Efficient Scalable Multi-Attribute Index Selection Using Recursive Strategies

Rainer Schlosser, Jan Kossmann, Martin Boissier
Hasso Plattner Institute, University of Potsdam, Germany
{firstname.lastname}@hpi.de

***Abstract*—An efficient selection of indexes is indispensable for database performance. For large problem instances with hundreds of tables, existing approaches are not suitable: They either exhibit prohibitive runtimes or yield far from optimal index configurations by strongly limiting the set of index candidates or not handling index interaction explicitly. We introduce a novel recursive strategy that does not exclude index candidates in advance and effectively accounts for index interaction. Using large real-world workloads, we demonstrate the applicability of our approach. Further, we evaluate our solution end to end with a commercial database system using a reproducible setup. We show that our solutions are near-optimal for small index selection problems. For larger problems, our strategy outperforms state-of-the-art approaches in both scalability and solution quality.**

Publication: This paper has been published in the proceedings of the 35th IEEE International Conference on Data Engineering 2019. The final publication is available at IEEE Xplore via http://dx.doi.org/10.1109/ICDE.2019.00113.

I. The Index Selection Problem

Developments in hard- and software have shifted the break-even point for sequential scans and secondary index scans in terms of performance. In [1], Kester et al. have investigated the impact of large main memory capacities and modern CPU cache hierarchies for both access paths. They show that even though sequential scans have overall become more efficient, secondary indexes are still necessary to achieve the best performance in many settings.

Besides the performance aspects, indexes often consume a major share of the available main memory, a typically scarce resource [2], [3]. The sheer size of indexes and their potential performance gains emphasize the need for mechanisms to efficiently balance memory consumption and performance.

But determining the most beneficial set of indexes is a complex and time-consuming problem. The number of theoretical index candidates offered by large enterprise systems is unmanageable [4]. Large tables of enterprise software can have hundreds of attributes [5], workloads often include thousands of queries, and, hence, the number of accessed attribute combinations is enormous.

The selection process is typically conducted by Database Administrators (DBAs) who have to rely on their intuition and index advisors with often sub-optimal heuristics to find acceptable solutions. Chaudhuri et al. [6] even stated that expenses for DBAs became the key factor in the Total Cost of Ownership (TCO) with increasing complexity of databases and their queries. Workloads which change over time complicate the selection process further.

Moreover, increasingly large cloud database deployments require database vendors to minimize database administration efforts. Assisting DBAs [7] with these complex tasks has the potential to lead to more beneficial index configurations (and improved system throughput) and to a lower TCO by both reducing (i) the time taken for manual index tuning and (ii) the main memory footprint due to removed superfluous indexes.

Index selection (or index tuning) belongs to the class of NP-hard problems [8]. Especially large real-world problem instances exhibit characteristics that most existing selection approaches cannot handle: (i) thousands of attributes to consider, (ii) workloads consisting of many thousand query templates, (iii) solution constraints such as memory budgets, and (iv) balancing reconfiguration costs with performance gains. The approach we are presenting in this paper is capable of handling large systems efficiently, adheres to given memory budgets, and incorporates reconfiguration costs.

Looking at existing selection approaches, we see two important aspects that render them unsuitable in real-world settings: (i) index interaction and (ii) large problem dimensionality. Most approaches do not explicitly take the effects of index interactions into account or prune potential index candidates too early [9]–[11], thereby degrading the solution's quality. We seek to address both aspects.

A. Index Interaction & Index Candidates

Typical relational database systems as used by enterprise applications contain a large number of multi-attribute indexes [2]. Multi-attribute indexes offer a good benefit-cost-ratio compared to single-attribute indexes. Hence, it is desirable for index selection systems to consider multi-attribute indexes.

The determination of multi-attribute indexes is computationally challenging as the number of candidate indexes, i.e., the number of used attribute *combinations*, typically exceeds the number of attributes by orders of magnitude. Hence, in general, optimal solutions cannot be derived anymore, and it is impossible to evaluate all candidate combinations. Finding the best selection of indexes out of a large set of potential indexes is highly challenging as the presence of index elements mutually affects their impact and the overall performance. This interplay is called index interaction (IIA) and defined by Schnaitter et al. as follows: "an index a interacts with an index b if the benefit of a is affected by the presence of b and vice-versa" [12].

Index interaction increases the complexity of index selection significantly as each index may influence the impact of all other indexes. Nonetheless, the integration of index interactions can significantly improve the solution quality.

Most existing approaches tackle index selection by having a two-step process. First, the number of potential index candidates is limited according to a certain heuristic. Second, this index candidate set is used by the actual index selection algorithm to compute a final index configuration. The main reason for the pruning of index candidates during the candidate selection is runtime performance. For approaches that exhaustively enumerate candidates, the solution space of possible configurations and their evaluation using what-if optimizers makes it unfeasible to consider all candidates as soon as the system in focus is sufficiently large.

The problem with an upfront candidate limitation is that it might exclude candidates that later turn out to be useful. Especially when accounting for multi-attribute indexes and the effects of index interaction, simple heuristics (e.g., using the top n single-attribute indexes for multi-attribute selection [13]) yield sub-optimal results. To our best knowledge, all existing approaches that determine multi-attribute index configurations use strong candidate pruning and consider fixed candidate sets.

Instead of the described two-step approach, our solution is a one-step approach. The key idea is to prune index candidates as late as possible and to construct index selections in an iterative way. The power of recursion allows (i) to deal with the enormous size of index combinations and (ii) to incorporate index interaction in each construction step. Further, in each step, new indexes are chosen or existing ones extended to consistently maximize the additional performance per additional memory. This technique has advantages compared to randomized shuffling or substitution heuristics used by existing state-of-the-art approaches.

To obtain accurate cost estimates for a configuration, we use the what-if capabilities of modern query optimizers, similar to other approaches [13]–[15]. Unfortunately, what-if calls are the major bottleneck for most index selection approaches (cf. [16]). Hence, a major constraint is to limit the number of what-if optimizer calls. And even though our approach does not limit the index candidate set, we are able to decrease the number of what-if calls using caching and by exploiting workload context information (cf. Section II-C).

B. Contributions

The proposed index selection solution is based on a general solution principle which allows for efficient computation. To demonstrate the performance and the applicability of our approach, we present extensive evaluations, consisting of simulated, reproducible examples, and a real-world workload. Evaluations show that our approach outperforms various heuristics and state-of-the-art approaches.

The results indicate both an improvement in the quality of the selected indexes and a greatly reduced search time. We focus on a comparison with the state-of-the-art approach CoPhy [14], which outperforms index advisors of commercial database systems resembling techniques presented in [10] and [17]. Compared to CoPhy, we show that our solution quality is higher for large problem instances, while at the same time being significantly faster.

Current state-of-the-art approaches spend almost half of the computation time for solving and the other half for what-if calls to the query optimizer [14]. Our solution improves both main contributors to the overall runtime for large problems: We calculate index configurations orders of magnitude faster while executing fewer what-if calls. Our main contributions can be summarized as follows:

- We propose a novel workload-driven index selection approach that builds on a recursive mechanism and effectively accounts for index interaction.
- Our approach is scalable and efficient, making it applicable even for large-scale problems.
- Using reproducible examples, we demonstrate the scalability and the end-to-end performance of our solution with a commercial database system compared to state-of-the-art techniques.
- We also demonstrate the improved performance of our approach over established techniques for real-world scenarios with the workload of a productive enterprise system.

The remainder of this paper is structured as follows. In Section II, we derive our index selection approach and discuss its conceptual differences to other state-of-the-art approaches. In Section III, we use a reproducible scalable setting to study the scalability and quality of our approach compared to optimal selections provided by CoPhy. In Section IV, we study the performance of our approach against other state-of-the-art tools using real-life workloads in a commercial DBMS. In Section V, we summarize our evaluation results and discuss the advantages of our approach. In Section VI, we present important related work in the area of index selection. Final conclusions and future work are given in Section VII.

II. Multi-Attribute Index Selection

In this section, we present our approach and discuss its conceptual differences compared to other approaches.

A. Problem and Model Description

We consider a system with N attributes. The problem is to choose secondary indexes for a workload, consisting of Q queries, such that the overall performance is maximized, e.g., by minimizing the execution time, I/O traffic, or the amount of transferred memory. Each of the Q queries is characterized by a set of attributes $q_j \subseteq \{1, ..., N\}$, $j = 1, ..., Q$, that are accessed during query evaluation. Note, a notation table is given in Appendix A.

