Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms

Jan Kossmann[1] * Stefan Halfpap[1] * Marcel Jankrift[2] Rainer Schlosser[1]
Hasso Plattner Institute, University of Potsdam, Germany
[1]firstname.lastname@hpi.de [2]marcel@jankrift.de *contributed equally

ABSTRACT

Indexes are essential for the efficient processing of database workloads. Proposed solutions for the relevant and challenging index selection problem range from metadata-based simple heuristics, over sophisticated multi-step algorithms, to approaches that yield optimal results. The main challenges are (i) to accurately determine the effect of an index on the workload cost while considering the interaction of indexes and (ii) a large number of possible combinations resulting from workloads containing many queries and massive schemata with possibly thousands of attributes.

In this work, we describe and analyze eight index selection algorithms that are based on different concepts and compare them along different dimensions, such as solution quality, runtime, multi-column support, solution granularity, and complexity. In particular, we analyze the solutions of the algorithms for the challenging analytical Join Order, TPC-H, and TPC-DS benchmarks. Afterward, we assess strengths and weaknesses, infer insights for index selection in general and each approach individually, before we give recommendations on when to use which approach.

PVLDB Reference Format:
Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *PVLDB*, 13(11): 2382-2395, 2020.
DOI: https://doi.org/10.14778/3407790.3407832

1. INTRODUCTION

Database systems use secondary indexes for speed up. However, the index selection problem, i.e., finding the optimal set of indexes for a given workload and dataset while considering constraints, e.g., a storage budget, is complex.

First, the solution space, i.e., the set of possible index combinations, is enormous. It increases with the number of indexable attributes, the number of columns per index, and the number of considered index types, for instance, B-trees, hash maps, or bit maps. Second, indexes interact with each other, i.e., the benefit of an index may depend on the existence of other indexes [45]. Third, it is challenging to efficiently quantify the impact of an index on the workload cost without deploying indexes and actually running queries, because cost estimates are inherently inaccurate [15].

The importance of performance enhancements by indexes in practice, in combination with the complexity of the problem, has led to a lot of research work in this area. Early work dates back to the 1970s [29, 41]. Since then, many (heuristic) algorithms, based on different approaches, have been developed to find optimized index configurations. The specific algorithms differ in calculation time, solution quality, optimization criteria, and the method for cost estimation.

To the best of our knowledge, there is no comparison of state-of-the-art algorithms in different dimensions, e.g., considering varying storage budgets, the algorithms' runtimes, and multiple workloads (see Figure 1). For example, Schnaitter and Polyzotis present a specific benchmark for online index selection and evaluate two index selection algorithms for changing workloads [44]. In contrast, in this paper, we compare eight algorithms for three workloads and focus on evaluating the algorithms' performance across multiple dimensions. While existing work, which presents new selection algorithms, usually contains comparisons, these are often limited regarding the evaluated dimensions and choice of compared algorithms.

In this work, we compare and evaluate eight existing algorithms for the index selection problem: [4, 9, 10, 14, 26, 43, 48, 49], and interpret their results in different scenarios. Such an investigation should (i) be reproducible, (ii) offer the flexibility to add further algorithms, database workloads, and database systems in the future, as well as (iii) automate the evaluation to facilitate practical use for database researchers and administrators. For these reasons, we developed an extensible and publicly available evaluation platform.

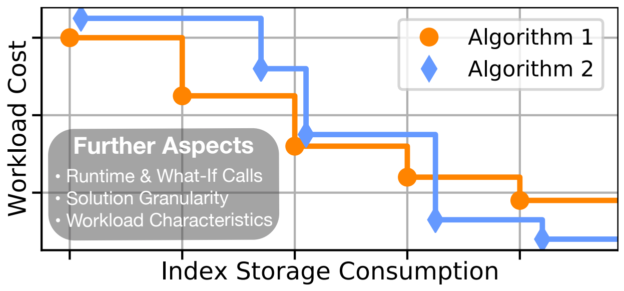


Figure 1: Various dimensions need to be considered for the evaluation of index selection algorithms.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407832

Contributions. With this paper, we present a comparative evaluation of eight index selection algorithms, namely the Drop Heuristic [49], AutoAdmin's index selection [10], DB2 Advisor [48], Relaxation [4], CoPhy [14], Dexter [26], Extend [43], and DTA Anytime [9]. This comparison includes:

- An explanation and detailed analysis of the functionality of eight diverse index selection algorithms.
- The experimental evaluation of the algorithms for three analytical benchmarks, namely, the TPC-H, TPC-DS, and Join Order Benchmark with regard to solution quality and runtime for varying storage budgets.
- An experimental analysis of the influence of different parameters of the algorithms.
- A deduction of insights and recommendations for selection algorithms depending on user needs.
- An open-source evaluation platform, which includes implementations of all compared algorithms, reproducibly automates setup, execution, and evaluation, and facilitates the extension by further algorithms or workloads.

In Section 2, we formalize the index selection problem and discuss different options for determining index benefits. Afterward, we describe, analyze, and compare the examined algorithms in Section 3. We present our methodology in Section 4 and Section 5 describes the implemented evaluation platform. In Section 6, we reveal the experimental results and examine the solutions found by the different algorithms, before we discuss our findings and mention the insights obtained from the experiments in Section 7. Section 8 concludes the paper and mentions ideas for future work.

2. INDEX SELECTION PROBLEM

In this section, we define the index selection problem as well as the vocabulary and notation used throughout this paper. Afterward, we discuss different approaches for estimating query costs, given a set of (hypothetically) existing indexes, which is essential for index selection algorithms.

2.1 Formalization

Secondary indexes are auxiliary data structures that are used to enhance the performance of database systems. Typically, there is a trade-off between improved performance and elevated storage consumption. While performance is essential for productive systems, storage is a scarce resource [51].

Index structures do not always accelerate query execution. In fact, additional indexes can even increase the workload processing time, e.g., because inserts or updates require expensive maintenance operations on the index structures [22].

The index selection problem is about finding the set of indexes for a given workload that results in the best performance while considering a set of constraints, e.g., a limited storage budget, the kind of indexes to create, and the runtime of the index selection. We use the following notation:

Workload. A workload is a set of Q queries, which are defined over a set of tables with N attributes. A query j is characterized by the subset of attributes $q_j \subseteq \{1, ..., N\}$ that are accessed for its evaluation, $j = 1, ..., Q$.

Query and workload costs are important input parameters for index selection algorithms. The total workload cost C is defined as the sum of the cost of all queries, i.e., $C = \sum_{j=1}^{Q} c_j$, where the costs c_j depend on the selected indexes. Naturally, also different frequencies (or occurrences) of queries within a workload could be taken into account to differentiate their importance during index selection.

Index. An index w with W attributes is represented by an ordered set of attributes $w = \{i_1, ..., i_W\}$, where $i_u \in \{1, ..., N\}$, $u = 1, ..., W$. W is also called the *width* of an index. In this work, the *size* of an index refers to its storage consumption. A single index cannot incorporate attributes of multiple tables but is only created on a single table.

Index Configuration. An index configuration k is a *set* of indexes. Query and workload costs can also be calculated considering an index configuration: $C(k) = \sum_{j=1}^{Q} c_j(k)$.

Potential Index. A potential index p with W attributes is any index that could be generated from an arbitrary combination of attributes (from the same table) that are part of the workload, i.e., $p = \{i_1, ..., i_W\}$, where $i_u \in \bigcup_{j=1,...,Q} q_j$, $u = 1, ..., W$.

Index Candidates. Potential indexes that are considered and evaluated by index selection algorithms are index candidates I. Usually (because of their large number and combination possibilities), index selection algorithms cannot consider all potential indexes, e.g., indexes with many attributes are ignored. Choosing index candidates from all potential indexes is an essential part of index selection algorithms. Many algorithms focus on *(syntactically) relevant indexes* that contain only columns that appear together in at least one query. Typically, algorithms evaluate different candidate sets and index configurations during the selection process.

Index Interaction. Indexes *interact* with each other, i.e., the benefit of one index can be affected by the presence of another index [45]. Thus, to find effective selections, the mutual interplay of indexes has to be taken into account. Index interaction increases the complexity of choosing suitable candidates and calculating the best selection significantly, as the benefit of indexes cannot be considered independently.

Index Selection. An index selection S is the set of chosen indexes out of a candidate set I. An algorithm's goal is to solve the problem $min_{S \subseteq I} C(S)$. Note, besides the selection S, also the candidate set I influences the solution quality.

Hypothetical Index. Hypothetical indexes [10], also called *virtual indexes* [48], do not physically exist. Their existence is only simulated to trick the optimizer into generating query plans and cost estimations that would also be created if the index was actually physically present.

2.2 Determining Query Costs

Index selection algorithms are required to quantify the benefit of an index, i.e., cost-savings when the index is utilized for executing the workload at hand. Furthermore, the index size, i.e., its storage consumption, is of interest. Both information is necessary for assessing and comparing different index candidates. A possibility to quantify the benefit of an index is to create the index physically and run each query whose costs are possibly affected by the index. Usually, this measurement-based approach results in accurate costs. However, this approach is prohibitively expensive for large workloads, in particular, because quantifying benefits in case of index interactions would require repeated recreations of the same indexes and executions of the same queries.

