LINEAR PROGRAMMING
OPTIMIZATION OF QUERY
ALLOCATION IN A
DISTRIBUTED CEP SYSTEM

Annica Ivert

Bachelor Thesis, Royal Institute of Technology, 2012
Acknowledgement

First of all, I want to thank my examiner Johan Montelius for believing in me and providing the contacts to make this thesis possible. I also want to thank Haruyasu Ueda for his support and for a warm welcome, both at Fujitsu and in his home. Also, thank you, Emeric Viel, not only for the help and discussions around this thesis, but also for helping me with anything that might have been troublesome for a foreigner in Japan. You are a true French gentleman!

I also want to thank the Scandinavia-Japan Sasakawa foundation for covering my traveling expenses to/from Japan.

And last by not least, a special thanks to everyone at Fujitsu for making me feel welcome!
Abstract

In Fujitsu’s up-and-coming Complex Event Processing service, the servers used for query-storage lie in a cloud environment. The cost of setting up the user-defined systems can therefore immediately be translated into the number of servers that is required and the amount of data that need to be sent between the servers.

The objective of this thesis was to provide a model that optimizes the cost of setting up these systems. The problem of query allocation has been modeled a linear program, such that the number of servers needed and the communication between them is minimized.

This turns out to be a kind of bin-packing-problem, which is known to be NP hard. Attempts to optimize the execution time were therefore made by implementing a customized solver that combined optimizations based on the problem-specific knowledge with the branch-and-bound-technique generally used by LP (Linear Programming) solvers. The customized solver was then compared to the commonly-used GLPK-solver and the comparison showed that the customized solver was to prefer in realistic scenarios where a lot of servers were needed. The GLPK solver did however often find the optimal solution early on, but could not discard the possibility of better solutions due to symmetry problems. The GLPK solver may therefore also be used if a certain fault-tolerance is acceptable.
## Contents

Acknowledgement ......................................................................................... i
Abstract ........................................................................................................ ii
1. Introduction ............................................................................................. 1
   1.1 Problem motivation ............................................................................ 1
   1.2 Problem formulation .......................................................................... 4
   1.3 Restrictions ........................................................................................ 4
2. Background .............................................................................................. 4
   2.1 Distribution strategies ....................................................................... 5
   2.2 Linear Programming .......................................................................... 6
   2.3 The branch and bound technique ....................................................... 7
   2.4 The GLPK library .............................................................................. 7
3. Linear programming formulation ............................................................. 8
   3.1 Objective function ............................................................................ 8
   3.2 Memory constraint ........................................................................... 9
   3.3 CPU constraints .............................................................................. 9
   3.4 Communication cost constraints ....................................................... 9
   3.5 Server relaxation constraints ............................................................ 10
   3.6 Server order constraints ² ................................................................ 11
4. Implementation using the GLPK library ............................................... 11
5. Separate solvers for distributable and non-distributable queries ......... 13
   5.1 Customized bin-packing solver for non-distributable queries ......... 13
   5.2. LP Relaxed model for the distributable queries ............................. 20
6. Customized solver for distributable and non-distributable queries ....... 21
7. Evaluation ............................................................................................... 24
   7.1 Test cases ....................................................................................... 25
   7.2 Results ............................................................................................. 26
8. Conclusions ............................................................................................ 37
9. Future work ............................................................................................ 38
References ................................................................................................. 41
1. Introduction

This initial chapter serves as an introduction to the problem of query allocation put into the context of distributed CEP systems, or more precisely the system currently being developed at Fujitsu. It will give a basic understanding of the CEP environment and explain both the gain and the restrictions involved in finding a more optimal query placement.

1.1 Problem motivation

Recently a lot of research has been done in the field of stream processing (1). Complex Event Processing (CEP) is a technique for processing and extracting relevant information from events in real time. Large streams of events are constantly arriving at the processing system from many different sources, which may include mobile phones, weather satellites or traffic tracking systems (2). The desired information is then extracted from these events by issuing pre-defined queries. The result is often a new event that may be subject to further queries.

A complex event is the aggregation of a number of simpler events. By gathering information from a lot of events one might be able to conclude the occurrence of some higher logic event. For instance, one end user might want to receive a notification when traffic congestions are detected on roads that he is likely to use. For this, a lot of information is needed. First we must know that this person is in his car, and which direction he is heading. We may also benefit from knowing which roads he usually uses. Also, to detect that a road that he is likely to use is congested, we cannot only consider one car driving slowly on that road – it might simply be that this car has some problem of its own. We need to have a number of events that indicate the occurrence of the same complex event to be able to draw a realistic conclusion.

In a CEP system, the queries themselves are stored in memory, and the events are processed as they arrive, which makes the resulting event an immediate consequence of the arriving event. If this method is compared to system based on first storing events in a database and then explicitly issue queries to retrieve the relevant information, one realizes that the CEP approach is more suitable when real time processing is needed.

Fujitsu will, in the future, offer a service called the Data Collection and Detection Service (3), where the user merely needs to specify which events he or she wants to be able to handle, along with the query definitions and then let this service construct and maintain the resulting system. The under-the-hood-workings are shown in Figure 1.
Figure 1. Relationship between the CEP components

The CEP compiler will take the used-defined EPL (Event Processing Language) file and construct a query graph, which describes how the queries are communicating with each other; i.e. which queries that produces events that need to be handled by some other query. This information is then passed on to the CEP manager, which places these queries onto a number of servers. It will distribute the queries among the servers based on the distribution strategies given by the compiler. Distribution strategies will be further explained in 2.1 Distribution strategies.

The CEP manager currently uses a very simple algorithm for the initial placement of the queries. However, one would of course benefit from knowing a more optimal placement. The servers used are placed in a cloud environment, and by minimizing the number of servers needed one would directly minimize the cost setting up this system.

There is also a gain in minimizing the communication between the servers used. Currently, the CEP manager does not take this fact into consideration.

Each server used also has a CEP engine running on it, which is responsible for handling the events. If a query on one server produces an event, the CEP engine on that server will know if this event can be handled locally or if it needs to be forwarded to some other server. The CEP engine will also send profiling information back to the manager about the memory and CPU usage on that server. The manager can thus detect if the system is in danger of being overloaded and will do automatic load-balancing if needed. This load-balancing is very fast and takes merely one second, but the system still has ways to improve.
As mentioned, the initial placement of the queries is done by the CEP manager in a very rudimentary fashion. A future aim is to develop a feedback system from the CEP manager, where profiling information is forwarded to the compiler. The initial query placement can thus be moved from the CEP manager to the compiler. This allows for more advanced query placements algorithms as the compiler does not need to provide an immediate placement plan. More information about the event topology (event rates, memory/CPU usage) will also be known and thus better choices can be made.

The objective of this thesis is to first develop a theoretical model for the optimal placement of a set of queries and then implement and evaluate the model. Preferably a mathematical model can provide an optimal placement. The approach taken in this thesis is therefore to formulate the problem as a linear program. Linear programming (LP) and further necessary background will be described in 2.2 Linear Programming. First, the objective of this thesis will be formulated in the next section followed by the restrictions.

A LP model for the problem is proposed in 3. Linear programming formulation. This is first implemented and evaluated with the GLPK (GNU Linear Programming Kit) library as described in 4. Implementation using the GLPK library. Further optimizations turned out to be necessary and a customized solver for two different approaches of the problem of query allocation was implemented. The first approach is described in 5. Separate solvers for distributable and non-distributable queries and the second approach in 6. Customized solver for distributable and non-distributable queries. For both of these approaches the GLPK solver was compared to the customized solver. The results are shown in Evaluation. In 8. Conclusions the results of the thesis is summarized and discussed, which leads to some future work suggestions in 9. Future work.
1.2 Problem formulation

The objective of this thesis is to formulate the initial query placement problem as a linear program and solve the resulting model. Previous research on similar problems (4) indicates that the execution time will be a major problem. It might thus be necessary to make further optimizations by implementing a customized solver for the specific problem of query allocation. The problem can thus be summarized with the following questions;

1) How can the query placement problem be formulated as a linear program, so that memory, CPU usage, distribution strategies and communication costs are all taken into consideration?