A (multi-attribute) index k with K attributes is characterized by an *ordered* set of attributes $k = \{i_1, ..., i_K\}$, where $i_u \in \{1, ..., N\}$, $u = 1, ..., K$. Further, to describe *index candidates*, we use a set of indexes denoted by I,

$$I \overset{e.g.}{:=} \{\{16, 3\}, \{1, 6, 2\}, \{5, 6\}\}$$

By the subset $I^* \subseteq I$, we denote an *index selection*. W.l.o.g., using binary variables x_k, we indicate whether a candidate index $k \in I$ is part of the selection I^*, i.e.,

$$I^*(I, \vec{x}) := \bigcup_{k \in I : x_k = 1} \{k\}$$

The costs of a query q_j, $j = 1, ..., Q$, are denoted by values $f_j(I^*)$, which depend on the selection I^*. Usually $f_j(I^*)$ is determined by what-if optimizer calls. The functions f_j assign costs to sets of indexes (which includes one-dimensional sets). Note, a query q_j can be of various type, such as a selection, join, insert, update, etc. or a combination of the above.

The total workload costs F are defined by the sum of query costs f_j of all queries q_j, multiplied by their number of occurrences denoted by b_j, $j = 1, ..., Q$,

$$F(I^*) := \sum_{j=1,...,Q} b_j \cdot f_j(I^*) \tag{1}$$

Further, we assume that the memory consumed by the selected indexes is not allowed to exceed a certain budget A. The necessary memory for a (multi-attribute) index k, $k \in I^*$, is denoted by p_k and can be arbitrarily defined. The total memory used by I^* amounts to

$$P(I^*) := \sum_{k \in I^*} p_k \tag{2}$$

Further, we allow for reconfiguration costs. By $R(I^*, \bar{I}^*)$, we denote the costs (arbitrarily defined) for changing an existing index selection $\bar{I}^*$ to a new selection I^* (i.e., create new indexes $I^* \setminus \bar{I}^*$ and delete unnecessary ones $\bar{I}^* \setminus I^*$).

Finally, (given an arbitrary but fixed current index selection $\bar{I}^*$) the multi-attribute index selection problem can be generally defined by

$$\underset{x_k \in \{0,1\}, k \in I}{\text{minimize}} \quad F(I^*(I, \vec{x})) + R(I^*(I, \vec{x}), \bar{I}^*) \tag{3}$$

$$\text{subject to} \quad P(I^*(I, \vec{x})) \leq A \tag{4}$$

Note, in problem (3) - (4) *both* the variables $\vec{x}$ and the index candidate set I are crucial and have to be optimized.

B. CoPhy's LP Approach

In this subsection, we consider an integer linear programming approach to solve index selection problems. The approach resembles the concept of CoPhy [14]. For ease of simplicity, w.l.o.g., we do not consider updates and reconfiguration costs.

CoPhy's model assumes that for each query at most one index (per table) is applied and that scan costs are additive separable for different tables. Moreover, it is assumed that the scan costs of a query when using a specific index are not affected by the presence of other indexes. Hence, if index k is applied to query q_j, CoPhy uses scan costs $f_j(k)$, $k \in I \cup 0$, where 0 describes the option that no index is applied to q_j.

Further, by $I_j \subseteq I$ we denote the set of index candidates out of I that are applicable to query q_j, $j = 1, ..., Q$. W.l.o.g., we assume that queries operate only on one table. For a given candidate set I, the essence of CoPhy's index selection LP approach can be written as:

$$\underset{\substack{z_{jk}, x_i \in \{0,1\}, i \in I, \\ j=1,...,Q, k \in I_j \cup 0}}{minimize} \quad \sum_{j=1,...,Q, k \in I_j \cup 0} b_j \cdot f_j(k) \cdot z_{jk} \tag{5}$$

$$\text{subject to} \quad \sum_{k \in I_j \cup 0} z_{jk} = 1 \quad \forall j = 1, ..., Q \tag{6}$$

$$z_{jk} \leq x_k \quad \forall j = 1, ..., Q, k \in I_j \tag{7}$$

$$\sum_{i \in I} p_i \cdot x_i \leq A \tag{8}$$

The family of constraints (6) guarantees that at most one index k is used for query q_j. The constraints (7) serve to identify which indexes k are used at all. Finally, constraint (8) ensures that the memory budget A is not exceeded. CoPhy takes IIA into account as the LP allows for all index combinations I^* out of I.

The complexity of the problem described by (5) - (8) is particularly characterized by the number of variables and constraints. Note, using $I_j \subseteq I$ instead of I allows to reduce the number of variables and constraints (from approx. $Q \cdot I$, cf. variables z_{jk} and constraint (7), respectively) to

$$|I| + Q \cdot \bar{I}_q \quad and \quad Q + Q \cdot \bar{I}_q + 1$$

where $\bar{I}_q := 1/Q \cdot \sum_{j=1,...,Q} |I_j|$ is the average number of relevant index candidates per query.

If an index k is only applicable to a query q_j under the condition that the first attribute of k denoted by $l(k)$ is part of q_j, then the relevant sets I_j can be defined as, $j = 1, .., Q$,

$$I_j := \bigcup_{k \in I : q_j \cap l(k) \neq \emptyset} \{k\}$$

In this case, the number of variables and constraints can be approximated as follows. On average, we have $|I|/N$ indexes that start with a specific attribute i, $i = 1, ..., N$. Further, let $\bar{q} := 1/Q \cdot \sum_{j=1,...,Q} |q_j|$ denote the average number of attributes occurring in queries. Hence, the average number of candidates applicable to a query is $\bar{I}_q \approx \bar{q} \cdot |I|/N$. Note, that the number of qualifying indexes can also be larger, as I typically contains indexes that start with attributes that overproportional occur in a workload's queries. Finally, the number of variables and constraints is approximately

$$Q \cdot \bar{I}_q \approx Q \cdot \bar{q} \cdot |I|/N \tag{9}$$

Hence, as expected, the problem complexity strongly increases in the number of queries Q and the number of candidate indexes $|I|$. As the linear programming formulations, cf. (5) - (8), require all cost coefficients $f_j(k)$, the number of what-if optimizer calls can also be estimated by $Q \cdot \bar{I}_q$, cf. (9). Note, that what-if cost estimations can be efficiently and accurately derived, e.g., via INUM [16].

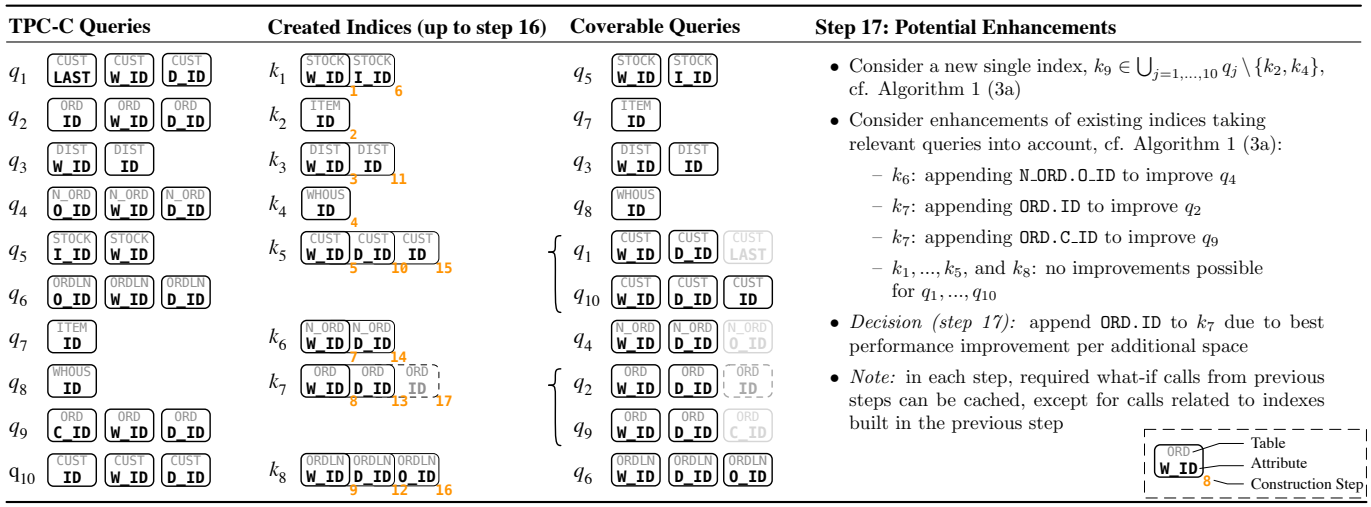

Fig. 1. Illustration of Algorithm 1 for index selection for TPC-C: construction steps, applicability of indexes to different queries, and consideration of relevant index enhancements.

