of the array L that are farther in the array than element x.

```
4  for level in 1..length(initial_rels):
5
6      accumulator[level] = array()
7
8      for rel in accumulator[level - 1]:
9          if level == 1:
10             other_rels = accumulator[0] after rel
11         else:
12             other_rels = accumulator[0]
13
14         make_rels(rel, other_rels, accumulator[i])
```

Joining only to base relations yields left-deep plans. The PostgreSQL planner prefers such plans and tries them first, but is also capable of building bushy plans. The next part of the algorithm deals with bushy plans.

The planner loops over pairs of relations from lower levels and tries joining them together. Again, the symmetry of the join building routine allows to only go halfway into the loop and stop processing when all pairs that together would form relations of cardinality n are processed (for algorithm level n).

```
15     k = 1
16     while true:
17         other_level = level - k - 1
18         if k >= other_level:
19             break
20
21         for rel in accumulator[k]:
22             if other_level == k:
23                 other_rels = accumulator[other_level] after rel
24             else:
25                 other_rels = accumulator[other_level]
26
27             make_rels(rel, other_rels, accumulator[i])
```

```
28
29          k += 1
```

Finally the planner checks if there where any relations created on the current level. If not, as last resort it tries to join the zeroth level relations but this time ignoring join restrictions, which effectively leads to producing Cartesian joins.

```
30      if empty(accumulator[level]):
31
32          for rel in accumulator[level - 1]:
33              if level == 1:
34                  other_rels = accumulator[0] after rel
35              else:
36                  other_rels = accumulator[0]
37
38              make_rels(rel, other_rels, accumulator[i], true)
```

Plan costs are determined be looking at the `Path` with the lowest cost contained in the topmost `RelOptInfo`. There is one more subtlety that has to do with keeping both the cheapest `Path` and the one that starts providing output tuples fastest. If the query being planned is going to be used by a SQL cursor, plans that produce output faster are preferred. The assumption is that the user will be fetching the output over a certain period of time, so even if the query takes more to complete, the perceived performance will be better if some result tuples will be available quickly.

2.3.2. Performance characteristic

The time complexity of the algorithm is exponential with the number of initial `RelOptInfos`. This is a direct result of the fact that the underlying problem is NP-hard. There are other issues with this way of solving the join ordering problem, though.

During the algorithm execution a list of all possible `RelOptInfos` is built, each one of them containing all possible `Paths` that can be used to form the join. That can easily exhaust the available memory, given a sufficiently large initial joinlist. Experiments have shown that removing the limitation for the maximum number of relation for which the standard algorithm is used caused the system to use all available memory long before it has reached execution times that could be deemed unacceptable. It is important to note, that even if the inevitable exponential growth of planning time would not be considered a fatal issue, the memory usage renders the standard planning algorithm useless for very large queries.

2.4. The GEQO module

The PostgreSQL optimiser contains a pluggable interface that allows external modules to implement parts of the planning algorithm. One module that is included in the default distribution of PostgreSQL is called the Genetic Query Optimiser (GEQO). When handling very large joins, it can significantly reduce the time and memory needed to plan queries, as it is designed to maintain strict memory usage constraints and uses a randomised genetic algorithm instead of exhaustive search, which prevents exponential runtime increase when the number of relations grows. GEQO's algorithm replaces the algorithm outlined in section 2.3.

2.4.1. Algorithm outline

GEQO is based on the GENITOR algorithm, developed at Colorado State University [18]. This algorithm has primarily been used to solve the Travelling Salesman Problem and has been adapted for the PostgreSQL project to be applicable for the join ordering problem. The join order as a `Chromosome` structure, which is a string of `Gene` objects. Each `Gene` has a one to one correspondence to a base `RelOptInfo`, that is one that has been created from a physical table or a recursively planned subproblem. The order of the `Genes` in `Chromosomes` influences the join order of relations, although as we will see later it does not explicitly determine it.

The cost model for GEQO is similar to the one used by the standard planner, with the difference that startup cost is ignored, and plans are only compared based on their total cost. Each `Chromosome` is assigned a fitness value that is inversely proportional to the total cost of the join order that it represents, which means that cheaper `Chromosomes` are considered more fit. The initial population of randomly chosen `Chromosomes` is then mutated, creating new `Chromosomes` that if they are fit enough can replace their parents in the pool. There are various mutation algorithms implemented:

- edge recombination crossover
- partially matched crossover [4]
- cycle crossover [12]
- order crossover [12]
- position crossover [2]

The choice of the mutation schemes used is taken at compilation time. The default source distribution only uses edge recombination crossover (ERX), the rest of the algorithms are left disabled. Some research suggests that ERX is indeed the method delivering the best results [15].

Each time a `Chromosome` fitness is computed, the `Genes` are visited in the order defined by the `Chromosome` and a join is created between one of the previously built `RelOptInfos` and the one corresponding to the `Gene` being visited. An important feature of the GEQO algorithm is that it only creates joins if there is a restriction clause that can be applied to the join, in other words it is avoiding forming Cartesian joins. To achieve that, the algorithm introduces a notion of `Clumps`, which are small wrappers around joined `RelOptInfo` structures that also remember the number of relations joined. The processing starts with one `Clump` created from the first `Gene` in the `Chromosome` and every time the `Gene` being visited points to a relation that cannot be joined to any of the existing `Clumps`, a new `Clump` is started. The `Clumps` are organised in a list sorted by the number of relations forming the `Clump`. This way the new relations are first joined to the biggest `Clump` and only if it fails, or if there are no join restrictions present, the next `Clump` is tried. After finishing the traversal of the `Chromosome` the `Clumps` are joined together, first preferring joins with restrictions and finally, as a last resort, forming Cartesian joins. The ability to join `Clumps` makes GEQO capable of producing bushy plans.

Observe that the algorithm uses the same primitives to construct join `RelOptInfos` as the standard planning module, which means that even if GEQO explores only some randomly chosen join orders, it always considers all `Paths` that can lead to a join. For the same reason, GEQO always considers relations involved in a join as both the inner and the outer join relation. As noted in section 2.3, the procedure responsible for creating new `RelOptInfos` is symmetrical.

2.4.2. Performance characteristic

Each computation of a `Chromosome` fitness is done in a separate memory context and the memory used for `RelOptInfo`, `Clump` and `Path` structures is freed afterwards. The internal planner relation cache is also truncated after every `Chromosome` computation, to force the rebuilding of all relations. Otherwise `RelOptInfos` retrieved from cache could contain `Paths` that refer to join relations that will not be present under the join order defined by the current `Chromosome`.

Freeing the planner resources after each recomputation gives GEQO a huge advantage over the standard planning module, which keeps all previously built structures in memory during the whole planning process. On the other hand it means that GEQO has to rebuild the entire join tree every time the fitness of a `Chromosome` needs to be calculated.

The algorithm ends after a certain number of mutations, which is linearly dependant on the number of relations being joined. The fittest `Chromosome` is chosen and a `RelOptInfo` is built according to the `Gene` order defined by that `Chromosome`. The obtained relation is the result of the GEQO algorithm. Having a fixed number of mutations before choosing whatever join order is currently fittest allows GEQO to constrain the time it uses for query planning.

2.4.3. Known deficiencies

The GEQO module has been repeatedly accused of being suboptimal by the PostgreSQL community. Historically, the mechanisms preventing Cartesian join creation were much more strict, which lead to situations where queries with few join restrictions or with many join order constraints could not be planned and were resulting in an error. The growing popularity of SQL generating software caused queries with numerous outer joins and complex subselects to become common. Such queries often have very strict constraints on the legal join order and lots of `Chromosomes` that GEQO was creating were simply invalid, because they were defining join orders leading to incorrect results under SQL semantics. This caused many `Chromosomes` to be discarded and sometimes could even result in running out of iterations before creating any valid join order. This has been since addressed by introducing the concept of `Clumps`, but now it is possible that queries with lots of join order restrictions will increase the planning time enormously because of having to check all joins for validity each time a `Chromosome` fitness is computed.

Members of the community also expressed concern about whether the algorithm that initially has been designed to solve the Travelling Salesman Problem is suitable for deciding join ordering. It is not clear if the heuristics that work well for TSP are equally efficient when planning a complex SQL query. There are no field proofs that ERX is indeed the best mutation scheme that can be used, as the other algorithms are disabled in the source tree and get no testing or user exposure. Finally, is the problem of queries with very strict join order constraints that can require excessive amounts of CPU to be planned and that cause repetitive and sometimes costly recalculation of long `Chromosomes`. Keep in mind that every time a `Chromosome` fitness is calculated, all possible `Paths` for a given join order are created. That is obviously a saving compared to the standard algorithm that would create every `Path` for *all* possible join orders, but still can result in unacceptable planning times for large joinlists.

Chapter 3

The SAIO module

This chapter will focus on the optimiser module that aims to replace GEQO. It takes a different approach than GEQO, choosing simulated annealing over genetic algorithms. We will shortly present the basics of solving optimisation problems with simulated annealing, how the ideas had to be adapted to fit the specific requirements of a PostgreSQL planner module and how does the module leverage existing infrastructure to produce good quality plans.

3.1. Algorithm overview

The algorithm starts by choosing any point from the solution space. It helps if the starting point does not represent an overly expensive solution. We will see later that being able to start from a decent solution allows the algorithm to produce much higher quality output. A copy of the solution with the lowest cost among all generated solutions is also being kept during execution.

An important concept that SAIO uses is the *temperature*. In general it is an arbitrary real positive number. It gets set at the beginning of the algorithm and is being maintained along with the current and minimal state during the whole run time.

```
1 state = min_state = any_state()
2 temperature = starting_temperature()
```

The body of the algorithm consists of two nested loops. In the innermost loop the algorithm generates a new state at random, based on the previous state. We will be calling the process of reaching a solution from another solution a *move*.

```
3 do:
4     do:
5         new_state = random_move(state)
```

The generated state can be either accepted or discarded. A discarded move is forgotten and the current state remains unchanged. If the move is *accepted*, the generated state replaces the current state. We call situations when generated states are cheaper than the initial ones *downhill* moves, whereas moves that take the algorithm from a lower cost state to a higher cost are called *uphill*.