2) Will this system be solvable for 20 – 30 queries with a standard MIP (Mixed Integer Programming) solver within a reasonable amount of time?¹?

3) Can the execution time be improved, compared to freely available integer programing solvers, by using the branch and bound technique and adjusting the node selection to the specific problem of query allocation with respect to the factors mentioned in 1)?

1.3 Restrictions

It will throughout this thesis be assumed that profiling information such as event rates and capacity demands are known. The maximum event rate, maximum CPU demand and maximum memory demand will then be considered. This is, of course, a restriction that may give a sparser placement than necessary. This thesis will still concentrate on finding an optimal placement based on the assumption that these maximum rates need to be used in order not to exceed the available capacity. Suggestions as how to extend the model to allow a more dynamical placement will be given in 9. Future work.

All servers will also be assumed to have the same capacity.

2. Background

This chapter aims to explain the more technical terms and techniques used in the rest of this thesis. Important concepts are distribution strategies, linear programming and relaxation.

¹ A reasonable amount of time was defined to a couple of days.
2.1 Distribution strategies

The arriving events should be forwarded to the corresponding query. In most cases a query can be placed on multiple servers. If there is no need to store any information any server with the correct query placed on it can handle any event that matches this query. Such a query is said to be freely distributable. The queries can also be key-distributable, which means that some information needs to be stored in a so called data window, but this data can be separated by a number of different keys. All events with the same key then need to be handled by the same server, but events with different keys do not need to share their data between each other. If one assumes that there are enough unique keys to distribute the queries by then they can be considered as freely distributable when one decides the query placement. There are generally a very large number of keys and thus this will be assumed to hold. Freely distributable queries and key-distributable queries will from here on simply be treated as (freely) distributable.

It might also be the case that all events need to be forwarded to the same server, since some general information from all incoming events is stored in a common data window and thus this information cannot be fragmented onto many different servers. If this is the case, the query is non-distributable.

![Query graph placed on servers](image)

Figure 2. Query graph placed on servers

Each query will thus be either non-distributable or distributable. Figure 3 shows an example query graph with five non-distributable (blue) and three distributable (orange) queries. The queries are placed on three different servers. The distributable queries can be “split” onto a number of servers. Of course, in reality, there will be one copy of the query on each server, but when one considers the incoming events; they will be directed to different servers depending on the “portion” of the query that
is placed there. In the above example, around 85% of $Q_2$ seems to be placed on server A. This means that 85% of the incoming events will be handled by $Q_2$ on server A and 15% by $Q_2$ on server B.

### 2.2 Linear Programming

Linear programming is a constrained optimization technique where the objective is to find the optimal value of a linear function called the objective function, with respect to a number of constraints. This is illustrated in Figure 3. It can be proven that the solution will lie in one of the corner points of the so called feasible region. This is also easily realized if one imagines starting with the objective function, $c$, being equal to minus infinity and then slowly move this line towards the feasible region. The entrance will always be at a corner point, and this will be the minimum value. The entrance might equal one of the lines enclosing the feasible region, but then this will also include the corner points of this line. (5)

In Figure 3 the function to minimize is $c = 2x + 3y$. The red/blue and yellow lines represent constraints that must be met. The solution must lie within this feasible region.

![Figure 3. Feasible region of a constrained optimization problem.](image)

Previous work at Fujitsu has regarded the placement of queries onto servers as a bin-packing problem, assuming a fixed sized of queries and server memory (6). This approach will also be used as a basis in this thesis, but the model will be further extended with CPU constraints and the possibility of placing queries onto multiple servers. A solution outline that minimizes the streams between servers has also been suggested by minimizing the sum of the number of connections between queries that are placed on different servers. The proposed solution in 3. Linear programming formulation takes on a similar approach and integrates this with the above mentioned factors.
This kind of problem will result in a binary integer program, which gives the same precise result as a pure linear program, but with the drawback that it is NP hard. This is of course a huge drawback compared to the easily solved linear equivalent, but there are ways to cut down on the computation cost, for instance LP relaxation and branch and bound techniques, which will be described in the next section. An EPL-file typically consists of 20-30 queries which are also around the upper limit for an integer linear program to be solved within a reasonable amount of time. A similar bin packing problem that placed virtual machines onto servers took 8.5 hours when 20 machines were to be placed, with the execution time quickly increasing with the number of VM’s (4). A more constrained problem may result in less execution time for the LP-solver, but most likely further optimizations will have to be applied in order to obtain an answer within the desired time limit.

### 2.3 The branch and bound technique

The branch and bound technique is a technique generally used by LP solvers when dealing with a binary/integer linear program. The solution space can be modeled as a tree, and the solver will need to examine all branches of this tree. When a feasible solution to the problem has been found it is compared with the currently best solution. If a better solution has been found this value is updated.

Whenever a branch of the tree is chosen, the LP relaxation of the current problem will be calculated. The LP relaxation of a binary linear program is the solution to the same problem with all binary variables relaxed to the interval [0,1]. This means that they are treated as continuous and the program becomes a pure linear program and thus solved quickly. The solution might not be feasible when the variables are seen as binary, but it will provide a lower bound (when minimizing) on the solution.

If this lower bound is worse than the currently best solution, a better solution cannot possibly be found by choosing that branch. The branch is cut, and the solver tries some other possibility.

### 2.4 The GLPK library

The GLPK (GNU Linear Programming Kit) library is a freely available LP solving library. It can solve pure linear programs as well as binary/integer programs and mixed integer programs (MIP’s). The GLPK library will be used both to model the problem and to solve the resulting linear program. A customized solver will be implemented as well, but this solver will still use the GLPK library when solving the LP relaxation of the programs.
3. Linear programming formulation

The main objective of this thesis was to provide a linear programming model for the problem of query allocation. This model is described in the following sections. It will, as with any linear program, consist of the objective function followed by a number of constraints. Together they will produce a system that, when minimized, outputs an optimal query placement given the restrictions in 1.3 Restrictions.

3.1 Objective function

The objective is to minimize the number of servers used, while at the same time taking the communication costs into consideration. Each server used should therefore be represented by adding a cost to the objective function. This cost might equal the actual cost of requesting a server in a cloud environment, but for simplicity it is simply one in the proposed model. The objective function can thus be expressed as:

Minimize \( c^T x \), where

\[
    e^T = [0 \ 0 \ 0 \ ... \ 1 \ 1 \ 1 \ ... \ -c_1 \ -c_2 \ ...]
\]

\[
    x^T = [q_{10} \ q_{11} \ q_{12} \ ... \ s_0 \ s_1 \ s_2 \ ... \ x_{120} \ x_{121} \ ...]
\]

The \( e \)-vector is known beforehand and describes the cost attached to using a server, or having to send events from one server to another. The \( x \)-vector represents the solution to the linear program. \( q_{in} \) specifies what fraction of query \( i \) that are placed on server \( n \). It is continuous in the event of a distributable query and binary for non-distributable queries. \( s_n \) specifies whether server \( n \) is used or not. It is logically binary, but it will be shown in 3.5 Server relaxation constraints that it may be treated as continuous if the rest of the problem (the query-variables) is binary without losing its binary nature. Each time a server is used we will thus add \( 1^*(\text{cost of one server}) \) to the objective function. \( x_{ijk} \) specifies whether query \( i \) and query \( j \) are both placed on server \( k \). The cost introduced when two communicating queries are placed on different servers is more easily modeled as a gain when the queries are placed on the same server. Each time two communicating queries are placed on the same server a small value will thus be subtracted from the objective function. This value depends on the fraction of the queries that are placed on this server and a communication constant, \( c_n \), which describes the gain of placing two communicating queries on the same server.
3.2 Memory constraint

The available memory in one server cannot be exceeded. This means that the summarized memory demand for all queries placed on one server needs to be less than the capacity of that server.

