Solving Index Selection Problems with General Linear Programming Models

Leonardo Hübscher, Oliver Nordemann, Marcel Weisgut
Hasso Plattner Institute, University of Potsdam, Germany
{firstname.lastname}@student.hpi.de

ABSTRACT

Fast processing of database queries is a primary goal of database systems. Indexes are a crucial means for the physical design to reduce the execution times of database queries significantly. Since indexes cause additional memory consumption and the storage capacity of databases is limited, the choice of indexes is also limited, making this an NP-hard problem. Therefore, it is of great interest to determine an efficient selection of indexes for a database management system (DBMS) and thus find a sufficient solution for the so-called index selection problem. This work aims to show how a specific basic index selection problem and various extensions of it can be solved using linear programming. For each defined index selection problem, we present a formal linear programming model that solves the problem. Furthermore, we implemented these solutions with the modeling language AMPL[1] and evaluated these implementations using synthetic data sets. We present these evaluation results and indicate possible points on how this project could be continued in the future.

1. INTRODUCTION

Indexes in a relational database system are auxiliary data structures used to reduce the execution time required for generating the result of a database query. The shorter the execution time of a workload's query set, the more queries can be executed in a limited time frame. Consequently, the goal of using indexes is to increase the throughput of the database. However, since indexes are data structures that have to be stored in addition to the stored data of a database itself, indexes come with specific additional memory consumption and thus increase the overall memory footprint of the database. Since memory is limited and therefore a valuable resource, it is important to take the memory consumption into account for decision making about what indexes should be used and stored in the system's memory.

For a single database query, multiple indexes may exist, each of which can improve the query execution time differently. Table 1 shows an exemplary scenario in which different combinations of indexes lead to different execution times of a single hypothetical example query. The first combination without any index leads to the longest execution time of the query with 500 milliseconds. The best execution time of 300 milliseconds can be achieved by using both index 1 and index 2. However, the combination which performs the best regarding the execution time also has the largest memory footprint. The second-best solution from an execution time perspective has an index memory consumption of only 40% of the optimal solution and is only about 17% slower. Although this is a strongly constructed example, it illustrates the need to consider the index memory consumption for selecting which indexes should be used.

usage of index 1	usage of index 2	total memory footprint	query execution time
false	false	0 MB	500 ms
true	false	100 MB	350 ms
false	true	150 MB	400 ms
true	true	250 MB	300 ms

Table 1: Sample index combinations with their memory consumption and the resulting execution times of a hypothetical query.

In real-world database scenarios, a DBMS processes more than only a single query. Instead, a set of queries is executed on a database with a certain frequency in a specific time frame for each query. The set of queries with their frequencies is referred to as workload. Executing a workload using a selection of indexes has a certain performance. This performance is characterized by the total execution time of the workload and the selected indexes' memory consumption. The workload execution time should be as low as possible, and the index memory consumption must not exceed a specific memory budget. An additional challenge of selecting the set of indexes that shall be present and used by queries is index interaction. "Informally, an index a interacts with an index b if the benefit of a is affected by the presence of b and vice-versa."[5] For example, a particular index i for a subset S of the overall workload may provide the best performance improvement for each query in that subset. There is also no other index that has a better accumulated performance improvement. Suppose i now has such a high memory consumption that the available index memory budget is completely spent. In that case, no other index can be created. Therefore, only queries of the subset S are improved by index i. Another index selection might be worse for the workload subset S but better for the overall workload. Consequently, a single index whose accumulated performance improvement is the highest is not necessarily in the set of indexes that provides the best performance improvement for the total workload.

In this work, we present optimization strategies based on

[1]`https://ampl.com/`

linear programming to solve complex index selection problems. Besides one basic problem, four extended problems and their solutions are explained. The remainder of this work is structured as follows. In section 2, the various index selection problems are formulated, and their solutions are presented. Section 3 then briefly describes how the linear programming models were implemented by using AMPL. An evaluation of the developed models is given in section 4. In section 5, we then provide ideas about how the project could be further evaluated and advanced as possible future work. Finally, section 6 concludes this work.

2. APPROACH

In this section, the various index selection problems are formulated, and the solutions for each problem are presented. For this purpose, section 2.1 first provides information about the context in which this work was developed and the given requirements. Section 2.2 briefly explains the concept of linear programming, which serves as the basis for the solutions of the index selection problems. Section 2.3 describes an index selection problem that is considered the basic problem in this work. In addition to the problem's description, the linear programming model is also formulated, with which this problem was solved. Sections 2.4, 2.5, 2.6 and 2.7 each describe an extended problem of the basic problem and explain which adjustments had to be made to the solution of the basic problem to solve the specialized problems. Finally, section 2.8 describes the problem in which all advanced problems were combined.

2.1 Given Task

This work was developed in the context of the master's course *Data-Driven Decision-Making in Enterprise Applications* under the direction of Dr. Rainer Schlosser which took place in the summer term 2020 at the Hasso Plattner Institute in Potsdam. The task specifies to solve the basic index selection problem using linear programming. Based on the initial problem, we should identify further problems and propose solutions to them.