Whether a move gets accepted or not depends on the current temperature and the the difference of costs between the old and the new state. The rule is that downhill moves are always acceptable and that uphill ones are accepted with some probability. The exact way of

computing the probability of accepting uphill moves will be discussed later. If the new state is cheaper than the cheapest one seen up to this point, it becomes the new minimal state.

```
6           if acceptable(new_state):
7               state = new_state
8           min_state = min(state, min_state)
```

The inner loop runs until the algorithm reaches a state called *equilibrium*. When this happens, the outer loop decreases the system's temperature and throws the execution back into the inner loop. The decision to determine if the system has reached equilibrium can be influenced by many variables and can get quite complex [17]. The equilibrium function used in the SAIO module will be presented later.

```
9       while not equilibrium() # inner loop
10      reduce_temperature()
```

Finally, the outer loop terminates when the system reaches yet another specific state, in which we call it *frozen*. Once the outer loop finishes, the minimal state is returned as the result of the algorithm. Again, different authors propose different ways of determining the frozen state [8].

```
11  while not frozen() # outer loop
12  return min_state
```

Looking closely at the pseudocode we can discover from where the name simulated annealing comes from. In metallurgy annealing is "a process that produces conditions by heating to above the re-crystallisation temperature and maintaining a suitable temperature, and then cooling" [1]. The algorithm follows a similar idea. We start with a hot system, where particles have high energies and the state changes rapidly. After the changes cease, which means the system reached equilibrium for the current temperature, we lower its energy and let the particles settle down once again. Iterating this process allows reaching the state with lowest energy.

We can see that simulated annealing is in fact a simple extension of a trivial random wandering algorithm, where the starting point is chosen randomly and then moves are taken in any direction that leads downhill. Such solutions have a severe problem of being easily trapped in local minima, where all neighbour states are more expensive, but there still exist points with lower cost in the solution space. The reason simulated annealing sometimes accepts uphill moves is to prevent such situation. If the algorithm wanders into a local minimum early in the execution, it has good chances of getting out of it and exploring other parts of the solution space.

3.1.1. Data model

SAIO uses a slightly different data structure than the default PostgreSQL planner, namely a `QueryTree`. They are binary trees, where every node has either zero or two children. The leaves are primitive relations and the inner nodes represent joinrels built from the children relations. This data model is much closer to the one presented in section 1.1, which is also the prevalent one in existing literature. In context of PostgreSQL, `QueryTree` leaves hold references to `RelOptInfos` that are the input of the join order planning module, while inner nodes hold `RelOptInfos` constructed by joining the children.

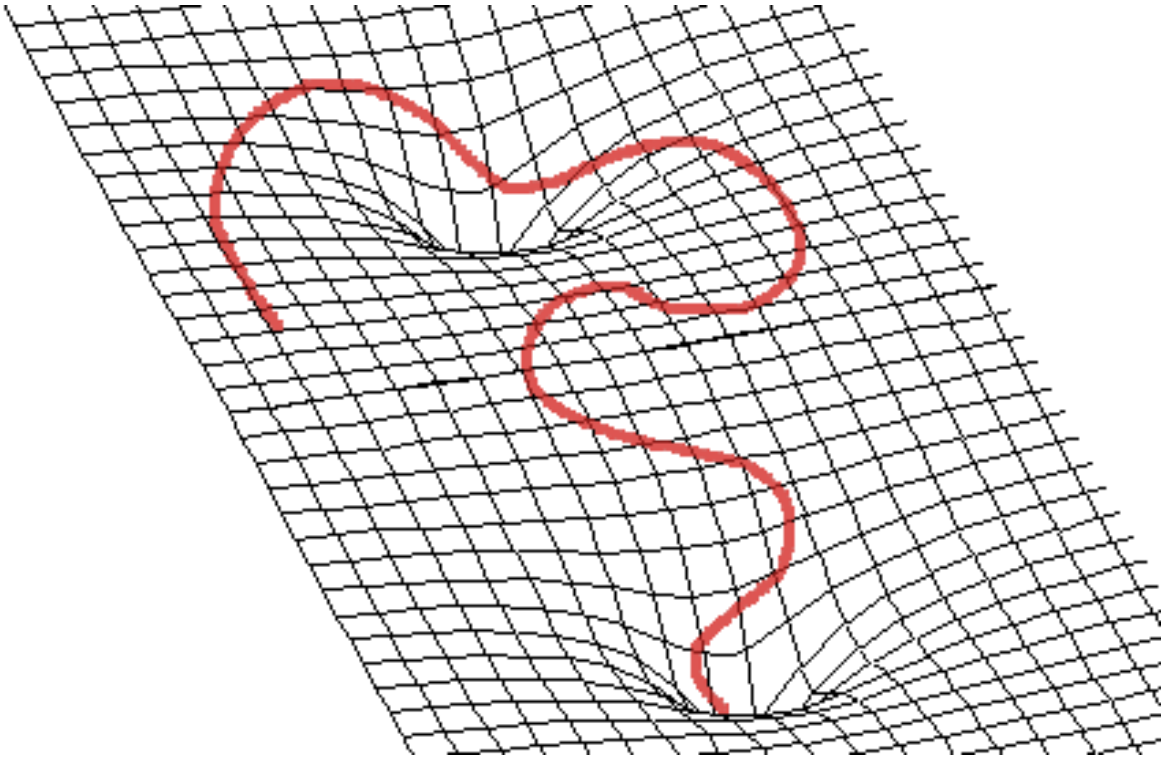


Figure 3.1: A visualisation of simulated annealing. The random walk manages to avoid the local minimum and then goes down in the global one.

3.2. Simulated Annealing challenges

The presented algorithm looks compact and straightforward. The difficult part is concretising the generic concepts that it uses. From previous section's discussion we can that implementing an optimisation module using simulated annealing requires solving the following problems:

a) finding an initial state
b) generating subsequent states
c) defining an acceptance function
d) determining the equilibrium condition
e) suitably lowering the temperature
f) determining the freeze conditions

Finding the initial state is done very similarly to how GEQO does it. Instead of `Clumps` SAIO uses `QueryTrees` but otherwise it starts by creating single node `QueryTrees` and then looping over them, creating joins while preferring ones that have some join restrictions until it produces a single `QueryTree`. That tree is always a valid solution for join ordering and the starting point of the algorithm.

Generating new moves turns out to be the most complex part of the problem. SAIO implements several ways of transforming a solution into another one, called *strategies*, that will be separately discussed in section 3.3.

The acceptance function is defined in terms of previous state cost, new state cost and current temperature. The probability of a move being accepted is defined as

$$P(accepted) = e^{\frac{cost_{prev} - cost_{new}}{temperature}}$$

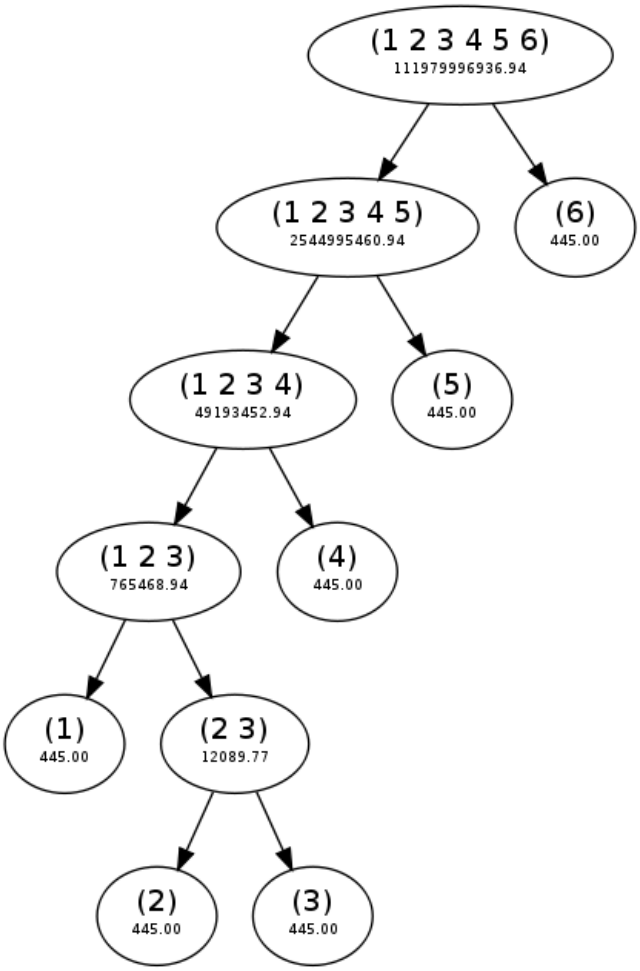


Figure 3.2: A `QueryTree` representing a six relation join. Each node is a reference to a `RelOptInfo` and also stores the cumulative cost of creating the joinrel. Nodes are labelled with sets of integer numbers that are indices of primitive relations in join. A node with label (1 2 3) represents a join of relations 1, 2 and 3.

As previously noted if $cost_{new} < cost_{prev}$ the move is always accepted. Initial temperature is chosen depending on the number of initial relations and drops geometrically after every step.

$$initial\_temperature = I \times initial\_rels$$
$$new\_temperature = temperature \times K \text{ where } 0 < K < 1$$

The coefficients used for temperature calculation are parameters of the algorithm. Usually I is in the order of tens or smaller and K is around 0.9. However, because SAIO operates on trees of `RelOptInfos` it is affected by the same specific PostgreSQL behaviour as GEQO — namely that is considers all possible `Paths` for every join relation. Strictly speaking this means that while exploring the solution space SAIO analyses several states in each algorithm step. It also means that generating subsequent moves and estimating their costs requires more computation time and memory, because every time it rebuilds all `Paths` for the current `QueryTree`. As we will see, SAIO includes measures to reduce the impact of this approach.

The equilibrium and freezing conditions are defined very simply [16]. The system is considered to be in equilibrium after a fixed number of loops, which is linearly dependent on the initial number of relations.

$$moves\_to\_equilibrium = N \times initial\_rels$$

This is somewhat counterintuitive when considering the annealing analogy, but turns out to work good enough. While there are more complex methods of determining equilibrium [17],

the simplicity of implementation, low computational cost and straightforward definition all vouch for the fixed loops number method. Conditions for considering the system frozen are equally simple — the temperature has to be below 1 and a fixed number of consecutive moves has to be rejected by the acceptance function.

$$frozen(state_N) \iff temperature < 1 \wedge \{state_i \,|\, i = N - F, \ldots, N \wedge acceptable(state_i)\} = \emptyset$$

If these two conditions are met, SAIO stops processing and returns the minimal state.

3.3. Moves generation

To understand the choices done in the move generation part of SAIO, some properties of `QueryTrees` must be noted. First is that in a tree there is no distinction between the left child and the right child. As we explained before, the PostgreSQL function that constructs a new join relation from two other relations is symmetrical and always considers each of the two arguments as either the inner or the outer side of the join. Another important observation is that the actual join order is in fact determined by the tree structure and the order of leaves in the tree. Because PostgreSQL considers all possible ways of joining two relations and because `RelOptInfos` are labelled with unordered sets of integers, the content of inner nodes can be always recovered by rebuilding the whole tree bottom to top. The only consideration is avoiding too many rebuilds, which are the most costly part of the algorithm.