With two servers and two queries, this constraint becomes:

\[
\begin{bmatrix}
  m_1 & m_2 & 0 & 0 & -C_m & 0 & 0 & \ldots \\
  0 & 0 & m_1 & m_2 & 0 & -C_m & 0 & \ldots \\
\end{bmatrix} x \leq \begin{bmatrix} 0 \\
 0 \end{bmatrix}
\]

*Constraint #1. Memory constraint.*

where \( m_i \) is the memory demand of query \( i \) and \( C_m \) the memory capacity of one server.

3.3 CPU constraints

Also, the CPU capacity of one server cannot be exceeded. To make sure that each server can handle the combination of queries placed on it, the following constraint is introduced:

\[
\begin{bmatrix}
  w_1 & w_2 & 0 & 0 & -C_{cpu} & 0 & 0 & \ldots \\
  0 & 0 & w_1 & w_2 & 0 & -C_{cpu} & 0 & \ldots \\
\end{bmatrix} x \leq \begin{bmatrix} 0 \\
 0 \end{bmatrix}
\]

*Constraint #2. CPU constraint.*

where \( w_i \) is the CPU demand of query \( i \) and \( C_w \) the CPU capacity of one server.

In addition, each query needs to be placed in such a way that all incoming events can be handled by some server. This is ensured by making the constraint that the sum the CPU-weight from all queries placed equals the total weight of that query:

\[
\begin{bmatrix}
  w_1 & 0 & w_1 & 0 & 0 & 0 & \ldots \\
  0 & w_2 & 0 & w_2 & 0 & 0 & \ldots \\
\end{bmatrix} x = \begin{bmatrix} w_1 \\
  w_2 \end{bmatrix}
\]

*Constraint #3. Query placement constraint.*

3.4 Communication cost constraints

The communication cost is, as earlier mentioned, rather modeled as a communication gain when two queries that communicate are placed on the same server. The gain can only be the minimum of the two queries placed on that server. If \( Q_i \) and \( Q_j \) are communicating and 45% of \( Q_i \) is placed on one server and 25% of \( Q_j \) is placed on that same server, then 20% of the outgoing events from \( Q_i \) still have to go to another server. The communication-variables will cause subtractions from the objective
function. Since the objective function is minimized these variables will try to take on as large values as possible. If we then set up constraints such that

\[ x_{ijn} = \text{“query i and j placed together on server n”} \leq \text{“query i placed on server n”} = q_{IN} \]

\[ x_{ijn} = \text{“query i and j placed together on server n”} \leq \text{“query j placed on server n”} = q_{IN} \]

this will serve the purpose.

The formal constraint becomes;

\[
\begin{bmatrix}
-1 & 0 & 0 & 0 & 0 & 1 & 0 & \ldots
\end{bmatrix}
\begin{bmatrix}
x
\end{bmatrix}
\leq
\begin{bmatrix}
0
\end{bmatrix}

Constraint #4. Communication constraint.

when \( q_1 \) and \( q_2 \) have a communication cost. The same constraint need to be added for all servers. In the above example the variable “\( q_1 \) and \( q_2 \) are placed on server 0” is set to 1.

3.5 Server relaxation constraints \(^2\)

Since a binary integer program is NP hard (7) one would, as much as possible, like to reduce the number of binary variables. By making further constraints on the servers, their binary nature can be relaxed to the interval \([0,1]\). Since the objective function is to minimize the number of servers, the variables representing the servers will take on a zero value whenever possible. If we create constraints such that, for each server, the corresponding server variable is always greater or equal to each of the query placements on that server, the variable will be zero when no queries are placed and one when at least one query is placed, assuming that the query-placement variables are binary.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & -1 & 0 & \ldots
\end{bmatrix}
\begin{bmatrix}
x
\end{bmatrix}
\leq
\begin{bmatrix}
0
\end{bmatrix}

Constraint #5. Server relaxation constraint.

\(^2\) The constraint will only hold for the case with separate solvers for distributable and non-distributable queries.
3.6 Server order constraints

As will be described in 4. Implementation using the GLPK library, the GLPK MIP solver does not seem to realize that the servers are simply identical placeholders. Since the problem will have n! different identical solutions (with the same solution permuted over the different servers) this will make the calculation considerably slower. A simple test of increasing the number of available server variables while keeping the problem size constant shows this. With four queries and four servers the problem is solved instantly. On the other hand, when the number of available servers is increased to 100, the problem seems to run forever.

To make the solver a bit smarter, constraints on the order in which servers are used are created. That way a lot of server combinations will be considered infeasible solutions and thus their branches will be cut. This way one does not lose as much execution time by specifying more server variables than needed. Still, all permutations within the first N servers used will be considered. The constraint will have the form \( s_0 \leq s_1 \leq s_2 \leq s_3 \leq \ldots \leq s_N \), i.e. the first servers will always be used.

\[
[0 \ 0 \ 0 \ 0 \ 1 \ -1 \ \ldots]x \leq [0]
\]

Constraint #6. Server relaxation constraint.

The example constraint specifies that server 1 can only be used if server 0 is also used.

4. Implementation using the GLPK library

The LP model formulated in 3. Linear programming formulation was implemented in java using the GLPK library. The linear program was represented by a large matrix, in which each row is a constraint and a column represents a variable that need to be set. The columns were given sensible names so that the corresponding column could be retrieved from a name-column-mapping.

Some examples were then solved using the GLPK MIP solver. How these example query graphs were generated is described in 7.1 Test cases. The examples solved contained only non-distributable queries and was thus an extended version of the NP hard bin-packing problem (7). Smaller problems were solved very fast as shown in 7.2.1 Customized solver for non-distributable queries but with around 16 queries one could note a significant increase in the execution time.
This increase was expected, but what was most surprising was the increase when the same problem (query graph) was solved, only modeled with some additional server variables. As described in 3.1 Objective function the server variables are binary and will be set to one when a server is used and zero when a server is not used. Solving the same problem, only with more available servers, should not be noticeably slower. The unused server variables only need to be set to zero.

This indicates that there is a problem in the GLPK solver. It does not seem to realize that these problems are equivalent and the likely reason for this is that it treats a permutation of the same solution onto different servers as different solutions to the problem. This is illustrated in Figure 4. Since all servers are assumed to have the same capacity these two situations are symmetrical, i.e. the objective function will have the same value in both cases. If the GLPK solver treats all symmetrical solutions as different solutions to the problem, it will not be able to cut the branches leading to these solutions until they are fully investigated, which will increase the execution time drastically with the number of server variables used.

![Figure 4. Symmetrical query placement](image)

The GLPK library includes callback routines (8) in which one can affect the solution process. This seemed promising at first, but after some further experiments this did not turn out to be very helpful. The GLPK solver creates a tree of available sub-problems and when a new branch is to be chosen one can direct the solution process by choosing the next sub-problem to be solved. The problem is that there is no real link between the sub-problems and the problem logic. Each new sub-problem will be given a number by the GLPK solver, and this number can be retrieved from the callback routine. One would have liked to be able to retrieve the variables set in that sub-problem, but there seems to be no way to do this.

Still, one would like to remove such symmetry as the one describe above. A customized solver for the specific problem of query allocation was
therefore implemented. It is of course very hard to make a better general LP solver than the GLPK solver, so this was not at all the intention. Since we have some specific knowledge about the problem to be solved, a customized solver might still be able to provide a quicker solution.

5. Separate solvers for distributable and non-distributable queries

Currently, the CEP manager first places the non-distributable queries together onto as few servers as possible by first filling up the servers one by one, and then placing the distributable queries onto nodes based on a round-robin-like algorithm (the number of available servers is thus decided in advance). This suggests that dividing the problem into two sub-problems (distributable and non-distributable queries) and solving each problem separately is an improvement over the current implementation even if the cost might be further reduced when the problem is solved as a whole. Solving two smaller sub-problems will reduce the execution time, but has the drawback that it will probably not give the most optimal placement.

The distributable queries was initially looked upon as binary variables that only could be split into a fixed number of parts, and thus the approach first taken in this thesis was a customized bin-pack-solver for the problem as a whole. When it later, after some discussion, was realized that the distributable queries could be treated as completely relaxed in the interval [0,1] the natural extension to the developed solution was to treat the problem as two sub-problems solved side by side.

