# Data Placement for Partial Replication

## Data-Driven Decision-Making in Enterprise Applications

Anton von Weltzien, Marvin Thiele, Daniel Lindner

September 2019

**Abstract**

The data partitioning problem in modern database systems was and still is a very relevant problem. In the age of Big Data, the need to properly replicate and access data in a distributed manner is bigger than ever. In this paper, we propose a system to solve the data partitioning problem using linear programming. Initially, we present a way to formalize the problem and showcase the challenges posed by this approach. Following these insights, we evaluate multiple approaches to make the problem computationally easier in order to be able to solve the problem at scale. For this, we propose tree-based decomposition heuristics, as well as using the solver's ability to provide a non-optimal solution. Moreover, we included a detailed discussion of the quality characteristics of such solutions and their trade-offs.

## 1. The Data Partitioning Problem

During the last decades, the utilization of database systems has shifted to new, heavy-workload scenarios. Enterprise applications use new data-driven models for business processes. Thus, they access large amounts of data for real-time purposes [2, p. 7]. As a consequence, new systems for data management, like HDFS[1], Cassandra[2] or in-memory databases, have evolved.

Distributed database systems have become a standard approach for use-cases where high scalability is required, e. g., in enterprise applications. In such a scenario, multiple nodes controlling a database instance serve the queries stated by the user. For optimal load balancing, the data can be *fully replicated*. This solution comes with multiple disadvantages. First of all, queries which modify the data need to be passed to every single node in the same order to avoid inconsistent states. Second, the total amount of storage space grows linearly with the number of nodes [3]. Additionally, the data might be to large to fit on one node entirely.

To overcome those issues, it is possible to only replicate parts of the total data. To achieve this, we partition the data into multiple distinct fragments [2, p. 65]. These fragments are then split between the nodes and replicated if necessary. For our model, we assume that the independent nodes should not need to communicate for one single query execution. Hence, we need to ensure that all of the common queries are executable on at least one single node.

We now have all the elements to formalize the problem. Our data is split into a set of fragments $F$, whereas each fragment $f \in F$ has a size $|f|$. Those fragments are distributed and replicated on a set of nodes $N$. The nodes process queries $Q$. We want to reduce the

[1] `https://hadoop.apache.org/`
[2] `https://cassandra.apache.org/`

overall storage consumption while all queries can still be executed, as well as creating a fair distribution of the workload, i.e., query processing, between the nodes. This problem is NP-complete [2, p. 65]. In this paper, we want to present different approaches to finding (close to) optimal solutions which we developed during our project[3].

## 2. Linear Programming Approach

One approach to solving optimization problems is to define them as a *linear program* and obtain a solution from a *solver*. That solver tries to find numerical solutions for declared *decision variables* to fulfill certain *constraints* and optimize a given linear *objective function* [4, p. $5f$].

In our initial solution, we wanted to compute an optimal solution for our problem. Every node should have the same load regarding one scenario of queries, where each query has a specific workload. This chapter presents a formalized linear program that takes the different goals of the problem into account.

### 2.1. Formulation as a Linear Programming Problem

As discussed in section 1, we want to distribute data fragments $f \in F$ with a specific size $|f|$ on nodes $n \in N$. Those nodes execute queries $q \in Q$, which have some cost measure and occur with a frequency. By combining these values, we obtain the workload caused by the query. This is shown in Equation 1. On the other hand, the overall workload the system needs to handle is the sum of the workloads for all queries, as stated in Equation 2.

$$\forall q \in Q : workload_q := cost_q \times frequency_q \tag{1}$$

$$total\_workload := \sum_{q \in Q} workload_q \tag{2}$$

We now introduce the decision variables. Most of those variables are two-dimensional and binary. One of them is the *query* variable, which for each query $q$ states whether it needs a specific fragment $f$. The *location* variable indicates whether node $n$ stores fragment $f$. *Runnable* shows if a node $n$ can execute query $q$. We assign a *workshare* of a query to a node. This means that this node executes a specific ratio of all runs of this query. The *workshare* variable is two-dimensional and continuous. It holds the mentioned ratio of query runs for all nodes.

When we sum up the sizes of the stored data fragments for every node, we get the overall size of the replicated data. Our objective is to minimize this value. Equation 3 shows the formal modeling of this relationship.

$$\min\Big(\sum_{f \in F,\ n \in N} location_{f,n} \times |f|\Big) \tag{3}$$

Our solution needs to satisfy several constraints. First of all, we want our system to serve all queries. The sum of the *runnable* values over all nodes needs to be at least 1 for every query, as shown in Equation 4. Another constraint sets the values of the *runnable* variable. We only want a query to be runnable on a node only if it stores all the fragments this query requires. To achieve this, we sum up all fragments a query needs which the node stores. This value should be greater or equal than the sum of all needed fragments times the *runnable* value for the query on the node. That formulation, stated in Equation 5, forces this value to be 0 if the equation is false, or 1 otherwise.

[3] `https://github.com/yT0n1/D3MEAP-Projekt`

Second, every run of a query requested by the user must be executed. Thus, Equation 6 states that the sum of all *workshare* fractions for every query must be 1.

Third, a node may only execute a *workshare* of query runs if the query is *runnable* on this node. Equation 7 forces the *workshare* of a node for a query to be 0 when the node cannot execute the query.

Fourth, we want to distribute the workload evenly across all nodes. Therefore, we sum up the *workshare* values for all queries on a node multiplied with the query workload. This workload per node should equal the total workload divided by number of nodes, as shown in Equation 8.

$$\forall q \in Q : \sum_{n \in N} runnable_{q,n} \geq 1 \tag{4}$$

$$\forall q \in Q,\ n \in N : runnable_{q,n} \times \sum_{f \in F} queries_{f,q} \leq \sum_{f \in F} location_{f,n} \times queries_{f,q} \tag{5}$$

$$\forall q \in Q : \sum_{n \in N} workshare_{q,n} = 1 \tag{6}$$

$$\forall q \in Q,\ n \in N : workshare_{q,n} \leq runnable_{q,n} \tag{7}$$

$$\forall n \in N : \sum_{q \in Q} workshare_{q,n} \times workload_q = \frac{total\_workload}{|N|} \tag{8}$$

With these constraints, a solver can obtain an optimal solution for a given workload scenario and a number of nodes.

## 2.2. Quality Criteria of a Solution

Solutions for the data partitioning problem come with several quality characteristics which make them comparable and desirable under different circumstances.

An obvious quality criterion is the optimization objective, namely the required replication space. The goal is to move as far away as possible from the trivial solution of full replication.

Another quality aspect is run-time. A solution has to be found in an acceptable timeframe in order to be able to potentially reconfigure the database to changing conditions.