2.2 Linear Programming

The process of decision-making using linear programming can be divided into three stages. In the first stage, the decision problem has to be formulated in natural language. This decision problem is then translated into a formal linear programming (LP) model in the second stage. Lastly, this model is then passed to a standard LP solver, which solves the problem and returns the optimal solution as output.

In this work, the focus is on the first two stages, i.e., formulating the decision problem informally in natural language and formally as a mathematical model. For the letter one, knowledge about the concept of linear programs is essential. In general, the three crucial components of a linear program are (i) the decision variables, (ii) the constraints for the decision variables, and (iii) one objective function [4]. The (i) decision variables are the controls that can be set by the decision-maker. The solver determines these variables. The (ii) constraints represent restrictions on the decision. They are in the form of linear expressions, which must not exceed ($\leq$), not fall below ($\geq$) or exactly equal ($=$) a specified value [7]. The (iii) objective function is a single linear expression to be maximized or minimized.

2.3 Basic Problem

Properties of the index selection problem described in this section are explicitly defined for this work and are not generally valid. The basic index selection problem is about finding a subset of a given set of index candidates used by a hypothetical database to minimize the total execution time of a given workload. The given workload consists of a set of queries and a frequency for each query. A query can use no index or exactly one index for support. Different indexes induce different improvements for a single query. As a result, the execution time of a query is highly dependent on which index is used for support. A query has the longest execution time if no index is used. For each query, it has to be decided whether and which index is to be used. Only if at least one query uses an index, the index can belong to the set of selected indexes. Each index involves a certain amount of memory consumption. The total memory consumption of the selected indexes must not exceed a predefined index memory budget.

designation	type	description
I	parameter	number of indexes
Q	parameter	number of queries
M	parameter	index memory budget
$t_{q,i}$	parameter	execution time of query $q = 1, .., Q$ using index $i = 0, .., I$; $i = 0$ indicates that no index is used by query q
m_i	parameter	memory consumption of index $i = 1, .., I$
f_q	parameter	frequency of query $q = 1, .., Q$
$u_{q,i}$	decision variable	binary variable whether index $i = 0, ..., I$ is used for query $q = 1, ..., Q$; $i = 0$ indicates that no index is used by query q
v_i	decision variable	binary variable whether index $i = 1, .., I$ is used for at least one query

Table 2: Parameters and decision variables of the basic LP model.

Table 7 shows the formal representation of the given parameters and the decision variables that have to be determined by the LP solver. The binary variable $u_{q,i}$ is used to control if an index i is used for query q. Variable v_i is derived from the values stored in $u_{q,i}$ and only used to calculate the overall memory consumption of selected indexes. The formal index selection LP model is defined as follows:

$$\underset{u_{q,i} \in \{0,1\}^{Q \times I}}{minimize} \quad \sum_{q=1,...,Q,\ i=0,...,I} u_{q,i} \cdot t_{q,i} \cdot f_q \tag{1}$$

$$\text{subject to} \quad \sum_{i=1,..,I} v_i \cdot m_i \leq M \tag{2}$$

$$\sum_{i=0,...,I} u_{q,i} = 1 \qquad \forall q = 1, ..., Q \tag{3}$$

$$\sum_{q=1,...,Q} u_{q,i} \geq v_i \qquad \forall i = 1, ..., I \tag{4}$$

$$\frac{1}{Q} \cdot \sum_{q=1,...,Q} u_{q,i} \leq v_i \qquad \forall i = 1, ..., I \tag{5}$$

The objective (1) minimizes the execution time of the overall workload taking into account the index usage for queries, the index-dependent execution times and the frequency of queries. The constraint (2) ensures that the selected indexes do not exceed the given memory budget. Constraint (3) ensures that a maximum of one index is used for a single query. It is a linear equality constraint (=) since the lower bound of summation is 0, and $i = 0$ represents no index. Thus, if $u_{q,i}$ with $i = 0$ is true, no index is used for query q. The constraints (4) and (5) are required to connect $u_{q,i}$ with v_i. If no query uses a specific index i, constraint (4) ensures that v_i is equal to 0 for that index. If at least one query uses index i, constraint (5) ensures that v_i is equal to 1 for that index.

2.4 Chunking

The number of possible solutions of the index selection problem grows exponentially by the amount of index candidates. In modern enterprise applications we have hundreds of tables and thousands of columns. This leads to long execution times to find the optimal solution of the increasing problem. In this extension the idea behind chunking is to split all possible indexes into chunks. The index-selection problem will be solved only with the reduced amount of indexes for each chunk, and the indexes of the optimal solution will be returned. After solving the problem for each chunk, the best indexes of each chunk will get on. In the second round the reduced number of indexes will be used. The optimal solution for each preselection will be found. Chunking does not ensure that the optimal solution of the original problem could be calculated because some indexes of the optimal solution might not be a winner of their chunk. Another aspect is that the division of the problem in many smaller problems adds overhead which is not beneficial in every case. One more advantage is that in the first round all the chunks could be solved in parallel. With this approach the overall execution time could be reduced even further.