When it after yet further discussions was realized that the distributable queries generally have a much larger capacity demand than the non-distributable queries, yet another approach was motivated, as described in 6. Customized solver for distributable and non-distributable queries.

5.1 Customized bin-packing solver for non-distributable queries

As described in 4. Implementation using the GLPK library the GLPK solver does not seem to handle symmetric solutions very well. When the maximum number of servers is increased the calculation time also increases drastically. Since the GLPK solver is used for any type of linear problem it lacks some of the logic that is apparent when one has further insights into the specific problem of query allocation.
To reduce the calculation time most efforts has been put into recognizing different type of symmetrical situations in order to avoid unnecessary double-calculations. These types of symmetrical situations and their proposed solutions are described in the next section.

5.1.1 Solver logic

The customized solver borrows some of the logic from any branch-and-bound solver, but also directs the solution process based on the knowledge of the problem at hand. It is able to find and discard some symmetrical solutions early on that might be hard for a standard MIP solver to recognize merely based on the linear constraints.

As in any branch-and-bound solver, when a query is placed (i. e. a branch is chosen), a LP relaxation is made and the LP relaxed program is solved by the GLPK solver. The result is then compared to the currently best objective value. If the relaxation is worse than this value, a better solution cannot possibly be retrieved when continuing down that branch and it is discarded. The solver backtracks and tries another placement.

The solver specific logic is mainly based on the recognition of three symmetrical situations as described below. Attempts to find a good solution early one has also been made by selecting communicating queries before those not communicating. The customized solver also benefits from always placing a query, i. e. setting a variable to one. There is little gain in choosing a zero branch, which merely means that a query will not be placed on a specific server. Such placements will only move the solution process slightly forward.

The three symmetrical situations mentioned are as follows:

A. *The same query compositions permuted over a set of servers.* A solution with the same query compositions permuted over different servers should not be considered. $Q_1$ and $Q_2$ placed on server $i$ while $Q_3$ is placed on server $j$ is logically equivalent to $Q_3$ placed on server $i$ while $Q_1$ and $Q_2$ are placed on server $j$. This is the symmetrical situation described in 4. *Implementation using the GLPK library.* To avoid these types of situations, queries are assigned to servers in order starting with server zero. Once the first query is placed onto a server it will stay there unless the solution leading up to that point is changed, i. e. as long as the query composition on the previous servers is kept intact. This is illustrated in Figure 5.
If Q₁ is fixed on Server 4 until the previous (Server 1-3) query composition change, we cannot possibly end up with the placement in 4 b). This does not restrict the space of non-symmetrical solutions in any way since Q₁ needs to be placed on some server, and this might just as well be Server 4.

B. *The same queries permuted over a set of the servers.* Assume that we reach a situation where we have placed a set of queries on the first n servers. If a permutation of the same set of queries has been placed on the same servers at a previously stage we only need to consider this route when we have achieved a greater communication gain then the last time we were in this situation. This is illustrated in Figure 6.
In Figure 6 a) Q₁ and Q₂ are placed on Server 0 and Q₃ and Q₄ on Server 1. Q₅ and subsequent queries will be placed on Server 3 and subsequent servers. In 6 b) Q₁ is placed on Server 0 and Q₁, Q₂ and Q₃ on Server 1. These situations are not symmetrical, but from that point on the optimal solution will be exactly that one already calculated in a). It can never lead to a better solution unless we have a better communication gain on the first two servers than the last time we were in this situation.

To avoid unnecessary calculations, the query-server-state, i. e. a unique value for the placement of a certain set of queries on the first N servers, is put in a map together with the current communication gain each time a query is placed onto a new server. The current communication gain is maintained as a global variable. Before a query is placed on a new server this map is inspected and when we encounter a non-improvable symmetric situation we simply backtrack.

C. The same queries permuted within a server. Assume that we have placed Q₀, Q₁, Q₂ and Q₃ on server i, then backtracked and are now in the situation where Q₀, Q₁ and Q₃ are placed on server i. We want to avoid placing Q₂ once again on this server since this will be symmetric to previous situation which already has been examined. This can be achieved by letting the next query inherit the queries it needs to place from its parent (i. e. the query that placed this query). Q₁ has already placed Q₂ once and it has therefore been removed from Q₁’s list of next-placement-candidates. Another way of doing this is to by fix the variable that represents “Q₂ placed on server i” to zero when we backtrack from Q₂. There is a benefit in doing both. The heritage is simple and doesn’t require us to do an extra check if the query can be placed or not. But there is a benefit in fixing those variables that are known to be zero early on since it will give us more realistic values of the LP relaxation. If we know that Q₂ cannot be placed on a server since it is non-distributable and does not fit, the relaxed version can still place half of Q₂ on that server, which might give a nice communication gain and a much better value than the real MIP solution. If the LP relaxation gives more realistic (worse) values, more branches can be cut.

The fact that Q₂ has been fixed to zero is stored in Q₂’s parent, Q₁ since this symmetry only holds if we started off with Q₀ and
Q₁. If Q₁ is unplaced we need to be able to release the fixation and once again let Q₂ be placeable on server i.

The fixation of variables that are known to be zero, as mentioned in situation C, could have been made a point of its own. It will, as described, give more realistic LP relaxations, which in turn helps cutting more branches. Such fixations should be done in all situations where a variable is known to be zero. This includes those situations when a query does not fit on a server.

5. 1. 2 The algorithm

Based on the observations in 5.1.1 Solver logic a query placement algorithm was developed. The algorithm is customized to the specific problem at hand and can thus provide an alternative to the standard GLPK solver, which is a general MIP solver that bears no knowledge of the problem besides the objective function and the linear constraints. The solver based on this algorithm was then evaluated against the GLPK solver. The results are shown in 7. Evaluation.

The algorithm can be divided into five phases:

- **Initialization.** Initialize all structures used.
- **Place a query.** Try to place the next query, primarily on the current server. Whenever a query cannot be placed appropriate variables is set to zero and a new attempt is made by selecting the next query in turn. If a query can be placed state-tracking variables are updated. The LP relaxation of the current program is also calculated and if the relaxation shows that an optimal solution can no longer be retrieved the algorithm moves on the **Backtrack** phase.
- **Compare.** When all queries are placed, the solution is compared to the current minimum and updated accordingly.
- **Backtrack.** Remove the latest placed query and restore all state-information. Inspect the, at this point, most recently placed query and either **Place a query** from its placement-candidates or **Backtrack** once again if all candidates have been evaluated.
- **Terminate.** All possible solutions have been examined or discarded. The solution is printed.

The steps above outline the basic features of the algorithm, which is described in its entirety below. This section can however be skipped without losing any necessary information needed for the following sections.
1. **Initialization.**
   
a. Add all queries to the set of not yet placed queries, \( S := \{0, 1, 2, 3, \ldots \ N \} \).
b. Push zero to the stack representing the current server memory.
c. Push zero to the stack representing the current server CPU.
d. Create a stack where the placed queries will be pushed.
e. Set the current best solution, \( Z \), to zero.
f. Set the current communication gain to zero.
g. Set the current server to zero.
h. Place the first query on server 0. A structure for this query is created. The structure contains the queryID, the server it is placed on and two lists. In the first list the queries that have communication to/from this query are added. In the second list the rest of the queries that are not yet placed are added. The PlacedQuery class also contains further logic to keep track of the communication gain and backtracked queries, i.e., queries that are set to zero on a server. A boolean variable also indicates whether or not the query is the first one placed onto a server.
   
i. Let baseQuery equal the query created in 1. h.