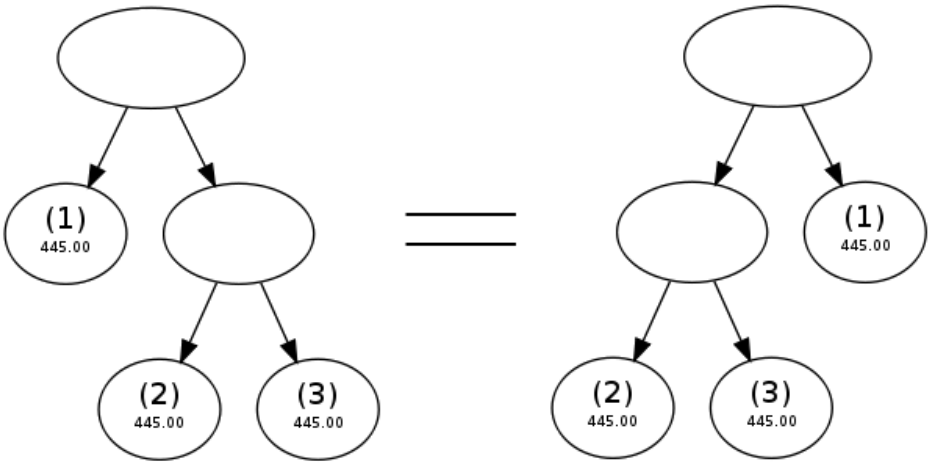


Figure 3.3: The two depicted `QueryTrees` are identical. Children nodes can be swapped around without changing the join order encoded by the tree, and inner nodes can always be recovered by rebuilding the tree.

Another important property the move generation algorithm has to have is freeing the memory used in every step before continuing to the next one. As we explained in section 2.3 keeping all built `RelOptInfos` in memory is simply not possible when planning a large query. To control memory usage, SAIO only keeps in memory the objects that are currently useful to rebuild the final joinrel. There are additional problems that this approach brings, because during the planning PostgreSQL modifies its internal structures making repeated planning of the same part of a query difficult. On the other hand, the additional complexity to deal with that brings substantial savings in runtime. The currently implemented version of the SAIO module includes several move generating strategies that can be chosen by the user.

All move strategies operate on a `QueryTree` and return a boolean result, true if the move has been accepted, false if it has been rejected. After a move is rejected the `QueryTree` structure is left unchanged, if it is accepted, it is modified to reflect the state after the move. As noted earlier, only the links between nodes and the order of leaves are important. Most move strategies start with a `QueryTree` that has its inner nodes uncalculated and rebuild them in each step. There is one strategy that tries to avoid that overhead, at the price of increased complexity. All implemented strategies are be presented in detail below.

3.3.1. SAIO move strategy

The basic strategy is called *move*. It starts by creating a `MemoryContext` structure that is used to track all memory allocation happening during the runtime. This allows easier control over memory usage and makes sure that all resources used for the current step are freed after the step is finished. The functions entering and leaving the context, apart from tracking and freeing memory used in the context, also save the state of the internal planner relation cache, and upon leaving the context truncate it to avoid issues described in section 2.3.

```
1  function saio_move():
2      context = MemoryContext()
3      context_enter(context)
```

To execute a *move* step, two nodes from the query tree need to be chosen. The first one is picked simply at random from all existing nodes, excluding the root of the tree. The second node is chosen from all remaining nodes, eliminating the ancestors of the first node, its children and its sibling node.

```
4      choices = children(root)
5      node1 = random_node(choices)
6      choices -= (node1 + children(node1))
7      choices -= (ancestors(node1) + sibling(node1))
8      node2 = random_node(choices)
```

After choosing the nodes, the subtrees rooted at them are swapped around and the whole tree is recalculated. During the recalculation process an invalid join might be detected, as the strategy is just swapping any two nodes in the tree and this might lead to violating some of the join order constraints. If such situation is detected, the move is treated as unacceptable and reverted. The restrictions of choices for the second swap node are important. Because whole subtrees are swapped, the chosen nodes cannot be in a parent–child relationship. As the swap procedure is symmetrical, both ancestors and children of the first node must be eliminated from the choices. Because the procedure used by PostgreSQL to construct `RelOptInfos` is symmetrical, the sibling node also needs to be discarded, as swapping sibling nodes would yield the exact same `QueryTree` as before the swap. Due to all that elimination process, it might not be possible to pick the second node. In that case, the step ends in a failure.

```
9      if not node2:
10         context_exit(context)
11         return false
12     swap_subtrees(node1, node2)
13     if not recalculate(root):
14         swap_subtrees(node1, node2)
15         context_exit(context)
16         return false
```

If the reconstruction is successful, the acceptance function is called with the old state's cost, the new state's cost and the current temperature. Accepted moves are kept, unaccepted moves are reverted.

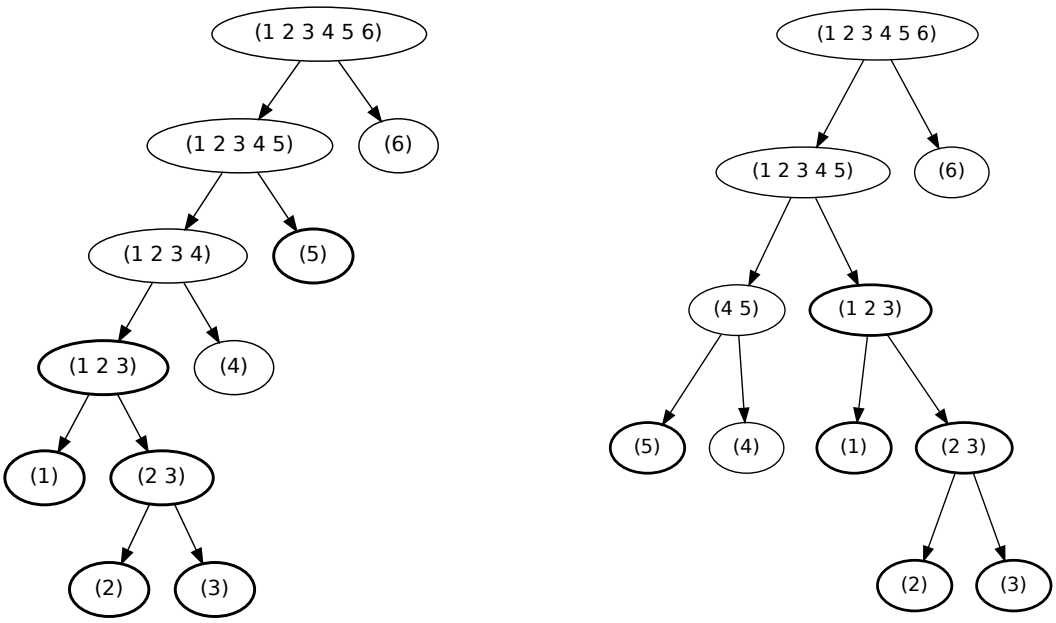
```
17      if not acceptable(old_cost, new_cost):
18          swap_subtrees(node1, node2)
19          context_exit(context)
20          return false
21      context_exit(context)
22      return true
```

The recalculation procedure is a simple recursive descent, building joinrels from lower level relations and short-circuiting the computation in case of failure. It can be given an optional argument that instructs it to omit part of the tree when rebuilding it. We will see later that some SAIO strategies take advantage of that argument.

```
1  function recalculate(node, cutoff=NULL):
2      if node == cutoff:
3          return true
4      if not node->left:
5          return true
6      ok = recalculate(node->left)
7      if not ok:
8          return false
9      ok = recalculate(node->right)
10     if not ok:
11         return false
12     node->rel = make_join_rel(node->right->rel, join->left->rel)
13     if not node->rel:
14         return false
```

The *move* strategy is useful for some cases, but also has some issues. The first is that swapping two random nodes can change the join tree radically. It might look like simulated annealing would work better with strategies that introduce localised changes, so the system could settle on a global minimum while slowly going from one state to another. It turns out that simply swapping nodes is an effective enough solution, the fact that all `Paths` are always being rebuilt compensating for the somewhat erratic exploration of the solution space. Indeed, with the current implementation the tree changes a lot, but every rebuild of the query tree is in fact visiting several solution points simultaneously.

Another problem is that if the resulting query tree represents an invalid join order, the recalculation procedure will eventually fail, rendering the whole work done in the current step useless. With queries with strict join ordering constraints it has been observed that as much as 30% of moves end up being semantically invalid and have to be discarded. All `Paths` built have to be thrown away and the algorithm cannot advance. One can see that problem as a tradeoff: higher failure rate means more runtime wasted on calculations that then get discarded, and that prohibits setting the temperature reduction speed low enough to thoroughly explore the solution space. Savings in the computational cost of each move give immediate improvements in overall plan quality, simply because the algorithm can process more states before the time it takes becomes unacceptable.



(a) A query tree with two chosen subtrees.

(b) A query tree after swapping the subtrees.

Figure 3.4: Visual explanation of the SAIO *move* strategy. Both trees are depicted after being fully rebuilt. The *move* strategy rebuilds the entire tree after each move.

3.3.2. SAIO pivot strategy

Pivot is a strategy aimed at minimising failed recalculations and localising changes to the query tree. Instead of swapping around subtrees, it operates on just three neighbour relations, trying to apply the transformation

- $(A \bowtie B) \bowtie C \rightarrow A \bowtie (B \bowtie C)$

Since the three relations involved are already being joined in the original solution, the hope is to avoid disrupting the state too much by only changing the join order between them and to have a greater chance of reaching a valid solution after executing the pivot. To avoid wasting moves on transformations yielding invalid join orders, the move is being applied to a series of nodes until it gets executed successfully.