The last quality criteria is a fair division of labor, the workshare. In a perfect solution, every node has the same expected workload. This is the case if the workshare assigned to a node is the same for each node in the cluster.

A strongly related characteristic is the workshare deviation. This number expresses the total amount of percentage points in which the system deviates from a fair distribution of workshare per node.

To make this clear, we look at an example with two nodes and a solution with a workshare distribution of 25% for node one and 75% for node two. A fair workshare distribution, in this case, would be 50% for each node. Thus, this scenario would result in a workshare deviation of 50 percentage points because each node deviates from the fair distribution by 25 percentage points.

All quality characteristics are closely related and can affect each other, which we will discuss in the following sections.

## 2.3. Performance Analysis of Linear Programming Approach

Since the data partitioning problem is NP-complete, we have to expect exponential run-times with growing problem sizes. Therefore, we measured the run-time of obtaining an optimal solution with the *Gurobi* solver[4] to evaluate the performance of the linear programming approach. We evaluated the three main factors, namely number of nodes, queries and fragments. These three factors directly influence the number of variables in the LP-Problem.

To a lesser degree, the run-time is also influenced by the number of fragments per query, the fragment sizes, and the frequency and cost of queries as these parameters have an influence on the workshare variable. However, their impact is not as direct and therefore harder to measure and interpret. Thus, we only evaluate the three main factors.

In Figure 1, we can see that the run-time quickly increases when we simultaneously increase the number of queries, fragments and nodes.

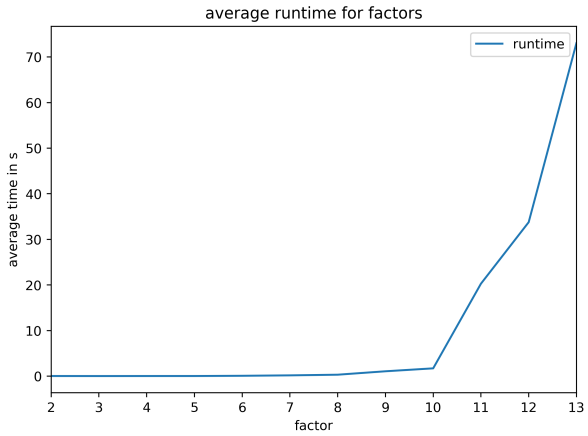



Figure 1: Run-time for Increasing Factor Sizes

To gain a better understanding of this effect, we try to measure the impact of each factor. For this, we selected a range from 2 to 11 and ran each combination of the number of nodes, fragments and queries of these values. Thus, running our implementation 1000 times, repeating it five times and taking the average over the results.

We then aggregated the results, so that the impact of each factor can be seen more clearly. Figure 2 plots the size of a factor against the run-time. The run-time is the aggregation of all possible configurations with one factor. That means that, for example, the run-time value for 10 fragments is determined by taking the mean over all tests run with the number of nodes and queries between 2 and 10 while fixing the number of fragments to 10.

[4] `https://www.gurobi.com`

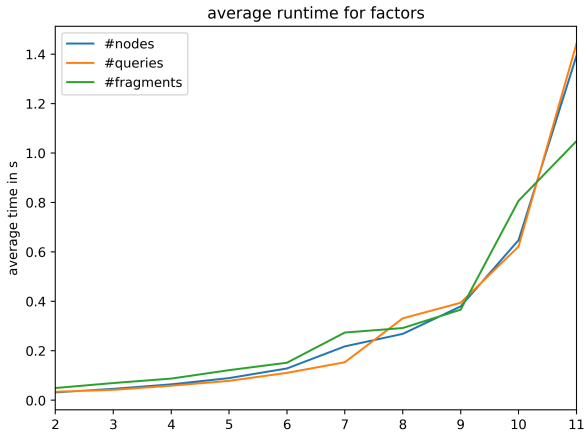



Figure 2: Aggregated Run-time for Different Node, Fragment, and Query Sizes

As we can see in Figure 2, all three factors have an impact on the performance, and no factor is dominating. However, it has to be taken into account that in a real-life scenario the number of fragments and queries will possibly be much larger compared to the number of nodes. Especially, if all factors have high values (i.e., 15) the run-time to find an optimal solution will exceed the ten-minute mark on a two-core laptop. Considering that the number of fragments or queries could potentially be in the hundreds or thousands, it is of interest to reduce the run-time for such cases.

To limit the run-time, the *Gurobi-Solver* offers the possibility to specify a timeout. This hyper-parameter will end the optimization process early with a valid, but non-optimal result. Since the most valuable optimization operations are often performed first, we can save time by cutting the fine-tuning.

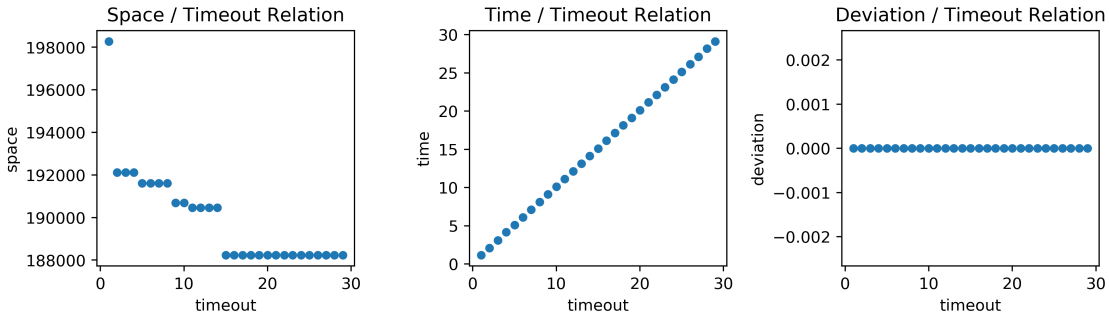



Figure 3: Relation between Timeout and Quality Aspects

In Figure 3, we can see that the space required by the solution is reduced mostly within the first 15 seconds, while there is no deviation from a fair workshare distribution at any point in time. After this point, the solution is barely improving, even if much more time is taken in the calculation. However, the point of diminishing returns is problem-dependent and therefore needs to be set by an expert.

## 3. Robust Optimization

The current problem statement expects only one workload scenario. However, real database systems often have different workload scenarios depending on circumstances like time of

day, day of the week or special events. In these scenarios, queries can occur with different frequencies. If our approach is meant to be successful in the real world, it should be feasible for different workload scenarios.