Therefore, most index selection algorithms do not measure but only estimate index benefits to avoid being thwarted too much by determining query costs. It is common to use the database system's optimizer and its cost model for these estimations, because the optimizer chooses the query plan and, in particular, which indexes are used. Furthermore, to not only avoid executing queries but also creating indexes, some database systems support *hypothetical* indexes. Hypothetical indexes are not physically created but only affect optimizer cost estimations. The optimizer is instructed to imagine that a specific index would exist and base its cost estimations on this imagination. So-called what-if (optimizer) calls are used to estimate query costs.

Although queries are not executed, what-if calls take a considerable amount of time, because a query optimization process is executed. In particular, for large workloads, where millions of optimizer calls must be made, what-if calls can become a significant bottleneck. Papadomanolakis et al. show that index selection algorithms spend, on average, 90% of their runtime in the optimizer [35]. We discuss the cost of what-if calls in Section 6.5.

Chaudhuri and Narasayya discuss techniques to decrease the number of optimizer calls by (i) reducing the set of configurations to evaluate and (ii) deducing costs from simpler configurations [10]. Papadomanolakis et al. present a cache-like approach (INUM) to reduce the number of what-if calls [35]. The authors claim to be three orders of magnitude faster without accuracy losses. Besides, Bruno and Nehme present with C-PQO [6] another method to speed up what-if based cost estimation.

In general, optimizer-based cost estimations can result in significant estimation errors due to cardinality misestimations [28] or inaccurate cost models [50]. As a result, index configurations predicted to be beneficial can, when actually executing queries, result in performance regressions [3, 13]. The solution quality of index selection algorithms is, to some extent, bounded by the accuracy of the cost estimations. In their recent work, Ding et al. demonstrate how machine learning classification approaches can be leveraged to predict which plan (with or without index) will be more efficient [15]. The cost models based on neural networks of Marcus and Papaemmanouil [30] could be used to mitigate the problems arising from inaccurate cost models.

3. INDEX SELECTION ALGORITHMS

In this section, we describe the compared index selection algorithms. Figure 2 gives a time-based overview of the examined publications as well as milestones in the field of index selection algorithms. The functioning and configurable parameters of the investigated algorithms are presented in Section 3.1 - 3.8. Section 3.9 summarizes the main characteristics of the algorithms, before Section 3.10 discusses machine-learning based approaches and Section 3.11 highlights specifics of commercial index selection tools.

Choice of Evaluated Algorithms. Index selection algorithms have been published since 1971, and differ in their underlying approaches and complexity. We included one of the *first* (*Drop*), *intermediate*, and *recent* (*Extend*, machine learning-based [47], *DTA*) algorithms (see Figure 2). Most algorithms were proposed in *research* papers. Some are related to *commercial systems* (*AutoAdmin*, *DB2Advis*, *DTA*). Dexter is an *open-source* algorithm. Regarding the underlying approaches, we evaluate *imperative* algorithms (i) starting with an empty index configuration and iteratively *adding* indexes (*AutoAdmin*, *DB2Advis*, *DTA*, *Extend*), and (ii) starting with a large configuration that is *reduced* (*Drop*, *Relaxation*), as well as *declarative* approaches based on (iii) linear programming (*CoPhy*) and (iv) machine learning [47]. Finally, the chosen algorithms differ in their complexity, e.g., caused by more (*DTA*, *Relaxation*) or less (*Drop*) sophisticated candidate selections and transformations to adapt the current index configuration.

3.1 Drop

One of the early index selection algorithms is Whang's Drop heuristic [49]. As the name implies, this approach successively drops index candidates. First, the initial candidate index set, $S_{|I|} = I$, comprises every potential single-column index and the processing costs for the given workload are determined with all index candidates in place, $C(S_{|I|})$. In each of the following drop phases n, $n = |I|, |I| - 1, ...$, every remaining index candidate $w \in S_n$ is removed from the current candidate set and the workload processing cost is re-evaluated, $C(S_n \setminus \{w\})$. The candidate w^* whose removal leads to the lowest cost is permanently removed for the next phase, $S_{n-1} := S_n \setminus \{w^*\}$. The original version drops index candidates until no further cost reduction is achieved. Furthermore, Whang's work states that costs are determined by characteristics of the data, not by the query optimizer.

Parameters. In our implementation, the maximum number of (finally) selected indexes $|I| - n$ can be configured.

3.2 AutoAdmin

Chaudhuri and Narasayya propose the *AutoAdmin* index selection algorithm for Microsoft SQL Server [10]. The iterative algorithm identifies multi-column indexes by incrementing the index width in each iteration. Iterations consist of two steps: First, candidates S_j are determined per query $j = 1, ..., Q$. The union of the candidates of all queries $\bigcup_j S_j$ serves as input for the second step, which considers all queries while determining the best index configuration. For both steps, the procedure only differs in the considered queries and index candidates. The algorithm is a combination of a complete enumeration of all subsets of index candidates with m elements and a greedy extension to find $n > m$ indexes. If $m = n$, the enumeration evaluates all index subsets with n elements to guarantee an optimal solution. The number of combinations may be prohibitively large to be evaluated in a reasonable amount of time. If $m = 0$, a pure greedy approach is used to decrease the runtime.

Using the single-column (m-column) indexes of the first (m^{th}) iteration, two-column ($(m + 1)$-column) indexes are created and evaluated in the second ($(m + 1)^{th}$) iteration. To create multi-column indexes, Chaudhuri and Narasayya propose two different strategies, selecting more indexes for better results or fewer indexes for faster computation times.

Parameters. While the authors mention a storage budget or the number of indexes as possible constraints, the latter is used throughout the original paper. Our implementation uses the maximum number of indexes as a constraint to be as close to the original as possible. Also, the number of naively enumerated indexes and the index width can be configured.

3.3 (Anytime) DTA

The Anytime algorithm of the Database Engine Tuning Advisor (DTA) for Microsoft SQL Server [9] is a continuously refined [1, 11, 12, 31] version of the *AutoAdmin* [10]

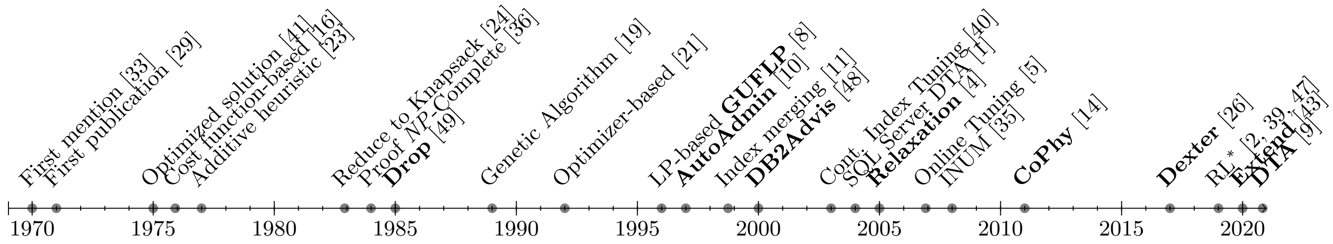


Figure 2: Timeline of milestones for index selection algorithms. Algorithms that are described, implemented, and evaluated in this work are highlighted with bold type. *RL refers to Reinforcement Learning.

index selection. The core approach to first determine index candidates per query and then identify an index configuration for the entire workload, based on the original greedy enumeration, is the same.

The approach uses the following main extensions: (i) Iterations with increasing index widths are unnecessary since multi-column indexes are considered from the start. (ii) After candidate selection, candidates and configurations are merged to determine further candidates that are beneficial for multiple queries. (iii) Index interactions are considered by identifying seed configurations to avoid complete enumerations that may evaluate many, certainly suboptimal, configurations. (iv) As the name suggests, the tuning times can be limited to guarantee solutions even for many candidates and seeds. (v) The algorithm is capable of simultaneously tuning indexes, materialized views, and partitioning criteria.

Parameters. The maximum width of index candidates and a limit for the tuning time that applies – in contrast to the original proposal – to the entire algorithm can be configured.

3.4 DB2Advis

Valentin et al. present the algorithm DB2 Advisor (short *DB2Advis*) to identify beneficial indexes for IBM's DB2 [48]. *DB2Advis* utilizes the optimizer's what-if capabilities and follows a three-step approach.

In the first step, *DB2Advis* determines index candidates. For each query j, $j = 1, ..., Q$, of the workload, hypothetical indexes are created on columns that, e.g., appear in equality or range predicates or in interesting order [46] lists. In addition, to not miss any candidate, all potential indexes are added as hypothetical indexes until a certain number of indexes is reached. Afterward, the optimizer is asked for the best plan for query j. Previously created hypothetical indexes that are utilized in the resulting plan are added to the set of index candidates (I) that is used in the next step.