In the context of `QueryTrees` it means choosing a node at random, excluding the root node and the leaves, and calling it the pivot root. The strategy is to execute the pivot in a loop until it succeeds.

```
1 function saio_pivot():
2     context = MemoryContext()
3     choices = children(root)
4     choices = choices - all_leaves(root)
5     while not empty(choices):
6         pivot_root = random_node(choices)
7         choices = choices - pivot_root
```

After that the sibling node of the pivot root gets swapped with one of its children. Because query trees are symmetrical it does not matter which child is chosen for the swap. The query tree then needs to be recalculated. In the pivot scenario the recalculation step is split into two parts. First the subtree below the pivot root is recalculated. The idea is that if the pivot introduced an illegal join, it will be detected faster if we focus only on its subtree.

```
8           swap_subtrees(pivot_root->left, sibling(pivot_root))
9           context_enter(context)
10          if not recalculate(pivot_root->parent):
11              swap_subtrees(pivot_root->left, sibling(pivot_root))
12              context_exit(context)
13              continue
```

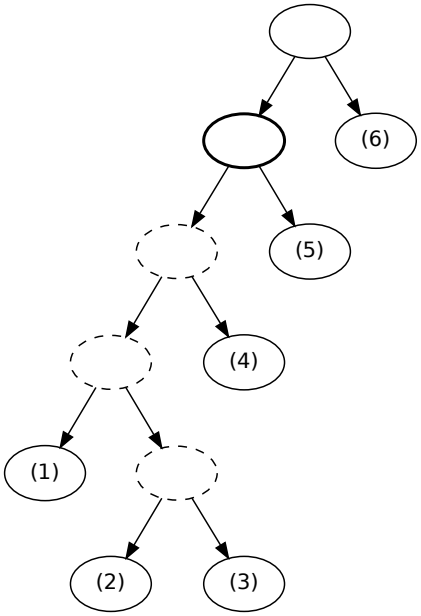
If the recalculation of the pivot root parent tree succeeded, we can be sure that the whole tree can be recalculated. This is because we know that the original tree could be recalculated and the label of the node above the pivot root is the same before and after the pivot. As `RelOptInfos` are identified by labels, we know that a relation with this label can legally appear in the position above the pivot root and we only modified the tree below it. The only way the move can now fail is if the acceptance function rejects it. To skip recalculation of the already built part the algorithm uses the cutoff parameter of the helper function.

```
14          ok = recalculate(root, pivot_root->parent)
15          assert ok
16          if not acceptable(old_cost, new_cost):
17              swap_subtrees(pivot_root->left, sibling(pivot_root))
18              context_exit(context)
19              continue
20          context_exit(root)
21          free(choices)
22          return true
```

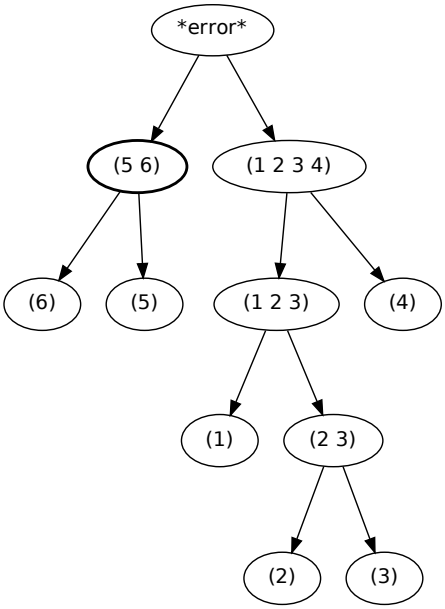
If after iterating through all the choices we fail to rebuild the whole tree, the move ends in a failure. This happens if the loop condition from line 5 becomes false.

```
23      assert empty(choices) # while loop exited
24      return false
```

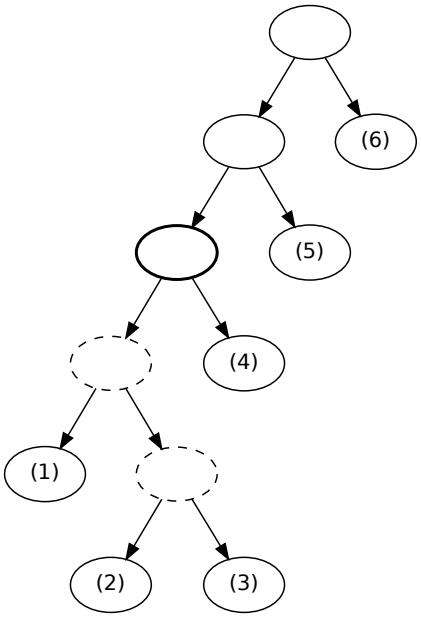
The *pivot* strategy can be effective on queries with few join order constraints. It lowers the probability of an odd constraint ruining the whole move, because in that case the pivot root will be simply pushed downwards in the tree, until a successful move is found. The price for that is big potential for duplicate computation, if the first pivot root happens to be high in the tree and several attempts at pivoting result in semantically invalid join orders. In fact, during one move execution the exact same join can be computed many times. Another group of problematic queries are ones that initially have very large cost due to one particularly badly chosen join. Localising changes to the tree does not pay off in this case, because until the pivoted relations happen to be the ones involved in the pessimal join, the cost will stay high. Overall, the *pivot* strategy leaves much to be desired and has no clear advantages over the simpler *move* strategy.



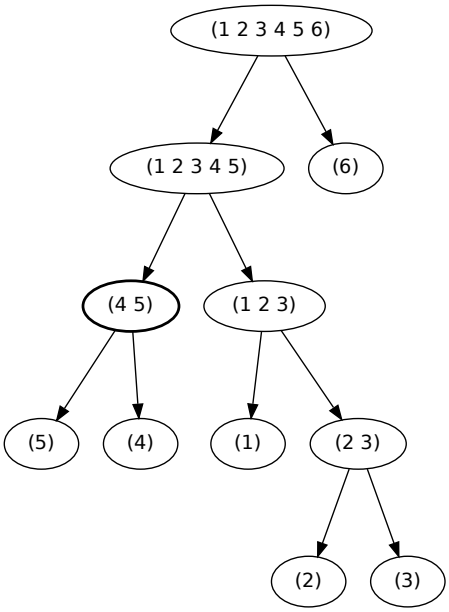
(a) A query tree with a pivot root and the rest of choices.



(b) A query tree after executing the pivot. The root node construction has failed.



(c) In case of failure, another next pivot root is chosen.



(d) Another pivot is attempted.

Figure 3.5: Visual explanation of the SAIO *pivot* strategy. One failed attempt at rebuilding the tree does not make the whole move to fail.

3.3.3. SAIO recalc strategy

An attempt to solve the deficiencies of *move* and *pivot* strategies, *recalc* is the most complex of all move generation routines that are currently implemented in SAIO. It is based on *move*, but differs from other strategies by not freeing all memory used by the previous step every time. Instead it tries to reuse the work done in previous moves and minimise wasted effort coming from repeated recalculation of the same joins.

Since the built `RelOptInfos` are being preserved between strategy invocations, they cannot all be kept in one `MemoryContext`. Instead, the recalc strategy keeps a separate memory context for each node of the query tree. This way it can free the resources used by a single `RelOptInfo` structure and all `Paths` that it contains. The `QueryTree` structure is also extended with a field to temporarily store another `RelOptInfo`. It is used to keep the results of partial recalculations separated from the last known to be valid move. The initialisation procedure for the *recalc* strategy rebuilds all join relations in the tree and sets the temporary relation field to NULL.

```
1 function init_recalc_strategy(node):
2     if not node:
3         return
4     if node->left:
5         node->rel = make_join_rel(node->left->rel,
6                                   node->right->rel)
7     node->tmp_rel = NULL
8     init_recalc_strategy(node->left)
9     init_recalc_strategy(node->right)
```

The strategy uses some helper functions, that first need to be introduced. The first one applies a boolean function to each element of a list, but stops if the function application returns false as a result. It is useful for implementing short-circuiting operations on lists of query tree nodes. The helper returns true if all elements of the list have been processed and false otherwise.

```
1 function walk_list(list, fun, extra_arg):
2     for elem in list:
3         if not fun(elem, extra_arg):
4             return false
5     return true
```

Then there are some functions that are used to transform parts of the query tree. They all take a single node as an argument, and a boolean extra argument that specifies whether they should operate on the temporary relation or the actual relation stored in the node. They are designed to be used with the previously defined list walking function. The `recalc` function recalculates the content of a node and stores the result either as the new node's relation or using the new temporary relation field, depending on the second boolean argument. An additional complication is deciding which fields from the child nodes should be used to form the joinrel. The rule is that if the temporary relation is present, it is chosen. Otherwise the main relation is used.

```
1 function recalc(node, real):
2     left = node->left->tmp_rel or node->left->rel
3     right = node->right->tmp_rel or node->right->rel
4     joinrel = make_join_rel(left, right)
```

```
5       if not joinrel:
6           return false
7       if real:
8           node->rel = joinrel
9       else:
10          node->tmp_rel = joinrel
11      return true
```

The `remove` function frees the `RelOptInfo` and all `Paths` contained in it. The boolean argument is used to decide whether the real relation or the temporary one should be freed. Apart from freeing resources it also removes the relation from the global planner cache, forcing a rebuild the next time a join with that label will have to be formed.

```
1  function remove(node, real):
2       if real:
3           rel = node->rel
4       else:
5           rel = node->tmp_rel
6       it rel == NULL:
7           return true
8       remove_from_planner(rel)
9       free(rel)
10      return true
```

The `nullify` and `switch` functions are simple and self-explanatory. Observe that the former pays attention to the boolean argument, while the latter does not.

```
1  function nullify(node, real):
2       if real:
3           node->rel = NULL
4       else:
5           node->tmp_rel = NULL
6       return true
7
8  function switch(node, real):
9       t = node->rel
10      node->rel = node->tmp_rel
11      node->tmp_rel = t
```

Finally, two functions are used to return the nodes that are common ancestors of two given nodes and to return the symmetric difference of their ancestors.

```
1  function common_ancestors(node1, node1):
2     a1 = ancestors(node1)
3     a2 = ancestors(node2)
4     return a1 ∩ a2
5
6  function exclusive_ancestors(node1, node2):
7     a1 = ancestors(node1)
8     a2 = ancestors(node2)
9     return a1 Δ a2
```

With these functions defined we can write the main part of the algorithm. It handles swapping two trees and recalculating parts of it, making sure wasted effort is kept minimal. The first step is swapping two nodes, just like in the *move* strategy, and gathering their ancestors. During the whole move we will only be rebuilding nodes that are ancestors of the two chosen nodes. The rest is not affected by swapping them around, and because we keep built relations between strategy invocations, we are sure they already exist in the planner.

```
1  function swap_and_recalc(node1, node2):
2      swap_subtrees(node1, node2)
3      ca = common_ancestors(node1, node2)
4      ea = exclusive_ancestors(node1, node2)
```

The exclusive ancestors are then recalculated. The results are put in the temporary relation slot. Note that none of the relations that will be rebuilt in this phase could have existed in the original tree, and so they are not in the global planner cache. This means that we do not have to remove any relations from the planner before executing this step. This also means that in case of failure during this step, we simply need to remove the newly built relations from the planner and clean up the temporary relation slots. After swapping the subtrees back, we will reach the original state, with all nodes containing valid `RelOptInfos`.

```
5      ok = walk_list(ea, recalculate, false)
6      if not ok:
7          walk_list(ea, remove, false)
8          walk_list(ea, nullify, false)
9          swap_subtrees(node1, node2)
10         return false
```

The common ancestors are a bit more problematic. The relations with these labels already exist in the planner cache, so we first need to remove them to force a rebuild after swapping subtrees in line 2. If the recalculation succeeds, we have thus rebuilt the root relation and can compare costs to determine acceptance.

```
11     walk_list(ca, remove, true)
12     walk_list(ca, nullify, true)
13     ok = walk_list(ca, recalculate, false)
14     if ok:
15         ok = acceptable(old_cost, new_cost)
```

If recalculating the common ancestors failed, they relations built so far need to be removed from the planner and the original state needs to be restored. Note that we do not have to recalculate the exclusive ancestors, for the same reasons we did not have to do it in lines 6–10.

```
16     if not ok:
17         walk_list(ea, remove, false)
18         walk_list(ea, nullify, false)
19         walk_list(ca, remove, false)
20         walk_list(ca, nullify, false)
21         swap_subtrees(node1, node2)
22         walk_list(ca, recalculate, true)
23         return false
```