2. **Place the next Query.** Call the method nextThisServer() in the most recently placed PlacedQuery (i.e., the one on top of the stack). The method will return the next query to place or -1 if all its query placements have been examined.
   
a. If a query id is returned:
   
i. If the query has a higher capacity demand than that available on the current server:
      1. Set the variable to “qid placed on server currentServer” to zero.
      2. GOTO 2.
   
   ii. Create the structure for this query as described in the initialization phase (1. h). Push it on the top of the placedQueries-stack.
   
   iii. Adjust the communication gain accordingly. If the parent query is communicating with this placement, add the communication factor to the current communication gain. If the parent query previously placed a communicating query that has been unplaced, subtract that communication factor from the current communication gain.
   
   iv. Calculate the LP Relaxation of the problem. If the relaxation is worse than Z GOTO 4.
   
   v. If there is queries left to place, GOTO 2.

b. Else -1 has been returned and a query should be placed on the next server.
   
i. Calculate the current query-server-state.
      1. If it does not exist in the query-server-state-map - put it in the map together with the current communication gain. Continue.
2. Else if it exists in the map and has a worse communication gain than the current: Replace the state in the map with the current. Continue.
   ii. Increase the currentServer variable.
   iii. Push zero to the currentServerCPU-stack.
   iv. Push zero to the currentServerMemory-stack.
   v. Create the structure for this query as described in the initialization phase (1. h). This query is the first placement on a server.
   vi. Calculate the LP Relaxation. If the relaxation is worse than Z GOTO 4.
   vii. If there are queries left to place, GOTO 2.
3. All queries are currently placed. Compare the new z-value with the current minimum, Z. If we have a better solution, update Z.
4. Backtrack.
   a. Inspect the top of the placedQueries-stack. If we are to backtrack from the baseQuery we have examined, or been able to cut away, all possible placements. GOTO 5.
   b. Backtrack. We are either:
      i. ..removing the first placed query on a server.
         1. Decrement the currentServer variable.
         2. Pop the first value from the currentServerCPU-stack.
         3. Pop the first value from the currentServerMemory-stack.
         4. Restore the [0,1] bounds for those variables that have been placed-and then-unplaced and fixed to 0 from this query.
         5. Decrease the currentCommunicationGain variable if the query placement caused a communication gain.
         6. Unplace query. Pop placedQuery from the placedQueries-stack.
         7. Add query number to the set of not yet placed queries, S.
         8. Calculate the new query-serve-state.
      ii. ..or removing a later placed query, Q_L. We should then:
         1. Replace the top element of currentServerCPU with the value when Q_L’s CPU demand has been subtracted.
         2. Replace the top element of currentServerMemory with the value when Q_L’s memory demand has been subtracted.
         3. Fix the variable "Q_L placed on server currentServer" to zero. The logic to restore this is created when the next query is placed.
         4. Do step 5-7 as described above (4.b. 5-7).
         5. GOTO 2.
5. Termination. All branches have been examined. Print the best solution.
5.2. LP Relaxed model for the distributable queries

The distributable queries can basically use the same model as the one proposed in 3. Linear programming formulation. Communication with respect to the same key must of course be placed on the same server in order to reduce the communication cost. The communication gain can only be the minimum of the fractions of the two communicating queries placed on the same server as described in 3.4 Communication cost constraints. This property will hold with the previously proposed model, since there exist constraints on the communication variables such that:

\[ \forall \text{ servers } i, \text{ if } a \text{ and } b \text{ are communicating:} \]
\[
\begin{align*}
gain \text{ for } a \text{ and } b \text{ on server } i & \leq \text{ fraction of } a \text{ placed on server } i \\
gain \text{ for } a \text{ and } b \text{ on server } i & \leq \text{ fraction of } b \text{ placed on server } i
\end{align*}
\]

The memory and CPU constraints can be treated the same way as in the non-distributable case. Their load is multiplied by the fraction of the query placed on the specified server instead restricting this to the set \{0, 1\}.

The only real change to the model is that the server relaxation constraints (3.5 Server relaxation constraints) must be removed for it to work correctly. These are not needed anyway since the whole problem will be relaxed to [0,1], but when the solution is inspected the server variables need to be rounded up to 1 to retrieve a correct value for the cost. This requires that the solver has filled up the servers as much as possible. The order constraints (3.6 Server order constraints) will somewhat help to accomplish this, but there is nothing that enforces the solver to fill up servers to the maximum.

One solution to this problem is run the solver twice. A test run will first determine the number of servers needed and this value will then be used for the maximum servers allowed in the next run. Since the execution time no longer is a problem when the entire program is relaxed, this is likely to be the most straightforward solution.

\[^3\text{ One hundred queries can easily be placed in less than a minute.} \]
6. Customized solver for distributable and non-distributable queries

One of the main problems with solving the two sub-problems separately is that the communication costs between queries of different types (distributable/non-distributable) are not taken into consideration. The other problem is that the servers used might be further reduced when we allow the distributable queries to fill the unused space that the non-distributable queries leave behind. Another motivation for using a combined approach is that the problem might actually be solved faster.

There is one important observation that can be made. The distributable queries normally account for most of the server capacity used. A realistic scenario might have 75% distributable queries that require 50-300% of one servers’ capacity. The non-distributable queries typically have a capacity demand of merely 10-20% of one servers’ capacity.

In a case where the distributable queries require more than half of the total capacity demand, the servers needed in the unrelaxed version of the program equals the servers needed in the LP relaxed version.

This is easily realized if one visualizes the distributable queries as a liquid and the non-distributable queries as objects that cannot be split. If we have a lot of liquid we will need a lot of containers. But if we have a lot of containers relative to the number of objects then it will be an easy task to place the objects in these containers. In the end we will just fill up the containers with the liquid.

That this holds already when the distributable queries take up more than half of the capacity demand can be proved by the following reasoning:

We only have non-distributable queries that together account for at most half of the total capacity of the servers. We need to be able to place these queries onto the number of servers retrieved by the LP relaxed version of the program. We only need to consider some cases;

1) If a single query, \( q_0 \), have a greater load demand than half of a server capacity, it can be placed on a server and we have thus “gained” capacity in the sense that the rest of the problem will be even sparser (contains an even greater part of distributable queries).
2) If a query, \( q_1 \), has less load demand than half of a server’s capacity, we can then place it onto one server. The remaining capacity of that server is \( C_r = C_{tot} - C_{q1} \).

\(^4\) This must hold for the memory-CPU-combination.
a. If all remaining queries have a greater load demand than \( C_r \) then the next query will be placed onto a new server. But we have than again “gained” capacity seen over these two servers (\( C_{q1} + C_{q2} > C_{tot} \)), and the rest of the problem will be sparser.

If there is no available server, the initial assumption could not have been true.

b. If there are smaller parts then \( C_r \) these can simply be placed on the same server as \( q_1 \) until we have used more than half of its capacity (step 1) or we reach the situation where we only have queries with a greater load demand, i. e. step 2 a).

This means that we no longer need to minimize the server used! The number of servers needed can be given by the LP relaxation of the whole program. The problem is thus reduced to that of finding the optimal placement with respect to the communication costs. When the number of servers used no longer need to be taken into consideration, the queries can then be placed directly according to the query graph. The only reason that we would consider placing two non-communicating queries on the same server would be that it could reduce the number of servers needed.

The problem thus becomes where to make cuts in the query graph with the constraint that the pieces cut cannot exceed the server capacity. This is illustrated in Figure 7.

![Figure 7. Query graph and corresponding cuts.](image)

This is a kind of knapsack problem. We need to fill a number of knapsacks (servers) with the most valuable things (those queries that maximizes the communication gain). This is again an integer programming problem, and thus NP hard. What makes this special case different from the ordinary knapsack problem is that the value of a query depends on which queries it is placed with. We never need to consider every combination of say a hundred queries, only the different
combinations of cuts in the query graph. Also, if the number of servers used is N, we will want to have at most N pieces cut from graph since there is no point in placing two unrelated pieces of the query graph on the same server.

One could then use the algorithm described in 5.1.2 The algorithm, but instead of trying all possible queries after a certain query placement one will only try those queries that are communicating with the last placed query.

If we know that we need N servers and that the non-distributable queries always can be placed on these N servers, we can remove the server variables completely. This is, as earlier mentioned, a safe assumption to make if the distributable queries have a heavier load demand than the non-distributable ones.

It must be noted that is no longer true those situations (symmetric situation B from 5.1.1 Solver logic) where the same queries are placed onto a set of servers, only permuted, are symmetrical, even if we have the same communication gain for the non-distributable queries. This is due to the fact that the distributable queries, which will be added later, also have communication. These query-server-state-cuts must therefore be removed.