In the second step, all index candidates in I are sorted by their benefit-per-space ratio in decreasing order. Following, index pairs w_1 and w_2 are combined if w_1 has a higher ratio, and its leading columns are equal to w_2's column permutation. In this case, the benefit of w_1 is updated, w_2 is removed from the list, and I is potentially resorted corresponding to its updated costs. Then, following the sort order, indexes are added to the final index configuration (S) until the storage budget (A) is exceeded.

Lastly, the current index configuration is randomly varied to improve its benefit and account for complex effects like index interaction: Sets of the previously calculated solution (S) are exchanged with sets of indexes that are not part of the solution (due to the budget constraint). If the variant leads to lower overall costs, it becomes the new solution.

Parameters. There are two main parameters: (i) the time for random variations and (ii) the maximum number of hypothetical indexes that are evaluated in the first step of the algorithm. Our implementation also allows configuring the maximum index width and does not support limiting the number of hypothetical indexes (ii). Such a preselection could be easily added, also to the other algorithms. We refrained from including it to compare the core algorithms.

3.5 Relaxation

Bruno and Chaudhuri propose to derive indexes from a (presumably too large) optimal index set by repeatedly transforming it to decrease the storage consumption [4]. First, they obtain optimal index configurations for each query. Therefore, they instrument the optimizer and exploit knowledge about the optimizer's index usage to avoid a brute force approach. Second, the optimal index configuration for the entire workload is defined as the union of all queries' optimal index sets. Afterward, this configuration is repeatedly *relaxed*, i.e., transformed, to lower its storage consumption while keeping the configuration's benefits as high as possible. They use five index transformations: (i) Merging two indexes into one. (ii) Splitting two indexes, i.e., creating an index with shared attributes and up to two (one per input index) indexes with the residual attributes. (iii) Prefixing an index by removing attributes from the end. (iv) Promoting an index to a clustered index. (v) Removing an index.

Parameters. The maximum width for index candidates can be configured as well as which of the transformations are permitted. Our tool currently uses a brute force approach to obtain the optimal index configuration per query. As clustered indexes are not considered in this work, we did not integrate this transformation.

3.6 LP-based Approaches (CoPhy)

Linear programming (LP) is a common approach to solve optimization problems by specifying the optimization goal and constraints with linear equations. Then, off-the-shelf solvers are used to calculate optimal solutions. Solvers are optimized to discard invalid and sub-optimal solutions early and, thus, outperform brute force approaches significantly. Commonly, index selection algorithms are formulated as integer LP problems, which are not scalable. The problem complexity of LP formulations for the index selection problem can be reduced by restricting the solution space, e.g., the number of index candidates by decreasing the allowed index width. Of course, limited candidate sets might lead to suboptimal solutions for the unrestricted problem.

GUFLP. Caprara and Salazar González derive an LP formulation [7, 8] for the index selection problem from a Generalized Uncapacitated Facility Location Problem (GUFLP). They reduce the complexity of the LP formulation with a restriction of the solution space by allowing only *a single index per query*. Thus, opportunities that only arise when multiple indexes exist simultaneously are not taken into account.

CoPhy. Dash et al. propose *CoPhy*, a more sophisticated LP formulation [14] for the index selection problem. In contrast to GUFLP, their approach considers multiple indexes per query (see also [34]) and multiple query plans that are possibly chosen by the optimizer depending on existing indexes. Below, we describe the essence of CoPhy's LP approach.

The costs of (a fixed query plan and) executing a query using specific indexes have to be at hand (e.g., based on what-if cost estimations). We consider a set K of different index combinations. If a *subset* of index candidates $k \subseteq I$, called *option* in the following, is applied to query j, the costs are $c_j(k)$, $k \in K \cup \{0\}$, where 0 describes the option that no index is applied to j, $j = 1, ..., Q$.

In the LP, binary variables z_{jk} are used to model whether an index option k is applied to a query j. Binary variables x_i indicate whether an index $i \in I$ with its corresponding storage consumption a_i is selected (as part of at least one chosen option k). The constraints of the LP guarantee that a unique index option is used for each query j and that the used indexes i do not exceed the storage budget A.

The complexity of the LP problem is characterized by the number of variables and constraints. As LP formulations require all cost coefficients $c_j(k)$, the number of necessary what-if calls can be estimated. In general, LP approaches do not scale as the problem complexity sharply increases in the number of queries Q and the number of options $|K|$. Hence, solver-based approaches are (i) either not applicable for large problems (see [43], Table I), or (ii) lead to *suboptimal* results as the candidate set sizes, cf. K, need to be reduced (see [43], Figure 3-4). To mitigate such solver-based scalability issues, heuristic decomposition approaches can be used [42].

Parameters. The maximum width of index candidates and the number of applicable indexes per query (1 corresponds to GUFLP) can be specified.

3.7 Dexter

Dexter is an open-source index selection tool for PostgreSQL [25, 26] and was developed by Andrew Kane. The algorithm builds on hypothetical indexes and is divided into two phases. First, the processed queries, together with information about their runtime, are gathered from the plan cache. Queries with the same template but different parameter values are grouped.

The second phase involves multiple sub-steps. (i) The initial costs (of the current index configuration) of the gathered queries are determined by using the `EXPLAIN` command. (ii) Hypothetical indexes are created for all potential single- and multi-column indexes that are not yet indexed. (iii) Again, cost estimations and query plans are retrieved from the query optimizer. The hypothetical indexes created in step (ii) that are part of these query plans become index candidates for the corresponding queries. (iv) For all queries, the index candidate with the most significant cost-savings compared to the costs obtained in step (i) is selected.

Dexter does not consider already existing indexes for deletion. Indexes are created independent of their storage consumption and cannot contain more than two columns.

Parameters. The tool offers a couple of parameters. The most important one is the *minimal cost-savings percentage* (default value 50%). It defines the minimal cost-savings that must be achieved by an index candidate to be selected.

3.8 Extend

The iterative selection algorithm for multi-column indexes of Schlosser et al. focuses on efficiency without limiting the set of index candidates a priori [43]. The cost inputs for the algorithm can rely on what-if estimations or manually crafted cost models (e.g., [27]).

The algorithm works in a greedy stepwise fashion, either adding or extending one index per step based on the highest benefit per storage ratio. The algorithm starts with an empty solution set $S = \emptyset$. Initially, the single-attribute index with the highest cost reduction per storage is added to S. Afterward, there are generally two options: either a new single-attribute index $\{i\}, i \in \{1, ..., N\}$ with $\{i\} \cap S = \emptyset$ is added, or an existing index $w \in S$ is extended by appending attribute $i \in \{1, ..., N\}$. Again, the option with the best ratio of further cost reduction and additional storage consumption is chosen. This procedure accounts for index interaction as in each step, the effect of already chosen indexes is considered. The algorithm terminates if the storage budget is reached or no further cost improvement can be realized.

Parameters. Originally, *Extend* does not limit the index width. In our implementation, the index width can be limited to make it comparable with the other approaches.

3.9 High-level Comparison

Table 1 summarizes the main characteristics of the eight compared index selection algorithms. When selecting indexes, algorithms consider either the benefit per storage consumption (*CoPhy*, *DB2Advis*, *Extend*, *Relaxation*) or the pure benefit (*AutoAdmin*, *Dexter*, *Drop*, *DTA*). Algorithms also differ in the stop criterion. Some algorithms allow specifying an arbitrary upper limit of the total index storage consumption. All algorithms but *Drop*[1] support the selection of multi-column indexes. Dexter is limited to two-column indexes. In particular, in the case of *CoPhy*, multi-column indexes increase the number of what-if calls drastically. The consideration of index interaction in *DB2Advis* is limited, mainly late in the final variation phase.

3.10 Machine Learning-based Approaches

Recently, machine learning-based approaches for index selection have been proposed. For example, Sharma et al. [47] demonstrate how deep reinforcement learning (RL) can be used for index selection with a publicly available implementation[2]. Sadri et al. describe the prototype of another RL-based approach to identify beneficial indexes in a replicated database scenario [39]. Basu et al. specify a general RL approach for database tuning problems [2]. The framework is applied to index selection for the TPC-C workload. Index

[1] *Drop* theoretically supports multi-column indexes. However, the configuration tests (and, thus, runtime) would significantly increase with the number of candidates.

[2] Source code: `https://github.com/sh-ankur/autoindex`

Table 1: Summary of the compared algorithms in chronological order.

	Drop	AutoAdmin	DB2Advis	Relaxation	CoPhy	Dexter	Extend	DTA
Minimization goal	Costs	Costs	$\frac{Costs}{Storage}$	$\frac{Costs}{Storage}$	$\frac{Costs}{Storage}$	Costs	$\frac{Costs}{Storage}$	Costs
Stop criterion	# Indexes	# Indexes	Storage	Storage	Storage	Savings (%)	Storage	Storage
Multi-column indexes	No	Yes	Yes	Yes	(Yes)	(Limit 2)	Yes	Yes
Index interaction	Yes	Yes	(Yes)	Yes	Yes	Yes	Yes	Yes

interaction is ignored to reduce the problem's complexity.