C. Our Index Selection Approach

In the following, we define our approach for multi-attribute index selection. Many index selection concepts are characterized by a two-step approach of (i) defining a well-sized index candidate set and (ii) deriving an optimized selection for a fixed memory budget (using heuristics or linear programming techniques). Instead of these two steps, we use a *constructive* approach to compute index selections which is based on a recursive solution with increasing budgets.

Our approach uses a step-wise construction of an index selection following a recursive *context-based* selection mechanism. Algorithm 1 recursively adds attributes to a set of indexes. Attributes can either extend the current set of indexes I as a new single-attribute index or they can be appended to the "end" of existing indexes out of I (cf. morphing). The decision which attribute is added in which way is determined by comparing the additional performance, cf. $F(I) + R(I)$, to the associated additional utilized memory, cf. $P(I)$, see (2). Figure 1 illustrates the procedure for a TPC-C example[1].

In a nutshell, the approach seeks to approximate the efficient frontier (regarding performance and size) associated to the multi-attribute index selection problem.

Algorithm 1. *To determine multi-attribute index selections, we use the following recursive construction:*

1) *Let* $I := \emptyset$.
2) *Determine the single-attribute index* $\{i\}$, $i = 1, ..., N$, *that minimizes the ratio between costs and the associated memory consumption, i.e.,* $(F(\emptyset) + R(\emptyset, \bar{I}) - F(\{i\}) - R(\{i\}, \bar{I}))/p_{\{i\}}$. *Extend the current index set* I *by the best single-attribute index* $\{i^*\}$: $I := I \cup \{i^*\}$.
3) *Consider the current set* I *with a calculated cost of* $F(I) + R(I, \bar{I})$ *and utilized memory* $P(I)$. *For each elementary index element* $\{i\}$, $i = 1, ..., N$, *and indexes* $k \in I$ *consider the following two types of construction steps:*
   a) *Add* $\{i\}$ *as new single-attribute index, if* $I \cap \{i\} = \emptyset$.
   b) *Append* $\{i\}$ *at the end of index* k.

   All potential attachments lead to a new index set $\tilde{I}$ *with* $F(\tilde{I}) + R(\tilde{I}, \bar{I})$ *and* $P(\tilde{I})$. *Choose the new index set with the best ratio of cost reduction* $F(I) + R(I, \bar{I}) - F(\tilde{I}) - R(\tilde{I}, \bar{I})$ *and its required additional memory* $P(\tilde{I}) - P(I)$. *Save the chosen construction step and let* $I := \tilde{I}^*$.
4) *Repeat Step (3) until either* $P(I)$ *exceeds the given memory budget, a predefined maximum number of construction steps is reached, or if no further improvement can be made.*

Step (3) of the presented algorithm allows for time-saving optimizations. If the elementary enhancements $i = 1, ..., N$ are considered for indexes $k \in I$ it is not necessary to *fully* recalculate costs $F(\tilde{I})$ using the what-if optimizer. It is sufficient to only recalculate costs for queries that might be affected by the potential attachment, cf. Step (3a or 3b). The costs of most queries do not change.

Algorithm 1's applicability is independent of specific values of F, R, P, and f, respectively. The algorithm can be generally applied and is independent of particular cost models or optimizers. The only requirements are that query costs need to be quantified and relative performance improvements of index attachments must be comparable, cf. Step (3).

Note, the set of index candidates is not restricted in advance. There is also no maximum number of attributes used in a multi-attribute index. Algorithm 1 provides a series of construction steps, which recursively and thus efficiently determine growing sets of indexes I and their associated memory consumption $P(I)$. The recursive selection allows to take cannibalization effects between different indexes into account as the value of an index candidate is measured by its additional performance

[1]We aggregated the distinct conjunctive selections of all TPC-C transactions: https://git.io/pytpcc

improvement in the *presence* of other earlier selected indexes (cf. IIA).

The following remark suggests several ways how Algorithm 1 can be further improved and extended.

Remark 1. *Extensions and generalizations of Algorithm 1:*

1) *Algorithm 1 can be accelerated, e.g., by just considering the n-best single-attribute indexes in the construction steps, cf. Step 2.*
2) *It is possible that indexes that have been built in earlier steps are not used anymore as new indexes (requiring higher memory budgets) allow for more efficient scans. In such cases, Algorithm 1 can be optimized by excluding unnecessary indexes from the current index set.*
3) *To be able to identify different indexes that have the same leading attributes, the estimated impact of missed (e.g., second best) opportunities to extend indexes can be stored and potentially used in a later step. Note, the performance impact might have to be re-estimated.*
4) *In case what-if calls are inexpensive or sufficiently accurate cost models are present, the algorithm can be generalized to consider not only single attributes but also certain pairs of attributes to build or extend indexes.*

Finally, we define our index selection heuristic. In addition, we define three heuristic rules as well as two index selection approaches resembling concepts used in [9] and [13].

Definition 1. *Our approach and state-of-the-art concepts:*

(H1) := Pick indexes of a given index candidate set I with the most used attributes, measured by the number of occurrences $g_i := \sum_{j=1,...,Q, i \in q_j} b_j$, $i = 1, ..., N$, in queries as long as the given memory budget A is not exceeded.

(H2) := Pick indexes of a given index candidate set I with smallest selectivity $s_i := 1/d_i$, $i = 1, ..., N$, as long as the budget A is not exceeded (d_i the number of distinct values).

(H3) := Pick indexes of a given index candidate set I with the smallest ratio of selectivity and number of occurrences, i.e., s_i/g_i, $i = 1, ..., N$, as long as A is not exceeded.

(H4) := Pick indexes of a given index candidate set I with the best performance as long as A is not exceeded; exclude candidates that are not efficient (concerning performance and size) for at least one query (skyline candidates), cf. [13].

(H5) := Pick indexes of a given index candidate set I with the best performance per size ratio as long as A is not exceeded, cf. starting solution of [9].

(H6) := Apply the series of construction steps, cf. Algorithm 1, as long as A is not exceeded.

D. Discussion of Different Selection Approaches

The index selection problem cannot be solved exhaustively as the number of combinations is enormous and the number of what-if calls that can be processed to estimate each cost improvement due to a specific index is limited.

Hence, the set of index candidates to be considered has to be limited. As a limited candidate set can easily shrink the solution quality the index candidate set has to be (i) either very large or (ii) well defined. While (i) makes it difficult to find an optimal selection, it is also not easy to solve (ii), as it resembles the index selection problem itself.

Existing approaches work as follows: First, (large sets of) index candidates are chosen based on simple metrics (e.g., access frequency, selectivity, or size) and individually evaluated using what-if optimizer calls for all relevant queries. Second, for a given memory budget the final index selection is picked according to a certain mechanism.

In [13], final indexes are selected greedily based on their individually measured performance. While this procedure scales, IIA is not explicitly considered. Selected indexes are likely to cannibalize their impact.

CoPhy's linear programming approach allows identifying *optimal* index selections while taking also IIA into account. However, due to the complexity of the LP, the applicability of the approach is limited to small sets of index candidates.

In [9], indexes are initially picked greedily based on their individually measured performance/size ratio (starting solution). To account for IIA and to increase performance index elements are *randomly* substituted. Thereby, the approach is applicable to large candidate sets. However, as the starting solution is often far away from optimal and the shuffling is not targeted, it can take a long time to obtain optimized results, particularly, when candidate sets are large.

On first sight, our algorithm appears similar to greedy approaches like [9]. However, our approach has crucial differences. We apply greedy selections in a recursive subsequent way such that IIA, i.e., the presence of other (earlier selected) indexes, is taken into account.

By comparing cost estimations, it can be determined for which queries a potential new index (when added to a current index set) would be applied, cf. (1), and hence, by how much the current total performance would be improved. In each step of Algorithm 1, an index is chosen such that the "additional performance" per "additional required memory" is consistently optimized. Hence, in contrast to [9], we address IIA in a *targeted* way.

