The Index Selection Problem with Configurations and Memory Limitation: A Scatter Search approach

Raslan Kain[a], Daniele Manerba[b1], Roberto Tadei[c]
[a]Dept. of Electrical and Computer Engineering, American University of Beirut, Beirut (Lebanon)
[b]Dept. of Information Engineering, Università degli Studi di Brescia, Brescia (Italy)
[c]Dept. of Control and Computer Engineering, Politecnico di Torino, Turin (Italy)

Abstract

Within the physical designing process of relational databases, the Index Selection Problem aims at finding the subset of indexes to build for accessing the stored information. More precisely, given a database workload, each query must be served by at most one predefined set of indexes (named *configuration*) to maximize the net gain in terms of time. This gain is made up of the time gain obtained to serve the queries through the configurations minus the fixed time needed to create and maintain those configurations. The clustering of indexes into configurations and the limited amount of memory available to store the indexes characterize our variant of the problem. At the same time, established approaches in the literature have only considered those two aspects separately. We model this setting as a generalization of the Uncapacitated Facility Location Problem with budget constraint and propose an Integer Linear Programming formulation for it. Then, to find near-optimal solutions in a reasonable computational time, we develop a Scatter Search meta-heuristic exploiting the specific facility location features of the problem. We test our algorithm over a broad set of benchmark instances and compare it with an exact solver and an efficient state-of-the-art heuristic method.

Keywords: physical database design, index selection problem with configurations, facility location problem, memory limitation, scatter search.

1. Introduction

The physical design is a fundamental step in developing any structured data storage, e.g., a relational database (DB). Its main objective is to translate a logical data model (e.g., an entity-relation scheme) into the technical specifications needed to create the relative DB structure on a physical machine or computational device. This translation consists in defining an appropriate set of structures to access the stored information, offering a good compromise between memory occupation and the time required for data retrieval and maintenance. From a more technical point of view, the design depends upon the database management system (DBMS) used on the target device and, in turn, its specific features. For example, the supported access structures (such as

[1]Corresponding author: daniele.manerba@unibs.it
via Branze 38, 25123 Brescia, Italy. Phone: +39 030 371 5935.
Other e-mail addresses: rhk44@aub.edu.lb (R. Kain), roberto.tadei@polito.it (R. Tadei).

hash functions, *links*, *inverted indexes*, *clustering of data*), as well as the implemented methods for accessing the data (such as *join algorithms*, *index intersection methods*), must be taken into account. Moreover, the access structures must deal with multiple requests (queries) on the DB, each one possibly having its impact. It follows that the task at hand can get very complex, despite the limited set of features offered by a DBMS.

Information in a DB table can be accessed by scanning the entire table, but, in that case, the access time needed could be too high. Ad-hoc access paths called *indexes* are commonly used to reduce such a time. In most modern and advanced DBMSs, there also exists the possibility of combining several indexes into predefined sets called *index configurations* to gain efficiency by exploiting the joint action of different, but in general few, indexes. According to Bruno (2011), the complete DB index selection process is a complex activity composed of three main tasks: candidate selection, cost model definition, and optimal assignment. In the first task, the search space needs to be characterized by finding the candidate physical structures (single indexes or configurations) that could be selected later. In the second task, a cost model must be defined to evaluate and compare the different candidates. In the third task, candidate physical structures (or a subset of them) must be created and assigned to a precise set of queries (*workload*) to maximize the efficiency of the system response. In practice, each of the above tasks is a non-trivial problem itself. For example, the candidate selection is commonly affected by a combinatorial explosion of the physical structures to consider. Therefore, candidate access paths are almost always heuristically chosen based on the configuration of each input query. Moreover, the tasks are somehow linked together, and, in end-to-end implementation of an automated physical design system, they should be treated as a whole. For instance, the gains/costs are commonly obtained through simulations named *what-if calls* that rely on the results coming from the assignment optimization itself (see, e.g., Bruno and Chaudhuri, 2005). Finally, specific additional constraints or limitations may affect each task. However, in this work, we will focus only on the final optimal assignment task, which corresponds to a well-studied combinatorial optimization setting called *Index Selection Problem* (ISP). We will assume that a candidate set of physical structures has already been found, and the costs and gains for each candidate have been estimated. Note that our optimization approach will not assume any specific condition on developing the two previous tasks. Therefore, it can be coupled with any available approach tailored to define the candidate structures and the cost model.

The classical ISP is about maximizing the gain in terms of execution time minus the time required to create and maintain the selected indexes while ensuring to access the data that at most one index will be used for each query. When configurations are allowed, the so-called *Index Selection Problem with Configurations* (ISPwC) aims at finding a subset of configurations maximizing the gain in terms of execution time (now depending on the configurations used) minus the time required to create and maintain the indexes needed to activate the selected configurations. Moreover, at most, one configuration can be assigned to each query. In this work, we study an ISPwC, but we also explicitly consider a limited amount of memory available to store the indexes to make the problem more realistic. This paper is the first to deal simultaneously with the clustering of indexes into configurations and memory capacity to the best of our knowledge. According to Caprara et al. (1995), the ISP can be formulated as an Uncapacitated Facility Location Problem (UFLP) and, therefore, our ISPwC results to be a generalization of the UFLP with budget constraint. Note that the memory limit is not a marginal aspect. It has been considered as one of the significant features to study in the context of the automated physical DB design in many previous works (see, e.g., Bruno and Chaudhuri, 2005, Dash et al., 2011, Schlosser et al., 2019). This feature's significance comes partly because, intrinsically, there is a limited amount of available memory space in any physical system. More importantly, in a DBMS, the memory used to store the indexes should not compromise its capacity to store data and their relations. Usually, storage capacities are associated with a certain percentage of the total memory available for the DBMS. Such capacity could represent an interesting parameter for developing a sensitivity analysis in DB performances (Schlosser et al., 2019). Nowadays, computational devices tend to become smaller and smaller in terms of physical size and memory capacity to be easily embedded into other components. Then, the memory limitation represents a challenging aspect of today's industrial applications and the future. On the other hand, it is also shared that the DB workloads have several thousands of queries and that the DBMS can manage an enormous number of indexes and possible configurations. While, in general, index selection optimization is addressed by quick-and-dirty heuristics, there is a practical need for more effective and robust methods.

To the best of our knowledge, this is the first paper in the literature providing a reliable and effective meta-heuristic solving the ISPwC with memory limitation. More precisely, we develop a Scatter Search meta-heuristic to find near-optimal solutions in a reasonable computational time. We test our method on a broad set of instances generated through benchmark procedures, demonstrating its accuracy, efficiency, and flexibility concerning different instances' features and sizes. A comprehensive comparison with a commercial solver and a state-of-the-art heuristic is also provided. Our algorithm has turned out to be able to deal efficiently with instances representing a workload composed of up to 5000 queries and a DBMS supporting up to 5000 indexes organized in up to 50000 different configurations. Therefore, the method is suitable to be embedded into realistic DB design software to improve its efficiency or even benchmark other procedures' quality through off-line simulations. Finally, our heuristic can be used to solve different problems with a similar combinatorial structure, e.g., simultaneous selecting and assigning problems. The paper is organized as follows. Section 2 reviews the scientific literature related to the problem under consideration. In Section 3, we present the ISPwC and its mathematical programming formulation, while in Section 4, we describe in detail our algorithmic approach based on the Scatter Search meta-heuristic. Section 5 is devoted to presenting and discussing the computational experiments conducted to demonstrate our solution algorithm's accuracy and efficiency. Finally, Section 6 concludes the paper and sketches some promising future research lines.

2. Literature review

The physical DB design is an essential issue for commercial DBMS, and great attention has been dedicated to making it efficient both from academics and practitioners.

Some classical approaches related to specific commercial products also focusing on practical issues like the DB workload characterization. Chaudhuri and Narasayya (1997) addressed the optimization of an end-to-end system tailored for *Microsoft*$^{®}$ *SQL Server* in which indexes are combined into configurations, as in our setting. The goal is to select the optimal configurations set for a given workload while respecting an upper bound on the index number. Three stages compose the approach. The first stage reduces the number of indexes to be considered by maintaining only the configurations that contain a specific subset of *atomic configurations* (defined on workload properties and interaction among indexes). The second stage selects the minor cost configuration for each query independently and creates the related indexes without exceeding the cardinality constraint. The third stage combines single-column indexes into multi-column indexes through an iterative approach. Instead, Valentin et al. (2000) implemented an index recommendation system called *DB2 Advisor* for relational DBs. The ISP is simply modeled as a Knapsack Problem (KP) variant. The solving approach uses a combination of two algorithms recursively. The first one defines so-called *virtual indexes* (i.e., indexes whose statistics are temporarily introduced into the schema only for the optimization process) by analyzing query of predicates (`equal`, `join`, `range`, etc.) and clauses. The second one, instead, enumerates all possible indexes. A final greedy approach selects the indexes with the best benefit-to-cost ratio until a memory limit is reached. Yu et al. (1992) developed a *Relational Database Workload Analyzer* (REDWAR) to study the structure and the complexity of SQL statements, the makeup of transactions and queries, and composition of relations and views in a *IBM*® *DB2*-like environment. Statistics are gathered on query types and usage, as well as data access patterns. A clear output of this analysis is the massive effect of variations on the indexes selection's optimality in response time and the number of tuples examined before getting a qualified one. More recently, Boronski and Bocewicz (2014) observed that several commercial DBMS's (e.g., *Oracle Access Advisor* or *Toad*) do not consider relationships between group of queries for the index selection process. So, an extended ISP for grouped queries is addressed by an evolutionary algorithm. The approach reduces, over many DBs and query groups, the execution time and the number of indexes built with respect to the *Oracle Access Advisor* tool. However, the algorithm's average CPU time is several hours (or even days), making it impossible to implement commercial software.