After the common ancestors are rebuilt and the acceptance function returns true, we only have to promote the already built `RelOptInfos` to the main relations of the nodes. We first

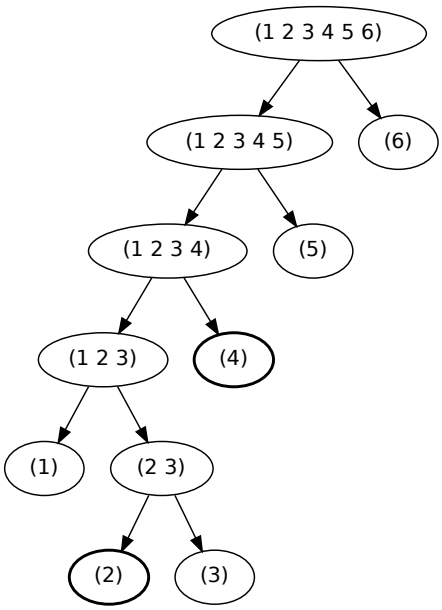
promote the exclusive ancestors, removing the relations that formerly were the main ones and replacing them with the temporary ones. After that we switch the roles of relations in common ancestors. Note that we already removed their old versions in line 11. Finally we clean the temporary relation slot.

```
24      walk_list(ea, remove, true)
25      walk_list(ae, switch_contexts, true)
26      walk_list(ca, switch_contexts, true)
27      walk_list(ea, nullify, false)
28      walk_list(ca, nullify, false)
29      return true
```

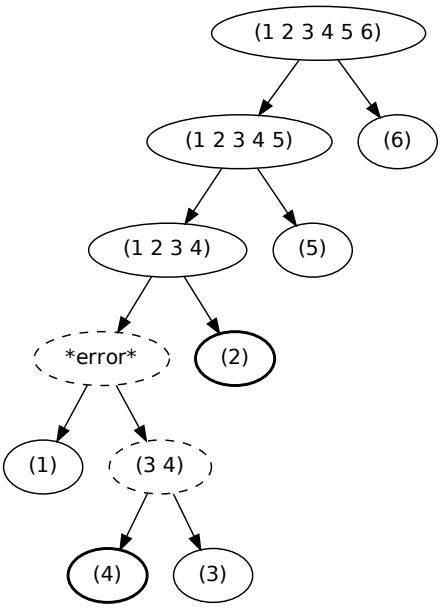
With that function defined, it is easy to write the main *pivot* strategy procedure. It is similar to the *move* procedure, but simple swapping is replaced by the `swap_and_recalc` function that takes care of rebuilding necessary parts of the query tree.

```
1  function saio_recalc():
2      choices = children(root)
3      node1 = random_node(choices)
4          choices -= (node1 + children(node1))
5          choices -= (ancestors(node1) + sibling(node1))
6      node2 = random_node(choices)
7      if not node2:
8          return false
9      if not swap_and_recalc(node1, node2):
10         return false
11     return true
```

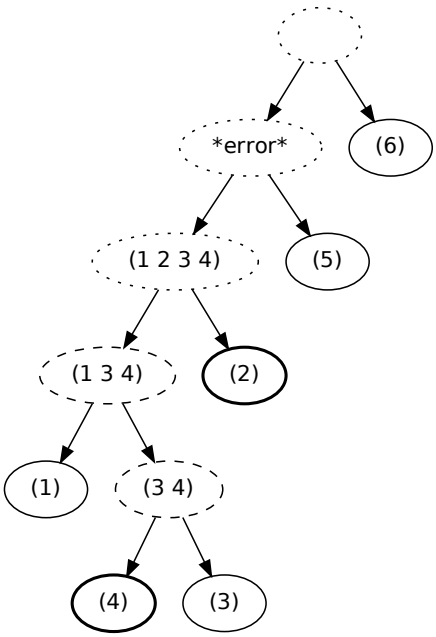
The *recalc* strategy turns out to be the one that gives best results. Complexity brought be tracking each `RelOptInfo` separately is compensated by lower CPU consumption due to recalculations. Simple subtree swapping turns out to be an efficient enough method of morphing the query tree compared to pivoting. Because the algorithm takes care to avoid needless computation, localising tree changes is less important. On the other hand, *recalc* does not do anything to lower the number of invalid states visited by the algorithm. It only makes the failures be detected earlier and recovery from them cheaper. As we will see in chapter 4 it is enough to get comparable or even better results than GEQO.



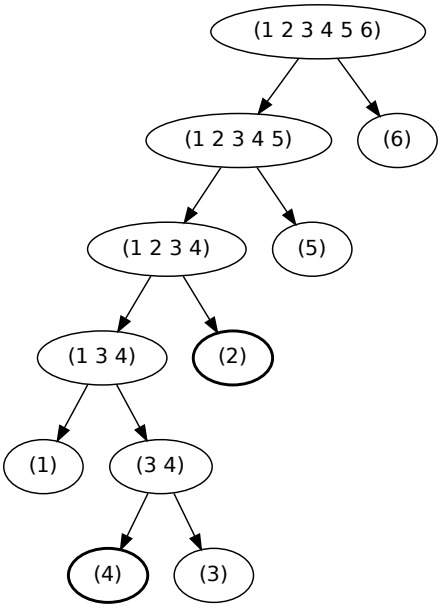
(a) A query tree with two chosen nodes.



(b) In case of an early failure no relations are rebuilt, just the ones drawn as dashed are discarded.



(c) If the failure occurs when building a common ancestor the dashed nodes get discarded and the dotted ones need to be rebuilt.



(d) If there are no failures, the whole tree is rebuilt.

Figure 3.6: Visual explanation of the SAIO *recalc* strategy. Only the necessary relations are rebuilt in case of a failure.

Chapter 4

Results

This chapter presents the comparison results between GEQO and SAIO. Comparisons with the standard planner module are not done, as they would usually not be possible. The standard planner, if forced to run over the test cases presented here, would quickly exhaust all available memory. Next we present resource consumption reports that compare memory usage, CPU utilisation and plan quality of SAIO according to various algorithm parameters. As mentioned in section 1.3, the quality of the plan is measured by the estimated total execution cost. The goal of the optimiser is minimising the estimated cost, not the execution time — the latter is a task of the executor. None of the test queries were actually executed, only planned.

4.1. Test environment

Measurements were done with two different queries provided by members of the PostgreSQL community. It has been a deliberate choice to use real-world queries that originated in production systems, rather than ones that have been automatically generated. The simulated annealing module aspires to be a general purpose optimiser plugin and using artificial queries would risk not stressing critical parts of the algorithm, like ones responsible for handling outer joins. One of the queries used is moderately complex, with around 100 relations. The other one is more complex, with relation number in the range of 1000. Both queries in question and the schema definition can be found in the code appendices.

All tests were done using the development version 9.1 of PostgreSQL as of July 2010, compiled from source. The cluster was running on an Intel Core2 Quad CPU machine with a 2.83GHz processor clock and 8GB of main memory. No modifications were done to the default configuration file of the database cluster. It is important to note that all tests were conducted after disabling the limits discussed towards the end of 2.1.1, namely the query flattening and subquery pulling limits. The idea behind that is that they constraint the solution space to a small subset in which it is difficult to immediately see the quality of an algorithm, as it has little room to prove its usefulness.

4.2. Comparison with GEQO

The first group of tests is a head to head comparison of execution time, memory consumption and plan quality between GEQO and SAIO. The genetic optimiser was run with the default settings, the simulated annealing planning was repeated with different algorithm parameters. Each query has been planned five times and the average results for all values are presented.

Let us also remind the formulas for the number of equilibrium moves, initial temperature and cooldown factor.

$$moves\_to\_equilibrium = N \times initial\_rels$$
$$initial\_temperature = I \times initial\_rels$$
$$new\_temperature = temperature \times K \text{ where } 0 < K < 1$$

algorithm	N factor	I factor	K factor	cost	runtime	memory
GEQO	n/a	n/a	n/a	1557.470000	0.523016	43849.600000
SAIO	4	2.000000	0.600000	1576.644000	0.389347	44441.600000
SAIO	6	2.000000	0.400000	1584.180000	0.323950	44464.800000
SAIO	6	2.000000	0.600000	1569.574000	0.567734	44577.600000
SAIO	8	2.000000	0.400000	1572.380000	0.453518	44476.000000
SAIO	12	2.000000	0.600000	1575.320000	1.262268	44746.400000
SAIO	12	2.000000	0.800000	1574.166000	3.009681	45197.600000

Table 4.1: For a moderately sized query GEQO gives slightly better results

For the first test query GEQO produces better plans than any combination of SAIO parameters. On the other hand, the difference in cost is not large and with low number of loops before reaching equilibrium and a large temperature reduction factor SAIO is significantly faster. Memory consumption for both modules is very similar, which is not unexpected, as they both release most used memory after reconstructing the root `RelOptInfo`, keeping just small amounts of bookkeeping information.

algorithm	N factor	I factor	K factor	cost	runtime	memory
GEQO	n/a	n/a	n/a	22417.210000	809.662594	227880.000000
SAIO	4	2.000000	0.600000	21139.506000	122.251778	196618.400000
SAIO	6	2.000000	0.400000	20853.768000	100.508593	199447.200000
SAIO	6	2.000000	0.600000	20921.834000	193.073352	205004.000000
SAIO	8	2.000000	0.400000	20698.076000	146.922708	195714.400000
SAIO	12	2.000000	0.600000	20584.972000	408.599637	218168.800000
SAIO	12	2.000000	0.800000	timeout	timeout	244882.400000

Table 4.2: For a very complex query GEQO gives clearly inferior results

The second test query is much larger, with thousands of relations taking part in the join ordering process. It also includes numerous outer joins, which create strict join order restrictions and cause many moves to be invalidated for semantic reasons. On such a query with the flattening limits removed SAIO is clearly superior than GEQO, both in runtime and in plan quality. However, for very high parameter values it failed to provide a result within a limit of 900 seconds. A dependency between the equilibrium and temperature reduction factors and result quality and execution time can also be observed.

4.3. Influence of SAIO parameters

Choosing different parameters for the simulated annealing algorithm can can influence both the quality of the results and the runtime. It is quite obvious that increasing the K factor and decreasing the I factor will increase runtime. It turns out that for moderately sized queries it does not materially improve the output quality. This result suggests that for queries that are not sufficiently large the benefits of simulated annealing are not fully visible.

On the other hand, results for the more complex query reveal that increasing the number of loops the annealing algorithm does before reaching equilibrium does influence the obtained result. The experiments confirm an intuitive understanding — more computation before declaring the state to be in equilibrium give the system more chances to reach a lower cost, which in turn produces better plans. The same happens with the temperature reduction factor. The higher it is, the more fine grained the process of cooling down the system is and the plan has the opportunity to gradually move towards a more optimal state. Sudden temperature drops that happen with cooling factors in the order of 0.5 make it more difficult to obtain satisfying results. An interesting observation is that even if the temperature reduction factor is very low, high number of equilibrium loops can still shift the balance enough for the algorithm to produce good plans. A possible explanation might be that a lot of moves are rejected as invalid, and so the temperature does not play such a big role. Essentially, the acceptance function rarely gets called. Making the inner loop longer makes finding a succeeding move more probable, and so large equilibrium loops factor can outweigh a low temperature reduction factor.