2.5 Multi-Index Configuration

Our first basic problem introduced in section 2.3 could not handle the interaction of indexes as described in the introduction: One query could be accelerated by more than one index and the performance gain of an index could be affected by other indexes. We tackled this part of the index selection problem by adding one level of indirection called index configurations. An index configuration maps to a set of indexes. Assuming the index selection problem has 10 indexes, then the first configuration (configuration 0) means, that no index is used for this query. The next 10 configurations point to the respective indexes, e.g., configuration 1 points to index 1, configuration 2 points to index 2, configuration 10 points to index 10. The subsequent configurations map to sets containing combinations of two indexes. Database queries could be accelerated by more than two indexes, but we simplified the configurations in our implementation so that they can consist of a maximum of two different indexes. We created a binary parameter $d_{c,i}$ which indicates whether configuration c contains the indexes i. Furthermore, we assume that 10 percent of all possible index combinations will interact in configurations. The index selection algorithm operates on configurations in the same way it works on indexes directly. The constraints (3, 4, 5) of the basic problem in section 2.3 are adapted in the following way for multi-index configurations.

$$\sum_{c=0,...,C} u_{q,c} = 1 \qquad \forall q = 1, ..., Q \tag{6}$$

$$\sum_{q=1,...,Q} u_{q,c} \cdot d_{c,i} \geq v_i \qquad \forall c = 1, ..., C \quad \forall i = 1, ..., I \tag{7}$$

$$\frac{1}{Q} \cdot \sum_{q=1,...,Q} u_{q,c} \cdot d_{c,i} \leq v_i \qquad \forall c = 1, ..., C \quad \forall i = 1, ..., I \tag{8}$$

The binary variable $u_{q,c}$ is used to control if configuration c is used for query q and C is the number of index configurations. Similar to the basic approach, $c = 0$ represents a configuration that contains no index. Constraint (6) ensures that one single query uses exactly one configuration instead of one index. In Constraint (7) and (8) where the used and unused indexes are connected, the parameter $d_{c,i}$ is added.

2.6 Stochastic Workloads

Until now, we considered a single workload only. However, in the context of enterprise applications, we could imagine that each day of the week has a different workload. For example, the workloads on the weekend could contain fewer requests compared to a workload during the week. In this section, we propose a solution that can take multiple workloads into account. The solution should provide a robust index selection, where robust means that it should not be too sensitive for one specific workload.

First, the expected workload cost T across all workloads is being calculated, where W is the number of workloads. Additionally, we introduce a workload frequency k_w to be able to manually tweak the algorithm.

$$T = \sum_{w=1,...,W} \frac{g_w * k_w}{\sum_{w2=1,...,W} k_{w2}} \tag{9}$$

The execution time g_w of a workload w is determined by formula 10, with $f_{w,q}$ being the frequency of query q in workload w.

$$g_w = \sum_{q=1,...,Q,\ c=1,...,C} u_{q,c} * t_{q,c} * f_{w,q} \tag{10}$$

The information whether a configuration c is being used for a query q of a workload w is shared between the workloads, leaving it to the solver to minimize the total costs across all workloads.

Figure 1 shows exemplary total workload costs when minimizing the global execution time. It can be seen that the actual costs of each workload differ a lot, leading to poor performances for $W1$ and $W3$ in favor of $W2$ and $W4$. We use

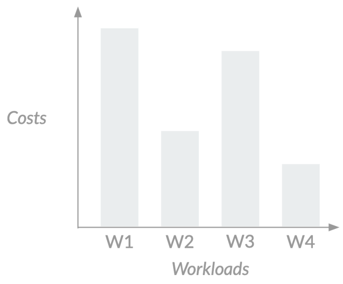

Figure 1: The costs for each workload when minimizing the global costs *(lower is better)*

two different approaches to make the index selection more robust. The first one optimizes the worst-case by punishing the total costs with the maximum workload costs as additional costs. The maximum workload costs l are determined by the following constraint:

$$\textit{subject to} \qquad l \geq g_w \qquad \forall w = 1, ..., W \quad (11)$$

The LP includes the maximum workload cost into the objective using the penalty factor a.

$$\underset{u_{q,i} \in \{0,1\}^{Q \times I}}{\textit{minimize}} \Big(\sum_{q=1,\dots,Q,\ i=0,\dots,I} u_{q,i} \cdot t_{q,i} \cdot f_q \Big) + a * l \qquad (12)$$

Figure 2 shows that no workload cost exceeds the worst-case cost. However, the costs of $W2$ and $W3$ increased, leaving a bigger gap between $W4$ and the rest. The second approach

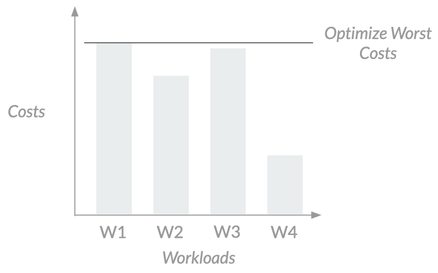

Figure 2: The workload costs, when increasing the robustness by optimizing the worst-case costs. *(lower is better)*

uses the mean-variance V based on all workload costs as a penalty to minimize the cost differences. Again, a factor b is applied to control the influence of the penalty.

$$V = \sum_{w=1,\dots,W} \Big(g_w - \frac{\sum_{w2=1,\dots,W} g_{w2}}{W}\Big)^2 * \frac{k_w}{\sum_{w3=1,\dots,W} k_{w3}} \qquad (13)$$

Using the above formulas results in Figure 3. As you can see, all costs are now within the mean-variance. However, in comparison to the previous figure, not only have been $W1$, $W2$ and $W3$ brought into the mean-variance, but also $W4$. The total costs of $W4$ may only be increased artificially to drag the costs into the mean-variance, which makes it indeed more robust but less effective in the end.