Several other works have studied different ISP variants under various assumptions and various modeling and solving approaches. Comer (1978) addressed selecting a minimum-size set of attributes that can serve as the key for a file on secondary memory. Fotouhi and Galarce (1989) solved the same problem in the ISP context using a Genetic Algorithm and a Learning System model to observe the DB request patterns and change the existing choices dynamically. Finkelstein et al. (1988) considered a set of clustered and non-clustered indexes on DB tables, assuming that a table has no more than one clustered index, and no columns have both clustered and non-clustered indexes. Given a set of tables and a set of statements (together with the relative expected frequencies of use), their ISP problem consisted in selecting, for each table, the ordering rule for the stored records (which determines the clustered indexes) and a set of non-clustered indexes, so to minimize the total processing cost. A limit on the entire index space is also considered. The authors develop a methodology called *DBDSGN*, which mixes heuristics and exact method procedures. Other similar ISPs have been proposed in Choenni et al. (1993), Gupta et al. (1997), and Heeren et al. (2003), where straightforward cost models (i.e., where the size of the smallest covering index represents the cost of a query) are assumed. However, when answering some queries through a configuration of several indexes together, there exist interactions between indexes and, therefore, the cost model must take into account such inter-dependencies. Later, Chaudhuri et al. (2004) considered a set of already-chosen candidate indexes partitioned into different sets corresponding to the DB tables where the benefit of a set of indexes is calculated through a sophisticated cost model that embeds indexes correlations. The problem aims at finding the set of indexes that maximizes the benefit in terms of response time for query requests, selecting at most one index per table, and not exceeding memory storage. The problem, modeled as a KP and proved to be $\mathcal{NP}$-hard for both clustered and non-clustered indexes, is solved using a greedy approach, in addition to a pruning strategy for the relevant index subsets, to minimize the number of calls to the optimizer for calculating the benefits. Asgharzadeh Talebi et al. (2013) proposed an integer programming model for a combined view-and-index selection problem in the context of on-line analytical processing (OLAP) in DBMSs. The problem aims to select a subset of views and indexes to minimize the evaluation time for a given collection of queries and considering a storage limit. The cost model is similar to the one defined by Gupta et al. (1997). There, the cost of answering a query using a view without indexes (i.e., the number of rows of the portion of the view that must be scanned to construct the query result) is added to the cost of answering with a view and an index (i.e., the number of rows of the portion of the view referenced by the index). The problem is tackled through a plain solver after a significant reduction of the search space by using both formal properties and a greedy heuristic method. Ameri et al. (2015) presented a data mining-based approach to address the ISP for non-relational DBs. A candidate set of indexes is obtained using different mining algorithms that analyze frequent queries and then use the DB's query optimizer to select optimal indexes. Together with their type, indexes are ordered according to rules based on the properties and the queries' particular needs. Ameri (2016) continued the previous work by discussing the implementation of a refined and parallel mining algorithm that aims to minimize the response time based on the read-write ratio of workload operations, the selectivity of attributes, and memory limitations. The ordering rules give higher priority to indexes with many served queries and with a lower number of write operations. Moreover, the DB query optimizer is helped by a Support Vector Machine algorithm that can extract a representative but a limited sample of the DB to perform the final selection. Finally, Subotić et al. (2018), proposed an automated index selection scheme to achieve optimal memory usage for *Datalog* programs. Such programs work on relations and on indexes that are based on attribute sequences. Since the relations are stored as in-memory index-organized tables to reduce lookup times, an ISP (with some peculiarities) arises. In particular, the authors address a *Minimum Index Selection Problem* seeking the minimum set of indexes covering frequently repeated calls to reduce creation and maintenance costs and the associated memory occupation. The problem is solved by computing a chain cover of the searches that exploits the relationship between the indexes and the chains' search spaces using a lexicographic order.

The research stream focused on solving the ISP as a UFLP started with the work by Caprara et al. (1995), which contains the assumptions needed to justify the coherence of such a combinatorial structure. The authors provide a branch-and-bound algorithm improved with the introduction of ad-hoc valid inequalities. Lower bounds are enhanced by using two heuristic approaches that were compared to the best available solution. The first heuristic selects indexes that maximize the global likelihood of having those indexes in an optimal solution. In contrast, the second sorts indexes by decreasing the objective function's value and enumerates over subsections of the ordered indexes. However, the algorithm can tackle only small instances for the current industrial needs. Caprara and Salazar González (1996) generalized the above problem by introducing the configuration concept to serve the queries, with precisely the same meaning as in our paper, but relaxing the memory limitation constraint. By reformulating the problem as a Set Packing, the authors derive strong valid inequalities and develop an effective branch-and-cut method. In a later paper (Caprara and Salazar González, 1999), the same authors proposed an effective exact separation procedure for a well-defined family of lifted odd-hole inequalities. More recently, Kratica et al. (2003) proposed a Genetic Algorithm for the same problem. The method adopts an elitist approach in the generational replacement of the population of solutions and a *Least Recently Used* caching strategy to shorten objective value evaluation time. The results show excellent CPU time for challenging instances of the problem while maintaining a good quality of the obtained solutions. However, this algorithm cannot solve our ISPwC, which must also comply with a limited amount of memory. In this paper, we will follow this facility location-based modeling perspective, which is also particularly tailored for deriving insightful analyses (see Fadda et al., 2021).

Some other works share some similarities with ours. Dash et al. (2011) addressed an index tuning problem in which configurations of indexes are obtained from a heuristically generated pool of candidates. An integer linear programming model taking advantage of the inherent structure of the solutions space and a method that employs a linearly composable query cost function to speed up what-if calls (as in Papadomanolakis et al., 2007) are used. The approach, called *CoPhy*, which uses an off-the-shelf solver and a Lagrangean relaxation technique, results in general but can address only instances up to 1000 queries. Very recently, Schlosser et al. (2019) employ a recursive-based heuristic (RH) into an end-to-end system to solve massive ISP instances (where hundreds of tables in a DB are involved) and without limiting the pool of index candidates in the search. The approach also considers the interactions between indexes by taking advantage of *multi-attribute indexes*, where each attribute relates to a specific column in the DB table. RH works by first selecting the single-attribute index with the best ratio of the cost in serving a query to its memory occupation. Then, the solution set is recursively updated by choosing a cost-improving index attribute greedily to be considered either as a single-attribute index or as part of a multi-attribute one. RH thus reduces by construction the number of what-if calls needed to estimate the costs for an index configuration and outperforms CoPhy for large problem instances, obtaining faster solutions with better quality.

To the best of our knowledge, the present paper is the first one dealing simultaneously with clustering of indexes into configurations and limited storage memory. Moreover, RH by Schlosser et al. (2019) seems to be the state-of-the-art heuristic for our problem. Hence we will use it as a benchmark method to assess the quality and efficiency of our SS.

3. Problem definition and formulation

In this section, we formally present our ISPwC and its formulation. As already said, our focus is on the optimal assignment part of the complex process constituting the physical design of a DB. Therefore, we assume that the candidate access structures (indexes and configurations) and the cost function have already been defined.

3.1. Problem statement

Let Q be the set of queries composing the DB workload, and I be the set of possible indexes that can be built to serve the queries. Since not only single indexes but also sets of indexes can be used for a query, let C be the set of the possible predefined configuration of indexes. The set of indexes composing a specific configuration is the subset $I_c \subset I$ and, in turn, the set of configurations composed by a particular index is $C_i = \{c \in C : i \in I_c\}$. A configuration c is called *active* if all its indexes $i \in I_c$ are built. A configuration can be used to serve a query only if it is active. Note that a single index cannot be used anymore to serve a query directly. However, configurations composed of only one index are allowed. Each index $i \in I$ can be built using $m_i > 0$ units of memory at a fixed cost $f_i > 0$, corresponding to the time needed for its creation and maintenance. Moreover, let $g_{cq} \geq 0$ be the gain in terms of execution time of using index configuration $c \in C$ for query $q \in Q$. This gain is calculated for the time needed to serve the same query without the support of any single index or configuration. Given the variety of queries and configurations possibly existing in realistic DBs, the gain is a measure of how much a configuration of indexes is tailored to a specific query type's requirements. The gain values for all possible configurations and queries are estimated during the cost model definition phase (see Section 1). A null gain $g_{cq} = 0$ either means that configuration c cannot be used for query q or, simply, that it does not affect its execution time.

Our ISPwC aims at selecting the indexes to build for serving the DB workload, ensuring that, at most, one active index configuration is used for each query and the total memory usage does not exceed a predefined maximum capacity M. The objective is to maximize the net time gain (i.e., the total gain obtained by serving queries through the configurations activated minus the indexes creation and maintenance times) necessary to execute all the DB workload queries.

An example of the problem and a feasible solution structure is given in the following.

Example 1. *Let us consider a sample instance of the ISPwC (graphically represented in Figure 1) involving a set of 5 indexes $I = \{i_1, i_2, i_3, i_4, i_5\}$, a set of 4 configurations $C = \{c_1, c_2, c_3, c_4\}$, a DB workload composed by a set of 3 queries $Q = \{q_1, q_2, q_3\}$, and a memory capacity $M = 300$. In particular, above each index (represented by a circle in the upper tier), the relative fixed cost*

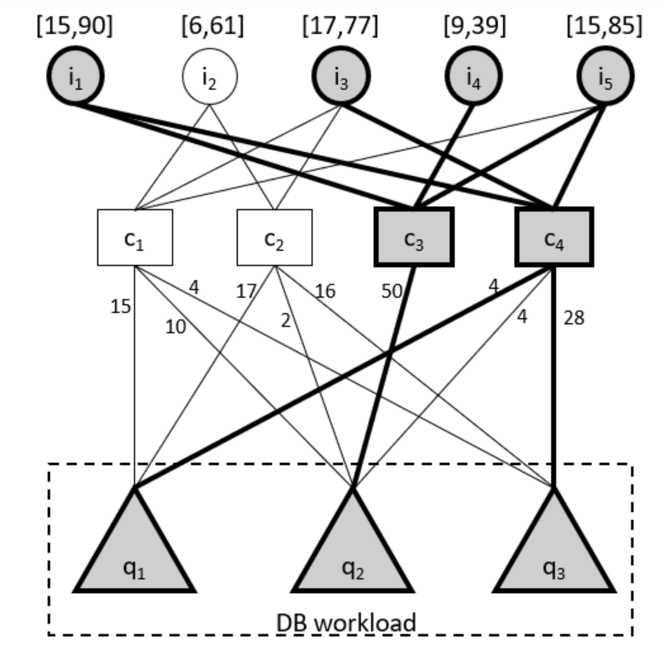


Figure 1: A sample ISPwC instance and its optimal solution

f_i memory occupation m_i are reported in square brackets. In the medium tier each configuration

is represented as a rectangle. The links between the indexes and the configurations represent the configuration composition, i.e., a link exists between index i and configuration c if and only if $i \in I_c$. In the lower tier each query composing the DB workload is represented as a triangle. Links between a configuration and the queries represents strictly positive gains, i.e., a link exists between configuration c and query q if and only if $g_{cq} > 0$ (the value of the gain is reported beside each link). Figure 1 also shows (through bold font and shaded gray) the optimal solution of the selected sample instance. In particular, configuration c_3 has been chosen to serve query q_2 while configuration c_4 serves queries q_1 and q_3. Accordingly, the indexes selected are all but i_2. This yields a net time gain of 26 (total gain of 82 and fixed cost of 56), with a feasible memory occupation of 291.

3.2. Mathematical model

Let us define a binary variables y_i taking value 1 if index $i \in I$ is built and maintained, and 0 otherwise, and a binary variables x_{cq} taking value 1 if index configuration $c \in C$ is activated to serve query $q \in Q$, and 0 otherwise. Then, our ISPwC can be stated as follows:

$$\max \quad \sum_{c \in C} \sum_{q \in Q} g_{cq} x_{cq} - \sum_{i \in I} f_i y_i \tag{1}$$

subject to

$$\sum_{i \in I} m_i y_i \leq M \tag{2}$$

$$\sum_{c \in C} x_{cq} \leq 1 \qquad q \in Q \tag{3}$$

$$\sum_{c \in C_i} x_{cq} \leq y_i \qquad i \in I, q \in Q \tag{4}$$

$$y_i \in \{0,1\} \qquad i \in I \tag{5}$$

$$x_{cq} \in \{0,1\} \qquad c \in C, q \in Q. \tag{6}$$

The objective function (1) maximizes the total net time gain, composed by the gain raised by using specific configurations for the queries minus the fixed cost to create/maintain the indexes built. Inequality (2) is a classical knapsack constraint ensuring that the memory available is not exceeded. Constraints (3) ensure that at most one configuration is activated for each query. Constraints (4) link logically variables y and x. More precisely, if any configuration c is activated to serve a query q, then all the indexes composing that configuration must be built. *Vice versa*, if an index is not built (i.e., $y_i = 0$), then all the configurations composed by that index cannot be activated (i.e., $x_{cq} = 0, \forall c \in C_i, q \in Q$). Finally, constraints (5)–(6) state binary conditions for the variables.