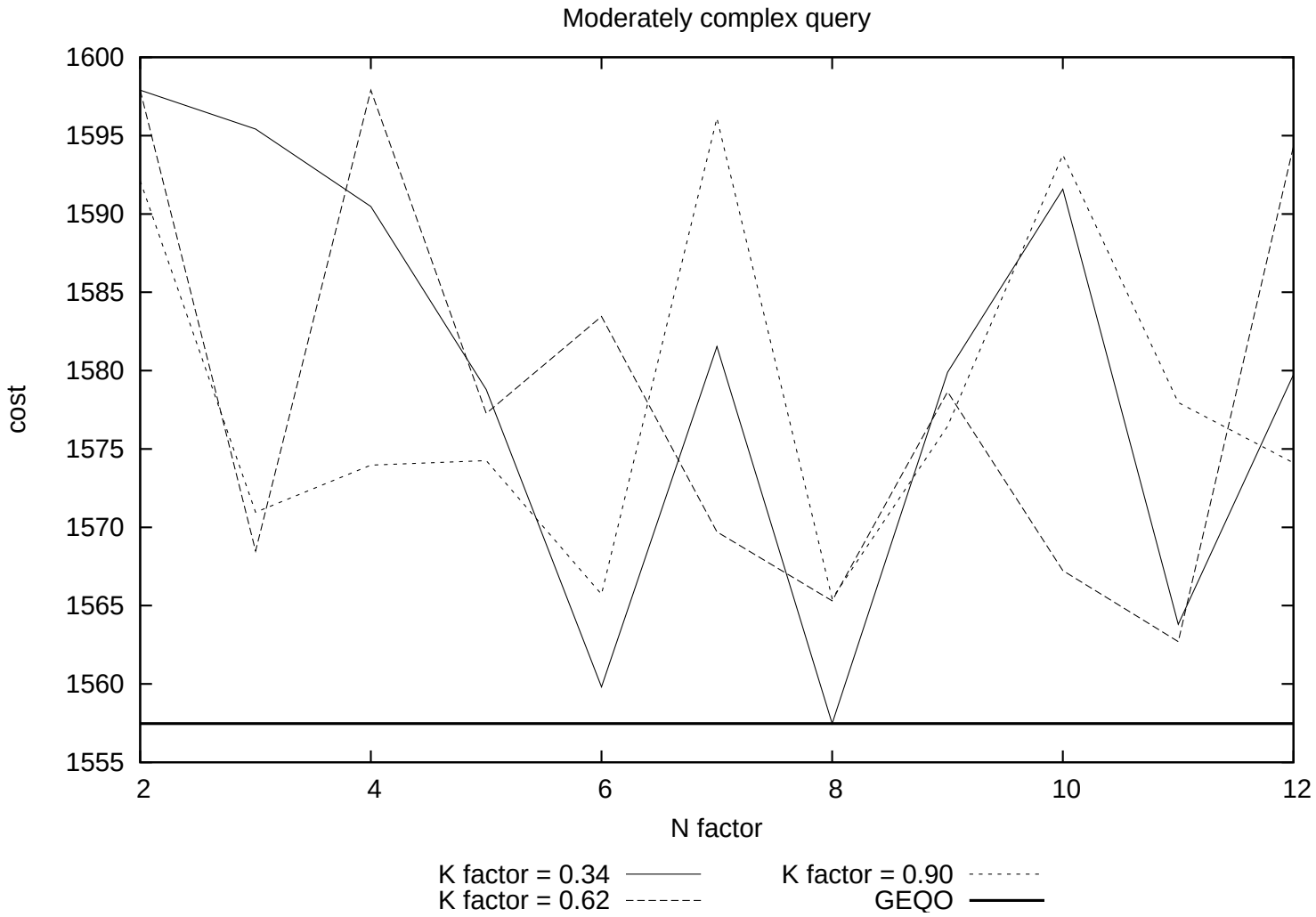


Figure 4.1: The output costs are not very different for a query that is not sufficiently large.

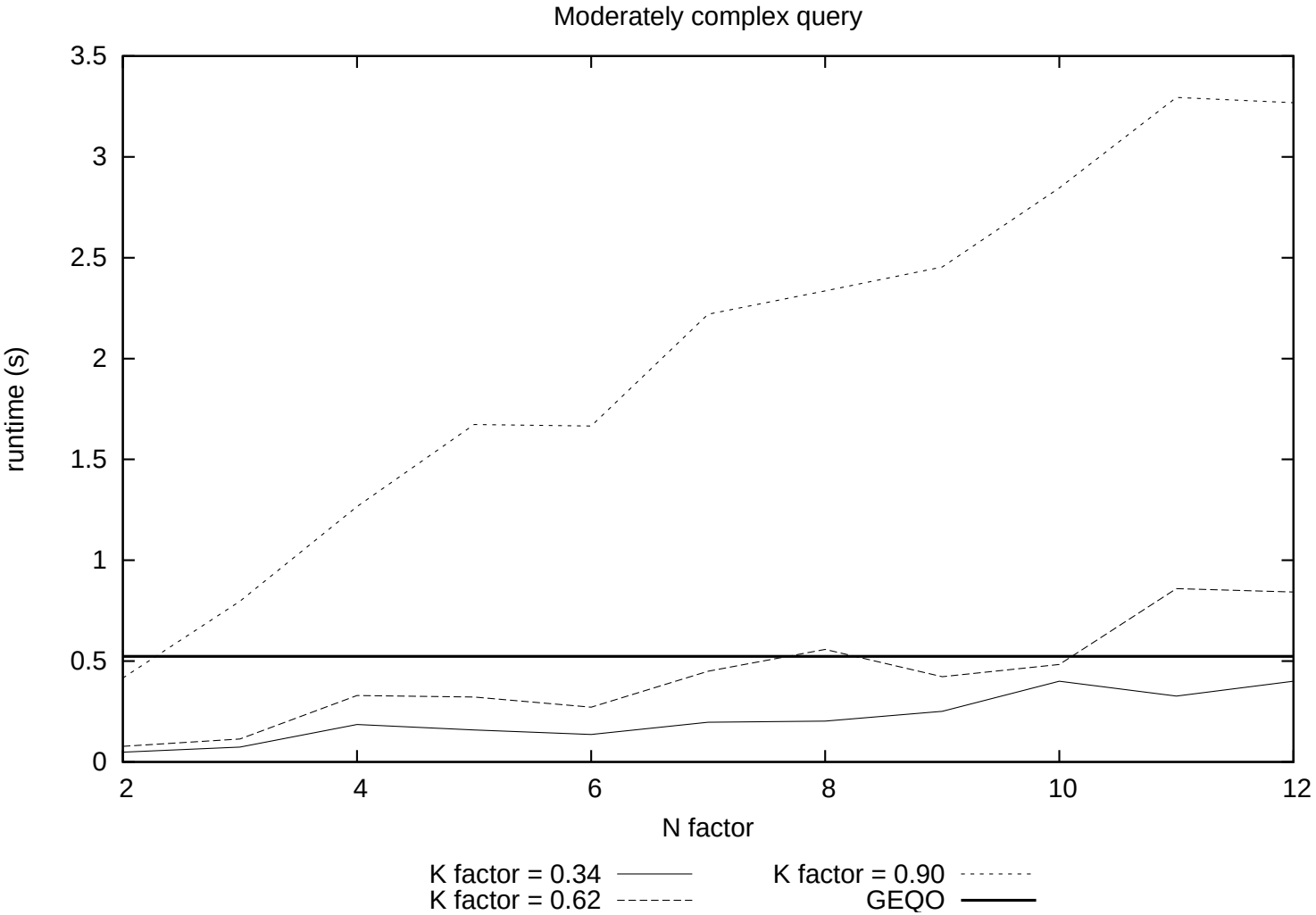


Figure 4.2: Runtime increases as the temperature reduction falls and equilibrium loops increase.

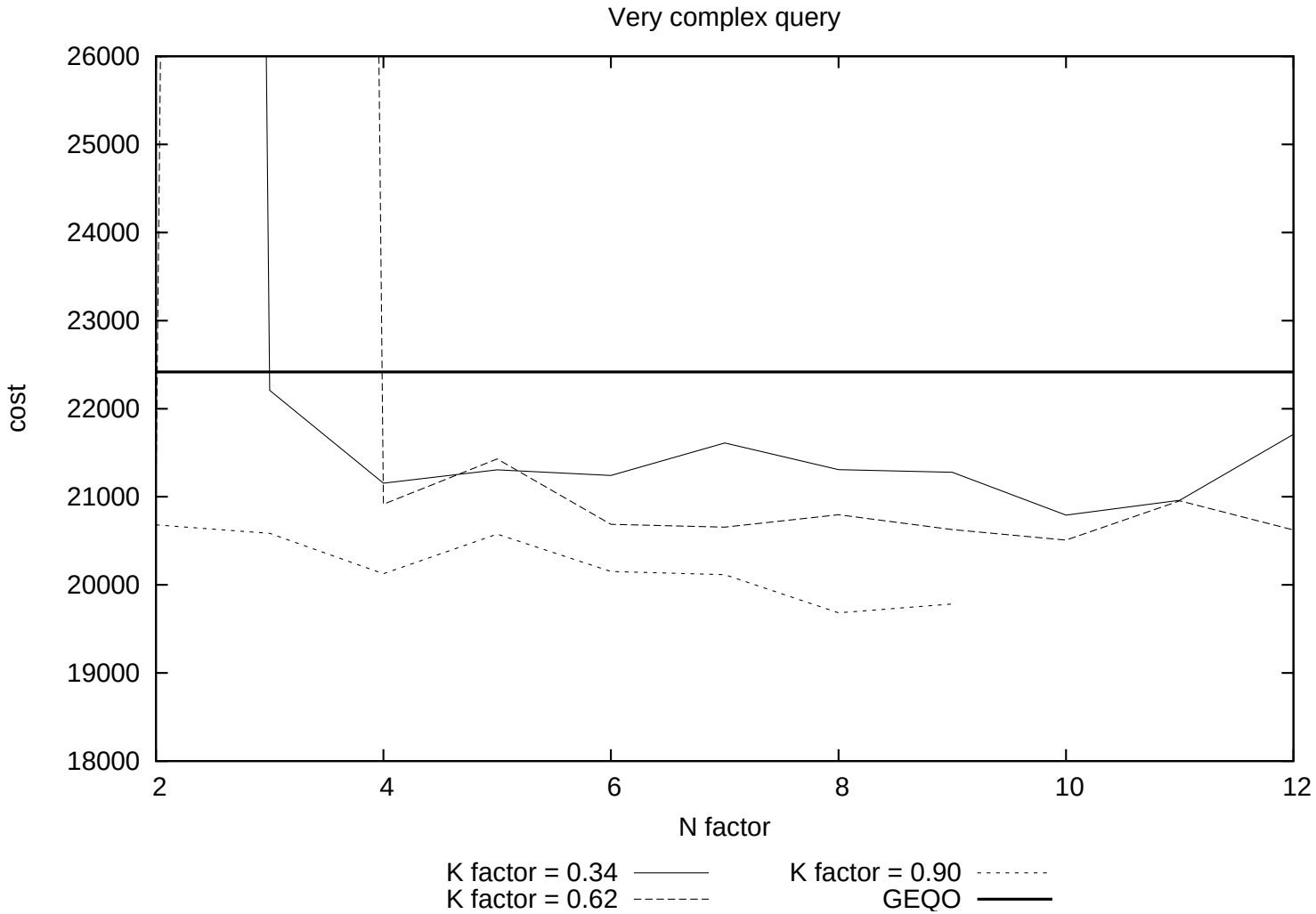


Figure 4.3: For complex queries simulated annealing reaches better plans than the genetic algorithm, but for small cooling factors it starts timing out.