The main difference from the previous algorithm is still what queries to consider as the next-placement-candidates. When the first query, Q₁, is placed onto a server, the queries that should be considered to be placed on the same server are those who directly communicate with Q₁, i.e. all queries that send/receive events to/from Q₁. The fact that we only need to explicitly place the non-distributable queries adds an extra factor to the query selection problem. If a non-distributable query, call it Q₂, has an edge in the query graph to a distributable query, say Q₃, we cannot simply discard that query. When the distributable queries are finally placed, it might very well happen that Q₃ is placed with Q₂. If Q₃ in turn is sending events to a non-distributable query, say Q₄, we would have benefited by placing Q₄ on that same server as well.

Therefore we need to follow the query graph in all directions until we find the first non-distributable query down that path as described in the example below.
Assume that \( Q_0 \) from Figure 8 is placed and we need to find out which queries to consider for the next placement. \( Q_1 \) is an obvious candidate. But we also must consider \( Q_3, Q_6, \) and \( Q_9 \), since there is a path through distributable queries that leads to these non-distributable ones. The only query that we can exclude from our next-placement candidates are \( Q_7 \). Thus, in well-connected graphs with a lot of distributable queries we do not save that many candidates.

Also, if we place the query \( Q_0 \) on a server, \( S_0 \), it might have outgoing connections to more than one query and the placement of all those queries onto \( S_0 \) should be considered. It would thus be a mistake to simply pick one query, \( Q_1 \), and in the next step only consider only those queries that directly communicate with \( Q_1 \). Instead, when a query is placed on the same server as its parent query, it should first inherit all communicating queries from its parent and then add its own neighbors, if they are not already present in the list.

### 7. Evaluation

The evaluation compares the customized solver with the GLPK solver for various more-or-less realistic scenarios. Since there wasn’t yet any profiling information available at the time of the evaluation there was some discussion involved before a more realistic scenario could be defined (many, large distributable queries). Some introductory less realistic scenarios will therefore mainly serve as an indication of the problems with the GLPK solver. All in all, further evaluation is necessary to provide upper bounds on how many queries that can be placed given some specific time constraint. Some conclusions can still be drawn as described in 8. Conclusions.
The linear programing model will also be put to the test by illustrating and analyzing a query graph example and the resulting query placement.

7.1 Test cases

A test case consists of a query graph with one input adapter. Since the queries reached from separate input adapters are completely separated there is no point in using more than one input adapter. Each query has a random memory demand, a random CPU demand and a randomly generated distribution strategy (distributable/non-distributable). Each node is linked to the rest of the graph with at least one edge. A query only has output connections to queries with a higher query-id than the query itself. This is done to prevent loops in the graph and does not introduce any restrictions on the possible graphs generated since one could always find a numbering of the nodes in any query graph that meets this requirement.

The maximum memory, maximum CPU demand, number of queries and a seed for the random generator are given as input parameters when a test case is generated.

A test case is then generated by the algorithm described below.

1. CREATE query graph, input adapter, input adapter-stream
2. FOR (i =0; i < nrOfQueries; i++)
   a. CREATE query Qi
   b. ASSIGN memorySize ∈ RAND(0, maxMemory) to Qi
   c. ASSIGN cpuDemand ∈ RAND(0, maxCPU) to Qi
   d. ASSIGN distributionStrategy ∈ {Non-distributable, Distributable} to Qi
3. CONNECT the input adapter and Qi by a sub-stream
4. FOR (i =0; i < nrOfQueries; i++)
   a. CREATE the main output stream from Qi, Si
   b. IF (i != 0) /* Connect Qi to the existing graph, i.e. the already processed queries */
      i. LET m ∈ RAND(0, i-1)
      ii. IF (!∃ stream between Qi and Qm)
          1. CREATE substream between Qm and Qi
          2. ADD substream TO Sm
   c. LET c ∈ RAND(0, communicationFactor)
   d. FOR (j = 0; j < c; j++) /* Create a number of output connections*/
      i. LET m ∈ RAND(i+1, nrOfQueries-1)
      ii. IF (!∃ stream between Qi and Qm)
          1. CREATE substream between Qi and Qm
          2. ADD substream TO Si
      iii. ELSE GOTO 4.d.i.
7.2 Results

In all of the following test cases each server is assumed to have the same fixed memory/CPU capacity. The communication gain when two queries are fully placed onto a server is retrieved from an array with event rates that has been pre-calculated. These are random, but have largest values from queries placed close to the input adapter. The number of servers variables used was adjusted so that the minimum required was always given. The GLPK solver would show too poor results otherwise. The GLPK solver can quickly detect when a problem lacks a feasible solution. This means that one could quickly retrieve the number of servers needed by increasing the number of server variables used step by step.

7.2.1 Customized solver for non-distributable queries

There is only one scenario with relatively few test-runs displayed for the customized bin-packing solver as the second approach (described in section 6) will provide the most optimal solutions - and as the time went by, it become more and more apparent that this was the approach focus on. The solver for the problem as a whole will still have to be further evaluated, as explained in the subsequent sections, and if it cannot provide fast enough placements to query graphs of the desired sizes one might still want to go back to treating the problem as two separate parts.

Scenario 1: Server memory $\in (0, 10]$, CPU $\in (0, 1/2]$.

<table>
<thead>
<tr>
<th># Queries</th>
<th># Servers used</th>
<th>Customized, Execution time (s)</th>
<th>GLPK, Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>90</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>60</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>200</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>60</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>40</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>600</td>
<td>&gt; 6000</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>2000</td>
<td>&gt; 400</td>
</tr>
</tbody>
</table>

Table 1. Non-distributable queries treated separately.
The GLPK solver is initially faster than the customized solver, but when 16 queries need to be placed the execution time of the GLPK solver drastically increase. Five servers are needed, but when 15 queries are placed on five servers the problem is still solved very fast. This seems to be a bit of a strange exception, but still, if one combines these results with the results in the second part of 6. Customized solver for distributable and non-distributable queries it seems likely that the customized solver is faster when more than five servers are needed for the pure non-distributable approach as well.

7.2.2 Customized solver for the combined problem

It has already been indicated that the problem with the GLPK solver is that it cannot handle a lot of servers well. If the non-distributable and distributable queries are modeled as one large linear program, there will be more servers needed relative to the number of non-distributable queries. This indicates that the GLPK solver will be slow.

7.2.2.1 Model validation

The LP model itself is of course subject to errors. To sanity check the proposed model, described in 3. Linear programming formulation, a test case is displayed graphically. This, of course, gives no proof of the correctness, but will still help to verify the soundness of the model.

![Figure 9. Query graph that was given to the solver](image)

The query graph in Figure 9 displays four non-distributable queries (blue) and five distributable (orange). If the corresponding linear program is solved the resulting query placement is as follows:
If one inspects these placements and their resulting communication costs in Figure 10 a couple of observations can be made. First, all communication between the distributable queries have been completely avoided. Secondly, the distributable queries are placed on two of the three servers and these servers have been filled to the maximum with a 100 % CPU usage. The last server only contains distributable queries.

Query 0 has a lot of out-going connections and since it is non-distributable it needs to be fully placed on a server and therefore all its communicating queries also need to be fully placed in order to completely avoid all communication costs. However, this is not possible due to the capacity constraints – reducing all communication costs can only be done if the whole graph can be placed on one server.
The query placement on server A thus makes sense. Query 3 is also non-distributable and the communication costs between these two queries are completely reduced. Equal fractions from the paths $Q_2 - Q_6$ and $Q_1 - Q_7 - Q_8$ is placed onto the server. This saves some communication costs between $Q_0 - Q_1$ and $Q_0 - Q_2$ at the same time as we still may completely reduce the communication costs between the non-distributable queries.

The fact that the servers use up the maximum capacity on the servers that contain non-distributable queries maximizes the communication gain with the non-distributable queries. This does not restrict the communication gain among the distributable queries since they can be placed as fractions on different servers and as long as the same fractions of the distributable queries that communicate are placed on the same server we will minimize the communication cost. There is no need to maximize server utilization if a server only consists of distributable queries.