Note that our ISPwC is $\mathcal{NP}$-hard, being a generalization of both the UFLP and the KP, which are known to be $\mathcal{NP}$-hard.

4. Scatter Search implementation

The Scatter Search (SS) algorithm is a population-based meta-heuristic (such as Genetic Algorithms), firstly introduced by Glover (1977). Given its capability to solve both combinatorial and continuous optimization problems, SS has been successfully applied to various hard optimization problems. See, e.g., the school bus routing problem (Corberán et al., 2002), the capacitated multi-commodity network design problem (Ghamlouche et al., 2003), and even non-linear or multi-objective problems (Rahimi-Vahed et al., 2007). SS was chosen as it provides, similarly to other evolutionary algorithms, the right balance between accuracy and efficiency. However, compared to Genetic Algorithms, SS shows advantages because it utilizes strategic designs and unifies principles for joining solutions. Additionally, SS can be easily coupled with Tabu Search features, such as adaptive memory and aspiration criteria, to influence individuals' selection in the reference set. As exhaustively explained in Martí et al. (2006), SS consists of five basic methods/steps that we summarize in the following:

1. a *Diversification Generation Method*, to generate a set of diverse solutions by taking a so-called *seed* solution as input;
2. an *Improvement Method*, to enhance the quality of the candidate solutions;
3. a *Reference Set Creation/Update Method*, to build and maintain a small reference set consisting of the "best" solutions, in terms of quality or diversity, found so far;
4. a *Subset Generation Method*, to operate on the reference set by producing a subset of its solutions as a basis for creating combined solutions;
5. a *Solution Combination Method*, to transform a given subset of solutions produced by the previous method into one or more combined solution vectors.

However, SS is very flexible, and its elements can be combined in several ways and implemented with different degrees of sophistication.

In the following, we will discuss in detail our SS implementation.

4.1. Main framework

First, we represent our solution through a binary encoding of the set of configurations set. We define a vector $\mathbf{w}$ of bits representing whether an index configuration has been used to serve at least one query (value 1) or not (value 0)[2]. This encoding is sufficient to achieve a complete solution $(\mathbf{x}, \mathbf{y})$ for our ISPwC. In fact, given a binary value w_c for each configuration $c \in C$ and defining the set $\bar{C} := \{c \in C \mid w_c = 1\}$, a ISPwC solution can be uniquely obtained by setting:

- $y_i = 1$ for each index $i \in I_c$, $c \in \bar{C}$, and $y_i = 0$ otherwise;
- $x_{cq} = 1$ for each query $q \in Q$ if $g_{cq} \geq g_{c',q}$ for each $c, c' \in \bar{C}, c' \neq c$, and $x_{cq} = 0$ otherwise.

The configuration-query assignment follows the logic of maximizing the objective function, providing the best net gain. Note that a solution obtained as before may not be feasible due to the available memory capacity constraint in Eq. (2).

Our SS algorithm implementation is schematized in Figure 2. The black circles represent configuration vectors corresponding to solutions that could be unfeasible, while the gray circles represent vectors corresponding to surely feasible solutions. The latter result from applying our particular Improvement Method, which also guarantees the restoring of feasibility.

The precise pseudocode of the SS heuristic can be found in Algorithm 1. The method generates a

[2]Interesting enough, the Genetic Algorithm proposed in Kratica et al. (2003) for the unlimited-memory version of our problem uses a binary encoding of the set of indexes instead.

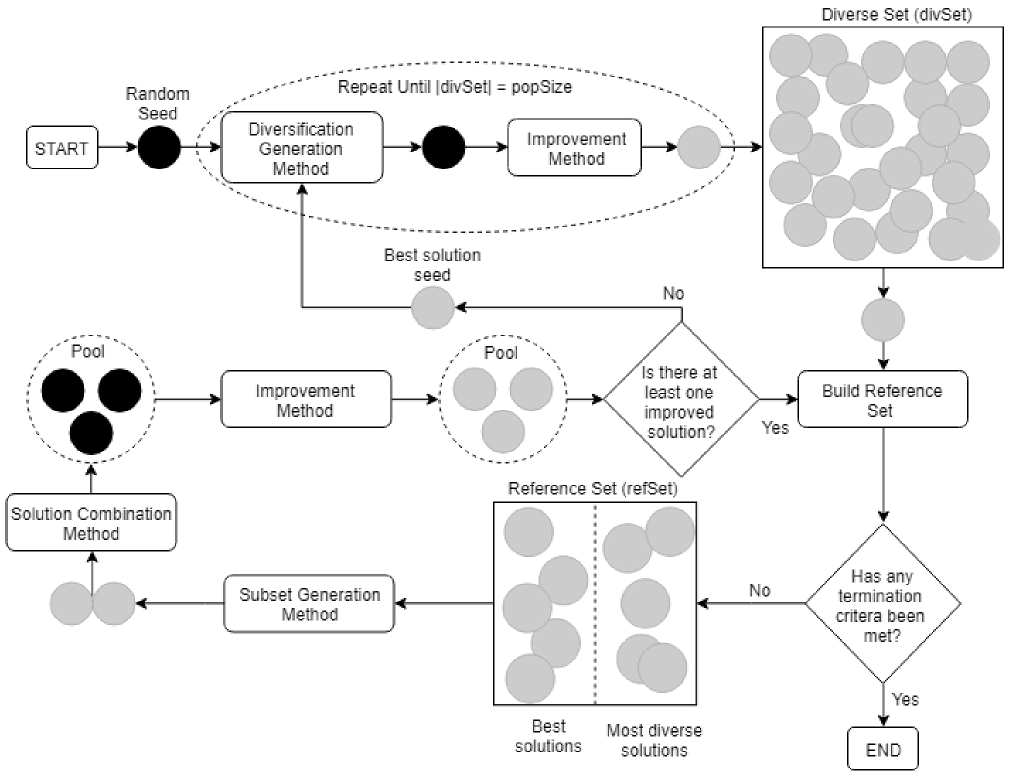


Figure 2: Schematic representation of our Scatter Search algorithm implementation.

set called *divSet* of different index configuration vectors using a Diversification Generation Method. The very first time, this method takes as input a random *seed* solution and inverts some of its elements by using specific rules (described later in detail). Any time a new vector is generated, the Improvement Method is applied to the solution for two primary purposes. Namely, to ensure the solution's feasibility (in terms of memory usage concerning the capacity) and possibly improve its objective function value through greedy procedures. The SS algorithm's peculiarity is the construction and maintenance, through the Reference Set Creation/Update Method, of a Reference Set ($refSet$), i.e., a small subset of the configuration vectors that correspond to solutions having high quality and diversity. Those vectors are used in the main iterative loop of the SS. Using the Subset Generation Method, all the possible subsets composed of two solutions of the $refSet$ are created and then combined to form a new vector using the Solution Combination Method. This approach produces a pool of new vectors, possibly corresponding to infeasible solutions. Therefore the Improvement Method is again applied. According to a scoring function, the highest-score vectors in the pool are used to update the $refSet$. We do that for multiple iterations until no new vectors corresponding to an improved solution are found regarding the objective function value. In the case of non-improvement, the algorithm restarts from regenerating the *divSet* by applying the initial Diversification Generation Method, using the current pool's best solution as a *seed*. The aim is to provide a different $refSet$ to restart the main SS loop.

Finally, notice that the algorithm stops once a termination criterion is met. As usual, this criterion is based upon thresholds on numbers of iterations or CPU time. During the run-time of the algorithm, the best solution found so far is stored in memory. When the algorithm stops, it

```
Require: $popSize$, $refSize$
Ensure: a vector $\mathbf{w}$ corresponding to a feasible ISPwC solution $(\mathbf{x}, \mathbf{y})$

$divSet \leftarrow \emptyset$
while $|divSet| \neq popSize$ do
    Use the Diversification Generation Method to generate a vector $\mathbf{w}$
    Apply the  Improvement Method on $\mathbf{w}$
    if $\mathbf{w} \notin divSet$ then
        Add $\mathbf{w}$ to $divSet$ (i.e., $divSet = divSet \cup \mathbf{w}$)
    else
        Discard $\mathbf{w}$
    end
end
Build a $refSet$ of cardinality $refSize$ (using half the best and half the most diverse configurations)
$newSolFound \leftarrow$ TRUE
while any termination criterion is not satisfied do
    if $newSolFound$ then
        $refSet$ updating:
        Apply Subset Generation Method to create the set $subsets$ containing all the possible subsets of $refSet$ of
          cardinality 2
        $newSolFound \leftarrow$ FALSE
        while $subsets \neq \emptyset$ do
            Select the next subset $s$ in $subsets$
            Apply the Combination Method to $s$, for different threshold $r$ values, to obtain new vectors and add
              them to $pool$
            Apply Improvement Method to all the vectors in the $pool$
        end
        Update $refSet$ by selecting the best $refSize$ solutions in $refSet \cup pool$
        if refSet has at least one new solution then
            $newSolFound \leftarrow$ TRUE
        end
    else
        Apply the Reference Set Criterion/Update Method to rebuild $refSet$:
        Delete the worst $refSize/2$ vectors from $refSet$
        Build a new $divSet$ using the Diversification Generation Method with the best solution in the current
          $refSet$ as a $seed$
        Add the new most diverse vectors generated to $refSet$
    end
end
```

Algorithm 1: Scatter Search (SS) pseudocode

returns the best solution found so far.

4.2. Detailed description of the SS components

A detailed description of the main components of our SS is provided in the following.

Diversification Generation Method. This method aims at initializing the *divSet* with a population of cardinality *popSize* of different configuration vectors. It takes as input a configuration vector called *seed* (of length $|C|$), which is manipulated to generate other vectors. More precisely, given a parameter $2 \leq h \leq |C| - 1$ and a parameter $q = \{1, \dots, h\}$, the following rules are applied:

R1) generate a vector by inverting the elements in position $q + kh$ of the vector *seed*, where $k = \{0, 1, \dots, \}$, while $q + kh \leq |C|$;

R2) generate a vector by inverting all the elements of the vector generated by rule R1.

An example of such diversification is shown in Table 1. In our implementation, following Glover (1997), the parameter h is bounded by $|C|/5$ to decrease the number of diverse solutions considered. Moreover, since the parameter *popSize* results to be a critical for the efficiency of the algorithm, an accurate calibration is needed (see Section 5.2).

Table 1: Configuration vectors generated from the input *seed* $[0,0,0,0,0,0,0,0,0,0]$.

h	q	R1	R2
2	1	[1,0,1,0,1,0,1,0,1,0]	[0,1,0,1,0,1,0,1,0,1]
2	2	[0,1,0,1,0,1,0,1,0,1]	[1,0,1,0,1,0,1,0,1,0]
3	1	[1,0,0,1,0,0,1,0,0,1]	[0,1,1,0,1,1,0,1,1,0]
3	2	[0,1,0,0,1,0,0,1,0,0]	[1,0,1,1,0,1,1,0,1,1]
3	3	[0,0,1,0,0,1,0,0,1,0]	[1,1,0,1,1,0,1,1,0,1]
4	1	...	...