By allowing multiple workload scenarios, we are faced with another problem. If we continue to enforce a completely fair workshare distribution, it is possible that the only valid solution is full replication. To avoid this, we introduce a so-called "Quetschvariable," which essentially is a border creating an optimization goal for pushing the most-unfair workshare values in the direction of a fair workshare distribution. The factor which controls how much this goal is weighted in the objective is called the penalty factor $\varepsilon$.

This strategy optimizes the worst case; prioritizing the minimum quality of all workloads instead of optimizing for a single workload scenario or the average of workload scenarios.

For our tests, all parameters of a workload, such as the number of fragments and queries are generated randomly in user-specified ranges. This allows us to test the proposed system under real-world, randomized circumstances and avoid overfitting. For each experiment five of these random problems with each problem containing five scenarios were created and the average performance was taken as the result. The experiments were run on a Windows laptop featuring a four-core, 3.50GHz Intel i5-7300HQ processor and 16 GB of RAM.

## 3.1. Formulation as a Linear Programming Problem

To extend our problem and integrate robust optimization, we need to introduce more parameters. We now have a set of different workload scenarios $W$. In these different workload scenarios, the frequencies of the queries may differ. Therefore, we store the work caused by one single query in each scenario in a parameter *query_workload*.

By summing up this *query_workload* times the respective workshares one node executes for each query, we obtain the workload for one node in a workload scenario, as seen in Equation 9. All the *query_workload* values summed up for one workload scenario $w$ result in the *total_workload* for this scenario.

$$\forall n \in N,\ w \in W : node\_workload_{n,w} := \sum_{q \in Q} workshare_{q,n} \times query\_workload_{w,q} \tag{9}$$

Previously, all the node workloads used to be the same, however, as we want to be more lenient with the *fair* distribution of workload we introduce the "Quetschvariable" $\alpha$. For this reason, we modify the constraint about this distribution. Equation 10 shows that the ratio of the work one node executes of the total workload should be less than or equals to $\alpha$ for every node and workload scenario. As a result, $\alpha$ corresponds to the highest amount of work one node has to handle. Since we want to split up the problem solution to multiple subproblems later on (see subsection 4.2), we normalize that mentioned ratio by the part of the workshare a subproblem should solve (i.e., *workshare_split*).

$$\forall\ n \in N,\ w \in W : \frac{node\_workload_{n,w}}{total\_workload_{w}} \times (1 - workshare\_split_{n}) \leq \alpha \tag{10}$$

Second, we update our objective according to Equation 11. It now consists of two addends. The first is the sum of the sizes of stored fragment on all nodes, normalized by the overall size of all fragments. The second one is the penalty value $\varepsilon$ times the "Quetschvariable" $\alpha$. As we minimize the objective, the solver tries to reduce both the overall storage and the highest amount of work a node does. Thus, depending on the penalty factor $\varepsilon$ the solver is encouraged to avoid unfair work distribution.

$$\min\left(\left(\sum_{f\in F,\ n\in N}\frac{location_{f,n}\times|f|}{total\_fragment\_size}\right)+\varepsilon\times\alpha\right) \tag{11}$$

With these adaptions, we can now handle multiple workload scenarios. Since we introduced a new hyper-parameter $\varepsilon$, we want to analyze how this value influences our solution.

## 3.2. Effect of Epsilon on Quality

Going from a strictly enforced fair workshare distribution to a relaxed version of this constraint using the epsilon penalty introduces several new dynamics.

In addition to the trivial solution of full replication, we also see that having the data only on one node is now also a possible trivial solution. This can occur of the epsilon value is too low, which decreases the importance of a fair workshare distribution. This effect highlights the importance of tuning the penalty value in order to gain the right balance between having a strictly fair workshare distribution and a totally unfair, but very space-efficient workshare distribution.

A strict workshare contribution corresponds to a high epsilon value. The values we propose for this hyper-parameter range between 10 and 1,000,000, depending on the desired results.

To be able to better understand the correlation of epsilon with the different result characteristics like run-time, space and fairness of the distribution, we created a Pareto front with each of these criteria in relation to different values of epsilon.

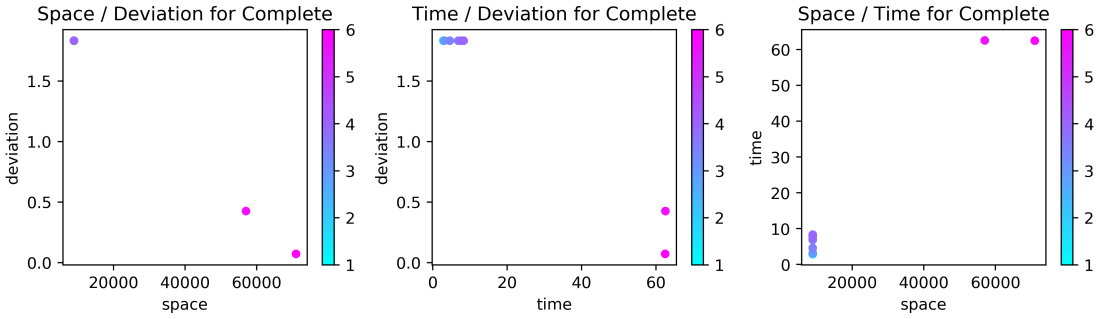



Figure 4: Pareto Front for The Complete Split Strategy regarding the quality dimensions

In this figure, we can see how different epsilon values affect the different quality criteria. For this test, we have chosen epsilon values ranging from $10^1$ to $10^6$. The scale on the color-bar is normalized by $log_{10}$. In the first chart, we can see a linear correlation between space and deviation. The more we relax our epsilon penalty factor, the more space we can save while having more workshare deviation. From the second chart, we conclude that solutions with lower epsilon values require less time to compute. This makes sense since the solution space is effectively less constrained.

## 3.3. Effect of Early Exit on Quality

We already discussed in subsection 2.3 that the Gurobi-Solver offers the possibility to specify a timeout.

When we apply this timeout with epsilon penalty tuning turned on, we see a different picture than the one we saw in Figure 3. We can observe that the deviation from a fair workshare is no longer always zero.