### 7.2.2.2 Comparison with GLPK

The first two scenarios are not very realistic test cases, but will still show the characteristics of the GLPK solver. Scenario 3 is a more realistic case, but the generated graphs may have too much connectivity to give good results. The customized solver would benefit from more realistic query graphs with less connectivity.
Scenario 1: Server memory $\in (0, 10]$, CPU $\in (0, 1/2]$. Number of non-distributable queries fixed to $(1/3+1)$ of the total number of queries.

<table>
<thead>
<tr>
<th># Queries (# Non-distributable)</th>
<th># Servers used</th>
<th>GLPK, Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 (6)</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>15 (6)</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>17 (6)</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>17 (6)</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>18 (7)</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>18 (7)</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>19 (7)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>19 (7)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>20 (7)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>20 (7)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>21 (8)</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>21 (8)</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>25 (9)</td>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>25 (9)</td>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>27 (10)</td>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>27 (10)$^5$</td>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>29 (11)</td>
<td>8</td>
<td>1300</td>
</tr>
<tr>
<td>30 (11)</td>
<td>9</td>
<td>300</td>
</tr>
<tr>
<td>31 (11)</td>
<td>9</td>
<td>60</td>
</tr>
<tr>
<td>32 (12)</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>33 (12)</td>
<td>8</td>
<td>2000</td>
</tr>
</tbody>
</table>

Table 2. Comparison between the customized solver and the GLPK solver with a fairly large capacity demand.

---

$^5$ From here on only the customized solver is evaluated. The GLPK solver will be too time-consuming.
Figure 11. Comparison between the customized solver and the GLPK solver with a fairly large capacity demand
**Scenario 2:** Server memory $\in (0, 5]$, CPU $\in (0, 1/4]$. Number of non-distributable queries fixed to $(1/3+1)$ of the total number of queries.

<table>
<thead>
<tr>
<th># Queries (#Non-distributable)</th>
<th># Servers used</th>
<th>Customized, Execution time (s)</th>
<th>GLPK, Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 (7)</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20 (7)</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>23 (8)</td>
<td>4</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>23 (8)</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>25 (9)</td>
<td>4</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>25 (9)</td>
<td>5</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>27 (10)</td>
<td>4</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>27 (10)</td>
<td>4</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>29 (10)</td>
<td>5</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>29 (10)</td>
<td>4</td>
<td>70</td>
<td>20</td>
</tr>
<tr>
<td>31 (11)</td>
<td>4</td>
<td>70</td>
<td>10</td>
</tr>
<tr>
<td>31 (11)</td>
<td>5</td>
<td>40</td>
<td>500</td>
</tr>
<tr>
<td>33 (12)</td>
<td>6</td>
<td>100</td>
<td>&gt; 10000</td>
</tr>
<tr>
<td>33 (12)</td>
<td>5</td>
<td>1000</td>
<td>800</td>
</tr>
<tr>
<td>34 (12)</td>
<td>6</td>
<td>8000</td>
<td>&gt; 12000</td>
</tr>
<tr>
<td>34 (12)</td>
<td>6</td>
<td>2000</td>
<td>&gt; 6000</td>
</tr>
<tr>
<td>35 (12)</td>
<td>6</td>
<td>200</td>
<td>2000</td>
</tr>
<tr>
<td>35 (12)</td>
<td>6</td>
<td>5000</td>
<td>&gt; 9000</td>
</tr>
</tbody>
</table>

*Table 3. Comparison between the customized solver and the GLPK solver with a smaller capacity demand.*
The test cases above indicate that, as the number of servers increase, the customized solver shows better results compared to the GLPK solver. With smaller queries the GLPK solver is initially better than the customized solver. This is due to the fact that few servers are needed. Figure 11 and Figure 12 also suggests a correlation between the number of servers needed and the execution time of the GLPK solver. Whenever an extra server is needed the GLPK solver has a peek and when a test case that needs one less server than the previous one is generated the GLPK solver has a valley. This correlation is not further investigated since this was to be expected and very apparent from just increasing the number of available server variables.

The values used for the capacity demand does not really provide realistic query graphs but still helps to show the characteristics of the GLPK solver. What was also noticeable was that the customized solver and the GLPK solver in all cases concluded the same value for the objective function. This indicates that the customized solver does indeed provide valid results. These values are however not displayed in the tables since they give no extra information other than this equality.

As mentioned, the GLPK solver is slow when the problem requires a large number of servers. If one inspects the bounds in the cases when the solver was interrupted before the final result was retrieved one saw that
the upper bound was equal to the solution from the customized solver in all cases. This means that the optimal solution has been found but the solver has not been able to cut away all branches.

In fact, the GLPK solver often finds the correct solution very fast; it merely takes a long time to decide that it is in fact optimal. The advantage with the customized solver is that we can say for certain that the best solution has been found. An interrupted GLPK solve will always have some error margin. If one can tolerate a certain error the GLPK solver might still be preferable in cases such as the ones described above.

One can also note large variations in the results, with query graphs consisting of the same number of queries. This is due to the randomness of the generated query graphs. All connections are completely random and well-connected graphs where one can find a path through the distributable queries from most non-distributable queries to most others will show noticeably worse execution times. As the number of variables grows the LP relaxation of the problem will also become considerably slower, even if still very fast. Since the LP relaxation of the program is solved at each placement a solving time of one second will still drastically slow down the execution time.
Scenario 3: Server memory $\in [10, 60]$, CPU $\in [0.5, 3]$ for distributable queries (i.e. 50-300% of one server's capacity) and server memory $\in [2, 4]$, CPU $\in [0.1, 0.2]$ (i.e. 10-20% of one server's capacity). 75% of the queries are distributable.

<table>
<thead>
<tr>
<th># Queries (# Non-distributable)</th>
<th># Servers used</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 (5)</td>
<td>27</td>
<td>10</td>
</tr>
<tr>
<td>20 (5)</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>20 (5)</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>20 (5)</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>20 (5)</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>20 (5)</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>30 (7)</td>
<td>43</td>
<td>50</td>
</tr>
<tr>
<td>30 (7)</td>
<td>41</td>
<td>600</td>
</tr>
<tr>
<td>30 (7)</td>
<td>44</td>
<td>100</td>
</tr>
<tr>
<td>30 (7)</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>30 (7)</td>
<td>45</td>
<td>500</td>
</tr>
<tr>
<td>30 (7)</td>
<td>47</td>
<td>40</td>
</tr>
<tr>
<td>35 (8)</td>
<td>52</td>
<td>50</td>
</tr>
<tr>
<td>35 (8)</td>
<td>48</td>
<td>20</td>
</tr>
<tr>
<td>35 (10)</td>
<td>47</td>
<td>900</td>
</tr>
<tr>
<td>35 (8)</td>
<td>52</td>
<td>100</td>
</tr>
<tr>
<td>35 (8)</td>
<td>52</td>
<td>100</td>
</tr>
<tr>
<td>35 (8)</td>
<td>51</td>
<td>300</td>
</tr>
<tr>
<td>40 (10)</td>
<td>53</td>
<td>1000</td>
</tr>
<tr>
<td>40 (10)</td>
<td>53</td>
<td>200</td>
</tr>
<tr>
<td>40 (10)</td>
<td>63</td>
<td>1000</td>
</tr>
<tr>
<td>40 (10)</td>
<td>54</td>
<td>1000</td>
</tr>
<tr>
<td>40 (10)</td>
<td>55</td>
<td>60</td>
</tr>
<tr>
<td>40 (10)</td>
<td>60</td>
<td>900</td>
</tr>
</tbody>
</table>