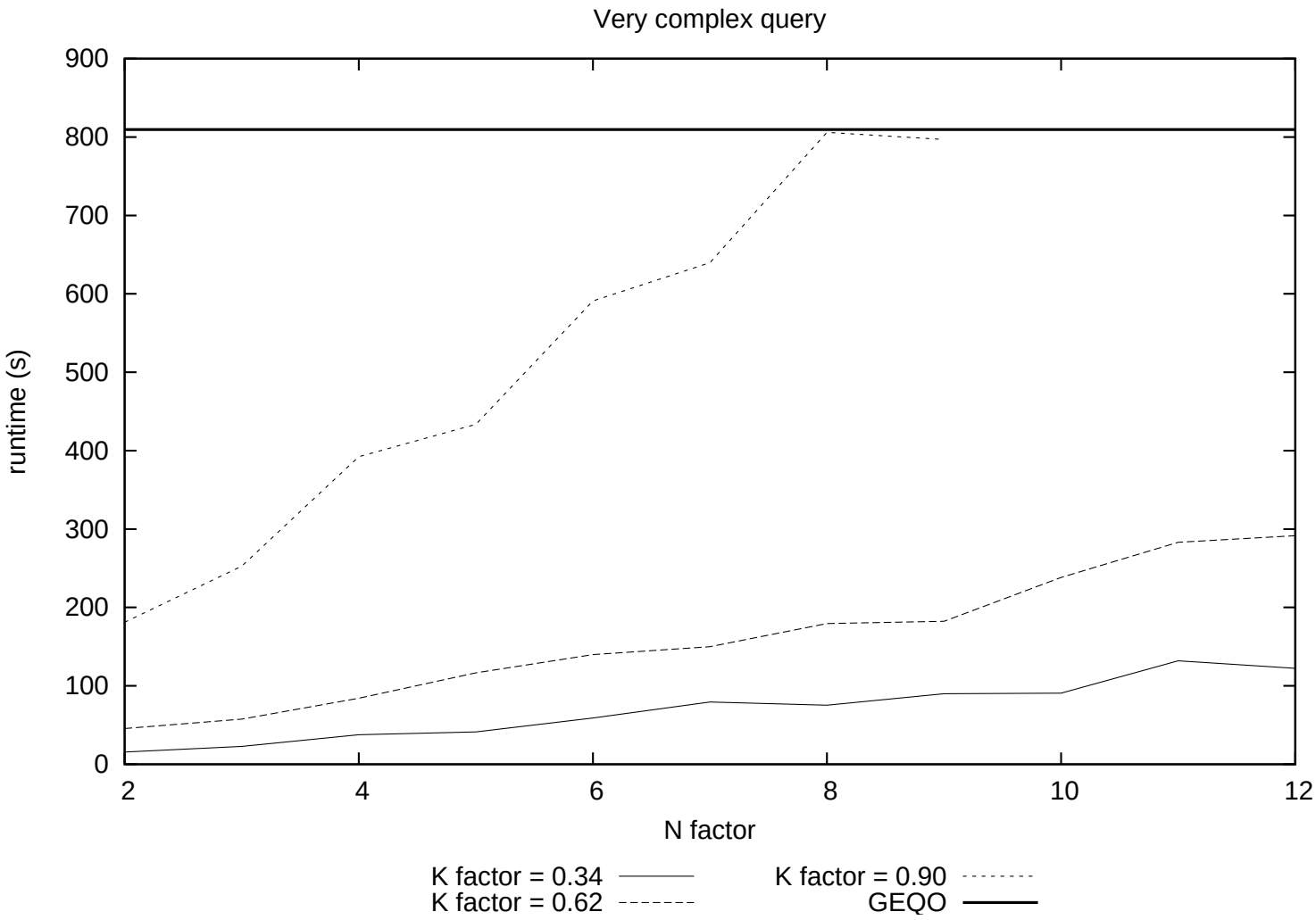


Figure 4.4: The runtime for complex queries is much smaller for SAIO, if the parameters are chosen wisely. For large equilibrium loops and small cooling factors, the times reach values close to GEQO, and then start to grow uncontrollably.

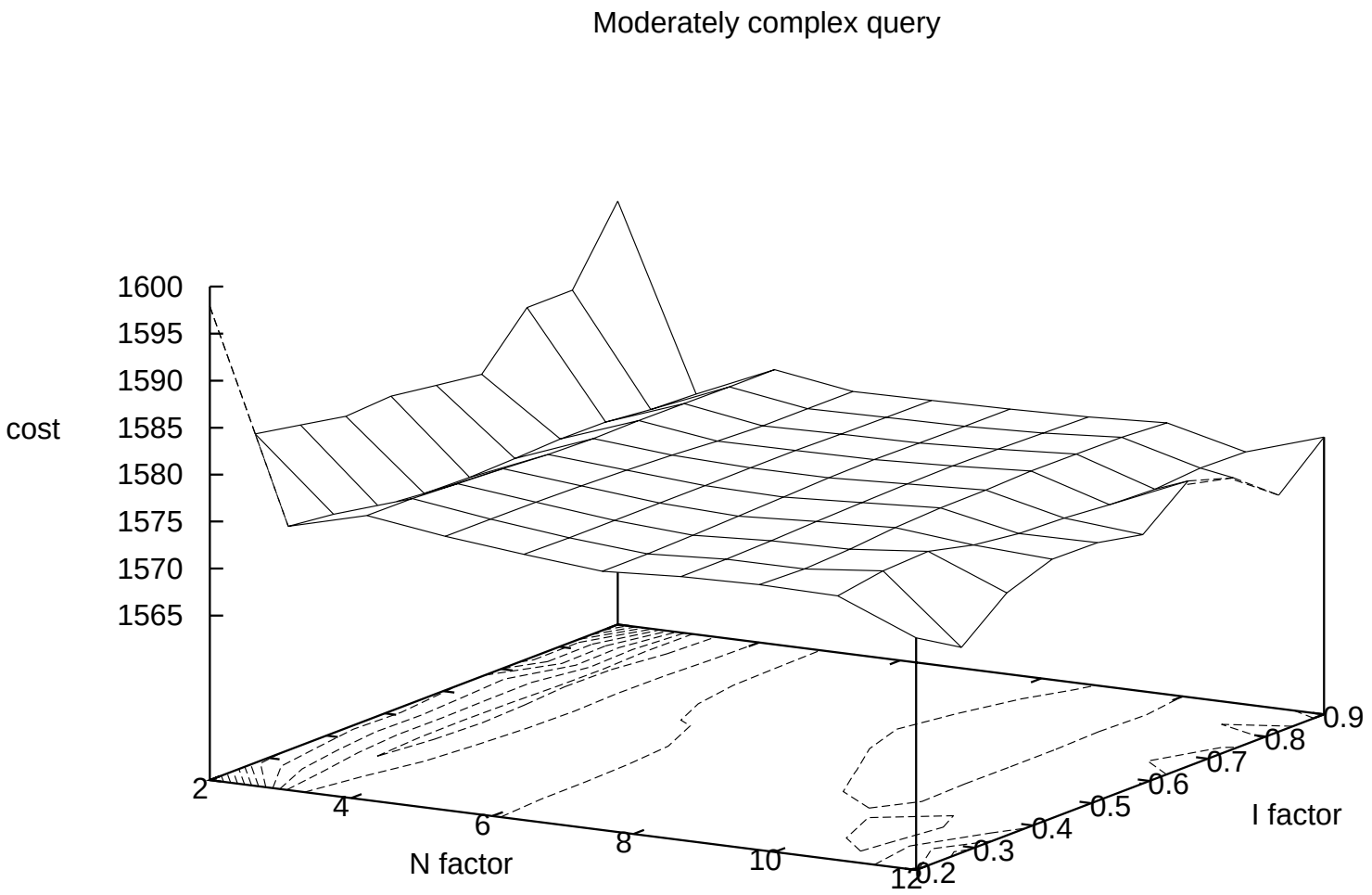


Figure 4.5: In a moderately complex query the result quality oscillates regardless of annealing parameters.