A third option would be to use the semi-variance. However, this would leave us with a non-linear problem, which is the reason why we did not pursue this. This extension now enables us to use workloads that are known from the past to improve the index selection in the present in an ongoing matter.

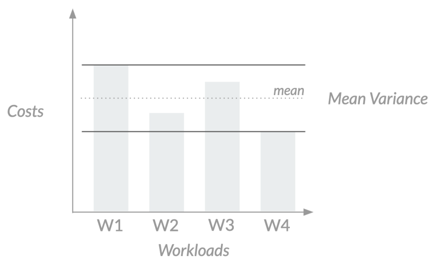

Figure 3: The workload costs, while increasing the robustness by optimizing using the mean-variance. *(lower is better)*

2.7 Transition Costs

In the previous chapter we have shown how to deal with different workloads on consecutive days. But we also need to think about how to transition between two index selections. We assume that in the real world the database would need to remove indexes that are no longer used and load indexes that are going to be used into the memory. Typically, the database would need to do some I/O operations, which are time expensive and generate additional costs. We decided to model such creation and removal costs in our final extension to reduce such transition costs. To adapt the index configurations, the algorithm identifies the differences between the previous configuration of v_i^* and target configuration of v_i. For each removal at index i, the algorithm looks up the removal costs rm_i for index i and adds them to the total removal costs RM. Analogous, the algorithm proceeds to calculate the total creation costs MK using the creation costs mk_i of the index i. The sum of the removal costs and creation costs is then being added to the objective which results in avoiding high transition costs. The resulting formulas are the following:

$$RM = \sum_{i=1,\dots,I} v_i^* * (1 - v_i) * rm_i \qquad (14)$$

$$MK = \sum_{i=1,\dots,I} v_i * (1 - v_i^*) * mk_i \qquad (15)$$

Table 3 describes an explanatory calculation of the transition costs between two workloads. The previous workload uses the indexes one and three. The next possible index selection uses index 1 and 2. The resulting transition costs are 50.

index			$\mathbf{v}_i^*$	$\mathbf{v}_i$	**RM+MK**
#	mk_i	rm_i			
1	50	10	1	1	0 (keep)
2	20	5	0	1	20 (create)
3	100	30	1	0	30 (remove)
4	10	1	0	0	0 (skip)
total transition costs					**50**

Table 3: Transition cost calculation example

2.8 Combined Problem

All extensions that have been described in the previous sections have been developed on top of the basic approach. To show the encapsulation of all extensions, we decided to build a solution that integrates all extensions in an *all-in-one solution*. In most of the cases, this was straight forward. One minor issue that showed to be more complex was the integration of the chunking extension, as it required to carefully think about when constraints or variables should work on chunks. On the one hand side, a lot of smaller changes were required. Listing all changes would therefore lead to adjusting nearly all formulas and would go beyond the scope of this work. On the other hand side, we were able to integrate all extensions into a single solution. In the end, we have shown the independence of all extension implementations.

3. IMPLEMENTATION

To be able to evaluate the described approach, we implemented the program in AMPL[2], which is a modeling language tailored for working with optimization problems. The syntax of AMPL is quite close to the algebraic one and should be easy to read and understand, even for the readers, who have never seen AMPL syntax before. AMPL itself translates the given syntax into a format linear solvers can understand.[1][6]

The solver is a separate program that needs to be specified by the developer. As described in section 2.2, the approach is based on linear programming using integer numbers. Both solvers, *CPLEX*[3] and *Gurobi*[4], are suited to solve the index selection task. A first test showed that Gurobi is faster than CPLEX in most cases, which is the reason why we use Gurobi.

The main entry point is the *.run*-file, which contains information about the selected solver and loads the specified model and data specifications. After a given problem was solved, the solution is displayed. The *.mod*-file is the most interesting one, as it contains the description of the mathematical model, such as parameters, constraints, and objectives. The input data, which is required for solving a certain problem, is specified in the *.dat*-file. All code files are available at GitHub[5].

The implementation in AMPL allows us to evaluate our approach and its extensions.

4. EVALUATION

In this section, we evaluate the implementation of our approach concerning the used evaluation setup, the input data, and several disputes regarding our design decisions. First, we reflect on the scalability of our basic approach. Then we investigate when index chunking is beneficial for the performance compared to the basic approach and reflect on the cost trade-off that the heuristic entails. Afterward, we determine the computational overhead of the *multi-index extension*. Lastly, we take a more in-depth look into the *stochastic workload extension*, evaluating the impact of the different robustness measures and the trend of costs depending on the number of training workloads.

[2]`https://ampl.com/`

[3]`https://www.ibm.com/de-de/analytics/cplex-optimizer`

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

[5]`https://github.com/mweisgut/DDDM-index-selection`

4.1 Evaluation Setup

All performance measurements have been made on the same machine, featuring an Intel i5 8th generation (4 cores) and 8GB memory storage. All measurements have been repeated three times. For each time measurement we used the AMPL build-in function *_total_solve_time*. It returns the user and system CPU seconds used by all solve commands.[3]

The final value was determined by the mean of all three measurement results. All non-related applications have been closed to reduce any side effects of the operating system.