Improvement Method. The Improvement Method is applied to every new generated vector $\mathbf{w}$ to ensure that the corresponding solution $(\mathbf{x},\mathbf{y})$ is feasible for the ISPwC problem. Moreover, the method also tries to improve the net gain of the solution greedily.

The procedure changes the bits of a configuration vector $\mathbf{w}$ according to the following steps:

1. consider the bits (configurations) of $\mathbf{w}$ in non-decreasing order, according to the following performance metric (PM):

$$\mathrm{PM}(\mathbf{w},c) = \sum_{q\in Q} g_{cq}\cdot x_{cq} - \sum_{i\in I_c} f_i y_i. \tag{7}$$

   This metric trivially follows the objective function (1) of the ISPwC problem, but considers only a single bit c of the configuration vector;
2. while the solution remains unfeasible, switch w_c from 1 to 0 (the idea is to eliminate the use of configurations not to pay for the creation of their indexes);
3. as soon as the solution becomes feasible, switch w_c from 0 to 1 unless this change causes an infeasibility (the idea is to use as many configurations as possible without exceeding memory occupation threshold).

Within the above algorithmic framework, two other different performance metrics are proposed. These variations, called PM_1 and PM_2, are described in the following Equations:

$$\mathrm{PM}_1(\mathbf{w},c) = \sum_{q\in Q} g_{cq}\cdot x_{cq} - \sum_{i\in I_c} \frac{(f_i+m_i)y_i}{2|AC_i|}, \tag{8}$$

$$\mathrm{PM}_2(\mathbf{w},c) = \sum_{q\in Q} g_{cq}\cdot x_{cq} - \sum_{i\in I_c} \frac{f_i y_i}{|AC_i|}, \tag{9}$$

where AC_i is the set of active configurations in the considered solution $\mathbf{w}$ using index i. PM_1 provides some gain for the queries where $x_{cq}=1$ and subtracts a sort of average of the fixed cost and memory occupation. In PM_2, instead, the gain is subtracted by the normalized fixed cost of the candidate solution. Some tests about the best performance metric are shown in Section 5.2.

Reference Set Creation/Update Method. The $refSet$ contains a mix of the configuration vectors that are the most diverse and corresponding to the best solutions. In particular, given a $refSet$ of cardinality $refSize$, half vectors will be chosen because of their quality and half because of their diversity.

The solutions' quality is assessed using the objective function in Eq. (1) of the ISPwC problem. Instead, to find the most diverse vectors from the population set, we calculate each vector's most considerable average Hamming distance with respect to the rest of the $refSet$. A calibration of the $refSize$ parameter is performed in Section 5.2.

Subset Generation Method. The Subset Generation Method simply generates all subsets of the $refSet$ composed by two configuration vectors. In Algorithm 1, we call *subsets* the set of all the above subsets.

Combination Method. In the main iterative loop, the vector pairs appearing in each subset of the $refset$ are combined to generate a *pool* of new possible configurations. Since the combination does not guarantee the resulting solution's feasibility, the Improvement Method is applied to each resulting vector. The new vectors which correspond to solutions with larger value than the solutions already found are added to the $refSet$.

The Combination Method is based on the use of a scoring function (SF). More precisely, given vectors $\bar{\mathbf{w}}$ and $\bar{\bar{\mathbf{w}}}$ to be combined, SF is computed for each configuration c as follows:

$$SF(\bar{\mathbf{w}}, \bar{\bar{\mathbf{w}}}, c) = \frac{PM(\bar{\mathbf{w}}, c) \cdot \bar{w}_c + PM(\bar{\bar{\mathbf{w}}}, c) \cdot \bar{\bar{w}}_c}{\sum_{c \in C}[PM(\bar{\mathbf{w}}, c) + PM(\bar{\bar{\mathbf{w}}}, c)]}. \tag{10}$$

Eq. (10) calculates the sum of the individual contribution of each configuration of the two vectors with respect to the sum of the objective values associated with the solutions being combined. We then use the scoring function as a probability for setting each bit being considered for the new solution as 1 or 0. More precisely, in the new combined solution

$$\mathrm{w}_c = \begin{cases} 1, & \text{if } r \leq SF(\bar{\mathbf{w}}, \bar{\bar{\mathbf{w}}}, c) \\ 0, & \text{otherwise} \end{cases} \qquad \forall c \in C, \tag{11}$$

where r is a threshold number randomly drawn in the range $[0, 1]$.

After obtaining combined and improved solutions from the reference set's different subsets, they are placed into *pool*. The best solutions in *pool* are added to the $refSet$. If none of the new solutions is different from the original reference set (i.e., there are no better solutions in the current *divSet*), then the reference set is rebuilt but, instead of a random *seed* as done in the initialization step, the best vector is taken as a *seed* to repopulate the *divSet*.

5. Computational experiments

In this section, we first discuss the generation of the instances used for the computational experiments. After some parameters calibration, we present and analyze extensive results to assess our Scatter Search accurately and efficiently. All the computational experiments have been run on an *Intel(R) Core(TM) i7 CPU 860@2.80GHz* machine with 16.0GB RAM and running *Windows 10 Pro* x64 operating system.

Our Scatter Search algorithm has been implemented through the C++ language. The code is freely available at `https://github.com/RuslanKain/odbdp-ssa`. The SS time limit is set to 180 seconds. The optimal benchmarks, when possible, have been obtained by merely inputting the model presented in Section 3.2 into CPLEX v12.8.0 MIP solver through its C/C++ Concert Technology's APIs and letting it run for 1 hour. Finally, several comparisons have been made with the recursive-based heuristic (RH) proposed by Schlosser et al. (2019). A Python implementation of the RH code, made available by the authors, has been migrated to the C++ language to have a fair comparison. We emphasize that RH has been implemented without the explicit distinction between single-attribute and multi-attribute indexes (which does not hold in our problem), and considering a configuration of multiple indexes in substitution for a multi-attribute index. As such, the RH algorithm's design remained unaltered but only fitted to work with the problem instances targeted by the SS.

In the following, whenever two solution methods $\mathcal{A}$ and $\mathcal{B}$ are compared in terms of quality, we will denote by $\%gap_{\mathcal{A},\mathcal{B}}$ the percentage gap between the objective function value of the best solution found by $\mathcal{B}$ ($obj_{\mathcal{B}}$) and by $\mathcal{A}$ ($obj_{\mathcal{A}}$), i.e.

$$\%gap_{\mathcal{A},\mathcal{B}} := \frac{obj_{\mathcal{A}} - obj_{\mathcal{B}}}{obj_{\mathcal{A}}} \times 100.$$

Hence, a negative value for $\%gap_{\mathcal{A},\mathcal{B}}$ will indicate that method $\mathcal{B}$ outperforms method $\mathcal{A}$.

5.1. Benchmark instances

To assess the performance of our algorithm, we create an excellent testbed of new random instances. Based on those proposed in the literature for similar problems, the generation method is intended to simulate realistic DB workloads and obtain hard-to-solve instances.

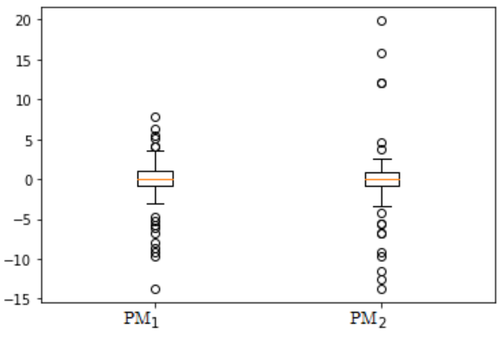
In Caprara and Salazar González (1996), the authors propose two different methods (*Class A* and *Class B*) to generate ISPwC instances. However, *Class B* instances have resulted in being too easy-to-solve and have been abandoned in further studies. Instead, *Class A* instances have also been used in other more recent papers (see, e.g., Kratica et al., 2003) since they are still challenging. Therefore, we adopted the *Class A* generation method as a general framework. Moreover, we improve the generation method by introducing both randomnesses to diversify the instances better and some parameter values proportional to the instance size (e.g., indexes or queries). More precisely, for each index $i \in I$, f_i is uniformly generated in $[90, 110]$. For each configuration $c \in C$, the indexes composing the set I_c are selected randomly in I, given that the cardinality of each I_c is chosen according to a uniform distribution in $[1, |I|/10]$. Similarly, for each configuration $c \in C$, we randomly select in $[1, |I|/10]$ the number of queries for which the gain $g_{cq} > 0$. The exact value of those positive gains are randomly drawn in $[1, \alpha * |I_c|]$, where the maximum value depends on a parameter α representing the percentage gain powered by each index $i \in I_c$ with respect to a particular query $q \in Q$. The parameter α allows to modify the proportion between gains and fixed costs in terms of time. Finally, we need to complete the instances with a cost in terms of memory for each index and a total available memory. Following Caprara et al. (1995), the memory occupation m_i for each index $i \in I$ is uniformly generated in $[450, 2500]$. The total available memory M is calculated as a certain percentage β of the memory needed to store all the created indexes, i.e., $M = \lfloor \beta \sum_{i \in I} m_i \rfloor$. The higher the value of β, the milder the memory limitation in constraint (2).

Eventually, we have generated three sets of instances, called *Small*, *Large*, and *Extra-Large*, respectively. The *Small* set contains one instance for each combination of $|I| = \{50, 100\}$ indexes, $|Q| = \{50, 100\}$ queries, $|C| = \{500, 1000\}$ configurations, $\alpha = \{25, 50, 100\}$, and $\beta = \{20, 50, 80\}$. The *Large* set contains one instance for each combination of $|I| = \{500, 1000\}$, $|Q| = \{500, 1000\}$, $|C| = \{5000, 10000\}$, and all the combinations of the above α and β values. Finally, the *Extra-Large* set contains 8 very huge instances with $|I| = 5000$, $|Q| = 5000$, and 20000 or 50000 configurations. Such instances are freely available at `http://www.orgroup.polito.it/resources.html`.

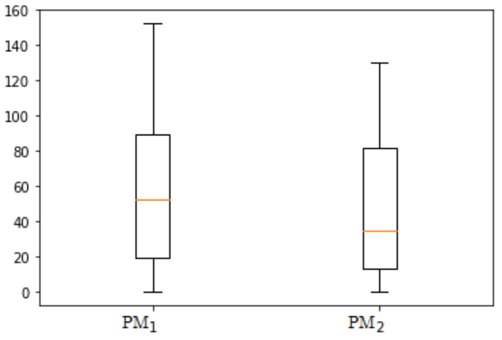
5.2. Parameters calibration

Hereafter, we propose an empirical calibration of the best performance metric to use and the best main parameters to adopt in our SS.

Performance metric calibration. Since the basic performance metric in Eq. (7) has appeared ineffective in some preliminary experiments, we tested the performance metrics PM_1 in Eq. (8) and PM_2 in Eq. (9) over 10 independent runs of SS for a subset of 24 *Small* instances. This subset is representative enough since PM_1 and PM_2 are not correlated to the instances' main dimensions ($|Q|$, $|I|$, and $|C|$). The results are assessed by calculating the percentage gap $\%gap_{CP,SS}$, where CP indicates the model solved by CPLEX within 1 hour threshold, and the time-to-best (ttb) of the heuristic, i.e., the time in seconds that SS needs to achieve the best solution. Note that a negative value for $\%gap_{CP,SS}$ may happen since CPLEX cannot solve all the ISPwC instances optimally within the threshold time. The distributions of the results obtained for $\%gap_{CP,SS}$ and ttb are shown in the box-plots of Figure 3a and 3b, respectively. In Figure 3a, the average $\%gap_{CP,SS}$



(a) Distribution of $\%gap_{CP,SS}$.