Limitations Even though the conceptual idea of applying machine learning to index selection is promising, we did not include the aforementioned approaches in this evaluation because, currently, they are still limited and cannot be applied to the benchmarks utilized for evaluation. While there are no implementations or details for reimplementation available for some approaches [2, 39], Sharma et al.'s work [47] lacks support for operators different from filters, e.g., joins. Also, the concept of RL, where an agent independently develops a strategy to solve a problem, requires massive amounts of training data, i.e., example queries, to develop a broadly applicable strategy. While generating this training data is challenging in itself, a large amount of training data results in long training times. Even further, it is vital for RL-based approaches to quickly evaluate the agent's actions (index creation/removal) to determine its reward. However, in this case, the efficiency of the evaluation is limited by the database system because (hypothetical) indexes must be created and costs must be determined. This particular process cannot be accelerated by GPUs or TPUs as often done for RL.

Nevertheless, we evaluated the implementation of Sharma et al. [47] on the TPC-H dataset. For ten filter-only queries, the reinforcement model was almost on par[3] with the traditional approach after 8 hours of training on CPUs.

3.11 Commercial Index Selection Tools

While some of the chosen algorithms are related to tools employed in commercial DBMS products, the re-implemented algorithms do not fully reflect the behavior and performance of the original tools, which may be continuously enhanced and optimized. Such tools need to set further focus points, such as robustness, scalability, time-bound tuning, and integration [1, 12]. Microsoft's Database Engine Tuning Advisor (DTA) [31] and the DB2 Design Advisor [52] are able to simultaneously consider multiple physical design aspects, e.g., indexes, partitioning, and materialized views. Further, commercial tools must be able to tune dynamic workloads in a production environment and support user interaction [12].

4. METHODOLOGY

An objective comparison of index selection algorithms is challenging because the quality of an algorithm depends on the input (workload and index benefits), the algorithm's configuration, as well as on the constraints (budget) and optimization goals (benefit, ratio of benefit and storage), and the granularity[4] of the solutions. In other words, the best algorithm may depend on the specific evaluation scenario. Thus, we tried to cover a broad range of scenarios for our comparison. In this section, we describe our evaluation setup.

First, we detail the investigated workloads in Section 4.1 and how cost estimations are realized in Section 4.2. Following, we explain constraints, optimization goals, and discuss our setup.

4.1 Workloads

For our comparison, we use three benchmark workloads of different scales and characteristics, e.g., the number of potential indexes, number of queries, or whether they are based on synthetic or real-world data, to investigate the workload's potential impact on the algorithms' solution quality and runtime. In the following, we highlight the differences of the TPC-H [37], TPC-DS [32], and Join Order Benchmark [28].

Table 2 displays relevant metrics for index selection for the three benchmark workloads and schemata. The TPC benchmarks are standardized analytical benchmarks and can be scaled with a scale factor. The JOB is based on reasonable queries over the Internet Movie Data Base (IMDB). In contrast to the synthetic datasets, the IMDB comprises real-world data with realistic cardinalities and dependencies, and focuses on the processing of joins.

The metrics demonstrate the different scales of the benchmarks. The TPC-H benchmark is relatively small and can be utilized for quick evaluations while the TPC-DS benchmark represents more realistic scenarios and is more abundant in every dimension. Even though the JOB's workload consists of the most queries, the number of potential (multi-column) indexes is comparably low. Most JOB queries contain many columns, but most of them belong to different tables and indexes are only created over attributes of the same table.

4.2 Determining Query Cost

All compared algorithms require a (more or less) large number of query cost determinations (given a fixed index configuration) and/or cost evaluations with large configurations (see Section 3.1 - 3.8). Although it is theoretically possible to obtain query costs by physically creating/dropping indexes and executing queries (repeatedly), it would take too much time and restrict feasible algorithm settings, especially the number of index candidates, by a too large degree. Therefore, we use hypothetical indexes for cost (and size) estimations.

Although these estimates are inaccurate (see Section 2.2), they offer a reasonable combination of *speed, accuracy, and accessibility* and are consistent for all algorithms. Note, in this work, instead of assessing cost estimation approaches, we focus on the evaluation of index selection algorithms.

Based on the fact that many of the index selection algorithms proposed in the literature require what-if calls for practical application, it is surprising that database systems do usually not expose (a well-documented) interface to retrieve cost estimates for hypothetical indexes.

[3]Results: `https://git.io/EvaluationIndexSelectionRL`

[4]Granularity refers to the ability to identify solutions whose storage consumption exploits any provided budget.

Table 2: Metrics for the evaluated benchmark schemata and workloads. The number of relevant index candidates was determined by generating all permutations of all syntactically relevant indexes.

Benchmark	Dataset	Relations	Attributes	Queries	Relevant n-column candidates			
					$n = 1$	$n = 2$	$n = 3$	$n = 4$
JOB	Real-world	21	108	113	73	218	552	1 080
TPC-H	Synthetic uniform	8	61	22	53	398	3 306	29 088
TPC-DS	Synthetic skewed	24	429	99	248	3 734	68 052	1 339 536

Therefore, database options for the evaluation were limited. We chose PostgreSQL with the extension HypoPG [38]. HypoPG enables creating, dropping, and size estimation of hypothetical indexes. Using PostgreSQL's `EXPLAIN` command, query plans with arbitrary hypothetical index configurations can be inspected. In doing so, we can determine which indexes are used and the estimated total execution cost for the plan. By calling `ANALYZE` before an index selection evaluation process is initiated, we ensure the existence of up-to-date statistics that are used for cost estimations and for predicting the storage consumption of hypothetical indexes. Thereby, single-column statistics (PostgreSQL holds histograms and the 100 most common values by default) are built for all attributes. `EXPLAIN` reports costs in arbitrary units, which are based on and can be tuned with parameters, e.g., specifying the relative performance of processing a row vs. fetching pages sequentially from disk. We use PostgreSQL planner's default cost parameters and could not find significant cost estimation differences between actual indexes and HypoPG's hypothetical ones[5].

Write statements are not differently costed if indexes are present, neither by HypoPG nor by PostgreSQL's optimizer, which is not an issue for our evaluation of purely analytical workloads. For transactional workloads, PostgreSQL's optimizer could be complemented by a manual cost model to approximate index maintenance costs.

4.3 Constraints and Optimization Goal

The optimization goals and possible constraints of the compared algorithms differ (see Section 3).

Optimization goal. Most of the compared algorithms optimize the ratio of index benefit and storage consumption. Therefore, we evaluate all algorithms concerning the storage consumption of selected indexes and the *estimated*[6] workload costs. *Drop*, *DTA*, *Dexter*, and *AutoAdmin* minimize workload costs without considering storage consumption (*General Insights (vii)* in Section 7). Although it is possible to adapt these algorithms to consider the ratio of benefit and size, we decided to evaluate their original implementation.

Index Selection Limit. Algorithms differ in the way they constrain the number of the selected indexes. *Drop* and *AutoAdmin* (also supports a budget constraint) limit the number. Dexter limits by the minimal cost-saving percentage. Most commonly, algorithms limit the storage budget. For all workloads and algorithms, we report workload costs with varying storage consumption. For *Drop*, *AutoAdmin*, and *Dexter*, the number of selected indexes and the minimal cost-saving percentage allow to implicitly control the storage consumption, which is, in general, increasing for both a higher number of indexes and a lower minimal cost-saving percentage. We denote the exact storage consumption, which may be equal or lower than the provided budget, for differing parameters in the figures of Section 6 for all algorithms.

Runtime. Only *DTA* originally supports to limit the runtime of the selection process. In general, runtimes can be indirectly controlled by limiting the investigated index candidates, e.g., index width or naively enumerated combinations (only *AutoAdmin* and *CoPhy*). The number of cost requests and, thus, evaluated configurations is the main cost factor for selection algorithms [35]. Further, *DB2Advis* allows specifying the time for random substitutions after a preliminary index selection was conducted (see Section 3.4). We chose algorithm settings in a way that runtimes did not become too large. We report runtime details and a detailed cost breakdown in Section 6.

5. EVALUATION PLATFORM

To automate the comparison of index selection algorithms, we developed an evaluation platform. Besides enabling the reader to *reproduce all results* and retrace the algorithm's selection steps completely, the platform facilitates the integration of additional algorithms, workloads, or database systems in the future.

5.1 Implementation

The open-source[7] evaluation platform is implemented in Python 3. The main goal of our implementation is to automate the evaluation of varying algorithms, settings, and workloads. The automation includes the setup, i.e., data generation and loading, query generation, the evaluation of different parameters for the algorithms, and the collection and summary of the results.

The centerpieces of the implementation are the compared selection algorithms. We implemented all algorithms (except *Dexter*), including unit tests, based on the original descriptions and tried to keep them as close to the original as possible. For *Dexter* (Section 3.7), we used the publicly available implementation [25] and embedded it into Python to offer the same interfaces for all algorithms. For LP-based approaches, we implemented the input generation with Python and the model in AMPL [20] using the Gurobi solver (v8.1.0). Algorithms can access `Query` and `Table` objects to generate index candidates of varying widths or for specific queries.