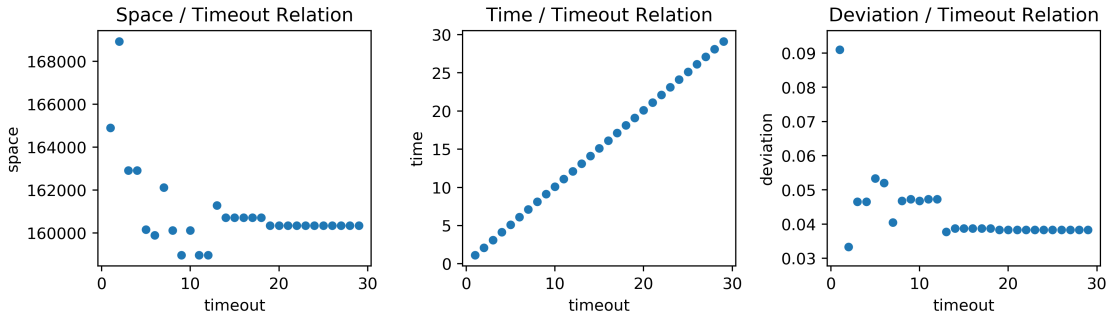



Figure 5: Relation between Timeout and Quality Parameters with the Epsilon Penalty

Instead, it deviates between a range of values in an effort to find the optimal balance between total space and workshare deviation. This has the effect that an early solution might require less replication space because the workshare is not fairly distributed, and thus, some fragments do not need to be replicated. After giving the solver enough time, we see that the quality of the solution does not further improve.

Thus, when using early exit in combination with robust optimization, one needs to be aware that early solutions might have an unfair workload distribution and few nodes doing most of the work.

## 4. Tree-based Decomposition Heuristic

As we have seen in subsection 2.3, the performance of the LP-Approach deteriorates for larger problem sizes. We have already seen that an early-exit of the solver can help reduce the run time at the cost of an optimal solution. In this chapter, we want to present additional strategies to decreasing run-time while still getting close to optimal solutions and analyze their performance.

Since the number of nodes, queries and fragments have a significant influence on the performance, we divide our problem into separate smaller sub-problems, which are easier to solve.

This approach generates a *split-tree* similar to the one shown in Figure 6. Problems are divided between the sub-problem nodes. It is also still possible for a sub-problem node to compute only a fraction of the requests for a certain query, as was the case in the LP-based approach.

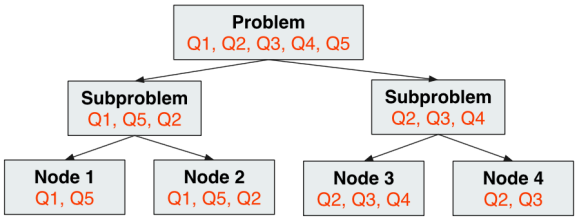



Figure 6: Example of a Split-Tree

This leads to fewer nodes having to be dealt with in a single split and can also reduce the number of queries and fragments if a query was not assigned to a sub-problem because it was fully assigned to a different sub-problem.

### 4.1. Tree Split Strategies

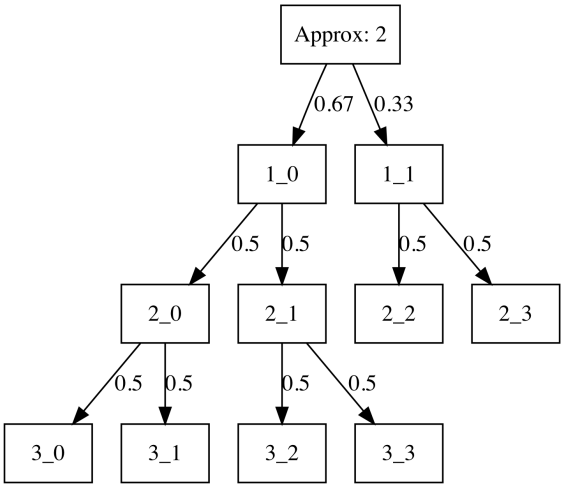
Selecting the best split strategy is non-trivial. A balanced binary tree might be the first strategy coming to mind. However, it is not the only option. While each sub-problem reduces the problem size, it also presents a loss of information, as the sub-tree loses the ability to take the queries assigned to the other side into account. Thus, it could be possible that larger splits, while taking more time, maintain a higher quality.

We developed three types of split strategies. The first is to approximate an *n-ary tree* for a given number of nodes as can be seen in Figure 7a and Figure 7b.

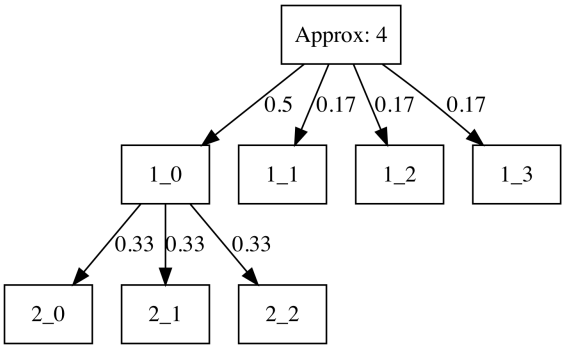
The second is to create a *one vs. all* split where at each level a new node is decided as seen in Figure 7d. The left side always represents the server-nodes in that case.

The third approach is the prime factor-based, where a given number of nodes is decomposed into its prime factors, which then constitute the split-tree like in Figure 7c. Should the number of nodes be a prime number, the strategy will mimic the complete split.

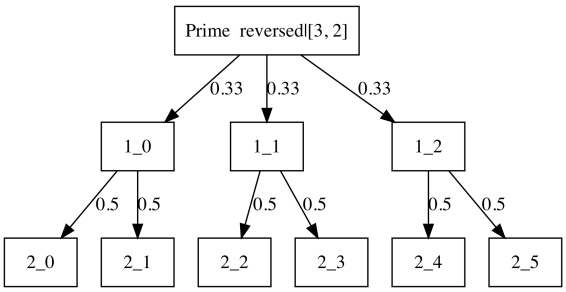
In all the examples, the leaves of the trees always constitute the actual server-nodes of the problem, while all other nodes represent the sub-problems. All split-strategy examples in Figure 7 also show on their edges how the workload is split at each sub-problem node, so that in the end all nodes have an equal workload independent of where they are located in the strategy split-tree.