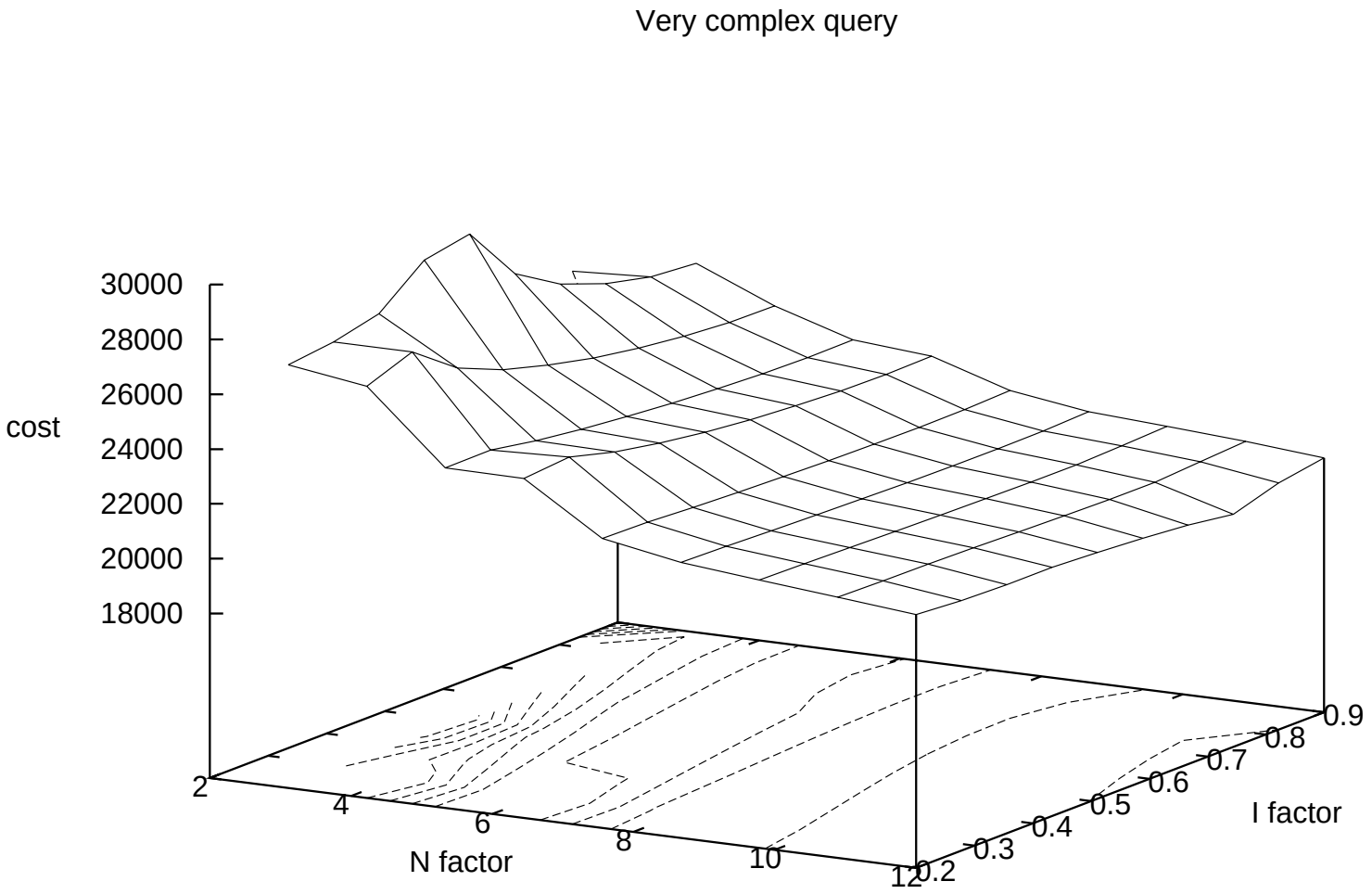


Figure 4.6: In a complex query the plan quality becomes better when cooling factor drops and equilibrium loops raises.

Chapter 5

Future development

There is a number of improvements that could be implemented in SAIO to address some of its issues, make it produce better plans and take less CPU time. Some of them are enhancements of the existing algorithm, others are different approaches that could be integrated into the module code. Many implementation choices done during the development of SAIO were influenced by the way the PostgreSQL planner works internally and by the interface that it exposes. The module has been developed without any modifications to the source code of PostgreSQL. As a result, it contains some workarounds and unnecessary complexity that stems solely from the need to integrate with the structures used by PostgreSQL. When the module will be considered mature enough, changes in the upstream system could be requested. A number of core developers expressed positive views on modifying the standard query planner to make modules such as SAIO easier to write.

In this chapter a few ideas of improvements for SAIO will be discussed. They range from simple and straightforward to generic and open-ended. All of them have as the goal improving plan quality — ways of introducing better support for simulated annealing into PostgreSQL will not be covered here.

5.1. Cost estimation function

The cost estimation function used by SAIO simply looks at the cost of the cheapest `Path` in the root `RelOptInfo`. This is exactly what the existing genetic algorithm does, but it is not ideal. First, it ignores the startup cost, which for queries planned for usage in SQL cursors can result in reduced plan quality, as perceived by the user. Addressing this would be relatively straightforward. One would have to take note of the planner flag telling whether the query is planned for full execution or as part of creating a cursor and use the cheapest startup `Path` accordingly.

A much more complex issue is whether the total cost is the correct value to use when comparing plans. One idea would be to include the number of Cartesian joins in the cost computation. Joins without join conditions usually result in increasing the planned runtime cost, but sometimes that increase is not big enough to make a move unacceptable. If the cost estimation function would heavily penalise plans that *increase* the number of Cartesian joins, it could prevent the algorithm from generating such join orders, and this in turn would lower the risk of the Cartesian join staying in the tree forever, due to the nodes that form it never again being chosen for the swap. Not that GEQO never creates Cartesian joins, unless it is the only way to produce a valid join order. This is a difference from SAIO, where only the initial state is generated with care to avoid Cartesian joins. After that the module trusts

that clauseless joins will cause such cost increase, that they will always get rejected by the acceptance function.

5.2. Smarter move generating

The three implemented strategies still leave a lot do be desired. In section 5.1 we discussed the dangers of introducing Cartesian joins. Even under the assumption that the acceptance function will reject trees with too many joins without join constraints, it means that all computation done in the move will be wasted. That is the reason why sometimes GEQO obtains better results than SAIO — it avoids cross joins in every iteration of the optimisation loop. An intelligent move generating strategy would recognise clauseless joins and avoid continuing with the move if one is detected, or at least would include a random (but rather strong) rejection mechanism for such situations. The implemented *recalc* strategy includes a prototype of that approach, omitted in the pseudocode in 3.3.3 for brevity, but it still could get improved.

Another omitted optimisation the *recalc* strategy has is early rejection of joins that turn out to produce much more costly plans than the original join order. The idea behind it is that sometimes a swap of two subtrees can result in one particularly expensive join, that will then carry its cost up to the higher levels of the query tree, eventually causing the whole move to be rejected, but forcing all `RelOptInfos` taking part in the join to be recomputed. What *recalc* does to mitigate the problem is that when rebuilding each node, if the join relation has been successfully built, it checks if its cost is not too high. More specifically, it compares the cost of that particular `RelOptInfo` with the cost of the root `RelOptInfo` from the previous state. If the former is higher, the total cost of the tree after rebuilding everything will certainly be at least as high, so the acceptance function will be invoked. Instead of waiting for the rest of the relations to be rebuilt, *recalc* checks for acceptance immediately, using the cost of the current `RelOptInfo`. If it gets rejected, the join is treated as if it has been illegal. This of course means that if the acceptance function returns true, rebuilding the parent of the current `RelOptInfo` will also cause an acceptance test, because the parent's cost will include the cost to construct the child. Experiments show however, that inner nodes with cost higher than the previously built root node are almost always signs of a tree that will eventually be rejected, so the heuristic with early acceptance testing is justified.

Finally there is the question of invalid join orders. If the random swap results in a join ordering that is semantically invalid, the whole move is wasted. Currently the *recalc* strategy focuses on early detection of such situations, but ideally that would be avoided altogether. The GEQO module, because of the way it rebuilds the join tree encoded in the `Chromosome`, does not have this problem. Every `Chromosome` represents a valid join order and can be used to construct a single `RelOptInfo` that represents the result of the query. The difficulty of avoiding invalid join orders lies in the way PostgreSQL keeps track of which relations can and which cannot be joined to each other. Designing an algorithm that would be capable of choosing nodes to swap while being able to guarantee validity of the resulting tree is an interesting and nontrivial undertaking in itself. Even more difficult would be making such algorithm fast enough to avoid dominating the CPU cost of the planning process. If implemented correctly, it could potentially vastly increase the quality of plans SAIO is able to produce.

Somewhat related is the fact that internal PostgreSQL procedures always consider all `Paths` when building a `RelOptInfo`. Typically, a simulated annealing algorithm would regard changing the access pattern for a relation as a valid move. PostgreSQL folds all these possibilities into one move, the result being very expensive `RelOptInfo` construction. An

indirect result of this is that failed moves are so destructive — recalculation costs lots of CPU time and the dominating factor in the process is join relation construction.

5.3. Two phase optimisation (TPO)

Algorithms based on simulated annealing work best if their starting point is not too far from an acceptable solution. If the initial state is chosen completely at random, the first couple of moves will not bring it to a more reasonable one quickly. This is due to the temperature being initially high, and so the chances of taking an uphill move are also substantial. On the other hand, if the starting tree is already in the part of the solution space with small local minima and little differences between neighbouring states, simulated annealing works fairy well. This observation has led to the proposal of two phase optimisation [6]. In TPO the optimisation process consists of two parts. In the first one, the acceptance function is set to reject every uphill move. This part is sometimes called iterative improvement. It is aimed at reducing the initial cost quickly without straying too much due to accepting moves that do not really help avoiding local minima, as the algorithm is still in the range of very high costs. After the first phase ends, a simulated annealing process is initiated, starting from the point in which iterative improvement ended. Yet another variant of TPO starts with several randomly chosen points and runs iterative improvement on all of them. The one that yields lowest cost is takes as a base for simulated annealing.

Adding TPO features to SAIO would in principle be straightforward. All needed functions and infrastructure is already implemented. However it is uncertain if two phase optimisation would be able to improve results significantly. As we have seen, the biggest problem for SAIO is high calculation cost of single relations. TPO would only eliminate the cost of executing the acceptance function, which is completely overshadowed by CPU time necessary to build a relation. The optimisations present in the *recalc* strategy could indiscriminately reject nodes with higher cost than the previous tree, but the savings are likely to be small.

Bibliography

[1] Unknown author. Wikipedia article on annealing. `http://en.wikipedia.org/wiki/Annealing_(metallurgy)`.

[2] L. D. Davis and Melanie Mitchell. Handbook of genetic algorithms. *Van Nostrand Reinhold*, 1991.

[3] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[4] D. Goldberg and R. Lingle Jr. Alleles, loci, and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and their Applications*, 1985.

[5] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[6] Y. E. Ioannidis and Younkyung Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 312–321, New York, NY, USA, 1990. ACM.

[7] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. *SIGMOD Rec.*, 16(3):9–22, 1987.

[8] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892, 1989.

[9] Rosana S. G. Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 493–504, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[10] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 191–202. VLDB Endowment, 2002.

[11] Thomas Neumann. Query simplification: graceful degradation for join-order optimization. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 403–414, New York, NY, USA, 2009. ACM.

[12] I. Oliver, D. Smith, and J. Holland. Study of permutation crossover operators on the traveling salesman problem. In *Proceedings of Second International Conference on Genetic Algorithms and their Applications*, 1987.

[13] Wolfgang Scheufele and Guido Moerkotte. On the complexity of generating optimal plans with cross products (extended abstract). In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 238–248, New York, NY, USA, 1997. ACM.

[14] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM.

[15] T. Starkweather, S. Mcdaniel, D. Whitley, K. Mathias, D. Whitley, and Mechanical Engineering Dept. A comparison of genetic sequencing operators. In *Proceedings of the fourth International Conference on Genetic Algorithms*, pages 69–76. Morgan Kaufmann, 1991.

[16] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

[17] Arun Swami and Anoop Gupta. Optimization of large join queries. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 8–17, New York, NY, USA, 1988. ACM.

[18] L. Darrell Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 116–123, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.