The `CostEvaluation` class implements the determination of query/workload costs for given index configurations. This class can be used by the algorithms transparently, i.e., algorithms do not have to consider how the costs are determined, e.g., whether hypothetical indexes are used or not. The

[5]Estimations based on actual and hypothetical indexes: `https://git.io/CostEstimationAccuracyHypoPGIndexes`

[6]Note, we do not report actual runtimes for physically created index selections, because they *could arbitrarily differ* from the estimated costs. Thus, they are not adequate for assessing the algorithms, which *only consider estimates* during selection.

[7]`https://git.io/index_selection_evaluation`

`CostEvaluation` takes care of creating and dropping (hypothetical) indexes based on the current and last specified index configuration. The `CostEvaluation` also handles pruning and caching of cost estimations. Section 5.2 describes these optimizations in detail.

The `DatabaseConnector` builds an abstraction layer for different database systems. It provides a consistent interface for using what-if capabilities and hides different SQL dialects. Currently, the platform contains connectors for PostgreSQL and SAP HANA, while the latter does not support hypothetical indexes. We have compared[8] the cost estimations by PostgreSQL and Microsoft SQL Server for different benchmark queries. The relative differences between the query costs in a particular DBMS were similar. While the abstraction of the `CostEvaluation` facilitates adding new database systems, hypothetical indexes are not entirely documented, and their sizes cannot be obtained in SQL Server.

Configuration options, e.g., scale factors, the database system, parameter values for the algorithms, and whether and how often the workload is executed with the calculated index selection can be controlled via a central JSON file.

5.2 What-If Call Optimizations

Obtaining cost estimates for queries given a particular index configuration makes up a large part of the total runtime of index selection algorithms because it requires optimizer invocations (see Section 2.2) as well as inter-process (or network) communication. In the literature, the reduction of optimizer invocations is often a main focus, e.g., with *atomic configurations* [18] in the AutoAdmin work [10, p. 3]. To facilitate efficient evaluations, our `CostEvaluation` (i) avoids unnecessary optimizer calls, and (ii) maintains a (cost estimation) cache. (i) If no index of a given configuration is applicable for a query, the costs without indexes are returned. (ii) Given a query and all possibly applicable indexes of a requested configuration, the cost estimation cache stores retrieved cost estimates. Note, multiple cache entries (with different sets of possibly applicable indexes) for the same query may exist. We do not limit the cache size and reset the cache at the beginning of each algorithm evaluation. Besides (i) and (ii), we did not implement any further techniques to reduce the number of optimizer calls (see Section 2.2).

6. EVALUATION

In this section, we evaluate the selected algorithms (see Section 3.1 - 3.8) for the TPC-H, TPC-DS, and JOB workloads. After the experimental setup (Section 6.1), we present results for each of the workloads (Section 6.2 - 6.4) regarding the quality of the identified solutions and their corresponding runtime depending on the storage consumption.

Afterward, Section 6.5 discusses the impact of hypothetical indexes and cost requests on algorithm runtimes, before Section 6.6 sheds light on the influence of different algorithm parameters, i.e., the index width, *DB2Advis*'s time for random variations, *AutoAdmin*'s number of naively enumerated indexes, and *DTA*'s runtime limit.

6.1 Experimental Setup

All experiments were executed on an Intel Xeon 8180 platinum CPU with PostgreSQL 12.1 and Python 3.7.5. For the vital part of what-if optimization and hypothetical indexes, we employed HypoPG, commit `238cca5`.

We used PostgreSQL's default index type, a non-covering *B-tree*, for the following experiments. However, the evaluation platform is not conceptually limited to non-covering B-tree indexes. The admissible index width was set to 2. Thereby, also the initial index candidate set contains all syntactically relevant candidates of width 2. The time for *DB2Advis*' `TRY_VARIATION` step was set to 0. While this step is part of the original algorithm, random exchanges could be added to all algorithms, and we want to evaluate the solution quality of the core algorithms. For *AutoAdmin*, the number of indexes selected by a naive enumeration was set to 1. The *DTA* algorithm is designed to be interruptible at anytime and to still deliver acceptable solutions. We granted a maximum runtime of 30 minutes.

Algorithms that stop after a certain number of indexes has been selected and do not use a storage budget were provided with an increasing maximum number of admitted indexes. The exact budgets, admitted indexes, and further configurations can always be obtained from the benchmark configuration files in our repository. The relative estimated costs of index configurations are based on a benchmark setup without indexes, i.e., there are no indexes or primary keys.

6.2 TPC-H

The TPC-H measurements are based on a scale factor of 10. For Figures 3(a) and 3(d) we excluded the queries 2, 17, and 20 because due to subqueries their estimated costs are orders of magnitude higher than for TPC-H queries on average in PostgreSQL (see Figures 4(a) and 4(b)). Without the exclusion, these three queries dominate the costs of the entire workload, thereby rendering the index selection problem less complex because an index that decreases the cost of at least one of these queries would always outperform indexes for other queries by orders of magnitude.

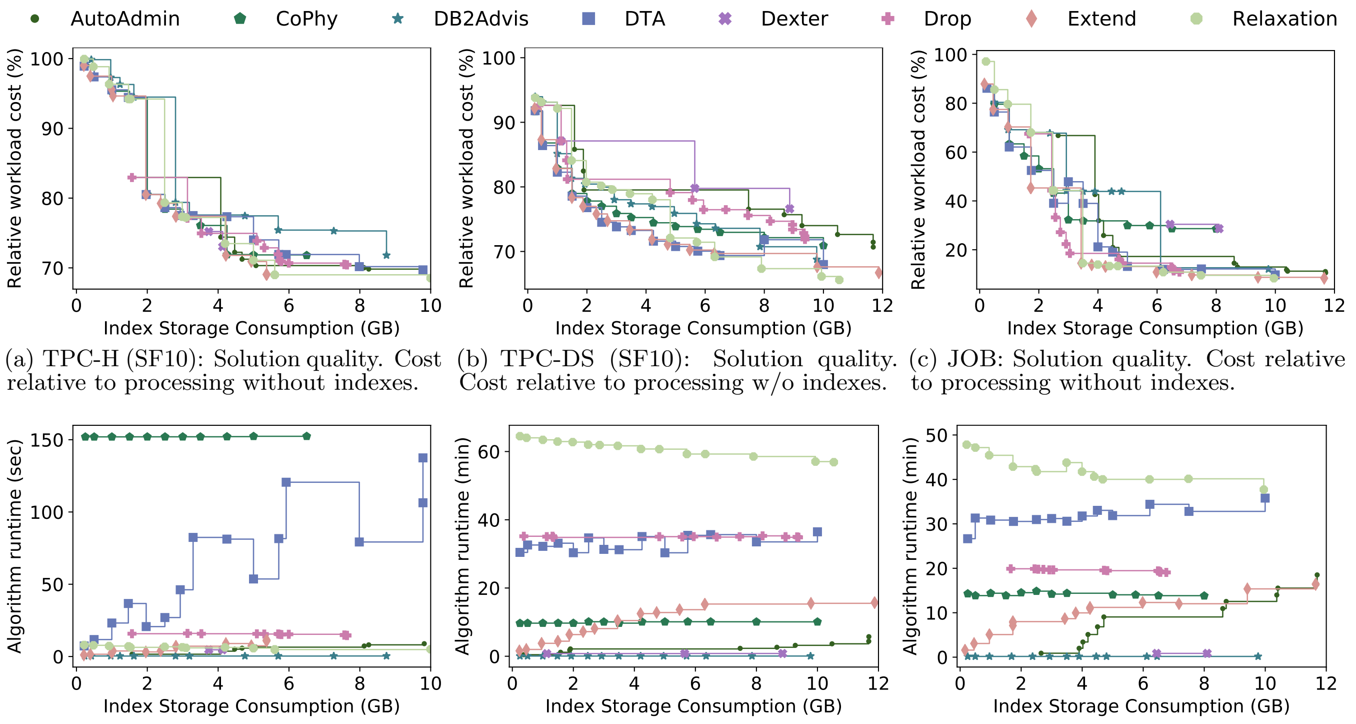
Figure 3(a) depicts the performance of the identified solutions for budgets from 0 to 10 GB. Each marker indicates an index combination identified by an algorithm for a particular storage budget. First, it becomes apparent is that most algorithms do not fully utilize the maximum budget of 10GB because no further indexes that reduce the workload cost can be found. For example, *Extend* uses only ≈ 6 GB.

Second, there is a structural difference between the algorithms that use the maximum number of indexes as a stop criterion and the budget-based algorithms. While the latter identify indexes for low budgets, starting with a few hundred megabytes, *Drop* and *AutoAdmin* need roughly 2 GB to add the first index because they first add the index with the largest cost improvement, independent of its size.