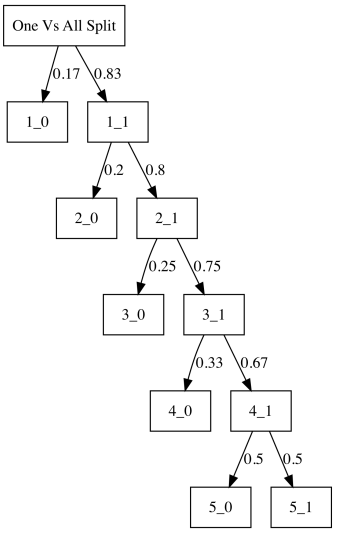

(a) Approximation of a Binary Tree




(b) Approximation of a Quaternary Tree




(c) Prime Factor Tree with Factors Reverse Ordered by Size

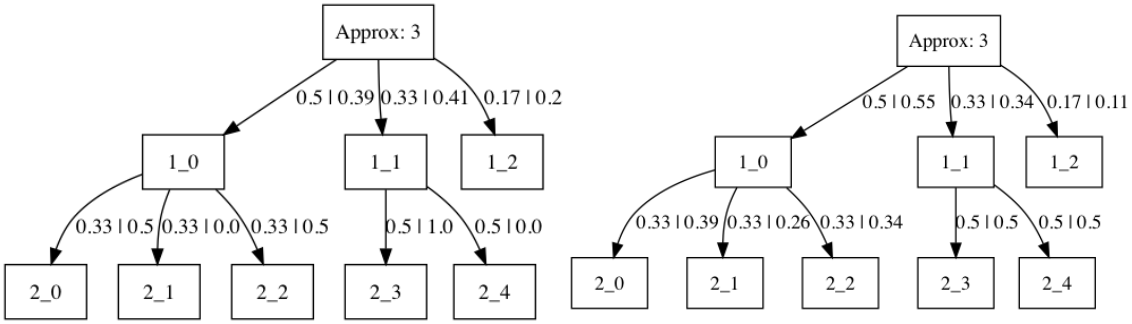



(d) One vs. All Tree

Figure 7: Examples for different Split Strategies for a Problem with six Nodes

## 4.2. Effect of Robust Optimization on Tree-based Decomposition

As robust optimization does not enforce a completely fair workshare, it can have some unintended effects when using it with the tree-based decomposition approach. In Figure 8a, we can observe that server-node *2_1* and *2_4* do not get assigned any work at all. The edges of the tree are annotated with (Fair Workshare | Actual Workshare). Fortunately, this can be counteracted by increasing the epsilon value as can be seen in Figure 8b. While the workshare is still not completely fair, all nodes get work assigned.




(a) Epsilon set to 100 000

(b) Epsilon set to 400 000

Figure 8: Ternary Trees with different Epsilon Values

# 5. Performance Evaluation

To compare our different splitting strategies, we generated a number of problems and let our strategies solve these problems under a varying node count. We use the number of nodes to increase the computational complexity while keeping all other factors the same. This allows us to draw conclusions on how the node count of a cluster affects the choice of the splitting strategy.

To also illustrate the effect of relaxing the fair workshare constraint, we will compare the results with and without robust optimization turned on. On the left side, robust optimization will be turned off, while the right side will have robust optimization enabled.

According to our quality characteristics defined in section 2.2, we will compare our results along the dimensions of replication space, deviation from fair workshare and runtime. Each diagram can be seen as a summary on its own, while a pair of diagrams mainly offers the chance to see the effect of the penalty factor epsilon in practice.

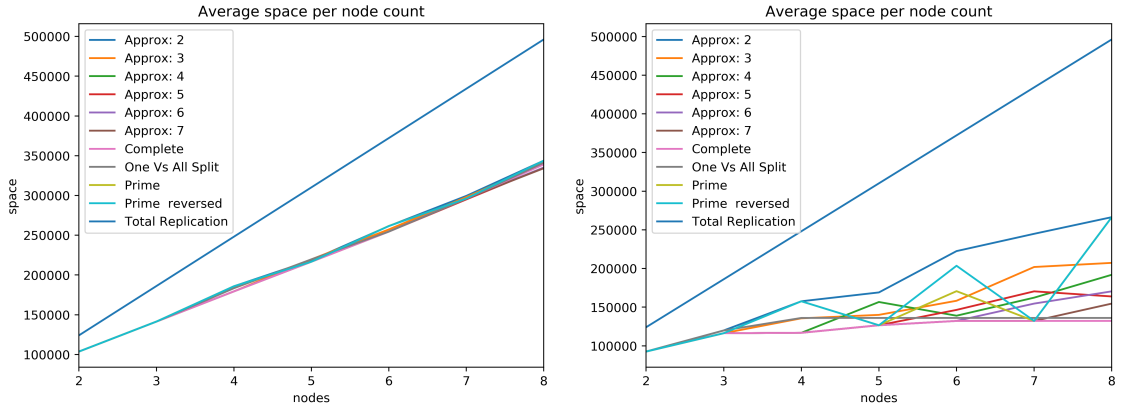
The ground-truth in the following figures is always the "Complete" splitting strategy. It corresponds to the approach presented in section 2 which can be interpreted as a tree with only one node an and $n$ leafs. Since it is not a heuristic it is not forced to make inherently non-optimal decisions. This can especially be seen in Figure 12, where all strategies are compared to the performance of the "Complete" strategy. As we have shown in subsection 2.3, however, the complete-split strategy requires a lot of time to compute an optimal solution. Thus, we have applied a timeout of 10 seconds to it.

## 5.1. Space

In Figure 9, we can see the amount of total space which a solution of a selected splitting strategy requires in our example problem. The most important thing we see is that all strategies are better than using the trivial solution of full replication. In addition, we notice that the gap between the total replication solutions increases with the node count. This observation is explained could be explained by node becoming more and more specialized to certain queries, which requires fewer fragments per node. Similar effects can be observed in societies with increasing size, where the individual becomes more and more specialized in his work [1].

Moreover, we see that in Figure 9a, the different strategies are performing very similarly. This effect is primarily good since it proofs that the heuristics perform as good or only slightly worse than the optimal solution.

On the right side in Figure 9b, we can see that all strategies using the robust optimization goal are having a lower total space than when using the strictly fair workshare distribution.

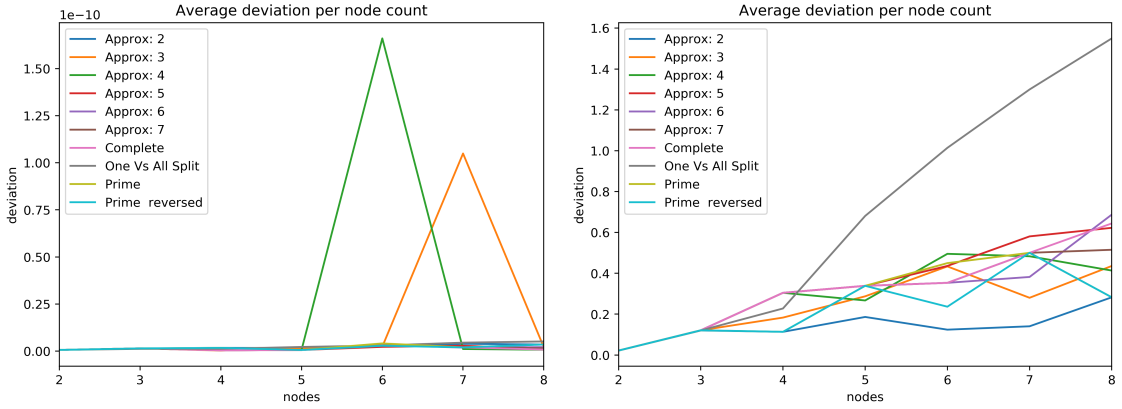



(a) Without Robust Optimization (b) With Robust Optimization

Figure 9: A comparison between the Average Space per Node Count per Strategy

We also see a lot higher variance of the total space between the different strategies. This can be more generalized by the fact that the more sub-problems a strategy creates, the higher the total space will be. We see that the strategies are performing better the bigger the split is they make with the exception of the *one vs. all strategy* which also performs very well.