The combinations and variety of indexes that can be composed by Algorithm 1 are not restricted in advance (as when starting with certain sets of index candidates). Yet, the number of necessary what-if calls is comparably small as we also prune index candidates: Due to the use of recursive incremental index extensions/alterations, in each step, only small subsets of potential new indexes are considered and few what-if calls are performed (cf. Section II-C).

Most importantly, the "direction" in which indexes are considered and selected follows the "additional performance per additional memory" criteria which reflects IIA.

To sum up, our approach (i) effectively includes IIA, (ii) performs a small number of what-if calls, and (iii) quickly identifies index selections for given budgets without using a solver - even for large problems.

III. Reproducible Evaluations

In this section, we use a reproducible scalable benchmark scenario to compare the runtime (Section III-A) and the quality (Section III-B) of our index selection approach, cf. Section II-C, to CoPhy's solution approach, cf. Section II-B.

A comparison of all presented index selection concepts including (H4) and (H5), cf. Definition 1, for real-life workloads as well as end-to-end evaluations follows in Section IV.

Example 1. *(Illustrating Reproducible Scalable Example)*

In order to compare our approach to CoPhy's solution, we consider the following setting.

(i) CoPhy applies at most one index per query. To obtain comparable results, we express query costs in the simplified "one index only" setting, i.e., given a selection I^* the costs of a query j are determined by:

$$f_j(I^*) := \min_{k \in I^* \cup 0} f_j(k) = \min(f_j(0), \min_{k \in I^*} f_j(k))$$

(ii) In order to illustrate the applicability of our approach in a reproducible setting, we fill the what-if optimizer calls for $f_j(k)$, $k \in I^* \cup 0$, according to an exemplary cost model described in Appendix Section B.

(iii) Further, we consider a (randomized) synthetic workload setting with $T = 10$ tables. We consider different problem sizes with $Q_t = 50, ..., 5\,000$ queries and $N_t = 50$ attributes per table, $t = 1, ..., T$. For details, see Appendix Section C.

(iv) We compare our heuristic (H6), cf. Algorithm 1 and Definition 1, to CoPhy's approach making use of the following three heuristics to define scalable sets of index candidates I:

- *(H1-M) For each $m = 1, ..., 4$ select h index candidates of m attributes $\{i_1, ..., i_m\}$ that occur most frequently in queries throughout the workload, $i_u = 1, ..., N$, $u = 1, ..., m$, $\sum_{j=1,...,Q, \{i_1,...,i_m\} \in q_j} b_j$ (in desc. order).*
- *(H2-M) For each $m = 1, ..., 4$ select h index candidates of m attributes $\{i_1, ..., i_m\}$ that have the smallest combined selectivity $\prod_{u=1,...,m} s_{i_u}$ (in ascending order).*
- *(H3-M) For each $m = 1, ..., 4$ select h index candidates on tuples of m attributes with the best ratio of combined selectivity (cf. H2-M) and number of occurrences (cf. H1-M) in ascending order.*

For M index candidates, let $h := M/4$ for each $m = 1, ..., 4$.

A. Applicability, Scalability, and Runtimes

We use Example 1 to compare the applicability of our approach to CoPhy's solver-based approach. For details see our implementation framework[2].

For different problem sizes (characterized by the number of queries and attributes), Table I shows the runtimes (excluding what-if calls) of our strategy (H6) – implemented with a single-threaded C++ program – and CoPhy's approach (CPLEX 12.7.0.0, mipgap=0.05, 4 threads, via NEOS Solver) for different sizes of index candidate sets, cf. Example 1. The number of selected indexes is roughly $|I^*| \approx w \cdot \sum_t N_t = 100$. The budget $A(w)$ is defined by the share w of the total memory required for all single-attribute indexes, $0 \leq w \leq 1$,

$$A(w) := w \cdot \sum\nolimits_{k \in \{\{1\},...,\{N\}\}} p_k \tag{10}$$

TABLE I

Runtime comparison: Solving time of our strategy (H6) vs. CoPhy's approach (with 5% optimality gap and $|I|$ index candidates) for different problem sizes with IC_{max} potential indexes, $T = 10$ tables, $\sum_t N_t = 500$ attributes, budget $w = 0.2$; without time for what-if calls; Example 1.

# Queries $\sum_t Q_t$	IC_{max} $\lvert \boldsymbol{I_{max}} \rvert$	# Candidates $\lvert I \rvert$	∅ Runtime CoPhy[3]	∅ Runtime (H6)
500	4 249	(100, 1K, IC_{max})	(0.35, 4.1, 19.7) s	0.276 s
1 000	7 504	(100, 1K, IC_{max})	(0.62, 7.3, 59.0) s	0.362 s
2 000	13 862	(100, 1K, 10 000)	(2.4, 470, DNF) s	0.587 s
5 000	29 111	(100, 1K, 10 000)	(5.4, DNF, DNF) s	1.121 s
10 000	54 622	(100, 1K, 10 000)	(6.1, DNF, DNF) s	2.163 s
20 000	97 550	(100, 1K, 10 000)	(15.3, DNF, DNF) s	5.552 s
50 000	194 065	(100, 1K, 10 000)	(16.3, DNF, DNF) s	12.230 s

The runtimes of (H6) were again collected on a consumer notebook with an Intel Core i5 and 16 GB of main memory. We observe that even for really large problem instances computations take only seconds and thus allow for basically immediate response times. To further accelerate the runtime of our strategy, Algorithm 1 could be parallelized.

We observe, that CoPhy's computation time significantly increases with the number of queries Q as well as the number of index candidates. The generation of candidate sets and calls to the what-if optimizer are not included. Note, the basic LP model, cf. (5) - (8), with $Q = 500$ queries and $|I| = 1\,000$ candidates typically has already more than 100 000 variables and 100 000 constraints.

Further, the number of combinations of indexes exponentially grows with the number of candidates (cf. [15]). Although solvers use pruning, they cannot circumvent that problem. Hence, for large problems the applicability of solver-based approaches is limited and, thus, the number of candidates has to be reduced to a small subset of potential indexes. Thereby, even though, the final selection of indexes is optimal (for the subset of index candidates), in general, the solution quality can be far from optimal.

Comparing the runtimes in Table I, we observe that our approach is still applicable if problems are large. The complexity of our algorithm is different: Due to the recursive nature of our approach, in each construction step, there is just a small number of possibilities which have to be evaluated. As current index selections remain, the number of possibilities in each step does not *multiply* – they just *add up*, leading to a dramatically lower complexity.

Moreover, our recursive approach does *not* require a particularly large number of cost estimations, e.g., what-if optimizer calls. Although in principle, a large number of index candidates can be constructed, not *all* of them are evaluated using what-if

[2]Reproducible framework for Example 1 for multi-attribute solutions, and re-implementation of CoPhy: https://git.io/ICDE19IndexSelection

[3]DNF: Did not finish within eight hours.

optimizer calls. The majority (e.g., often more than 50%) of optimizer calls occurs in the very first construction step, in which the scan costs of all potential single-attribute indexes $k = \{i\}$, $i = 1, ..., N$, have to be determined for all queries q_j, $j = 1, ..., Q$, i.e., we roughly have $\bar{q} \cdot Q$ what-if optimizer calls in the first step. In each of the following construction steps of Algorithm 1, only those queries have to be evaluated that could fully use a new potential (incrementally extended) multi-attribute index – otherwise the costs of a query do not change and have already been determined previously. Thus, in each construction step, just a small number of what-if calls has to be performed. Note, these numbers again just add up over all construction steps.

Whereas in our approach the number of what-if calls ($\approx 2 \cdot Q \cdot \bar{q}$) slowly increases with the number of necessary construction steps to reach the targeted memory budget, in CoPhy's approach the number of calls ($\approx Q \cdot \bar{q} \cdot |I|/N$, cf. (9)) is roughly linearly increasing in the number of index candidates $|I|$. Thus, our approach requires *fewer* cost estimations compared to CoPhy, especially if the number of candidates $|I|$ is large compared to the number of columns N.

Moreover, if *multiple* indexes per query are admissible the complexity of the index selection problem is further increased as the costs of a query cannot separately be derived for different indexes, cf. Example 1 (i). Instead, the costs of a query are context-based, i.e., they may depend on the entire selection I^*. Hence, Cophy's framework would have to be extended such that cost parameters $f_j(I)$ are available for all subsets I^*. In addition, the LP (5) - (8) is very likely to become nonlinear, and hence, might be hardly applicable anymore.