Further looking at both, the workload cost in Figure 3(a) and the runtimes in (d), it is not trivial to determine a winner. Using the platform, we can also create a table[9] that shows which indexes are selected for which budget by the algorithms. For many budgets, *Extend* and *Relaxation* identify the best solutions with acceptable runtimes. Due to the employed minimization goal, *AutoAdmin* and *Drop* need large budgets to find their first index, but this index is a substantial improvement over all other algorithms. Note, *DTA* would still identify usable, possibly the same, solutions if its runtime would be limited to, e.g., 20 seconds.

[8]Results: `https://git.io/CostEstimatesSQLServesPSQL`

[9]Table containing index selection steps:
`https://git.io/IndexSelectionSequenceTableTPCH`



(a) TPC-H (SF10): Solution quality. Cost relative to processing without indexes.

(b) TPC-DS (SF10): Solution quality. Cost relative to processing w/o indexes.

(c) JOB: Solution quality. Cost relative to processing without indexes.

(d) TPC-H (SF 10): Observed runtime for budgets from 0 to 10 GB.

(e) TPC-DS (SF 10): Observed runtime for budgets from 0 to 12 GB.

(f) JOB: Observed runtime for budgets from 0 to 12 GB.

Figure 3: (a), (b), (c): Estimated workload processing costs (relative to estimated costs without indexes); (d), (e), (f): Algorithm runtime including cost requests and index simulations. TPC-H (queries 2, 17, 20 excluded), TPC-DS (queries 4, 6, 9, 10, 11, 32, 35, 41, 95 excluded), and Join Order Benchmark on PostgreSQL.

DB2Advis has a constantly low runtime and still finds acceptable solutions. The low runtime is caused by its modus operandi (see Section 3.4 and Section 6.5): Most other algorithms generate many combinations and check their impact with what-if optimization calls, while *DB2Advis* only issues a fixed number of $2 \times Q$ cost requests. *Drop* shows an almost constant runtime since, for each round, it starts with the same large set of index candidates, drops them one by one, thereby, behaving similarly for every case.

Dexter's runtime is constant because it does not depend on a budget but on the number and complexity of queries. While its solution quality is close to the best, it cannot produce fine-grained solutions and identifies only two solutions for all evaluated budgets.

Finally, Figures 4(a) and 4(b) show the costs that are achieved with each algorithm's best index combination that does not consume more than 5 GB of storage on a per-query basis to better understand the mechanics of the different algorithms. The figures indicate that most algorithms, except for *Dexter* and *Drop*, manage to identify important indexes for queries where indexes are applicable, in particular for the expensive queries shown in (b). *DTA* and *Extend* achieve the lowest costs for all of the displayed queries while *Relaxation* performs only for the queries 9, 12, and 21 slightly worse. The platform allows generating the corresponding charts for other benchmarks and budgets.

6.3 TPC-DS

The TPC-DS measurements are based on a scale factor of 10. For all TPC-DS experiments, we excluded the queries 4, 6, 9, 10, 11, 32, 35, 41, and 95 because again, these queries have the potential to distort a fair assessment of index selection algorithms. The higher complexity of the TPC-DS benchmark compared to TPC-H (cf. Section 4.1) can be used to further emphasize the strengths and weaknesses of the algorithms.

Figure 3(b) shows the solution quality of the investigated index selection algorithms for memory budgets from 0 to 12 GB. In contrast to the TPC-H measurements, the differences, especially for realistic budget sizes, between 2 and 10 GB, are larger. The solutions identified by the algorithms vary to a larger degree because there are more (indexable) columns and more queries that benefit from indexes.

For the TPC-H experiments, *Drop*, and *AutoAdmin* found the first indexes for relatively large budgets, around 2 GB. For TPC-DS, they identify beneficial solutions much earlier because there is no single dominating table, such as the `lineitem` table for TPC-H, in the TPC-DS dataset. Thus, impactful indexes do not have to be that large.

For budgets of 2 GB and above, differences start to become apparent. *Extend* and *DTA* take the lead and generate the best solutions. In contrast to other approaches, they keep adding relatively small indexes to the solution for a larger budget range. For instance, *Extend*'s modus operandi causes this behavior (see Section 3.8): For each step, the index with the largest overall benefit-per-space ratio is added. For large budgets, the solution quality of all approaches is more homogeneous, and *Relaxation* identifies the best solution.

DB2Advis achieves again an almost constant runtime caused by its functioning and offers a good tradeoff between

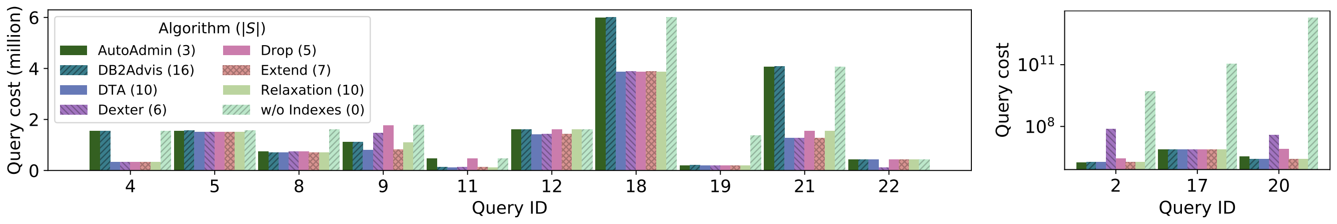


Figure 4: Estimated query processing costs for TPC-H (scale factor 10) on PostgreSQL. Queries 1, 3, 6, 7, 10, 13, 14, 15, and 16 are omitted as their costs were not affected by indexes for a budget of 5 GB. Expensive queries (2, 17, 20) depicted with log (right), others (left) with linear scale. S is the final index configuration.

runtime and solution quality. While there is quite some difference to *DTA* and *Extend* for medium-sized budgets, the solution quality of *DB2Advis* is comparable for small and large budgets. The runtime is in the range of seconds compared to minutes for *Drop*, *Extend*, *DTA*, *AutoAdmin*, and *Relaxation*, which shows the longest runtime caused by its functioning that requires many transformations to push the storage consumption below the given budget.

The solution quality of the ready-to-use approach *Dexter* is again not particularly bad if a solution is identified. However, the effect caused by its lack of fine-grained solution becomes more pronounced for the TPC-DS workload for which many, comparably small, indexes can have a substantial impact.

6.4 Join Order Benchmark

The Join Order Benchmark (JOB) measurements are depicted in Figures 3(c) and 3(f). For TPC-H and TPC-DS, *DTA* and *Extend* find the solutions with the lowest workload costs for most of the examined budgets. Medium-sized budgets (from 2 to 3.5 GB) are not handled well for this experiment because fine-grained solutions are missing. *Drop* identifies many small indexes that decrease the workload cost significantly for medium-sized budgets while it completely lacks solutions for small budgets similar to *AutoAdmin*. *CoPhy*, *DB2Advis*, *DTA*, and *Extend* find the best solutions for small budgets. However, for larger budgets, the lack of multi-column indexes and the limit of two indexes per query (both limitations due to complexity) become apparent for *CoPhy*. At $\approx$ 3.5 GB *Relaxation* and *Extend* start identifying the best solutions up to the largest evaluated budget. As for TPC-H and TPC-DS, *Dexter*'s solutions are coarse-grained.

Figure 3(f) shows similar results compared to the algorithm runtimes for TPC-DS. However, there are a few interesting aspects to note here. While *AutoAdmin* multiplies its runtime, *Drop* almost halves, and the runtime of *Extend* slightly decreases. The lower number of attributes can explain the last two observations. Thereby, fewer index candidates are generated by *Drop* and *Extend* resulting in fewer what-if optimizer calls per step. The increase in algorithm runtime for *AutoAdmin* can be explained by a higher number of beneficial single-column indexes compared to TPC-DS.

6.5 Cost Breakdown and Cost Requests

Most of the runtime of what-if based index selection algorithms is usually not spent on the algorithm logic, but on cost estimation calls to the what-if optimizer [14, 35]. Usually, algorithms request a cost estimate for a particular query and a given index combination from the (what-if) optimizer. These requests are expensive, but the estimated costs remain the same if neither the query and index combination nor the underlying data change. Therefore, most algorithms cache cost estimation calls. As mentioned in Section 4.2 and Section 5.1, with the developed framework, all algorithms use the same cost estimation implementation, and hence, the same caching mechanisms. However, the different strategies of the investigated algorithms result in varying cost request patterns and different cache opportunities, which is demonstrated in Table 3 for a budget of 5 GB for the TPC-DS benchmark. The platform allows generating this table for further budgets, configurations, and benchmarks.

The runtimes vary significantly, ranging from seconds to more than an hour, caused by different underlying approaches. Factors – besides the modus operandi of the algorithm – that influence the runtime include the number of evaluated configurations, simulated (hypothetical) indexes, cost requests, and cache rates. Table 3 indicates that the interplay of these factors is responsible for the resulting runtime.

Generally, for all algorithms, most of the runtime is spent on cost requests (what-if optimization calls), *Drop* being the only exception. However, most algorithms achieve high cache rates caused by the fact that they repeatedly evaluate similar configurations. The similarity of the evaluated configurations influences the cache rate. Note, retrieving costs from the cache does not come for free since the configuration must be looked up in the cache, and its cost must be obtained.