## 5.2. Deviation from Fair Workshare




(a) Without Robust Optimization (b) With Robust Optimization

Figure 10: A comparison between the Average Deviation per Node Count per Strategy

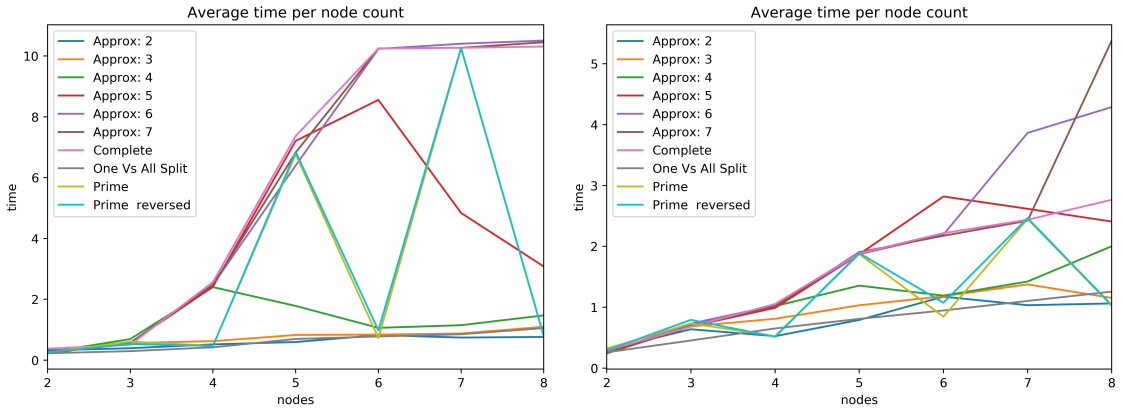
Within the deviation charts in Figure 10, we can see on the left side that enforcing a strict workshare distribution works. The peaks we see are numerical inaccuracies. This becomes clear when looking at the scale of the diagram, which is $10^{-10}$.

The more interesting part is the deviation with robust optimization enabled in Figure 10b. We can see that we achieve a relatively low deviation across most of the splitting strategies. When looking at "Approx: 6" strategy, which is one of the worse "Approx" strategies, we can see that it has a deviation of about 70 percentage points for eight nodes which equates to an average deviation of about nine percentage points per node. Interestingly, we see that the strategies performance is sorted in reverse order in comparison to the total space diagram. This makes sense since the more deviation from a fair workshare

distribution we sacrifice, the more space we can save by not having to replicate additional fragments.

The performance of the "One Vs All Split" can be explained by the fact that this strategy tends to ignore nodes completely according to the effect described in subsection 4.2. This effect is particularly present in the first sub-problem splits where server-node is relatively less important compared to the sub-problem node, as it is only meant to receive a small workshare as can be seen in Figure 7d.

## 5.3. Run-Time




(a) Without Robust Optimization

(b) With Robust Optimization

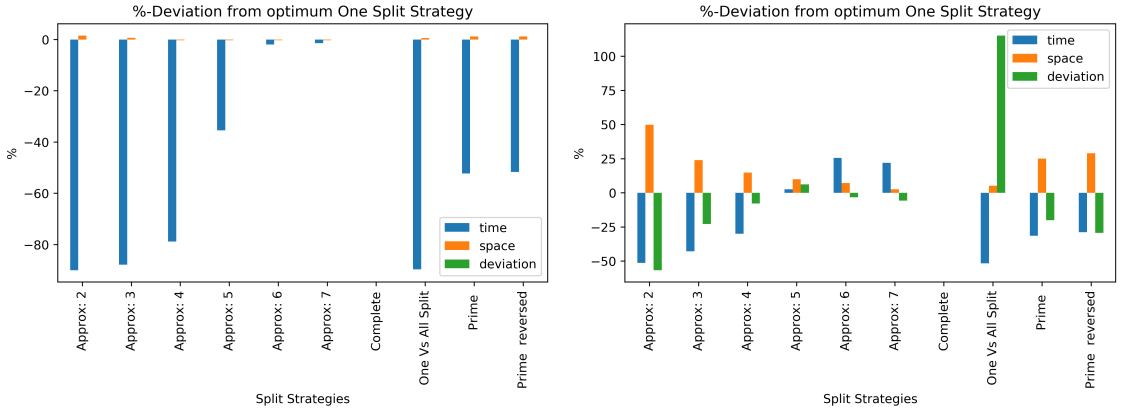
Figure 11: A comparison between the Average Time per Node Count per Strategy

The time comparison charts in Figure 11 allow us to see the efficiency of the proposed heuristics. When looking at the strict workshare distribution chart on the left, we see that the majority of heuristics are drastically faster than the optimal solution. It is important to note that a 10-second timeout was placed on the solver, which means that the unrestricted difference might be larger. As an example, we see that "Approx: 2" with a time of under a second is at least *a whole magnitude* faster than the reference solution while having a similar total space. Peaks on certain strategies can be observed because of the fact that some strategies are equal to the complete split for certain node counts. This is, for example, the case if the node number is smaller or equal than the approx number.

On the robust optimization side, we see that the strategies are grouped closer together and generally require less time. This can be explained by the fact that the constraint of a fair workshare distribution is relaxed by the penalty factor epsilon, which makes the problem computationally easier. All algorithms perform under cut-off time, which makes the comparison fair.

We can conclude from both charts that robust optimization generally makes the problem computationally easier to solve and therefore faster.

## 5.4. Aggregated Comparison




(a) Without Robust Optimization

(b) With Robust Optimization

Figure 12: A Summary View of the Strategies against the One Split "Complete" Strategy

In the two summary charts seen in Figure 12, we see results which are aggregated over the different node counts. This allows us to make statements about the general case instead of talking about certain node counts. The parameters we compare are the quality characteristics introduced in section subsection 2.2. The percentages given in the chart are relative to the results from the "Complete" strategy, meaning that for example a $-90\%$ value in run-time means that it requires 90% less time compared to the "Complete" strategy or alternatively is ten times faster.