Remark 2. *Algorithm 1 is applicable in settings with multiple indexes per query. In each step, what-if calls are made to evaluate potential extension where the current state, i.e., the current selection I, is taken into account. The only difference to the "one index only setting" is that some what-if calls performed in earlier steps have to be refreshed as the underlying current state (and in turn, query costs) might have changed.*

B. Performance Comparison with CoPhy

Next, we compare the performance of CoPhy's approach with our strategy. To study to which extent CoPhy's performance depends on the *size* and *quality* of index candidate sets, we consider Example 1 for different parameters h, the number of index candidates.

Figure 2 illustrates combinations of performance and memory used for our and CoPhy's selection approach in case different heuristics are used to define index candidate sets. We observe that CoPhy's results are affected by the assortment of candidate sets. Depending on the targeted memory budgets, heuristics can perform differently. Our strategy, however, provides excellent results for *any* budget.

Figure 3 depicts the corresponding results in case CoPhy's selection approach uses different *sizes* of candidate sets according to (H1-M). As expected, CoPhy's results are significantly affected by the size of candidate sets. The smaller the set of candidates is, the higher is the chance that important indexes are missing. The larger the set of candidates, the higher the solve time becomes.

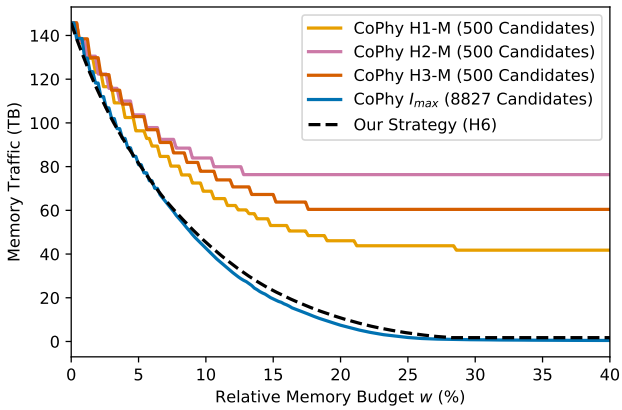

Fig. 2. Comparison of combinations of scan performance and associated relative memory budgets $A(w)$, $w \in [0, 0.4]$, for our strategy (H6) and CoPhy's concept with different sets of index candidates using (H1-M), (H2-M), (H3-M) with $|I| = 500$ and all candidates I_{max}; $N = 500$, $Q = 1\,000$; Example 1.

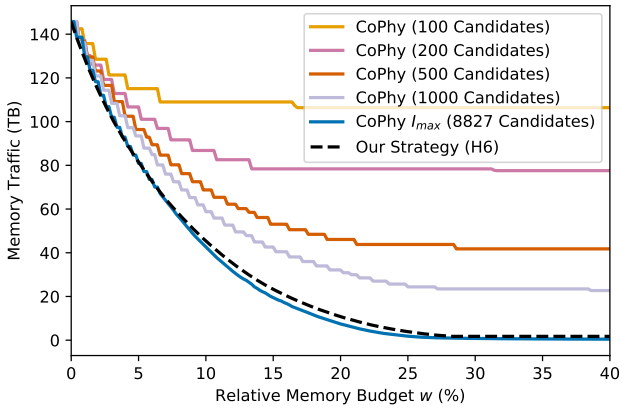

Fig. 3. Comparison of combinations of scan performance and associated relative memory budgets, cf. $A(w)$, for our strategy (H6) and CoPhy's concept with different sets of index candidates using (H1-M) and $|I| = 100, 1\,000, |I_{max}|$ candidates, $w \in [0, 0.4]$, $N = 500$, $Q = 1\,000$; Example 1.

CoPhy's solutions are *optimal* in case the *maximum* number of potential indexes (cf. I_{max}) is used as an exhaustive candidate set. The fact that our strategy (H6) performs close to optimal verifies the high quality of our solution. Recall, in terms of computation times, we clearly outperform solver-based approaches with large candidate sets which are necessary for solutions of high quality.

Finally, our solution can be used to replace as well as to *improve* traditional index selection approaches by *complementing* index candidate sets. If the indexes that our approach selects are used as additional index candidates, the problem complexity of solver-based approaches hardly increases while the solution quality may only improve.

IV. Enterprise Workloads and End-to-End Evaluations

In Section IV-A, we present benchmark results which are based on data and workloads from a productive enterprise system of a Fortune Global 500 company. In Section IV-B, we demonstrate that our solution's quality does not depend on the exemplary cost model used in Section II. We evaluate our approach with actual runtime costs measured with a commercial DBMS against all presented concepts including (H4) and (H5), cf. Definition 1.

A. Application to Enterprise Workloads

We evaluate our strategy for a real-world enterprise system. We extracted the largest 500 tables (by memory consumption) from the database system including the queries accessing these tables. These 500 tables consist of 4 204 relevant attributes. The tables had between ~350 000 and ~1.5 billion rows. 55 758 distinct queries (with $Q = 2\,271$ query templates) were executed and more than 50 million query executions took place during the recorded period. The workload can be characterized as mostly transactional with a majority of point-access queries but also contains few analytical queries (more information about a comparable system can be found in [18]).

Figure 4 shows the cost (calculated memory traffic) to process the aforementioned workload for varying memory budgets for our index selection strategy (H6) and CoPhy. In addition, the size of CoPhy's candidate sets is varied using (H1-M). The measurements reassure the results from Section III-A and Section III-B. Our approach clearly outperforms CoPhy's LP-based approach with limited candidate sets regarding the solution quality. The runtime of our approach amounts to approximately half a second whereas CoPhy with all 9 912 candidates needs several minutes.

The bad performance of the heuristics can be explained by the nature of real-world workloads. The aforementioned interaction between indexes plays an important role. Some attributes are often accessed together. Hence, an index on an attribute might degrade the performance of a potential index on another attribute. Rule-based heuristics like (H1) - (H3) cannot take these effects into account.

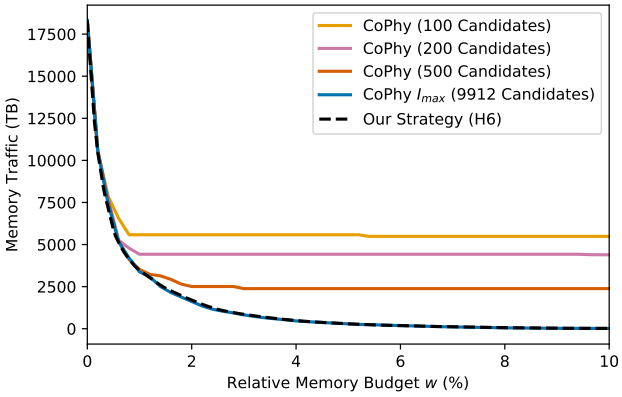

Fig. 4. ERP systems example: Comparison of combinations of scan performance and associated relative memory budgets, cf. $A(w)$, $w \in [0, 0.1]$, for our strategy (H6) and CoPhy's concept with different index candidates using (H1-M) and $|I| = 100, 1\,000, |I_{max}|$; $N = 4\,204$, $Q = 2\,271$.

B. Evaluation with a Commercial DBMS

In this section, we demonstrate the applicability of our solution in practice for real-world database systems. In addition, we show that our approach is independent of exact cost modeling or what-if optimization implementations.

The following performance evaluations are conducted using a commercial columnar main memory database system. While the usage of (what-if optimizer-based) cost estimations is necessary to enable the solution of large problem instances, it has also been shown to be too often inaccurate [19]. Hence, we ran all evaluations without relying on what-if or other optimizer-based estimations but executed all queries one after another with the aforementioned database system. The actual reported execution time is then used to determine a query's cost for a given index configuration. To determine the impact of index candidates on query performance, we also created all index candidates one after another and executed all queries for every candidate. These measured runtimes are then used (instead of what-if estimations) to feed the model's cost parameters. While this approach guarantees precise costs, it comes with a high evaluation time caused by evaluating every query for every index candidate multiple times. Therefore, we used a scalable workload as in Example 1 instead of the real-world ERP workload to enable these experiments.