Table 4. Execution time for query graphs with realistic capacity demands.
Table 4 will show the results from a more realistic scenario. 75% of the total number of queries are distributable. These have a large capacity demand (50-300% of one server’s capacity). This will quickly require to a lot of servers and therefore the GLPK solver will be very slow. In fact, it cannot be used in cases like this. A test-run with 20 queries that took a couple of seconds for the customized solver (only 5 queries need to be explicitly placed) did not finish in 1000 seconds with the GLPK solver. Therefore Table 4 only displays the results from the customized solver, since it does not make sense to make a comparison with the GLPK solver. Instead one would like to find out how many queries the customized solver can place within a reasonable amount of time. Since “a reasonable amount of time” was decided to two days, these experiments were too time-consuming to run, but Table 4 still gives an indication about the execution time for the shown 20-40 queries. However, some test cases with 50 queries were very slow and when these were more closely inspected it was realized that the first query placed communicated with all other non-distributable queries but one. If the graph is well connected a lot of different possible placements need to be considered. When this is combined with the fact that there are a lot of variables and thus the LP relaxation, which is calculated at each placement, also is noticeably slower, the execution time will drastically increase. Still, more realistic query graphs will probably not suffer as much from this problem, since they will have less random connections, and queries is likely to be grouped into different branches with few across-branch-connections.
Figure 13 does thus not in itself give a good picture of the execution time. The test runs merely indicates that 40 queries seem to be placed rather quickly, but this trend does not continue with 45-50 queries. These points are not displayed since they were very time-consuming and did not finish at all. The only conclusion one may draw is that at least 20-30 queries are easily placeable, but after that there is too much uncertainty.

8. Conclusions

In this thesis the problem of query allocation has been modeled as a linear program, minimizing both the number of servers used and the communication costs between the servers.

The freely available GLPK linear programming solver was used to model and solve the problem. With as few as 16 (non-distributable) queries the execution time was drastically increased and previous research on similar bin-packing problems (4) indicated that a problem with the proposed 20-30 queries would not be solvable within specified the time frame of two days.

To speed up the execution time a customized solver was implemented. This solver was then tested against the GLPK solver. Both solvers did, in all test cases, give the same minimum value of the objective function, which indicates that the customized solver indeed does find the optimal value. The execution time between the solvers varied a bit depending on which type of queries that was considered, with the GLPK solver generally being better at placing many small queries onto few servers. This however, is not a very realistic scenario – large distributable queries are generally common. As the number of servers needed increased the customized solver showed better performance compared to the GLPK solver.

It was also early on realized that there was a strong correlation between the number of servers needed and the execution time of the GLPK solver. Since a realistic query graph consists mainly of distributable queries with a very high capacity demand, the number of servers needed relative to the number of non-distributable queries is often very high. In such a case the GLPK solver cannot be used at all. There might of course be other LP solvers that can handle the symmetry problem described in 4. Implementation using the GLPK library in a better way, but it will probably still be a good idea to use a customized solver due to the of the observation in 6. Customized solver for distributable and non-distributable queries that lets us cut away all possibilities where we place non-communicating queries on the same server.
The customized solver generally showed better execution times than the GLPK solver when the problem size grew and especially when realistic cases with very large distributable queries (when the GLPK solver really cannot be used) were considered. The customized solver was still slow with around 50 queries, whereof 25% non-distributable. However, this is probably due to too much connectivity in the generated query graphs. It will most likely handle more realistic query graphs better. It does however handle the (in section 1.2) requested 20-30 queries well. Rather few test runs were made but with the slowest run in the interval 20-30 queries taking only 600 seconds there is not really a question whether or not two days will be enough for such cases. Future experiments should use more realistic query graphs to try and get a upper bound on the number of queries that can be placed within the time frame. In a realistic query graph the queries will generally branch into different paths and there will thus not be so much connectivity between the different paths. This will lead to smaller search trees and a reduction of the execution time for the customized solver.

9. Future work

- The customized solver should be evaluated with more realistic query graphs. As earlier mentioned, the query graphs generated often contained a lot of connectivity and did not provide very realistic test cases. Since the query graphs mainly consist of distributable queries one can often find a path from most non-distributable queries to most others. This means that we still have to consider a lot of different placements and we do not save that much when “only” considering the neighbors of one query, since most queries are neighbor-queries. In such a case the solver will be slow since the LP relaxation of a problem with a lot of variables also becomes slower. Such an LP relaxation might take a second in some cases and this will have a noticeable impact on the execution time of the customized solver as well, since the LP relaxation of the problem is solved after each placement. Thus, the next step is to test the solver with more realistic query graphs. One will then be able to provide an upper bound for the number of queries that can be placed within the specified time frame. Currently no such statements can be made and the test runs merely give some indications.

- An initial placement suggestion will be passed to the CEP manager along with the query graph, but whether or not this placement is suitable for future load balancing remains unknown. The main problem with the initial placement is that it only relies
on profiling information from past events. This, of course, gives us an idea about the event topology, but no efforts have been made to foresee the future. One would, as much as possible, like to be able to predict future abnormalities and not only give an initial placement proposal, but also suggestions as to how the system can be load-balanced depending on how the event topology change.

- The profiling information should be used in the best way to minimize initial placement cost. The profiling information may be very detailed and one might be able to find that the events follow certain distributions over a period of time. This has however not been taken into consideration in the proposed model in 3. Linear programming formulation. To ensure that all events can be handled, the maximum event rate has been used. The statement that the initial placement is optimal relies on the profiling information to give the exact event rates, the exact memory demand and the exact CPU demand. This will however not be the case; there will always be some fluctuations in the rate at which the events occur.

However it is possible to extend the model and also consider the distributions that the events follow. Currently only one memory constraint and one CPU constraint is given. This means that, at any point in time, the capacity demand of the queries placed on one server cannot exceed the capacity of that server.

Let's assume that the events all follow a periodic (let’s say weekly) distribution pattern, where some events have peaks on Mondays and some others on Wednesdays. One might then, instead of only one constraint, set up one constraint for each day of the week. The capacity demand that all queries placed on a server have on Mondays must then not exceed the capacity of that server. This must then hold for all days of the week. This extension does not introduce any new variables and should therefore not increase the execution time of the solver. In fact, it constrains the problem even more which actually might decrease the execution time. If some events have their peaks on Mondays and some others have their lowest event rates that same day, the concerned queries might successfully be placed on the same server. It might however be hard to find such situations in one query graph. Often the events are closely related and an increase of one event would cause an increase of many other events.

- The profiling information should be used in the best way to load-balance the system. One could instead also consider calculating
one placement for each day and thus suggest how the system should be load balanced. Of course, one does not want to make a completely new placement when doing such a load balancing, but rather only have to scale out the concerned queries when necessary.

If one sees that some events have large fluctuations these can be put in focus. One could then create seven copies of all servers, one for each day of the week, and make constraints such that all queries but the ones in focus must be placed on the same server every day on the week. These will have the form:

\[ \forall i \forall j \text{ “Fraction of } Q_i \text{ placed on server } j \text{ one day” } = \text{“Fraction of } Q_i \text{ placed on server } j \text{ on the next day”} \]

This will introduce a lot of new variables, which is never a good thing, but most of them will have equality constraints, so there is really only one placement to make for these queries, especially the non-distributable ones, since we can assume that they seldom have large fluctuations in their capacity demand. When using a customized solver the number of queries to place does not increase, but the LP relaxation might be slow, which will slow down the whole solving process.

The solution to this linear program would then suggest a steady placement for the majority of the queries, but also one placement per day for the queries in focus, which had large fluctuations. They will probably, to the larger part, be placed on the same servers each day due to the communication constraints, but if one wants to ensure this, one can also put further constraints on the system.

Since one has knowledge about the event rate distribution of these queries one might introduce special constraints for each on the concerned queries. Of \( Q_1 \) has the lowest event rate on Mondays, and the second lowest on Tuesdays, one can introduce a constraint that requires that

\[ \forall i \text{ “Fraction of } Q_1 \text{ placed on } S_i \text{ on Monday” } \leq \text{“Fraction of } Q_1 \text{ placed on } S_i \text{ on Tuesday”} \]

Similar constraints can be made for all days of the week. This will then give the optimal placement given that one wants to scale-out certain queries without having to move around any of the queries already placed.
References


4. Ferguson, Thomas S. Linear Programming – A Concise Introduction. p. 3.