On the left side, in Figure 12a, we see our general impression confirmed that the heuristics offer a drastically improved run-time while only deviating slightly in the total space result. Moreover, we also conclude that the optimal solution, doing one big split, is essentially the best solution space-wise. We do not compare deviation there as it is simply of a numerical nature.

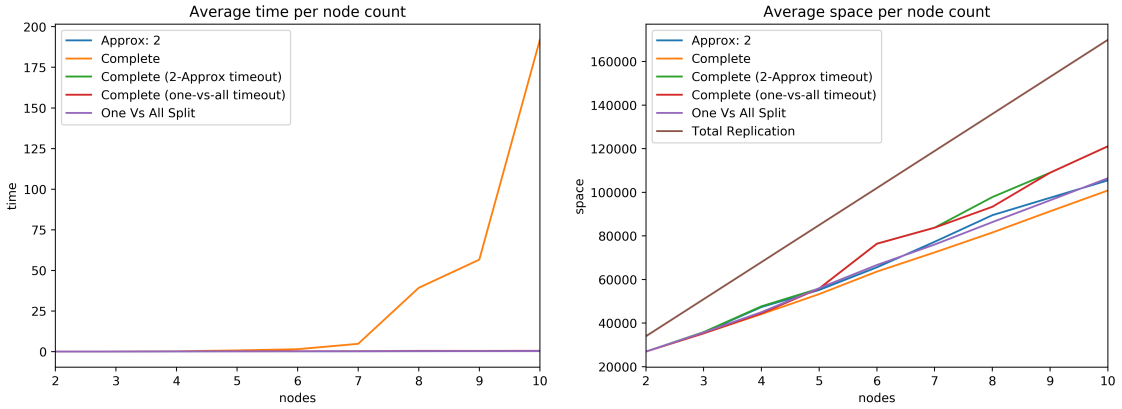
On the right side, in Figure 12b, we see a more heterogeneous image. We see that the run-time of most heuristics is still better than the optimal solution, but the differences in space are much larger. This makes sense since the deviation of those algorithms is also smaller. For most algorithms, you see a similar-sized deviation and total-space bar, which further illustrates the deviation-space trade-off. Surprisingly, only the result of the "Approx: 5" strategy could be considered bad, since all quality parameters are worse than the ones of the optimal solution.

## 5.5. Heuristic vs. Early-Exit Complete-Split

Finally, we want to investigate how the heuristical splitting strategies compare with the complete-split when the complete-split is given the same run-time. To achieve this, we run the heuristics first and then assign the complete-split a timeout equal to the run-time of the heuristic. If the complete-split is not able to produce a solution in the given time, the first solution is taken.

We decided only to run the two most promising heuristics. In Figure 13a and Figure 14, we can see that both the heuristics and the time-limited complete-split are able to save almost 100% of the run-time compared to the optimal complete-split. Additionally, we can observe in Figure 13 and Figure 14 that the heuristics need less replication space than the time-limited complete-splits and only slightly more space than the optimal complete-split.

When we conduct the experiment with robust optimization, we can observe the same behavior, albeit to a lesser degree. Though, the problem needs to be sufficiently complex (meaning the number of fragments and queries is large compared to the number of nodes), as the problem is otherwise trivially fast to solve for *Gurobi*. However, as we have seen in subsection 4.2 and Figure 12, the divergence can be quite a bit higher depending on the chosen epsilon parameter.




(a) (b)

Figure 13: Comparison of Heuristics and Early-Exit Complete-Split without Robust Optimization

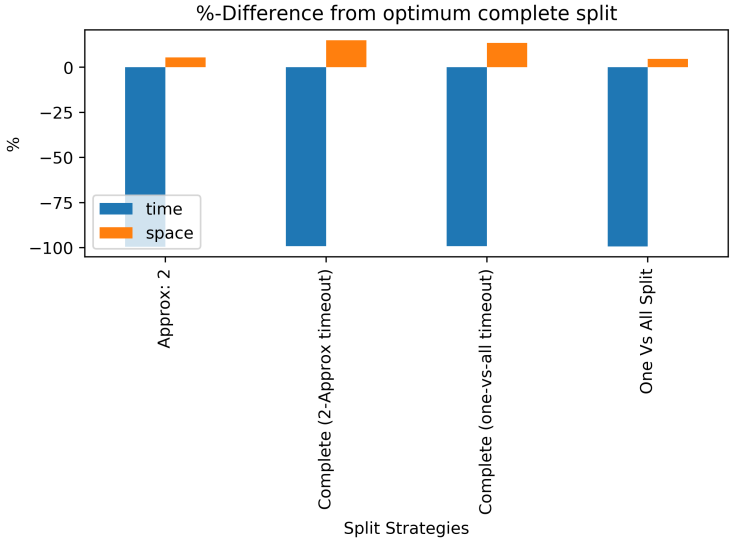



Figure 14: Aggregated Comparison of Heuristics and Early-Exit Complete-Split without Robust Optimization

## 6. Conclusion

In this work, we represented and solved the data partitioning problem using linear programming. Moreover, we introduced two approaches to tackle the scalablility of this in computationally NP-hard problem. First, we utilized the ability of the *Gurobi Solver* to perform an early exit to deliver a sub-optimal result. Second, we introduced multi-layer

tree-based decomposition heuristics to solve the problem more efficiently. Our experiments have shown that these heuristics can achieve an up to one magnitude faster run-time while showing a comparable space consumption and thus solution quality, as well as compare favorably to the already very good early-exit approach.

In addition to these strategies, we also show the effect of relaxing the fair workshare constraint using a worst-case optimization. While we can save up to 60% more space using this method, there are also other quality characteristics which suffer from this choice. However, we showed that even a small decrease in workshare equality can lead to large amounts of saved space. To illustrate this trade-off, we present a Pareto front to showcase the possible solution space.