(b) Distribution of ttb.

Figure 3: Distribution of $\%gap_{CP,SS}$ and ttb resulting from the use of performance metrics PM_1 and PM_2.

values for both performance metrics are close to zero, meaning that the performance is similar to the benchmark for most instances of the problem. However, for PM_2 there are several extreme values, which have significantly worse performance. In Figure 3b, PM_2 is shown to perform a bit better, on average, in terms of time to obtain the best solution. However, since our focus is more on the method's accuracy, we decided to use PM_1 as the performance metric for the extensive tests presented in the following.

Calibration of the population and reference set sizes. We experimentally tested different ways for setting the size of $divSet$ ($popSize$) and $refSet$ ($refSize$). Such parameters are set as proportions of specific instance dimensions, i.e., $popSize = 10 \cdot refSize$, where $refSize$ is set according to one of the following formulas:

$$\text{RS}_1: \ refSize = \frac{|C|}{|Q|}; \quad \text{RS}_2: \ refSize = \frac{|C||I|}{100|Q|};$$

$$\text{RS}_3: \ refSize = \frac{|C||I|}{50|Q|}; \quad \text{RS}_4: \ refSize = \frac{|C|}{100}.$$

In particular, we tested the four different settings over a representative subset of 24 *Small* instances and a representative subset of 24 *Large* ones; indeed, the algorithm has been executed 10 times on each instance. In the following, we report only the results obtained by the two best settings for each set, namely RS_1 and RS_2 for the *Small* instances (Table 2) and RS_3 and RS_4 for the *Large* ones (Table 3). For the sake of conciseness, for each calibration setting, in each table we only

Table 2: Percentage gap ($\%gap_{CP,SS}$) and time-to-best (*ttb*) in seconds of SS with two different calibrations of $refSize$ and *popSize* for *Small* instances.

Calibration	Indicators over all the instances	$\%gap_{CP,SS}$ statistics over 10 runs per instance				*ttb* statistics over 10 runs per instance			
		Avg	Stdev	Best	Worst	Avg	Stdev	Best	Worst
RS_1	Avg:	-0.74	0.35	-1.17	-0.18	60.49	43.49	15.63	141.68
	Stdev:	2.39	0.47	2.60	2.39	29.59	15.53	14.57	36.04
	Best:	-8.00	0.00	-8.80	-7.50	8.29	7.04	0.65	31.03
	Worst:	1.67	1.60	0.84	5.00	112.15	71.34	67.01	179.16
RS_2	Avg:	-0.70	0.25	-1.07	-0.35	59.59	44.10	13.22	138.40
	Stdev:	2.49	0.34	2.65	2.32	31.77	18.22	9.59	47.73
	Best:	-8.47	0.00	-8.80	-8.09	8.91	2.89	0.88	27.84
	Worst:	2.91	1.42	1.88	3.48	147.26	73.32	41.14	179.93

Table 3: Percentage gap ($\%gap_{RH,SS}$) and time-to-best (*ttb*) in seconds of SS with two different calibrations of $refSize$ and *popSize* for *Large* instances.

Calibration	Indicators over all the instances	$\%gap_{RH,SS}$ statistics over 10 runs per instance				*ttb* statistics over 10 runs per instance			
		Avg	Stdev	Best	Worst	Avg	Stdev	Best	Worst
RS_3	Avg:	-22.94	1.39	-25.06	-21.04	71.63	44.44	20.06	144.14
	Stdev:	13.86	0.94	14.40	13.53	34.75	11.27	16.25	34.65
	Best:	-50.88	0.00	-54.88	-48.37	7.59	14.12	1.39	46.62
	Worst:	-6.30	3.14	-7.33	-3.28	139.45	66.06	73.82	177.64
RS_4	Avg:	-22.38	1.33	-24.55	-20.81	75.53	45.99	16.59	150.34
	Stdev:	13.99	0.86	14.39	13.61	31.13	11.76	17.15	32.85
	Best:	-53.11	0.00	-55.39	-48.92	10.78	10.63	1.72	37.56
	Worst:	-4.63	3.61	-6.36	3.28	129.17	71.95	64.26	176.61

report some statistics (avg, stdev, best, and worst values) concerning some aggregate indicators of the percentage gap obtained by SS and of its time-to-best. In each cell of the tables it is reported a specific statistic (on the columns) calculated over 10 runs of a specific indicator (on the rows) aggregating the results for all the tested instances. For example, in cell $(1, 3)$ of Table 2, the value -1.17 represents the best value obtained in 10 different runs among all the average percentage gaps calculated over all the tested instances. Note that, since CPLEX is not a suitable solver for *Large* instances, in Table 3 the percentage gap reported ($\%gap_{RH,SS}$) is with respect to RH, which is the state-of-the-art heuristic proposed in Schlosser et al. (2019).

In Table 2, the RS_1 setting gave the best results for the *Small* instances, since it resulted slightly better in all the statistics for $\%gap_{CP,SS}$ (with an average improvement of -0.74% compared to the benchmark). In contrast, the time behavior (*ttb*) for the different settings showed similar performance, and the average time-to-best is about 1 minute. Concerning the *Large* instances, Table 3 indicates that the average $\%gap$ is -22.94% when using RS_3, while for RS_4 the average $\%gap$ is -22.38%. Moreover, on average, the *ttb* is 71.63 seconds for RS_3 and 75.53 seconds for RS_4. Thus, calibration RS_3 provides better solutions and finds them more quickly. So, according to the above calibration, we decided to perform the rest of the experiments by using the calibration RS_1 for the *Small* instances and calibration RS_3 for the *Large* ones. However, the SS quality and efficiency seem not significantly influenced by the various calibrations, indicating good robustness.

5.3. Extensive results and analysis

In the following, we present the assessment of the accuracy and the efficiency of our SS algorithm concerning the entire set of instances discussed in Section 5.1.

Accuracy of the Scatter Search. For each instance, SS was executed 10 times to obtain valuable statistics, while CPLEX and RH were executed once since they are deterministic methods. Table 4 contains a detailed comparison of the quality of SS and RH concerning the 72 *Small* instances. They were both compared to the CPLEX benchmark results. Instead, Table 5 contains a detailed comparison of the quality of SS with respect to RH concerning the 72 *Large* instances (the CPLEX benchmarks are not available).

Table 4: Percentage gaps of SS (over 10 runs) and of RH versus CPLEX on *Small* instances

Instance				$\%gap_{CP,RH}$		$\%gap_{CP,SS}$ ($\lvert I\rvert = 50$)				$\%gap_{CP,SS}$ ($\lvert I\rvert = 100$)			
$\lvert Q\rvert$	$\lvert C\rvert$	α	β	$\lvert I\rvert = 50$	$\lvert I\rvert = 100$	Avg	Stdev	Best	Worst	Avg	Stdev	Best	Worst
50	500	25	20	54.08	35.20	5.18	1.24	3.07	5.43	0.00	0.00	0.00	0.00
50	500	25	50	71.85	38.33	2.81	1.57	0.60	4.78	-0.23	0.08	-0.44	-0.18
50	500	25	80	70.17	43.35	6.27	2.34	0.76	7.78	1.82	0.83	0.87	3.16
50	500	50	20	78.58	37.42	0.00	0.00	0.00	0.00	-4.07	-2.21	0.73	-4.07
50	500	50	50	56.48	45.47	0.97	1.12	0.00	3.16	-1.47	0.41	-2.33	-0.98
50	500	50	80	75.43	49.91	0.01	0.00	0.00	0.01	0.00	0.00	0.00	0.00
50	500	100	20	81.51	38.29	0.04	1.91	0.00	4.52	-1.14	0.19	-1.33	-0.75
50	500	100	50	96.39	51.76	1.02	0.13	0.84	1.27	0.73	1.45	0.38	5.00
50	500	100	80	65.41	29.06	0.32	0.20	0.01	0.40	0.31	0.15	0.00	0.47
50	1000	25	20	66.86	32.65	3.41	4.75	0.77	11.33	-5.69	0.90	-6.63	-3.89
50	1000	25	50	70.55	41.94	0.41	1.38	-1.97	2.16	4.15	0.59	3.12	4.28
50	1000	25	80	77.10	57.67	0.49	0.65	-1.12	1.07	7.80	1.54	4.32	8.25
50	1000	50	20	79.09	29.62	1.89	1.41	0.52	4.99	-4.78	0.31	-5.30	-4.01
50	1000	50	50	73.75	37.15	0.00	0.00	0.00	0.00	-2.96	0.41	-3.12	-1.83
50	1000	50	80	93.39	68.54	0.41	0.38	0.00	1.11	-1.03	0.34	-1.40	-0.23
50	1000	100	20	70.99	41.52	1.09	0.00	1.09	1.09	-9.19	0.37	-9.82	-9.12
50	1000	100	50	72.10	41.68	1.43	0.46	0.00	1.45	-0.62	0.98	-2.54	-0.16
50	1000	100	80	58.60	47.45	0.47	0.15	0.00	0.48	0.25	0.06	0.24	0.42
100	500	25	20	73.07	50.61	0.00	0.00	0.00	0.00	5.38	1.09	2.38	5.81
100	500	25	50	71.34	48.57	2.24	1.00	0.68	4.11	-0.59	1.46	-3.34	0.93
100	500	25	80	77.18	73.03	0.00	0.00	0.00	0.00	0.80	1.45	0.00	3.67
100	500	50	20	98.39	57.35	0.00	0.00	0.00	0.00	-13.72	0.00	-13.72	-13.72
100	500	50	50	84.28	45.61	1.67	0.83	0.56	3.48	-6.16	1.60	-7.11	-3.41
100	500	50	80	83.44	11.16	0.00	0.00	0.00	0.00	0.38	0.01	0.38	0.42
100	500	100	20	73.94	16.56	0.00	0.00	0.00	0.00	-0.98	0.00	-0.98	-0.98
100	500	100	50	88.07	53.39	0.00	0.00	0.00	0.00	-5.27	0.09	-5.54	-5.27
100	500	100	80	87.79	37.31	0.00	0.01	0.00	0.03	0.22	0.15	0.12	0.63
100	1000	25	20	64.42	34.73	2.54	0.00	2.54	2.54	-8.72	1.19	-12.48	-8.71
100	1000	25	50	94.95	36.63	4.03	1.91	0.39	5.88	-1.92	1.75	-5.57	-0.96
100	1000	25	80	98.90	25.33	1.20	0.66	0.36	2.38	0.85	0.72	-1.13	1.04
100	1000	50	20	77.20	15.90	0.00	0.00	0.00	0.00	-0.78	0.50	-0.86	0.33
100	1000	50	50	81.91	44.33	0.64	0.29	0.00	0.69	-6.85	1.45	-8.80	-5.25
100	1000	50	80	97.99	44.87	0.00	0.00	0.00	0.00	0.43	0.03	0.40	0.45
100	1000	100	20	88.98	44.78	3.52	0.63	2.15	3.66	-9.73	0.05	-9.79	-9.67
100	1000	100	50	78.83	52.42	-1.46	0.00	-1.46	-1.46	-8.00	0.32	-8.56	-7.50
100	1000	100	80	88.65	53.32	0.20	0.11	0.00	0.30	0.61	0.00	0.61	0.61
			Avg:	78.38	42.03	1.13	0.64	0.27	2.02	-1.90	0.59	-2.83	-1.20
			Best:	54.07	11.16	-1.46	0.00	-1.97	-1.46	-13.72	0.00	-13.72	-13.72
			Worst:	98.9	73.03	6.27	4.75	3.07	11.33	7.80	1.75	4.32	8.25