The measurements were conducted with the newest version of the DBMS on a machine with 512 GB of main memory and 64 cores with a clock speed of 2.3 GHz. We repeated the measurements at least 100 times per query per index configuration to reduce measurement inaccuracies. The workload is created by the above-presented workload generator which mimics real-life workload characteristics. The code to reproduce the workload, data and measurements are organized as Jupyter notebooks and are available for download[4]. Based on the measured costs, we compute index configurations using CoPhy, heuristics, and our recursive strategy.

Figure 5 shows the performance of various index selection strategies. Up to a budget of roughly 30%, our solution is on par with the optimal solution CoPhy with all (2 937) candidates. The performance of our solution is always within 3% of the optimal solution.

The results of (H1) and (H4) with and without dominated candidates (skyline method [11]) based on the exhaustive candidate set are far away from optimal. Heuristic (H5) with the complete set of index candidates provides also good results. However, the results of (H5) are worse if the candidate set is too small (see Figure 3) or of lower quality (which is the case when, e.g., using H2-M or H3-M to define candidate sets, see Figure 2). Note, the results of CoPhy provide an *upper performance bound* if reduced candidate sets are used,

[4]Source code to reproduce end-to-end evaluation: http://bit.ly/index_selection_icde_submission

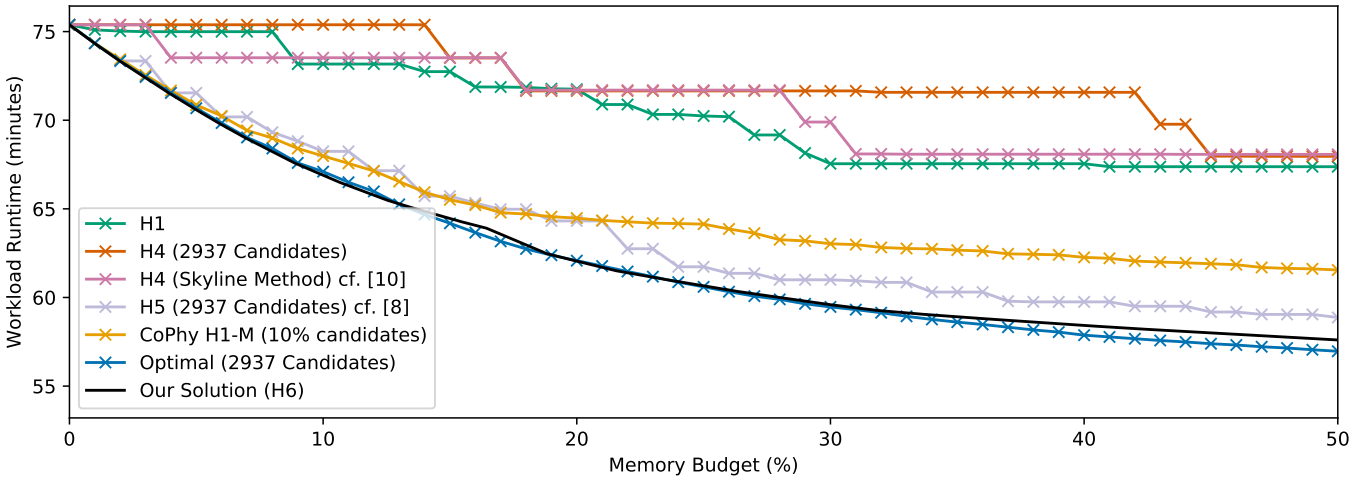

Fig. 5. End-to-end evaluation: Combinations of workload runtime and associated relative memory budgets, cf. $A(w)$, for our strategy (H6) vs. frequency-based heuristic (H1), (H4) without the skyline method and all 2 937 candidates, (H4) with the skyline method, (H5) with all candidates, and CoPhy's approach with 10% of the index candidates using the candidate heuristic (H1-M), and CoPhy with all candidates (optimal), $w \in [0, 1]$, $N = 100$, $Q = 100$, $|I_{max}| = 2\,937$. H2 and H3 as well as candidate set reductions via H2-M and H3-M are not presented due to their inferior performance.

since for a given candidate set CoPhy computes optimal selections. In this context, the results of CoPhy based on a 10% candidate selection (according to heuristic H1-M) show a significant performance decrease compared to CoPhy with all index candidates, cf. Figure 5.

The results can be summarized as follows: First, the choice of the candidate set can significantly influence the overall performance. Second, for a fixed candidate set the final selection mechanism crucially affects overall performance. Third, our approach does not depend on a suitable selection of candidates and provides near-optimal results.

Although the workload size of our example is relatively small, the problem complexity of CoPhy's LP is already surprisingly high. Both, the number of variables and constraints is linearly increasing with the size of the index candidate set (see Figure 6, Appendix Section D). The problem complexity, i.e., the solving time, typically increases super-linearly with the number of variables as well as the number of constraints. The exhaustive set of 2 937 index candidates leads to roughly 20 000 variables and constraints. The solving time for such problems can already take several minutes (cf. Table I) depending on the solver used, the specified optimality gap, the available hardware, etc. To reduce the number of candidates, permutations of multi-attribute index candidates can be substituted by their presumably best representative. However, in general, results will be negatively affected by removing other permutations.

Further, if problem instances are large, performance-based approaches like (H4) and (H5) are also affected. As in their approaches cost predictions have to be available for all index candidates, the required number of what-if optimizer calls can be (too) large. Hence, reduced candidate sets have to be used, which requires suitable candidate heuristics and often leads to an overall performance decrease, cf. Figure 5.

Finally, our results verify that our solution's results also hold in end-to-end scenarios: (i) we outperform simple rule-based selection strategies, (ii) we outperform CoPhy with restricted candidate sets, and (iii) we verify that our results are close to optimal in case of tractable selection problems. Thereby, we demonstrate that our solution finds close to optimal index configurations also when not relying on what-if estimations or specified cost models. Moreover, our solution is scalable as what-if optimizer-based costs are less frequently determined (see Section III-A) and no solver is required.

V. Discussion of Evaluation Results

We compared our solution against conceptual heuristics used by Valentin et al. (cf. (H5), [9]), Kimura et al. (cf. (H4), [11]), and CoPhy's approach. Note, for [9], its starting solution (H5) provides a lower bound while CoPhy yields an upper performance bound.

For reproducibility, in Section Section III-A (scalability) and Section III-B (performance), we used an exemplary illustrating cost model, cf. Example 1. For the end-to-end evaluation in Section IV-B, instead of a cost model, we used what-if optimizer-based cost predictions, which include complex effects of modern execution engines.

The results show that our approach scales and quickly provides near-optimal index selections. Our approach outperforms (H4) and (H5) as well as CoPhy's results if the set of candidates is small compared to the set of all potential candidates. Hence, our approach is a promising alternative to existing tools, particularly for large problems.

The scalability, i.e., the quick runtime, of our approach can be explained as follows: (i) during the recursive algorithm, we consider only a limited subset of index candidates, and (ii) instead of a randomized mechanism, we use a constructive deterministic one with a comparably small number of steps.

The performance of our approach can also be explained. Our approach exploits structures and properties that are typical for

real-world workloads and the performance of indexes: First, an index A can be applied to more queries than an (extended) index AB and requires less memory (Property 1). Second, similar indexes AB and AC typically cannibalize each other, i.e., when both are selected together they can only marginally increase the overall workload performance compared to a scenario where just (the best) one of them is selected (Property 2). Third, we can assume that taking out any index from a Pareto-efficient selection significantly affects performance. Hence, from Property 2 follows that selections with several similar indexes are not efficient and that including an index has to significantly increase performance (Property 3). Fourth, the Pareto-efficient frontier of "performance" and "required memory" is of convex shape, cf. Figure 2-5.

Our approach exploits those properties: If an index AB is beneficial, then typically index A also is, cf. Property 1, and hence, is identified (and extended) by our algorithm. Our algorithm does not construct similar indexes, cf. Property 2. Our recursion only realizes index selections/extensions with significant additional performance per size ratio while taking IIA (of the current state) into account, cf. Property 3. With an increasing number of our algorithm's steps, the realized additional performance per size typically decreases (diminishing returns), cf. Property 1 and Property 4. Thus, it is very unlikely that our algorithm misses a major improvement in a future step, and in turn, our approach resembles the efficient frontier of performance and required memory.

When Properties 1-4 are unsatisfied, our recursive approach has probably limitations. For instance, our approach might miss beneficial indexes in case they require a previous expensive but not directly beneficial index to append to. In this context, we refer to the discussion of potential extensions of Algorithm 1, see Remark 1 and Remark 2. To study optimality gaps of special cases, we recommend using CoPhy's solution as a reference. It will be challenging to derive performance guarantees as what-if optimizer-based costs lack functional structure.