DB2Advis and *Dexter* evaluate only two configurations. This behavior leads to few cost requests and low runtimes. For wider indexes, runtimes can increase dramatically as later discussed in this section and demonstrated in Table 4.

Drop and *Relaxation* diverge in the number of evaluated configurations but generate the largest number of cost requests. They are the two slowest approaches and follow – in contrast to all other approaches – the same general concept: they start with an extensive configuration and reduce it until it fits the given budget. For realistically sized budgets, this behavior leads to long runtimes.

Besides, according to Table 3, the impact of index simulation is almost negligible. Naive-2 simulates more than 70 000 indexes, which is only responsible for 2% of its runtime.

Furthermore, it is essential to note that not all cost requests are equally expensive. Query planning time depends on the query complexity and the number of available (what-if) indexes. Few available indexes lead to planning times in the range of milliseconds for our implementation. *AutoAdmin*, *CoPhy*, *DTA*, *Drop*, and *Extend* usually request costs for small index sets of a few dozen indexes at maximum. However, the modi operandi of *DB2Advis* and *Relaxation* can lead to large index sets for the cost requests, because costs are evaluated for all possibly applicable indexes per query

Table 3: Algorithm cost breakdown for the TPC-DS (SF 10) benchmark. Storage consumption ≈ 5 GB. *Configurations* are the number of uniquely evaluated configurations. *Index simulations* refer to the number of non-unique created hypothetical indexes. *Simulation* and *Costing* refer to the share of runtime that was consumed by index simulations or cost requests. *Naive-2* relates to *AutoAdmin* with two indexes selected by a naive enumeration. *Dexter's* original implementation does not provide all runtime details.

Algorithm	Configurations	Index simulations	Cost requests			Runtime		
			Total	Non-cached	Cache rate	Total	Simulation	Costing
AutoAdmin	129	10 991	33 851	11 676	65.5%	2.1m	2.0%	95.9%
Naive-2	816	73 504	240 441	73 440	69.4%	15.3m	2.0%	66.5%
CoPhy	3 983	3 982	394 317	52 177	86.8%	10.1m	0.6%	94.9%
DB2Advis	2	7 179	180	180	0.0%	0.1m	24.0%	58.7%
DTA	1 442	25 812	1 650 510	129 811	92.1%	32.2m	0.4%	87.2%
Dexter	2	3 982	180	180	0.0%	0.4m	n/a	n/a
Drop	203	29 144	2 601 450	18 348	99.3%	35.0m	0.6%	19.7%
Extend	594	11 295	812 430	53 472	93.4%	12.8m	0.5%	84.1%
Relaxation	1 898	51 680	2 982 690	170 863	94.3%	60.7m	0.4%	66.6%

(see Sections 3.4 and 3.5). Table 4 shows the cost request time for two complex queries of the TPC-DS benchmark. Cost estimations for large index combinations can take prohibitively long, which becomes especially pronounced when wide indexes are searched.

6.6 Parameter Influence

In this subsection, we investigate the influence of the index width, the time for *DB2Advis*' `TRY_VARIATION` step, *AutoAdmin*'s number of indexes that are selected by naive enumeration, and *DTA*'s runtime limits on the solution quality and runtime. The experiments were conducted with TPC-DS (scale factor 10) for budgets between 4 and 8 GB. Other benchmarks did not yield substantially different results.

Index Width. In our evaluations, the maximum number of columns per index does not have a huge influence for the investigated workloads on PostgreSQL. The impact could be higher for real-world workloads, e.g., enterprise systems utilize wide primary keys of up to 16 columns [17] and more sophisticated query processors might use wide indexes more efficiently. The maximum index width selected by algorithms during our experiments was 6. *AutoAdmin* and *Extend* show performance improvements of 1 to 3% when the index width is increased from 1 to 2 or 3. Improvements by wider indexes are below 1%. For *DB2Advis*, we found an anomaly: the performance improved by about 1% when the index width is increased from 1 to 3, but dropped by ≈ 5% when set to 2.

For *Extend*, runtimes did not significantly increase due to its functioning (Section 3.8). *DB2Advis*' runtime is comparable for index widths of 1 and 2 but grows by more than 10× when increased to 3 and is not feasible for larger numbers due to expensive cost requests for large index combinations (see Section 6.5). The runtimes of *AutoAdmin* and *DTA* increase with wider indexes, e.g., by a factor of 2 - 3 from single- to two-column indexes, because for each admissible index width, the enumeration steps are executed for all queries. As the candidate volume decreases per round because candidates are only drawn from previously beneficial indexes (Section 3.2), the increase in runtime declines over time.

DB2Advis - Try Variations. The measurable impact of *DB2Advis*' `TRY_VARIATION` step is often small. According to our experiments, variation achieves improvements of about 1% in workload cost, even when the core algorithm ran in ≈ 1 and the variation in 30 seconds. For massive candidate sets, variations can only be effective with a long runtime because chances, that beneficial candidates are becoming part of the final combination, decrease with a larger population.

AutoAdmin - Naive Enumerations. In our experiments, the number of indexes selected by naive enumeration affects the approach's solution quality only marginally. Sometimes, we even observed better results for smaller numbers. However, the runtime is significantly affected. For the conducted benchmarks, the runtime increased by factors between 3 and 10 when the indexes selected by naive enumeration were increased from 1 to 2. More indexes could not be selected via naive enumeration in an acceptable time.

DTA - Runtime Limits. We observed both aspects of the *anytime* ability of the approach: (i) returning acceptable solutions when interrupted and (ii) identifying improved solutions with longer runtime. For the TPC-DS benchmark (budget: 5 GB, maximum index width: 2), our *DTA* implementation generates ≈ 300 seeds. The first seed's calculation finishes after 9 minutes and identifies a solution that is within 3% of the best solution found after 14 hours.

Table 4: Cost request timings including index simulation for two TPC-DS queries; DNF exceeds 30min.

Index width	Relevant indexes		Time	
	Query 13	Query 64	Query 13	Query 64
1 column	22	49	13ms	12ms
2 columns	132	287	97ms	44ms
3 columns	870	1 889	33s	5s
4 columns	5 910	14 393	DNF	231s

7. LEARNINGS

In this section, we summarize general and algorithm-specific properties to explain the performance differences observed in the experimental evaluation. We give final recommendations on when to use which approach in Section 8. The following insights are only generalizable to a certain degree. They might not be transferable to other systems that rely on different optimizers and execution engines.

General Insights. (i) For different workloads or budget restrictions, different algorithms can perform best. (ii) The algorithm should be chosen based on the user's needs. Differences in runtime, solution quality, solution granularity, and multi-column index support are significant. (iii) There are two kinds of approaches: query-based (*DB2Advis*, *Dexter*) ones, which evaluate the benefit of all possible indexes at once, and index combination-based (*Extend*, *AutoAdmin*, *Drop*, *CoPhy*, *DTA*) approaches, which evaluate the benefit of comparably small index sets. Generally, the latter consider index interaction to a higher degree while query-based ones are orders of magnitude faster as long as the set of evaluated indexes per query is not too large (see Section 6.6). *Relaxation* combines both approaches. (iv) The cost of what-if calls or cost requests are not fixed, they depend on the query and evaluated index combination (see Section 6.5).

(v) The granularity of identified solutions is fundamental. Otherwise, the performance between identified solutions is suboptimal (cf. Figure 3(c)). (vi) Stop criteria are essential. Algorithms that halt after a maximum number of indexes, usually start with large indexes and do not find small indexes with a significant relative impact (cf. Figure 3(a)).

(vii) Ordering index candidates by benefit per space instead of purely by benefit showed to be efficient, especially for medium-sized budgets (see Section 6.3). This advantage could vanish for transactional workloads. Usually, benefit per space approaches favor configurations with many small indexes, which could lead to elevated maintenance costs and lock contention. Therefore, choosing the minimization goal accordingly might be beneficial (see Table 1).

(viii) Reduction-based approaches, i.e., *Drop* and *Relaxation*, become faster with larger budgets. However, for realistic budgets, their runtimes are rather long.

Drop. (i) The repetitive functioning causes a large number of cost requests, while it shows the highest cache rates (see Section 6.5). (ii) An extension to support multi-column indexes would result in an infeasible number of index candidates requiring some kind of pre-selection.

AutoAdmin. The approach finds reasonable solutions, but two properties weakened the algorithm in our evaluation. (i) The number of indexes selected by naive enumeration only had a minor influence on the solution quality. However, the runtime impact was enormous (3-10×, see Section 6.6). (ii) By ignoring the size of indexes, a multi-column index with an only slightly higher benefit than a single-column index but significantly higher memory consumption is favored, which acts in opposition to storage efficiency.

DB2Advis. (i) *DB2Advis*' solutions are reasonable for high budgets, while the runtime is low. In such cases, the index combination is optimized for every query. (ii) For wider indexes, e.g., $W \geq 3$, the runtime increases while the solution quality is not among the best. (iii) Random exchanges by `TRY_VARIATION` do not prove to be very effective (see Section 6.6). (iv) The approach may miss a locally inferior but globally superior index, e.g., indexes that are beneficial for many queries but never the best for an individual query.