Concerning the *Small* instances (Table 4), we observe that the larger the set of indexes, the better the performance of SS. More precisely, for $|I| = 50$, the SS underperforms compared to the solver with an average percentage gap performance of 1.13%, while for $|I| = 100$ the SS outperforms the solver by -1.90% on average. We also note that SS found the same solution as CPLEX (i.e., 0.0% gap) for 13 instances, primarily for $|I| = 50$, and outperforms the solver for 22 instances, especially for $|I| = 100$. SS significantly outperforms the benchmark in some individual instances, with the best improvement being -13.72%. Furthermore, SS shows stability in performance across

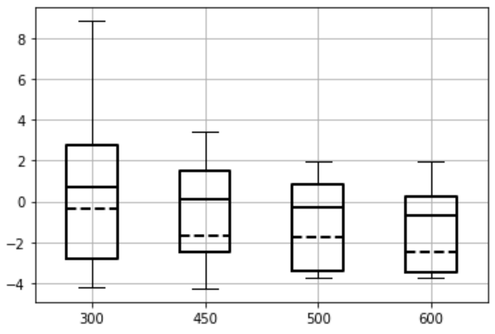
Table 5: Percentage gaps of SS (over 10 runs) versus RH on *Large* instances

Instance				$\%gap_{RH,SS}$ ($\lvert I\rvert = 500$)				$\%gap_{RH,SS}$ ($\lvert I\rvert = 1000$)			
$\lvert Q\rvert$	$\lvert C\rvert$	α	β	Avg	Stdev	Best	Worst	Avg	Stdev	Best	Worst
500	5000	25	20	-36.50	0.00	-36.50	-36.50	-12.64	0.28	-12.67	-11.78
500	5000	25	50	-32.73	1.85	-34.35	-28.53	-11.10	1.30	-13.04	-8.87
500	5000	25	80	-31.51	0.71	-32.86	-30.70	-10.64	0.79	-11.96	-9.38
500	5000	50	20	-28.52	1.97	-29.93	-23.72	-7.93	1.05	-10.02	-7.39
500	5000	50	50	-26.54	1.05	-27.96	-24.21	-7.91	0.90	-10.34	-7.25
500	5000	50	80	-25.23	0.67	-26.95	-24.89	-8.60	0.64	-9.83	-7.65
500	5000	100	20	-23.32	0.00	-23.32	-23.32	-9.27	0.43	-9.73	-7.96
500	5000	100	50	-26.08	0.79	-27.68	-24.70	-14.36	1.35	-16.53	-12.20
500	5000	100	80	-28.59	1.31	-30.00	-26.16	-11.12	0.71	-12.64	-10.18
500	10000	25	20	-45.70	0.00	-45.70	-45.70	-11.48	0.13	-11.64	-11.39
500	10000	25	50	-34.81	1.80	-38.89	-32.82	-11.70	0.79	-13.13	-10.58
500	10000	25	80	-35.77	2.60	-43.04	-34.33	-10.60	0.82	-12.88	-10.14
500	10000	50	20	-27.76	2.44	-34.77	-27.00	-21.32	2.14	-21.37	-16.30
500	10000	50	50	-29.22	1.06	-31.02	-28.07	-11.64	0.90	-13.12	-10.08
500	10000	50	80	-29.19	1.15	-31.43	-28.01	-11.53	0.65	-12.26	-10.18
500	10000	100	20	-31.99	3.14	-36.56	-27.97	-6.83	1.52	-7.34	-3.74
500	10000	100	50	-29.98	2.04	-33.95	-27.90	-14.24	1.07	-15.36	-12.18
500	10000	100	80	-29.99	1.62	-34.14	-28.17	-11.97	0.91	-14.64	-11.47
1000	5000	25	20	-25.00	0.14	-25.22	-24.93	-6.30	2.00	-7.81	-3.28
1000	5000	25	50	-32.02	3.24	-36.98	-26.95	-9.86	1.15	-12.53	-8.38
1000	5000	25	80	-41.56	1.78	-43.77	-38.66	-10.93	0.80	-13.10	-10.23
1000	5000	50	20	-23.74	2.59	-27.78	-21.47	-5.75	1.52	-9.78	-4.84
1000	5000	50	50	-30.27	2.04	-33.53	-28.08	-8.86	0.81	-9.07	-6.88
1000	5000	50	80	-44.36	3.02	-46.51	-38.11	-10.47	0.82	-11.46	-9.05
1000	5000	100	20	-12.78	2.35	-17.43	-11.11	-8.33	2.82	-11.65	-4.82
1000	5000	100	50	-32.31	2.34	-34.50	-28.47	-11.03	0.81	-11.77	-9.58
1000	5000	100	80	-42.60	1.54	-45.00	-39.59	-10.85	1.02	-13.38	-10.29
1000	10000	25	20	-32.06	0.88	-33.25	-29.66	-12.71	2.20	-17.57	-11.39
1000	10000	25	50	-35.89	2.65	-41.63	-32.88	-9.99	1.46	-11.32	-6.93
1000	10000	25	80	-49.97	2.90	-54.52	-44.63	-10.56	0.98	-11.38	-8.35
1000	10000	50	20	-30.61	1.84	-36.01	-29.14	-11.92	1.81	-13.83	-10.03
1000	10000	50	50	-31.89	2.64	-39.14	-30.39	-12.90	1.54	-15.24	-9.42
1000	10000	50	80	-47.22	3.15	-53.40	-41.71	-11.17	1.15	-11.99	-8.49
1000	10000	100	20	-28.37	3.04	-32.86	-26.54	-8.30	1.17	-11.09	-7.25
1000	10000	100	50	-40.38	2.44	-46.24	-37.74	-11.22	1.23	-12.77	-8.78
1000	10000	100	80	-50.88	2.30	-54.89	-48.37	-13.23	0.62	-14.27	-12.48
			Avg:	-32.93	1.81	-36.16	-30.59	-10.81	1.12	-12.46	-9.14
			Best:	-50.88	0.00	-54.89	-48.37	-21.32	0.13	-21.37	-16.30
			Worst:	-12.78	3.24	-17.43	-11.11	-5.75	2.82	-7.34	-3.28

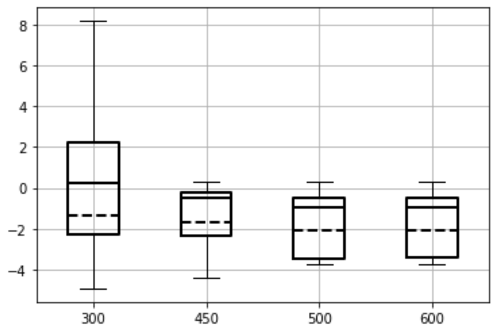
all instances, as indicated by the low standard deviations, 0.64 for $|I| = 50$, and 0.59 for $|I| = 100$, over the average results. This stability makes our method particularly suitable to solve this kind of instance, while CPLEX is not a reasonable alternative anymore, particularly for $|I| = 100$ instances. Lastly, SS outperforms RH in terms of solution quality, as indicated by the 78.38% average gap for $|I| = 50$ for and 42.03% average gap for $|I| = 100$ of RH compared to the benchmark. Concerning the *Large* instances (Table 5), SS outperforms RH since the average percentage gap with respect to RH is -32.93% and -10.91%, respectively. Moreover, SS can significantly outperform RH in some cases, as shown by the best gap value of -50.88%.We also note that SS outperforms RH for all the instances since the average $\%gap$ for all runs is always negative. Furthermore, SS shows stability in performance across all instances, as indicated by the low average standard deviations, which are 1.81 and 1.12 for $|I| = 500$ and $|I| = 1000$, respectively. However, the results also show that the SS and RH performances become closer as the problem's scale increases.

To further test our SS's scalability, we conducted experiments for the 8 *Extra-Large* instances characterized by an enormous number of indexes, configurations, and queries (namely, 5000 indexes and queries, and 20000 or 50000 configurations. The SS time limit has been therefore extended to 600 seconds. In Figure 4, we report results for 300, 450, 500, and 600 seconds in the form of box plot distributions of the $\%gap_{RH,SS}$ averaged over 10 runs, and using all the calibration settings for

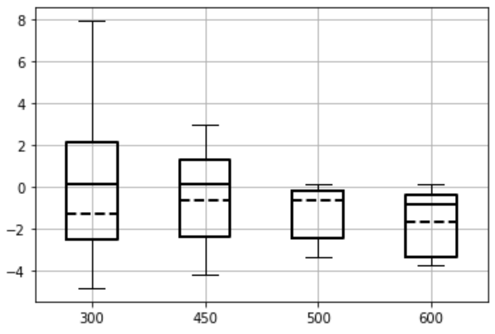
$refSize$ and $popSize$ proposed in Section 5.2 to explore a wider range of possible solutions. The box



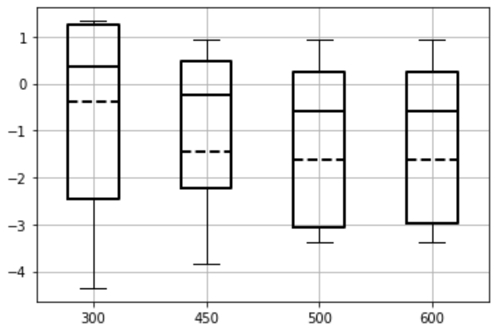
(a) RS_1



(b) RS_2



(c) RS_3



(d) RS_4

Figure 4: Percentage gap with respect to RH (y-axis) over time for *Extra-Large* instances with different parameters calibrations. The box represents the inter-quartile range (25% to %75) and reports the median value through a dashed line and the mean value through a solid line. Maximum and minimum values are represented out of the box.

plots show a pattern similar to that observed for *Small* and *Large* instances, i.e., that of a gradual improvement of SS over RH for more significant run-time limits. Up to 300 seconds, RH slightly outperforms SS, but the average percentage gap is always in the [0%, 1%] range. Moreover, the inter-quartile range (25%-75%) shifts downward, below the 0% gap, meaning that SS outperforms RH for most cases. Comparing the different calibrations settings, it appears that the average percentage gap is similar for all calibrations, i.e., between 0% and -1%. For RS_2 and RS_3, the inter-quartile range lies below the 0% gap. Upon closer inspection, for RS_3, the worst-case percentage gap has the lowest value compared to the others. For the sake of completeness, Table 6 reports the number of *Extra-Large* instances for which SS finds a configuration set returning a net gain strictly greater than 0 (i.e, a non-trivial solution), for the different time limits and for the different calibrations. The only difference seems regarding the 500-second mark. SS with calibrations RS_2 and RS_4 solved all 8 instances, while with RS_1 and RS_3 only one instance was not solved in time.