4.2 Datasets

The datasets that are being used for the evaluation are being generated randomly, using multiple fixed random seeds. Each dataset is defined by the number of indexes, queries, and available memory space. The algorithm provided in the *index-selection.data*-file then generates the execution time, that each query has, depending on the utilized index. Firstly, the "original" execution time for the query without using any index are chosen randomly within the interval [10; 100]. Based on the drawn costs, the speedup for each index is calculated by choosing a random value between the "original" costs and a 90% speedup. The memory consumption of a query can be an integer between one and 20. The frequencies can be between one and 1000.

The extensions that are applied on top of the basic approach introduce further variables that need to be generated. For the *stochastic workload extension*, we introduced a workload frequency, which gets drawn randomly for each workload. This also applies to the transition cost extension, where the creation costs and removal costs are random. The multi-index configurations package requires a more complex generation process since each index configuration should be a unique set of indexes. The configuration *zero* represents the option that no index is being used. The configurations 1 to I point to their respective single index. All other generated configurations consist of up to two indexes, whereas the combinations are drawn randomly. By using a second data structure, it is ensured that no index combination is used multiple times. The speedup s for a combination, existing of two indexes i and j, is then calculated by the following formulas:

$$min\_speedup_{s_{i,j}} = max(s_i, s_j) \tag{16}$$

$$max\_speedup_{s_{i,j}} = s_i + s_j \tag{17}$$

The minimum speed up and the maximum speed up are then passed to a function that returns a random number within the interval $[min\_speedup_{s_{i,j}}; max\_speedup_{s_{i,j}}]$.

Outsourcing the generation of input data into the *index-selection.data*–file allows for an easy replacement with actual data, e.g., benchmarking data of a real system. However, this also enables the questioning reader to validate basic example cases on their own.

4.3 Basic Index Selection Solution

"The complexity of the view- and index-selection problem is significant and may result in high total cost of ownership for database systems." [2] In this section we evaluate our basic solution. We show the scalability of our implemented basic solution and later we compare it to the chunking approach. We set up the memory limit with 100 units and assume 100 queries with random occurrences uniform between 1 and 1000. To test the scalability on our machine we generate data with 50, 100, 150, ..., 1450, 1500 index candidates. The outcome measurements are shown in Figure 4. The total solve time on the y-axis is a logarithmic scale.

The Figure 4 shows the growing total solve time while the number of indexes rises. With an increasing index count, the execution times vary more. Depending on the generated data, the solver could optimize the execution in some cases and find the optimum faster. In some cases over 1000 indexes the generated input could not be solved with our setup in a meaningful time. In general the possible combinations of the index selection problem grows exponentially, whereby the used solver could optimize the calculation to reduce the execution time. Furthermore Figure 4 shows nearly a linear increasing graph. With the logarithmic scale it seems to be an exponential growing total solve time in our evaluation setup.

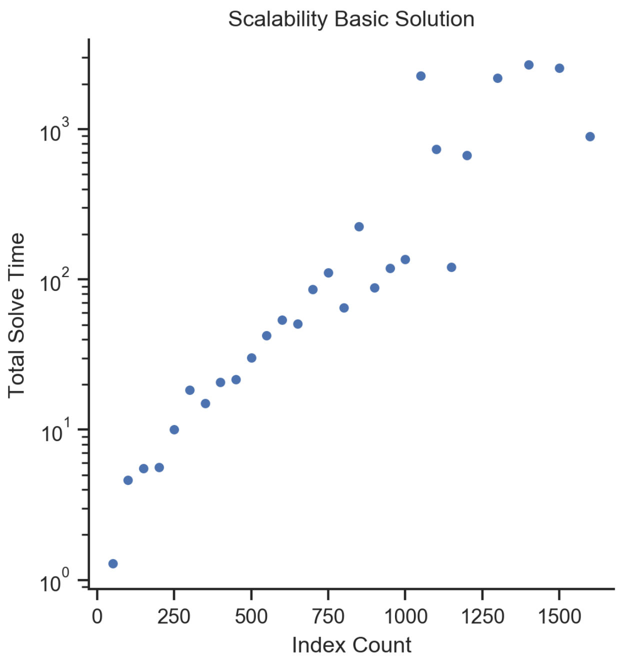

Figure 4: Execution times of basic solution *(lower is better)*

4.4 Index Chunking

To tackle the exponential growing index selection problem, we divide the problem into chunks, find the best indexes of each chunk and then find the best index of the winners of all chunks. Compared to the basic index selection solution, problems with much more indexes could be solved with chunking. Figure 5 shows the total solve time with chunking in orange and without chunking in blue. The other parameters were fixed (see previous section 4.3). The orange dots of the chunking approach show a linear relationship between an index count of 500 to 2500. In the beginning, the total solve time of the chunking curve has a higher gradient. The overhead introduced by chunking to split the indexes into