Relaxation. (i) Large candidate sets and its transformation rules lead to an enormous number of evaluated configurations and, thereby, to the best consideration of index interaction. (ii) Thus, for large budgets, *Relaxation* performed best in the evaluated benchmarks. (iii) The reductive functioning of the approach causes long runtimes for realistic budgets.

CoPhy. (i) The solution quality depends on well-chosen initial candidates and suitable choices of index combinations per query. TPC-H and TPC-DS selections improved with more candidates (wider indexes), whereas JOB improved with a higher number of indexes per query. (ii) With more candidates, cost requests dominate the runtime while the number of indexes per query impacts the solver runtime.

Dexter. Dexter identifies suitable index combinations and offers low, constant runtimes. However, the granularity of solutions is generally too coarse (cf. Figure 3(b)).

Extend. (i) *Extend* is the only algorithm that identified wide indexes ($W \geq 4$) in an acceptable amount of time. Due to its mechanics, the index width does not have a significant impact on the runtime (see Section 6.6). (ii) The runtime can decrease for larger budgets because fewer but larger indexes are identified. (iii) The tradeoff between solution quality and runtime is among the best for various budgets.

DTA. (i) The large number of seed configurations guarantees suitable configurations: *Anytime's* solutions, especially for small to medium-sized budgets, are usually among the best. (ii) Its runtime can be on par with most other approaches when the ability to interrupt the algorithm at *anytime* is considered. As shown in Section 6.6, the first seeds enable competitive solutions, while later improvements are often insignificant for the evaluated scenarios.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we compared eight different index selection algorithms. We evaluated the approaches regarding solution quality, runtime, solution granularity, and complexity with an extensible evaluation platform that promotes reproducibility.

Our recommendations for different scenarios and user needs are as follows. These conclusions are based on our evaluation on PostgreSQL. Their transferability to other engines might be limited. If fast solutions for large budgets are desired, and the index width is limited, *AutoAdmin* and particularly *DB2Advis* are advisable. If a higher runtime is feasible, we recommend *Relaxation*. For small problem sizes where all relevant candidate combinations can be exhaustively enumerated, *CoPhy*'s LP guarantees optimal solutions for any budget. *Dexter* is a simple, ready-to-use open-source option with low overhead. Compared to other algorithms, *Drop* is the easiest to implement. Overall, the recent approaches *Extend* and *DTA* provide the best combination of runtime and solution quality in a wide range of scenarios.

Our experiments showed that an efficient index selection remains challenging as, due to the problem's nature and the functioning of the approaches, different weaknesses surface in specific scenarios, which leaves room for improvement in future work. The revealed strengths and weaknesses of the algorithms make it possible to derive improved solution concepts, e.g., hybrid approaches and robust adaptions. Our platform facilitates such improvement work. For example, in the future, machine learning-based approaches that showed to be not competitive yet could be integrated into our extensible platform after becoming more mature.

Acknowledgments

We thank S. Chaudhuri, V. Narasayya, J. Rouhaud, A. Sharma, and D. Zilio for their support and detailed answers.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft SQL server 2005. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1110–1121, 2004.

[2] D. Basu, Q. Lin, W. Chen, H. T. Vo, Z. Yuan, P. Senellart, and S. Bressan. Cost-model oblivious database tuning with reinforcement learning. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, pages 253–268, 2015.

[3] R. Borovica, I. Alagiannis, and A. Ailamaki. Automated physical designers: what you see is (not) what you get. In *Proceedings of the International Workshop on Testing Database Systems (DBTEST)*, 2012.

[4] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 227–238, 2005.

[5] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 826–835, 2007.

[6] N. Bruno and R. V. Nehme. Configuration-parametric query optimization for physical design tuning. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 941–952, 2008.

[7] A. Caprara, M. Fischetti, and D. Maio. Exact and approximate algorithms for the index selection problem in physical database design. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(6):955–967, 1995.

[8] A. Caprara and J. J. Salazar González. A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem. *TOP*, 4:135–163, 1996.

[9] S. Chaudhuri and V. Narasayya. Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server. `https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server`, visited 2020-06-04, June 2020.

[10] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 146–155, 1997.

[11] S. Chaudhuri and V. R. Narasayya. Index merging. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 296–303, 1999.

[12] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 3–14, 2007.

[13] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri. Automatically indexing millions of databases in microsoft azure SQL database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 666–679, 2019.

[14] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6):362–372, 2011.

[15] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya. AI meets AI: leveraging query executions to improve index recommendations. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1241–1258, 2019.

[16] G. Farley and S. A. Schuster. Query execution and index selection for relational data bases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, page 519, 1975.

[17] M. Faust, M. Boissier, M. Keller, D. Schwalb, H. Bischoff, K. Eisenreich, F. Färber, and H. Plattner. Footprint reduction and uniqueness enforcement with hash indices in SAP HANA. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, pages 137–151, 2016.

[18] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems (TODS)*, 13(1):91–128, 1988.

[19] F. Fotouhi and C. E. Galarce. Genetic algorithms and the search for optimal database index selection. In *Proceedings of the First Great Lakes Computer Science Conference*, pages 249–255, 1989.

[20] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Thomson/Brooks/Cole, 2003.

[21] M. R. Frank, E. Omiecinski, and S. B. Navathe. Adaptive and automated index selection in RDBMS. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 277–292, 1992.

[22] G. Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, 2006.

[23] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1–8, 1976.

[24] M. Y. L. Ip, L. V. Saxton, and V. V. Raghavan. On the selection of an optimal set of indexes. *IEEE Transactions on Software Engineering*, 9(2):135–143, 1983.

[25] A. Kane. Dexter - The automatic indexer for Postgres, June 2017. `https://github.com/ankane/dexter`, visited 2020-06-04.

[26] A. Kane. Introducing Dexter, the Automatic Indexer for Postgres, June 2017. `https://medium.com/@ankane/introducing-dexter-the-automatic-indexer-for-postgres-5f8fa8b28f27`, visited 2020-06-04.

[27] 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 *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 715–730, 2017.

[28] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[29] V. Y. Lum and H. Ling. An optimization problem on the selection of secondary keys. In *Proceedings of the*

1971 26th Annual Conference (ACM '71), pages 349—356, 1971.

[30] R. C. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *PVLDB*, 12(11):1733–1746, 2019.

[31] Microsoft. SQL Server 2019 - Database Engine Tuning Advisor, January 2017. `https://docs.microsoft.com/en-us/sql/relational-databases/performance/database-engine-tuning-advisor?view=sql-server-ver15`, visited 2020-06-04.

[32] R. O. Nambiar and M. Poess. The making of TPC-DS. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1049–1058, 2006.

[33] F. P. Palermo. A quantitative approach to the selection of secondary indexes. In *IBM Research RJ 730*, 1970.

[34] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 442–449, 2007.

[35] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient use of the query optimizer for automated database design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1093–1104, 2007.

[36] G. Piatetsky-Shapiro. The optimal selection of secondary indices is NP-complete. *SIGMOD Record*, 13(2):72–75, 1983.

[37] M. Pöss and C. Floyd. New TPC benchmarks for decision support and web commerce. *SIGMOD Record*, 29(4):64–71, 2000.

[38] J. Rouhaud. HypoPG - Hypothetical Indexes for PostgreSQL, March 2015. `https://github.com/HypoPG/hypopg`, visited 2020-06-04.

[39] Z. Sadri, L. Gruenwald, and E. Leal. Online index selection using deep reinforcement learning for a cluster database. In *Proceedings of the International Conference on Data Engineering Workshops (ICDEW)*, pages 158–161, 2020.

[40] K. Sattler, I. Geist, and E. Schallehn. QUIET: continuous query-driven index tuning. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1129–1132, 2003.

[41] M. Schkolnick. The optimal selection of secondary indices for files. *Information Systems (IS)*, 1(4):141–146, 1975.

[42] R. Schlosser and S. Halfpap. A decomposition approach for risk-averse index selection. In *Proceedings of the 32th International Conference on Scientific and Statistical Database Management (SSDBM), forthcoming*, 2020.

[43] R. Schlosser, J. Kossmann, and M. Boissier. Efficient scalable multi-attribute index selection using recursive strategies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1238–1249, 2019.

[44] K. Schnaitter and N. Polyzotis. A benchmark for online index selection. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1701–1708, 2009.

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

[46] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 23–34, 1979.

[47] A. Sharma, F. M. Schuhknecht, and J. Dittrich. The case for automatic database administration using deep reinforcement learning. *CoRR*, abs/1801.05643, 2018.

[48] 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 *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 101–110, 2000.

[49] K. Whang. Index selection in relational databases. In *Proceedings of the International Conference on Foundations of Data Organization (FoDO)*, pages 487–500, 1985.

[50] 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 *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1081–1092, 2013.

[51] 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 *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1567–1581, 2016.

[52] 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 *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1087–1097, 2004.