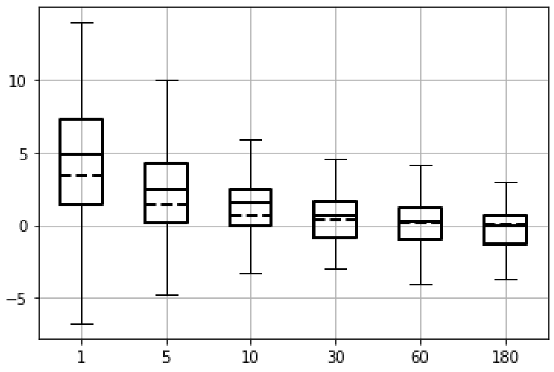
Efficiency of the Scatter Search. The experiments for the *Small* and *Large* instances were set to a run-time limit of 3 minutes to find an improved solution in an acceptable time range. In theory, if the algorithm is left running for longer, even better results could be obtained. However, in practice, long run-times are unacceptable for real-time DB management applications. Most of the time, it is important to obtain a good solution as fast as possible. Hence, to assess the efficiency

Table 6: Number of *Extra-Large* instances solved with the different calibrations and for the different time limits.

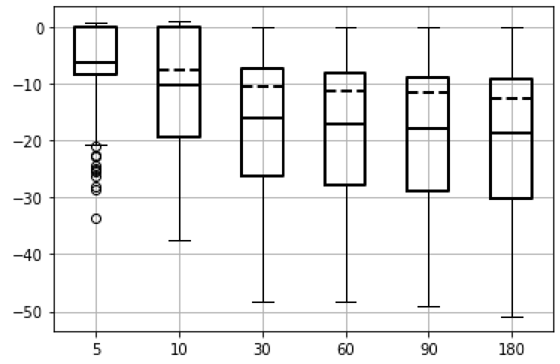
Calibration	Time limit (seconds)			
	300	450	500	600
RS_1	6	7	7	8
RS_2	6	7	8	8
RS_3	6	7	7	8
RS_4	6	7	8	8

of our SS, we study its performance over time.

In particular, for the *Small* instances, we report results at 1, 5, 10, 30, 60, and 180 seconds, and for the *Large* instances at 5, 10, 30, 60, 90, and 180 seconds. In Figure 5(a), we show the distribution of $\%gap_{CP,SS}$ for all the *Small* instances in the form of box plots over time, while the distribution of $\%gap_{RH,SS}$ for all the *Large* instances are similarly shown in Figure 5(b). We emphasize that, for all the *Small* instances, a configuration set returning a net gain strictly greater than 0 was found for all the run-time limits. Instead, all the 72 *Large* instances this happened only after 30 seconds, while, by the 5-second limit and the 10-second limit, 39 and 57 instances were solved, respectively. For the *Small* instances, in Figure 5(a), SS can find on average solutions 5% close to



(a) *Small* instances

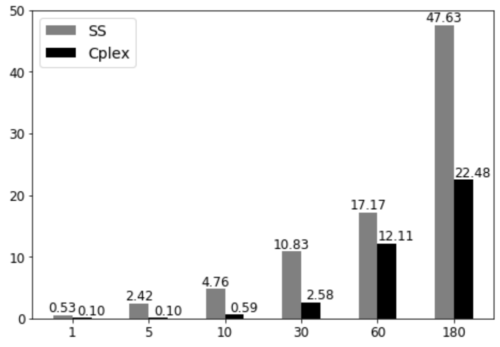


(b) *Large* instances

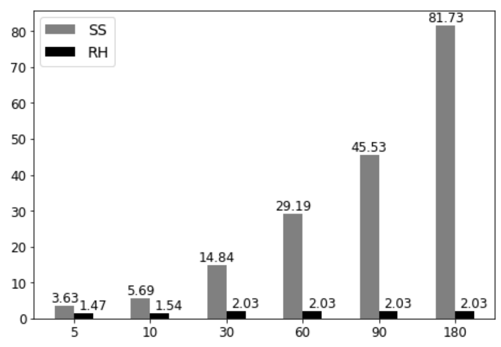
Figure 5: Box plot distribution of (a) $\%gap_{CP,SS}$ for *Small* instances and of (b) $\%gap_{RH,SS}$ for *Large* instances over time. The dashed line shows the median value, the solid line shows the mean value, and the circles indicate outliers.

the benchmark in 1 second. The quality rapidly increases, converging to have 0% gaps on average around 30 seconds. The best-to-worst range of the gaps reduces gradually from approximately 21% and -6% to 3% and -4%, thus, becoming more robust in terms of solution quality. Concerning the *Large* instances, in Figure 5(b), SS obtains on average solutions with gaps around -7% with respect to RH already after 5 seconds, with some outliers below the -20%. The average quality gradually increases with longer run-time limits, converging to around -16% at 30 seconds. The trend shows that the longer the time limits, the better the solution quality. The best-to-worst range consistently shifts downward, indicating that some solutions have increasingly better quality, thus shifting the average gap lower for longer run-time intervals.

In Figure 6, we show the speed performance (i.e., the average time-to-best) of all the approaches for both sets of instances. In both cases, when comparing SS to CPLEX for the *Small* instances in Figure 6(a) and to RH for the *Large* instances in Figure 6(b), we see that SS takes a longer time. For the benchmark, the *ttb* gradually increases with greater time limits, but not at the



(a) *Small* instances



(b) *Large* instances

Figure 6: SS time-to-best for the different run-time intervals for (a) *Small* instances with respect to CPLEX and for (b) *Large* instances with respect to RH.

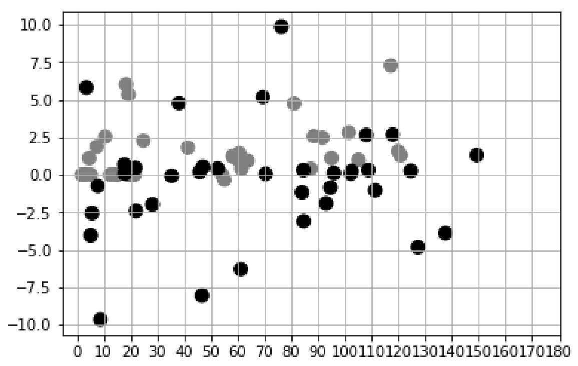
same rate as SS. RH remains relatively constant in its time-to-best, indicating its great speed in obtaining a solution is expected for a greedy-search based approach. Moreover, the consistency of RH *ttb* means that no further improvements are made after a couple of seconds, indicating that the searching scheme is stuck in a local optimum.

The overall SS results for both instance sets, in terms of percentage gaps versus the time-to-best, are shown in Figure 7. The circles represent results of problem instances and are differently colored depending on their main parameters This way, it is possible to discern the effect of the problem instance's parameterization on the obtained solution's quality and speed. Figure 7(a) shows that the improved solutions with respect to CPLEX mostly appear for $|I| = \{100\}$, which are more computationally hard to solve than problem instances for $|I| = \{50\}$. This is even more apparent in Figure 7(b), where the instances with $|I| = 1000$ are ground around the -10% gap and are almost all obtained past the 30 second mark, while the instances with $|I| = 500$ are dispersed between -50% and -20% gap. Figures 7(c) and 7(d) show that the best solutions are found for the greatest configurations size, i.e. $|C| = 1000$ and $|C| = 10000$ of the *Small* and *Large* sets, respectively. It seems that the availability of a larger space of solutions, increases the likelihood of finding improved solutions by our SS. Figure 7(e) shows that for $\beta = 20$ the solutions for the *Small* instances are found the soonest, and the same applies for the *Large* as shown in Figure 7(f).

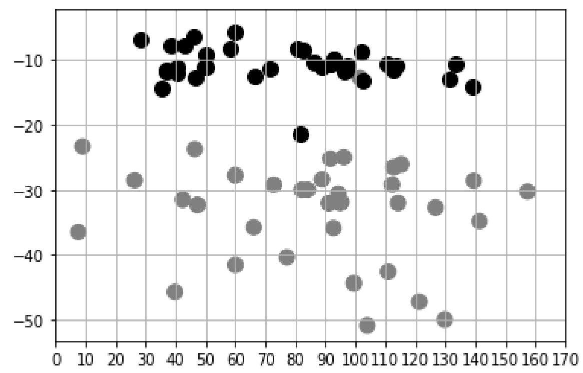
Finally, in Figure 8, we also analyze the *ttb* when solving for the *Extra-Large* instances for the different time limits and calibrations. We see that the *ttb* for SS follows a similar progression in all cases, with calibration RS_4 giving the shortest time overall. For RH, the *ttb* is better than SS for all cases. Therefore, we recommend for SS the use of calibration RS_2 when the solution quality is of high priority and calibration RS_4 when the *ttb* is deemed more valuable. We observe that RH, being a greedy heuristic, runs much more quickly than SS for *Extra-Large* instances of the problem. However, SS can find better quality solutions and help benchmark, simulations, and planning-level decision support given enough time.

6. Conclusions

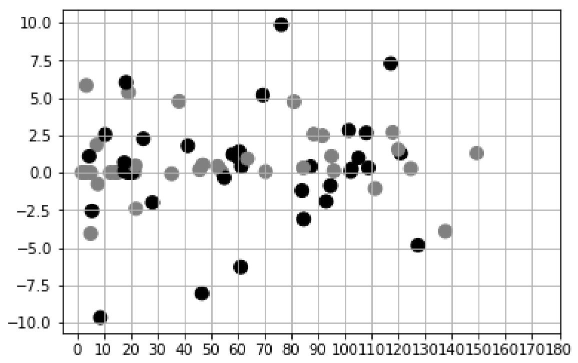
The design of a physical DB is a complex process involving several phases. In this paper, we have focused on the last phase of this process, optimizing data structures' assignment to a specific



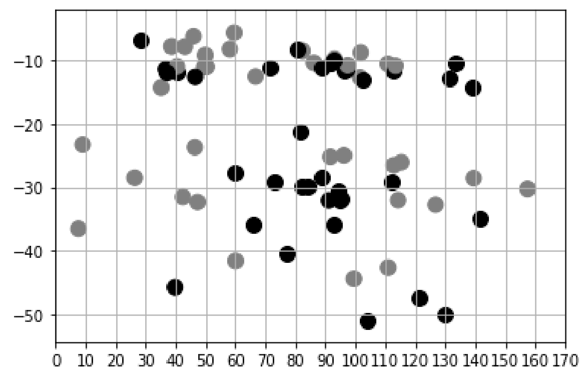
(a) *Small* instances (gray for $|I| = 50$, black for $|I| = 100$)



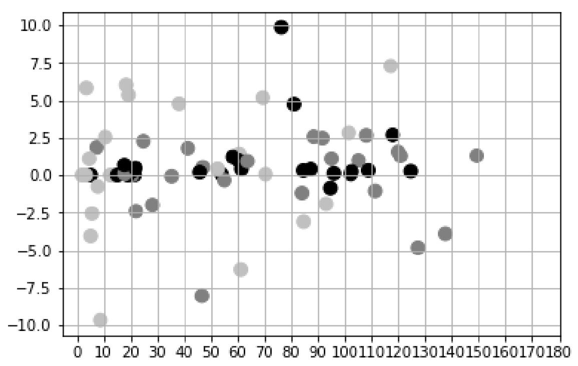
(b) *Large* instances (gray for $|I| = 500$, black for $|I| = 1000$)