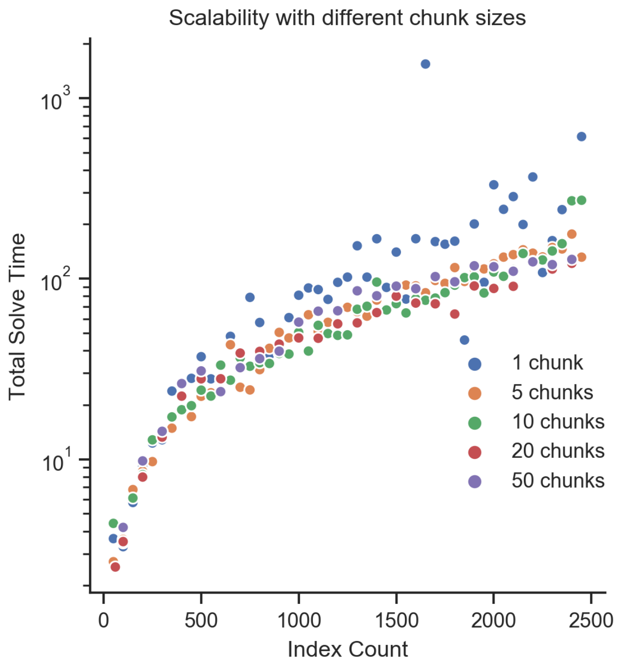

Figure 5: Execution Times of Basic Solution and with Chunking *(lower is better)*

chunks has not always a positive impact on the total execution time of our linear program. The chunking solution has a lower scattering than the basic solution. In each execution, some chunks could be solved faster than the mean and some other chunks need more time. The long and short solve times of single chunks balance each other and chunking lead to lower variations. As described in Section 2.4 chunking might cause that the final solution is not the optimal solution of the solved problem. Table 4 shows the worsening of the found solution to the optimal solution. The lower the chunk count is, the higher is the mean and the maximum of the worsening. With a lower chunk count the possibility that an index of the optimal solution is not a winner of the related chunk is higher. The more chunks the more indexes get on to the final round.

# chunks	mean growth	max growth
5	0.49 %	0.92 %
10	0.34 %	0.65 %
20	0.20 %	0.65 %
50	0.07 %	0.38 %

Table 4: Total costs growth with chunking compared to optimal solution in %

Chunking reduces the total solve time and fewer outliers with very long execution times occur. The worsening of the calculated solution is not high. The total workload costs growth is always lower than 1 %.

4.5 Multi-Index Configuration

In this section, we are evaluating the potential execution time overhead, which might get introduced by the multi-index configuration extension. Therefore, we compare the

# indexes	time overhead
10	533 %
50	120 %
100	-48 %
200	93 %
500	50 %
1000	24 %

Table 5: Percentual time execution time of the multi-index configuration extension compared to the basic approach

execution times of the extension with the basic approach. For both implementations, we tested multiple settings. The number of indexes defines a setting and is one of 10, 50, 100, 200, 500, or 1000. Independent of the setting, we assume 100 memory storage units and 100 queries. Figure 6 shows the execution time for both settings in comparison.

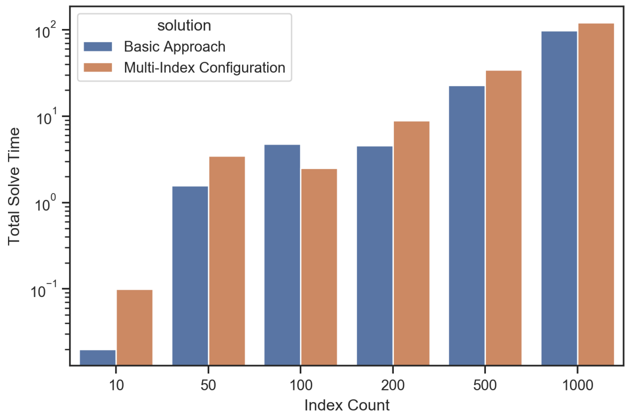

Figure 6: Execution times of basic approach and multi-index configuration extension in comparison *(lower is better)*

Besides the abnormality at 100 indexes, the multi-index configuration extension indeed has an execution time overhead. However, the percentage difference decreases with the number of indexes. Table 5 shows the actual overhead in percent.

4.6 Stochastic Workloads

In enterprise applications we have different use cases which produces different workloads for database systems. Each use case has different requirements. The output of some workloads is needed within a defined time, so we could set a maximum execution time as upper barrier for a workload. In other use cases the different workload costs should be robust without major deviations. Therefore we minimize the variance between the costs of different workloads. In other use cases we do not need robust workloads and the minimization of the total costs is the best decision for database systems. In this section we compare different objective functions. The evaluated index selection problem has 20 indexes, 20 queries, 30 available memory and 4 different workloads. The 4 different workloads W occur with different workload frequencies:

$$95 \times W1 \quad 19 \times W2 \quad 45 \times W3 \quad 7 \times W4 \tag{18}$$

We solve this problem with the following different optimization strategies: minimize total costs (tc), minimize the worst case (l) and minimize the variance (mv). For the 4th solution we have combined the three different optimization strategies with penalty factors:

$$minimize \quad 100 \cdot tc + 100 \cdot l + 1 \cdot mv \tag{19}$$

Figure 7 shows the different costs per workload for the 4 introduced strategies. Each bar shows the costs of one workload.