## References


[1] Jeanson, Raphaël ; Fewell, Jennifer H. ; Gorelick, Root ; Bertram, Susan M.: Emergence of Increased Division of Labor as a Function of Group Size. In: *Behavioral Ecology and Sociobiology* 62 (2007), Nr. 2, 289–298. http://www.jstor.org/stable/25511695. – ISSN 03405443, 14320762

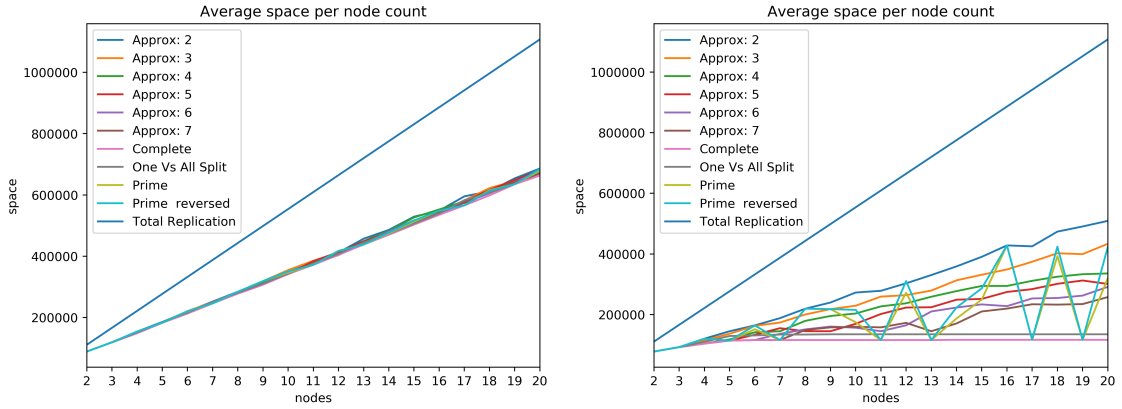
[2] Plattner, Hasso: *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases*. Second Edition. Berlin, Heidelberg : Springer-Verlag, 2014

[3] Rabl, Tilmann ; Jacobsen, Hans-Arno: Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017, S. 315–330

[4] Vanderbei, Robert J.: *Linear Programming: Foundations and Extensions*. Fourth Edition. New York : Springer Science+Business Media, 2014

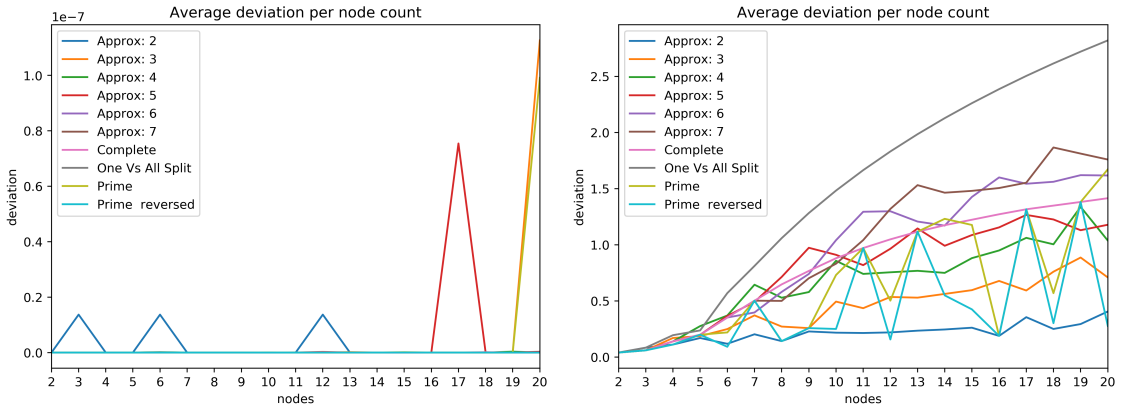
# Appendices

## A. Performance Charts



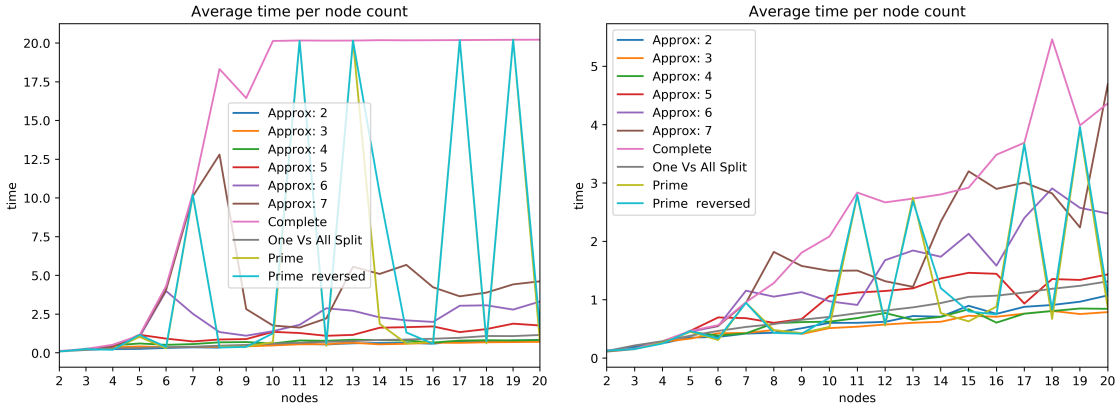

(a) Without Robust Optimization (b) With Robust Optimization

Figure 15: Average Space of different Heuristics




(a) Without Robust Optimization (b) With Robust Optimization

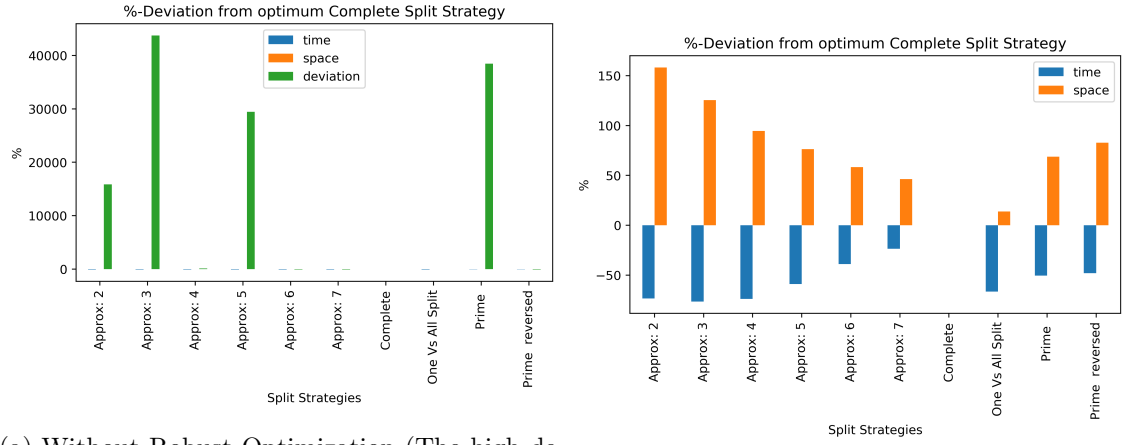
Figure 16: Average Deviations from fair Workshare of different Heuristics




(a) Without Robust Optimization

(b) With Robust Optimization

Figure 17: Average Run-Times of different Heuristics




(a) Without Robust Optimization (The high deviation is purely of a numerical nature and can be ignored)

(b) With Robust Optimization

Figure 18: Summary View of Aggregated Performance of Heuristics