VI. Related Work

An array of publications (e.g., [20], [21]) reduces the Index Selection Problem (ISP) to the Knapsack Problem (KP) [22]. Both problems share similarities and are to a certain extent comparable. The reduction to the KP as well as the work of Piatetsky-Shapiro [8] demonstrate the NP-completeness of the ISP. However, index interaction (cf. [12]) renders the reduction of ISP to KP an oversimplification of the problem.

Valentin et al. present the index selection approach of IBM DB2 [9] which greedily selects indexes until the given budget is exceeded. The authors propose to evaluate indexes by their ratio of runtime improvement vs. size consumption. To account for index interactions and maintenance costs, the determined configuration is randomly shuffled several times in search for potentially better configurations.

Chaudhuri and Narasayya presented the optimizer-based index selection tool for Microsoft SQL Server [13], [23]. They explained several techniques to reduce the complexity of the ISP. For example, to avoid considering every admissible index, they determine a set of index candidates as input for the actual index selection process. These candidates are chosen as follows: Only indexes which are the best index for at least a single query are considered as index candidates, potentially resulting in wasted potential. Moreover, their solution, cf. [13], [23], takes a *fixed number of indexes* as constraint and stop criterion. Instead, as indexes differ in size depending on the underlying data, we see their size as a more reasonable constraint and consider a *fixed memory budget*.

A more recent version of Microsoft SQL Server's index advisor handles compressed indexes [11]. Here, potential index candidates are first filtered for being efficient (i.e., are not dominated by others). Then, Kimura et al. propose a heuristic that – in contrast to the DB2 advisor – greedily selects indexes with the largest expected runtime improvement. Only after the budget is exceeded, index costs are considered when the advisor tries to "recover" from the budget violation and iteratively replaces indexes with more space-efficient alternatives.

The work of Idreos et al. [24] focuses on index selection for column stores. They introduced adaptive indexing, where the tuning of indexes is not a separate task but part of the query processing. The two techniques adaptive merging [25] and database cracking [26] either need a large number of processed queries to be fully functional or have high initial costs for the first queries compared to non-index scans. The authors presented a hybrid approach of adaptive merging and database cracking that performs close to optimal. But experimental evaluations, e.g., by Schuhknecht et al. [27], show that the success of these adaptive approaches depends heavily on query access patterns. In addition, the potentially highly-tuned database operator code needs to be adapted and it is not clear how well these indexes can be taken into account for the optimizer's cost estimations.

Often, solutions for autonomous database design, e.g., CoPhy [14], depend on plan selections or cost estimations of query optimizers. Especially for systems with large solution spaces for index configurations, query optimizer-based what-if calls can easily become the dominated performance bottleneck. Therefore, previous optimization results can be reused as presented by Papadomanolakis et al. [16] with INUM. They propose to cache already calculated query cost estimations to speed up the evaluation of similar configurations. Thereby, beneficial index combinations can be identified without fully enumerating them [14]. Their approach outperforms industry optimizers by orders of magnitude with respect to runtime and by up to a factor of $4\times$ in accuracy as they can evaluate larger solution spaces in the same time. We see INUM as a promising mechanism to speed up what-if calls.

The idea to formulate the ISP as an optimization problem is not new [14], [15], [28]. These works consider binary integer problems as well as associated continuous relaxations. As integer optimization problems are NP-complete [29], heuristics are usually used. Caprara et al. [28] use branch-and-bound algorithms to approximate solutions of their linear integer problem. Their model assumes that for each query at most one index is used which is a strong simplification. The benefit of

an index for one query is not affected by other index decisions. Thereby index interaction is often not considered which leads to suboptimal index configurations.

Papadomanolakis et al. [15] repeal the one index per query simplification by allowing more than one index per query. Furthermore, they take update costs into account. Dash et al. [14] presented with CoPhy the current state of the art solution for index selection. The authors claim to solve larger problem instances with well-known techniques from linear optimization efficiently. Their technique also relies on what-if optimizer calls to feed the input for their linear programming optimization; for details see Section II.

To mitigate scalability issues of index selection approaches, workloads can also be proprocessed. Chaudhuri et al. proposed to *compress* the workload while staying within the given error bound that the user accepts for the compressed workload [30]. For the first step of compression (i.e., the determination of query similarities) the optimizer is invoked. As such, the main bottleneck for index selection (i.e., what-if optimizer calls) is lowered but still remains. Zilio et al. [10] made similar observations, finding the proposed workload compression to be too slow. As an alternative, DB2 uses a simple approach that selects the top k most expensive queries [10].

VII. Conclusion & Future Work

We have proposed a novel index selection approach which allows computing efficient multi-attribute index selections. While either the applicability or the quality of traditional index selection approaches is limited, our strategy is applicable to large index selection problems and provides effective results.

Using various reproducible examples and comparisons to state-of-the-art index selection concepts, we have demonstrated both the general applicability of our solution approach as well as its high quality for large-scale workloads. Finally, we have also verified the performance of our solution for real-world systems using an ERP data set and workload.

The performance of state-of-the-art tools highly depends on both, (a) the quality of an initial index candidate selection and (b) the final selection mechanism accounting for index interaction. Instead, our approach deals with problems (a) and (b) simultaneously and combines four key properties for index selection: (i) the ability to consider workload-specific index interaction in general what-if optimizer-based cost models, (ii) an optimized usage of main memory resources for performance-enhancing indexes, (iii) short computation times even for large problem instances, and (iv) configurable reconfiguration costs.

In future work, we will analyze scenarios with stochastic workloads that change over time. To react to changing workloads, the model has to adapt the index selection successively. For such scenarios, it is vital to take reconfiguration costs into account in order to determine whether it is worth to reorganize the index configuration.

Further, historical workload information can be used to estimate anticipated query frequencies or even distributions of potential future workload scenarios, e.g. in the context of self-managing database systems [31], [32]. Weighting the impact of index selections for these scenarios (under risk considerations) allows computing robust data allocations under uncertainty.

References

[1] M. S. Kester, M. Athanassoulis, and S. Idreos, "Access path selection in main-memory optimized data systems: Should I scan or should I probe?" in *Proc. SIGMOD*, 2017, pp. 715–730.

[2] M. Faust *et al.*, "Footprint reduction and uniqueness enforcement with hash indices in SAP HANA," in *Proc. DEXA, Part II*, ser. Lecture Notes in Computer Science, vol. 9828, 2016, pp. 137–151.

[3] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *Proc. SIGMOD*, 2016, pp. 1567–1581.

[4] S. Chaudhuri, M. Datar, and V. R. Narasayya, "Index selection for databases: A hardness study and a principled heuristic solution," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 11, pp. 1313–1323, 2004.

[5] M. Boissier *et al.*, "Hybrid data layouts for tiered HTAP databases with pareto-optimal data placements," in *Proc. ICDE*, 2018, pp. 209–220.

[6] S. Chaudhuri and G. Weikum, "Self-management technology in databases," in *Encyclopedia of Database Systems*, 2009, pp. 2550–2555.

[7] K. Schnaitter and N. Polyzotis, "Semi-automatic index tuning: Keeping DBAs in the loop," *PVLDB*, vol. 5, no. 5, pp. 478–489, 2012.

[8] G. Piatetsky-Shapiro, "The optimal selection of secondary indices is NP-complete," *SIGMOD Record*, vol. 13, no. 2, pp. 72–75, 1983.

[9] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley, "DB2 Advisor: An optimizer smart enough to recommend its own indexes," in *Proc. ICDE*, 2000, pp. 101–110.

[10] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 Design Advisor: Integrated automatic physical database design," in *Proc. VLDB*, 2004, pp. 1087–1097.

[11] H. Kimura, V. R. Narasayya, and M. Syamala, "Compression aware physical database design," *PVLDB*, vol. 4, no. 10, pp. 657–668, 2011.

[12] K. Schnaitter, N. Polyzotis, and L. Getoor, "Index interactions in physical design tuning: Modeling, analysis, and applications," *PVLDB*, vol. 2, no. 1, pp. 1234–1245, 2009.

[13] S. Chaudhuri and V. R. Narasayya, "An efficient cost-driven index selection tool for Microsoft SQL Server," in *Proc. VLDB*, 1997, pp. 146–155.