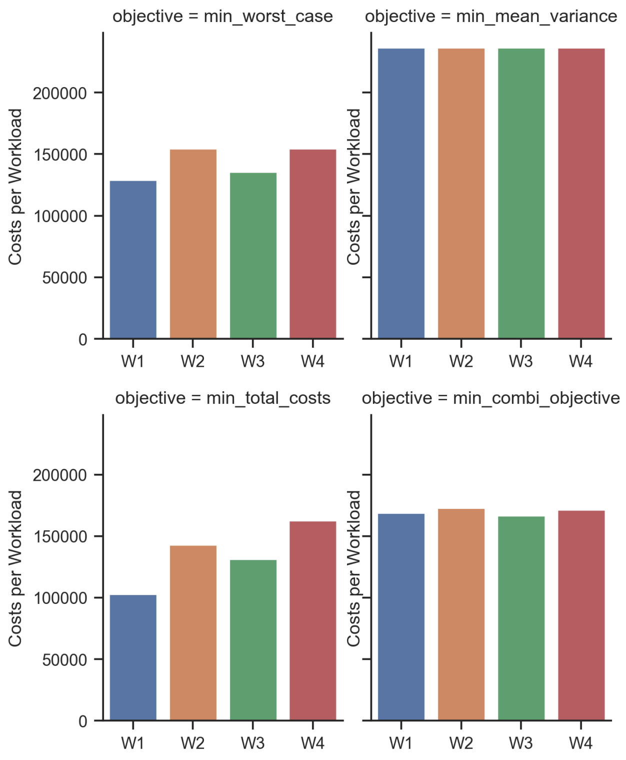

Figure 7: Workload costs with different optimization strategies: 1. Minimize worst costs, 2. minimize mean variance, 3. minimize total costs 4. combined approach *(lower is better)*

On the top left the worst case is minimized. The costs per workload vary between 129 017 and 154 602. On the bottom left the costs per workload for the minimization of the total costs are shown. The costs for each workload range from 102 907 to 162 762. The deviation between the different workloads are higher than the workload costs optimized with an upper barrier. The minimization of the mean variance objective (top right) harmonizes the four single workload costs. Each bar seems to have the same height. The cost per workload is between 236 571 and 236 611. Thus, the costs for each workload are significantly higher than any other approach. The combination depicted on the bottom right shows a much smaller deviation than the the minimization of the worst case and the total costs. The costs per workload

optimization strategy	worst case costs	cost variance	total costs
worst case	154 602	8 518 510 000	22 369 100
variance	236 611	6 188	39 272 300
total costs	162 762	34 962 000 000	19 539 400
combination	172 912	275 269 000	28 026 800

Table 6: Optimization strategies with their resulting performance metrics: worst workload costs, mean variance of the workload costs and the total costs

are between 166 480 and 172 912 if we optimize with the combination.

An index selection calculated with the mean variance minimization strategy leads to a fair workload cost distribution. However, this approach is increases the costs of all workloads. The problem is that workloads with lower costs than the mean workload costs worsen the value of the objective function. A second problem is that the calculation of the mean variance increases the execution time of our linear algorithm in some cases. The total solve time of the mean variance optimization was 248 although the solved linear problem seems to be small. However, the total solve time of the combined optimization strategy was only 0.78.

Table 6 displays the metrics of each optimization approach. The optimal total costs are 19 539 400. The worst case optimization lead to a growth by 14.5 %. The mean variance optimization has by far the highest total costs. On the other hand the variance is optimal. The worst case optimization has a worse variance by 6 orders of magnitude and the total costs minimization 7 orders of magnitude. If we optimize the worst case, we get $W4$ as the worst workload with costs of 154 602. With the total costs optimization the worst case is only 5 % higher. The combination is also only 11 % higher than the minimal worst case but the mean variance approach worst case is 53 % higher.

The mean variance solution is a fair solution for all workloads, but for database systems is important to execute the workloads as fast as possible. This approach raises unnecessarily the execution time of workloads, and the total cost of ownership for database systems also increases.

Some workloads should be executed in a specific time frame because a user waits on the results. In this case an optimization of the worst case is helpful. Another opportunity is to add a constraint for this single workload to tweak the total cost optimization. However, the application needs to specify this requirement and inform the database system in some way. If it is not important that some workloads should be executed within a maximum cost range, the total cost optimization strategy is the best one, because it reduces the total cost of ownership of database systems.

The worst case optimization and the total costs optimization have similar performance indicators. Both have their specific advantages and are good optimization strategies for the index selection problem.

5. FUTURE WORK

In section 2.6 we presented alternative objectives that use an upper bound to optimize the execution time of the worst workload or use the mean-variance to achieve robust execution times. Besides, we also showed combinations of different optimization objectives.

As a further extension, the mean-semivariance could be optimized. Optimizing the mean-variance will penalize execution times that are better than the average execution time. The better the execution times, the higher the penalty. However, short execution times are desirable from a database perspective. Consequently, by optimizing mean-semivariance, only the execution times that are higher than the average execution time are penalized.