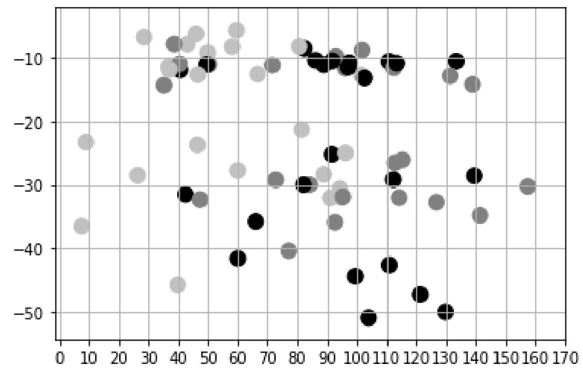
(c) *Small* instances (gray for $|C| = 500$, black for $|C| = 1000$)



(d) *Large* instances (gray for $|C| = 5000$, black for $|C| = 10000$)



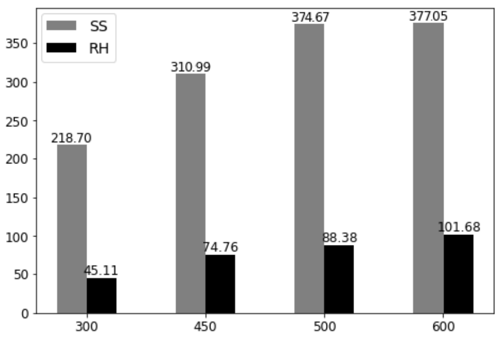
(e) *Small* instances (gray for $\beta = 20$, dark gray for $\beta = 50$, black for $\beta = 80$)



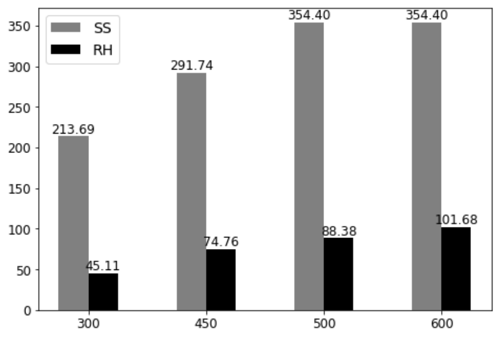
(f) *Large* instances (gray for $\beta = 20$, dark gray for $\beta = 50$, black for $\beta = 80$)

Figure 7: Percentage gap with respect to benchmarks (y-axis) vs time-to-best in seconds (x-axis) for: (a), (c), and (e) *Small* instances; (b), (d), and (f) *Large* instances. The parameters analyzed are: (a) and (b) number of indexes; (c) and (d) number of configurations; (e) and (f) value of β.

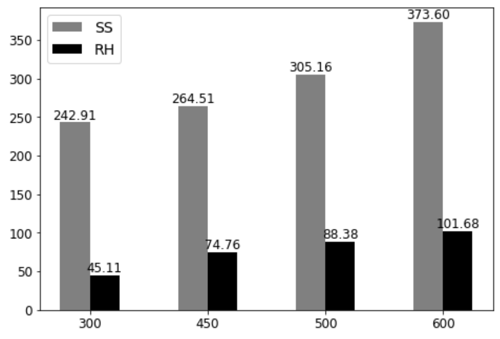
workload of queries. Therefore, we have proposed an Index Selection Problem considering configurations of indexes and a limited amount of memory simultaneously. The problem has been modeled as a generalization of the Uncapacitated Facility Location Problem with budget constraint through an Integer Linear Programming formulation. We have then developed a tailored Scatter Search



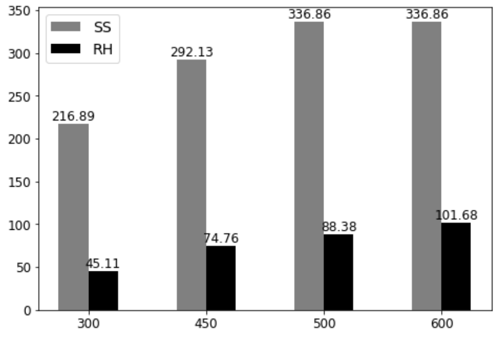
(a) RS_1



(b) RS_2



(c) RS_3



(d) RS_4

Figure 8: Average time-to-best in seconds (y-axis) for SS solutions (gray bar) and RH ones (black bar) for different run-time limits on *Extra-Large* instances, with different parameters calibrations.

meta-heuristic by exploiting several features of the problem's combinatorial structure. Our algorithm's accuracy and efficiency are tested with respect to CPLEX and a state-of-the-art heuristic method (RH) over an extensive set of instances. CPLEX has resulted in being very unreliable even when a significant amount of time is available. At the same time, RH (due to its greedy nature) runs quickly but provides very low-quality solutions for almost all the types and dimensions of instances. It follows that our SS method is the most suitable approach when the memory constraints are very binding, as it happens in several practical applications. The proposed solution method's excellent performance makes it possibly embeddable into realistic DB design software and usable not only for on-line developments but also for benchmarking other approaches in terms of quality.

Several future research lines can be delineated. First, our basic ISPwC can be extended by including specific features coming from the application at hand, such as incompatibilities among indexes or non-trivial functions, to derive the configuration gain for the marginal gain of the composing indexes. In that case, our algorithm could represent a solid basis for constructing an efficient solution procedure. Second, a combined approach of the best aspects of our Scatter Search and RH can be attempted, where an initial solution is obtained quickly by using the latter and then improved by the former. Third, given the high stochasticity affecting the possible workload of a DB, we could explicitly consider a model with uncertain parameters to find not only good but also robust solutions.

Acknowledgments

The authors are grateful to Dr. Riccardo Mereu (Politecnico di Torino, Italy) for his significant contribution in the development of an early version of the algorithm proposed in this paper and to Dr. Jann Kossmann (*Hasso Plattner Institute* in Potsdam, Germany) for having shared a working and customizable version of the RH method. The authors are also grateful to four anonymous reviewers for their valuable comments.

References

Ameri, P., 2016. On a self-tuning index recommendation approach for databases. In: 2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW). pp. 201–205.

Ameri, P., Meyer, J., Streit, A., 2015. On a new approach to the index selection problem using mining algorithms. In: 2015 IEEE International Conference on Big Data (Big Data). pp. 2801–2810.

Asgharzadeh Talebi, Z., Chirkova, R., Fathi, Y., 2013. An integer programming approach for the view and index selection problem. Data & Knowledge Engineering 83, 111–125.

Boronski, R., Bocewicz, G., 2014. Relational database index selection algorithm. In: International Conference on Computer Networks. Springer, pp. 338–347.

Bruno, N., 2011. Automated Physical Database Design and Tuning, 1st Edition. CRC Press, Inc., USA.

Bruno, N., Chaudhuri, S., 2005. Automatic physical database tuning: A relaxation-based approach. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data. SIGMOD '05. Association for Computing Machinery, New York, NY, USA, p. 227–238.

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

Caprara, A., Salazar González, J. J., 1996. A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem. Top 4 (1), 135–163.

Caprara, A., Salazar González, J. J., 1999. Separating lifted odd-hole inequalities to solve the index selection problem. Discrete Applied Mathematics 92 (2-3), 111–134.

Chaudhuri, S., Datar, M., Narasayya, V., 2004. Index selection for databases: a hardness study and a principled heuristic solution. IEEE Transactions on Knowledge and Data Engineering 16 (11), 1313–1323.

Chaudhuri, S., Narasayya, V. R., 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In: VLDB. Vol. 97. Citeseer, pp. 146–155.

Choenni, S., Blanken, H., Chang, T., 1993. Index selection in relational databases. In: Proceedings of ICCI'93: 5th International Conference on Computing and Information. IEEE, pp. 491–496.

Comer, D., 1978. The difficulty of optimum index selection. ACM Transactions on Database Systems (TODS) 3 (4), 440–445.

Corberán, A., Fernández, E., Laguna, M., Martí, R., 2002. Heuristic solutions to the problem of routing school buses with multiple objectives. Journal of the Operational Research Society 53 (4), 427–435.

Dash, D., Polyzotis, N., Ailamaki, A., Mar. 2011. Cophy: A scalable, portable, and interactive index advisor for large workloads. Proc. VLDB Endow. 4 (6), 362–372.

Fadda, E., Manerba, D., Cabodi, G., Camurati, P., Tadei, R., 2021. Comparative analysis of models and performance indicators for optimal service facility location. Transportation Research Part E: Logistics and Transportation Review 145, nb. 102174.

Finkelstein, S., Schkolnick, M., Tiberio, P., Mar. 1988. Physical database design for relational databases. ACM Transactions on Database Systems 13 (1), 91–128.

Fotouhi, F., Galarce, C. E., 1989. Genetic algorithms and the search for optimal database index selection. In: Great Lakes CS Conference on New Research Results in Computer Science. Springer, pp. 249–255.

Ghamlouche, I., Crainic, T. G., Gendreau, M., 2003. Cycle-based neighbourhoods for fixed-charge capacitated multicommodity network design. Operations Research 51 (4), 655–667.

Glover, F., 1977. Heuristics for integer programming using surrogate constraints. Decision Sciences 8 (1), 156–166.

Glover, F., 1997. A template for scatter search and path relinking. In: European Conference on Artificial Evolution. Springer, pp. 1–51.

Gupta, H., Harinarayan, V., Rajaraman, A., Ullman, J. D., April 1997. Index selection for olap. In: Proceedings 13th International Conference on Data Engineering. pp. 208–219.

Heeren, C., Jagadish, H. V., Pitt, L., 2003. Optimal indexing using near-minimal space. In: Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. PODS '03. ACM, New York, NY, USA, pp. 244–251.

Kratica, J., Ljubic, I., Tošic, D., 2003. A genetic algorithm for the index selection problem. In: Proceedings of the 2003 International Conference on Applications of Evolutionary Computing. EvoWorkshops'03. Springer-Verlag, Berlin, Heidelberg, pp. 280–290.

Martí, R., Laguna, M., Glover, F., 2006. Principles of scatter search. European Journal of Operational Research 169 (2), 359–372.

Papadomanolakis, S., Dash, D., Ailamaki, A., 2007. Efficient use of the query optimizer for automated database design. In: VLDB '07: Proceedings of the 33rd international conference on very large data base. pp. 1093–1104.

Rahimi-Vahed, A. R., Rabbani, M., Tavakkoli-Moghaddam, R., Torabi, S. A., Jolai, F., 2007. A multi-objective scatter search for a mixed-model assembly line sequencing problem. Advanced Engineering Informatics 21 (1), 85–99.

Schlosser, R., Kossmann, J., Boissier, M., 2019. Efficient scalable multi-attribute index selection using recursive strategies. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE). IEEE, pp. 1238–1249.

Subotić, P., Jordan, H., Chang, L., Fekete, A., Scholz, B., Oct. 2018. Automatic index selection for large-scale datalog computation. Proc. VLDB Endow. 12 (2), 141–153.

Valentin, G., Zuliani, M., Zilio, D. C., Lohman, G., Skelley, A., 2000. DB2 advisor: An optimizer smart enough to recommend its own indexes. In: Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073). IEEE, pp. 101–110.

Yu, P., Chen, M.-S., Heiß, H.-U., Lee, S., 1992. Workload characterization of relation database environments. IEEE Transactions on Software Engineering 18, 347–355.