[14] D. Dash *et al.*, "CoPhy: A scalable, portable, and interactive index advisor for large workloads," *PVLDB*, vol. 4, no. 6, pp. 362–372, 2011.

[15] S. Papadomanolakis and A. Ailamaki, "An integer linear programming approach to database design," in *ICDE Workshops*, 2007, pp. 442–449.

[16] S. Papadomanolakis, D. Dash, and A. Ailamaki, "Efficient use of the query optimizer for automated database design," in *Proc. VLDB*, 2007, pp. 1093–1104.

[17] N. Bruno and S. Chaudhuri, "To tune or not to tune? A lightweight physical design alerter," in *Proc. VLDB*, 2006, pp. 499–510.

[18] M. Boissier *et al.*, "Analyzing data relevance and access patterns of live production database systems," in *Proc. CIKM*, 2016, pp. 2473–2475.

[19] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton, "Predicting query execution time: Are optimizer cost models really unusable?" in *Proc. ICDE*, 2013, pp. 1081–1092.

[20] B. Falkowski, "Comments on an optimal set of indices for a relational database," *IEEE Transactions on Software Engineering*, vol. 18, no. 2, pp. 168–171, 1992.

[21] M. Y. L. Ip *et al.*, "On the selection of an optimal set of indexes," *IEEE Transactions on Software Engineering*, vol. 9, no. 2, pp. 135–143, 1983.

[22] G. B. Mathews, "On the partition of numbers," *Proceedings of the London Mathematical Society*, vol. s1-28, no. 1, pp. 486–490, 1896.

[23] S. Chaudhuri and V. R. Narasayya, "AutoAdmin 'what-if' index analysis utility," in *Proc. SIGMOD*, 1998, pp. 367–378.

[24] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe, "Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores," *PVLDB*, vol. 4, no. 9, pp. 585–597, 2011.

[25] G. Graefe and H. A. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," in *Proc. EDBT*, 2010, pp. 371–381.

[26] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *Proc. CIDR*, 2007, pp. 68–78.

[27] F. M. Schuhknecht, A. Jindal, and J. Dittrich, "The uncracked pieces in database cracking," *PVLDB*, vol. 7, no. 2, pp. 97–108, 2013.

[28] A. Caprara, M. Fischetti, and D. Maio, "Exact and approximate algorithms for the index selection problem in physical database design," *IEEE Trans. Knowl. Data Eng.*, vol. 7, no. 6, pp. 955–967, 1995.
[29] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
[30] S. Chaudhuri, A. K. Gupta, and V. R. Narasayya, "Compressing SQL workloads," in *Proc. SIGMOD*, 2002, pp. 488–499.
[31] M. Dreseler *et al.*, "Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management," in *Proc. EDBT*, 2019.
[32] J. Kossmann and R. Schlosser, "A Framework for Self-Managing Database Systems," in *Proc. ICDE Workshops*, 2019.
[33] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The design and implementation of modern column-oriented database systems," *Foundations and Trends in Databases*, vol. 5, no. 3, pp. 197–280, 2012.
[34] T. Kersten *et al.*, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *PVLDB*, vol. 11, no. 13, pp. 2209–2222, 2018.

APPENDIX A
NOTATIONS

TABLE II
NOTATION TABLE

Workload Parameters	N	number of attributes
	Q	number of queries
	T	number of tables
	n	number of rows of all attributes of a table
	d_i	number of distinct values in attribute i
	a_i	value size of attribute i, $i = 1, ..., N$
	s_i	selectivity of attribute i, $i = 1, ..., N$
	q_j	attributes accessed by query j, $j = 1, ..., Q$, subset of $\{1, ..., N\}$, e.g., $q_1 = \{8, 6, 13, 14\}$
	b_j	frequency of query j, $j = 1, ..., Q$
	g_i	number of occurrences of attribute i
	$\bar{q}$	average number of attributes accessed by queries
	$\bar{I}_q$	average number of index candidates applicable to a query
Cost Parameters	A	main memory budget
	w	relative main memory budget
	$f_j(0)$	cost of scanning query j without an index
	$f_j(k)$	cost of scanning query j with index k
	p_k	memory size of index k, $k \in I$
Variables	I	set of multi-attribute index candidates
	x_k	multi-attribute index k selected, yes (1) / no (0)
	$\vec{x}$	vector of decisions of all x_k for $k \in I$
	I^*	set of selected indexes out of candidates I
	$f_j(I^*)$	costs of query j for selection I^*
	$F(I^*)$	total costs of index selection I^*
	$P(I^*)$	occupied memory of index selection I^*
	$\bar{I}^*$	existing set of selected indexes (current state)
	$R(I^*, \bar{I}^*)$	index reconfiguration costs from $\bar{I}^*$ to I^*
	I_j	set of candidates out of I applicable to query j
	z_{jk}	decision variable: index k used for query j yes (1) / no (0), $j = 1, ..., Q$, $k \in I_j$

APPENDIX B
REPRODUCIBLE EXEMPLARY COST MODEL

(i) To define $f_j(I^*)$, we consider the following execution of scan operations for a query j given an index set I^*:

1) Choose the index k out of I^* for query j that produces the smallest result set. Take the selectivity of attributes of q_j and the order of attributes within k into account.
2) Scan attributes i of q_j that are coverable for a chosen index k, i.e., $i \in U(q_j, k)$, and accumulate scan costs, e.g., via

$$\log_2(n) + \sum\nolimits_{i \in k} a_i \cdot \log_2(d_i) + 4 \cdot \prod\nolimits_{m \in U(q_j,k)} s_m$$

Note, written position list elements amount to 4 bytes.

3) Choose the next best index out of I^* for the remaining attributes in q_j. Use this index and accumulate scan costs as described above.
4) Repeat these steps until no further index is used.
5) Scan all remaining attributes of q_j ordered by selectivity and accumulate scan costs F.

(ii) Let the memory utilized by a multi-attribute index k:

$$p_k := \lceil \lceil \log_2(n) \rceil \cdot n/8 \rceil + \sum\nolimits_{i \in k} a_i \cdot n$$

Note, the workload model chosen for the reproducible example resembles vector-at-a-time execution models (cf. [33]). However, the model is general and can also be applied to data-centric execution models (cf. [34]).

APPENDIX C
REPRODUCIBLE SCALABLE WORKLOAD DEFINITION

In Example 1, see Section III-A, we consider multiple tables and define the following workload:

$$\begin{aligned}
T &= 10 \\
N_t &= 50, \quad t = 1, ..., T \\
Q_t &= N_t, \quad t = 1, ..., T \\
n_t &= t \cdot 1\,000\,000, \quad t = 1, ..., T \\
d_{t,i} &= round(Uniform(0.5, n_t \cdot \left(\frac{N_t - i + 1}{N_t + 1}\right)^{0.2})) \\
Z_{t,j} &= round(Uniform(0.5, 10.5)), \quad j = 1, ..., Q_t \\
q_{t,j} &= \bigcup\nolimits_{k=1,...,Z_{t,j}} \{round(Uniform(1, N_t^{1/0.3})^{0.3})\} \\
b_{t,j} &= round(Uniform(1, 10\,000)), \quad j = 1, ..., Q_t
\end{aligned}$$

APPENDIX D
ADDITIONAL FIGURES

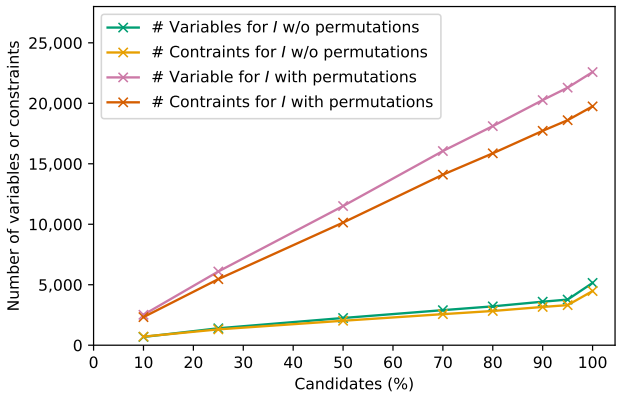

Fig. 6. Increasing problem complexity: number of variables and constraints of CoPhy's LP for different relative sizes of index candidate sets; end-to-end example with $N = 100, Q = 100$, $|I_{max}| = 2\,937$.