In this work, the created LP models were evaluated only with randomly generated values. The models should be evaluated with data from real database scenarios to obtain more information on the quality and practical applicability of the models created. For this purpose, the created AMPL programs can be executed with real database benchmark results (e.g., data of the TPC-DS benchmark). A database that supports what-if optimizer calls should be used to anticipate performance improvements of individual indexes, which also serve as input data for the models. If the created models also show promising results with real benchmark data as input, the index selection approach should be evaluated with the selected real database. Thus, the workload should be executed without indexes and with the indexes suggested by our index selection approach. Both the execution time and memory consumption of the different workload executions are then to be measured.

6. CONCLUSION

In this work, we defined a basic index selection problem and different variants of it and presented linear programming models as solutions.

The basic approach makes decisions for each index individually and only takes one workload consisting of a set of queries and their frequencies into account.

In the chunking variant, we divided the overall index selection problem into multiple small sub-problems, which are solved individually. The selected indexes of these sub-problems are then put together and the best selection among these candidates will be determined in a final step.

For the multi-index configuration variant, the granularity of the possible options was changed from the index level to the index configuration level, where each configuration represents a combination of a maximum of two indexes.

The stochastic workload variant takes multiple workloads into account. Based on multiple previous workloads of the past, the LP model learns from these workloads and calculates an improved solution. For this variant, different objectives were used to minimize (1) the total workload costs, (2) the worst workload costs, (3) the workloads' mean variance and (4) a combination of the first three objectives.

In the transition costs variant, we addressed the fact that indexes potentially have to be created and removed when the index configuration changes in a DBMS. Therefore, the objective is extended with total creation and total removal costs.

In the last variant, chunking, multi-index configurations, stochastic workloads and transition costs were combined in one solution.

We evaluated the basic approach and its variants in different quality characteristics. For the basic approach, we evaluated the problem-solving execution time, which scales exponentially with the number of indexes.

For the chunking variant, we compared the execution times of different scenarios with the execution times of the basic approach. The results show that chunking reduces the total solve time and that particularly large index selection scenarios can be solved where the basic approach does not finish in a meaningful time. The basic approach often provides a better solution, i.e., a solution with smaller workload costs. However, the total workload costs growth of the chunking variant is always lower than 1%.

For the multi-index variant, we evaluated the potential execution time overhead compared to the basic approach. The results show that the overhead is significantly high in small scenarios but decreases with an increasing number of indexes.

For the stochastic workload variant, the workload costs of the four different objectives were evaluated. The results show that optimizing the robustness, i.e. the workloads' mean variance, significantly worsens the overall workload costs. Minimizing the total workload costs provides the most sufficient results.

7. REFERENCES

[1] A. O. inc. Home - amplampl — streamlined modeling for real optimization, 2020. [Online; accessed 7-August-2020].

[2] M. Kormilitsin, R. Chirkova, Y. Fathi, and M. Stallmann. View and index selection for query-performance improvement: Algorithms, heuristics and complexity. 2(1):1329–1330, 2008.

[3] D. M. G. Robert Fourer and B. W. Kernighan. Ampl reference manual, 2003. [Online; accessed 7-August-2020].

[4] R. Schlosser. Slides (2nd lecture) of the master's course Data-Driven Decision-Making in Enterprise Applications. Hasso Plattner Institute, Potsdam, Summer Term 2020.

[5] K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: Modeling, analysis, and applications. 2(1):1234–1245, 2009.

[6] Wikipedia contributors. Ampl — Wikipedia, the free encyclopedia, 2020. [Online; accessed 7-August-2020].

[7] H. Williams. *Model Building in Mathematical Programming*, page 8. Wiley, 5th edition, 2013.

APPENDIX

Notation Table

parameters	C	number of index configurations
	I	number of indexes
	M	index memory budget
	Q	number of queries
	W	number of workloads
	a	maximum workload costs penalty factor
	b	mean-variance penalty factor
	$d_{c,i}$	binary parameter whether configuration $c = 0, ..., C$ contains the indexes $i = 1, ..., I$
	f_q	frequency of query $q = 1, ..., Q$
	$f_{w,q}$	frequency of query $q = 1, ..., Q$ in workload $w = 1, ..., W$
	k_w	frequency of workload $w = 1, ..., W$
	m_i	memory consumption of index $i = 1, ..., I$
	s_i	speedup of index $i = 1, ..., I$ in contrast to no index being used
	$t_{q,i}$	execution time of query $q = 1, ..., Q$ using index $i = 0, ..., I$; $i = 0$ indicates that no index is used
	mk_i	creation costs of the index $i = 1, ..., I$
	rm_i	removal costs of the index $i = 1, ..., I$
variables	T	total execution time of all workloads
	V	mean-variance
	MK	total creation costs
	RM	total removal costs
	l	maximum workload costs
	g_w	execution time of a workload $w = 1, ..., W$
	$u_{q,c}$	binary variable whether configuration $c = 0, ..., I$ is used for query $q = 1, ..., Q$; $c = 0$ represents an empty configuration with no indexes
	$u_{q,i}$	binary variable whether index $i = 0, ..., I$ is used for query $q = 1, ..., Q$; $i = 0$ indicates that no index is used by query q
	v_i	binary variable whether index $i = 1, .., I$ is used for at least one query
	v_{i^*}	binary variable whether index $i = 0, ..., I$ was used previously used for at least one query and thus is already created

Table 7: Parameters and decision variables of the used LP models.