Dynamic Query Refinement for Interactive Data Exploration[*]

Alexander Kalinin[†]
Vertica, a Micro Focus Company
Cambridge, Massachusetts
allex2001@gmail.com

Ugur Cetintemel
Brown University
Providence, Rhode Island
ugur@cs.brown.edu

Zheguang Zhao
Brown University
Providence, Rhode Island
zheguang_zhao@brown.edu

Stanley Zdonik
Brown University
Providence, Rhode Island
sbz@cs.brown.edu

ABSTRACT

Queries that can navigate large search spaces to identify complex objects of interest cannot be efficiently supported by traditional DBMSs. Searchlight is a recent system that aims to address this fundamental shortcoming by deeply yet transparently integrating Constraint Programming (CP) logic into the query engine of an array DBMS. This hybrid model enables exploration of large multi-dimensional data sets progressively and quickly.

Fast query execution is only one of the requirements of effective data-exploration support. Finding the right questions to ask is another notoriously challenging problem, given the users' lack of familiarity with the structure and contents of the underlying data sets, as well as the inherently fuzzy goals in many exploration-oriented tasks. To this end, in the context of Searchlight, we study the modification of initial query parameters at run-time. We describe how to dynamically refine (i.e., relax or tighten) the parameters of a query, based on the result cardinality desired by the user and the live query progress. This feature allows users to iterate over the datasets faster and without having to make accurate guesses on what parameters to use. Our experimental results show that the proposed techniques introduce little or no overhead while yielding considerable time savings compared to user-driven, manual query refinements. The result is a system that not only optimizes machine resource usage but also reduces user effort.

1 INTRODUCTION

Consider a researcher working with the MIMIC II dataset [1], which contains medical information for a number of ICU patients over a large period of time. Assume that she is studying historical ABP (Arterial Blood Pressure) signal readings and wants to identify time intervals that satisfy the following constraints:

- The length of the time interval can be from 8 to 16 seconds, and it can start at any point in time.
- The average signal amplitude must be within [150, 200] range for the interval.
- The maximum amplitude of the signal over the interval must exceed the maximum amplitude of the signal over its left and right neighborhoods by at least 80. The left (right) neighborhood is defined as an 8-second interval to the left (right) of the main interval.

This seemingly simple *search query* is difficult to express and even harder to optimize using traditional query languages and DBMSs [10]. Recent systems, such as Searchlight [10, 11], extends SciDB [2] with constraint-based search and optimization support, effectively integrating two mature technologies (DBMS and CP-based solver) to address queries that operate on large search spaces and over large data sets.

Now imagine that the user gets *zero* results after running the initial query, as illustrated in the top part of Figure 1. It turns out that the query was over-constrained! She then tries to guess the correct constraint parameters manually by *relaxing* the average amplitude constraint: now the average ABP amplitude must be within the [150, 250] range. When she runs this new query, she gets flooded with a large number of intervals, many of which overlap, as shown in the middle band of Figure 1. Since such a result can be overwhelming for her to parse and study, expecting that there might be more focused and "better" initial results she could work with, she then tries to *tighten* the interval to [150, 220]. After running the query again, she gets a reasonable number of results that she can explore in detail, as illustrated at the bottom of Figure 1.

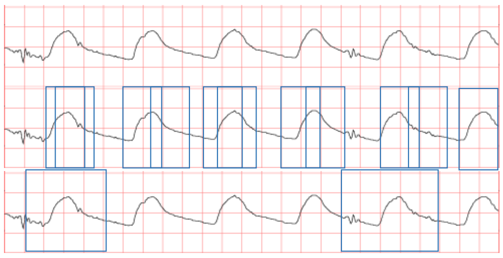


Figure 1: Exploring the ABP waveform data. *Top*: original query result. *Middle*: an over-relaxed query result. *Bottom*: final query result.

This scenario illustrates a number of problems:

- The user has to go through a series of guesses to identify a query that outputs a desirable number of results. This is often a frustrating and cumbersome process, especially if the user has limited knowledge about the data, which is often the case when exploring new data sets.
- The process of manual refinements (i.e., relaxation and tightening) might be quite complex even for seemingly

[*]This work was funded in part by NSF IIS-1526639.
[†]This work was done while the author was a PhD student at Brown University.

simple queries. In the earlier example, the user can modify the average amplitude constraint, or the two neighborhood constraints, or any combination of those, which collectively create an exponential number of possibilities.
- With manual refinements, no guarantees can be given for the final result. In general, users prefer to get results that are as close as possible to the original query parameters.

We address these problems in the context of Searchlight, which gives us an open DBMS platform to develop and test our solutions. At the same time, we emphasize that these problems would arise in any system supporting constraint-based queries and traditional solving technology. We also note that these issues are amplified when dealing with large datasets, as the query times can be larger and it becomes even more critical to perform such query iterations quickly and efficiently.

1.1 Motivation

A common initial step in data exploration is the identification of a small number of interesting cases that the user dives into for deeper study. In such cases, query refinements are applied to assure a target result cardinality to meet a "budget" constraint typically on user time or money. For example, a medical researcher may have the budget to run a survey on a specific number of patients with certain characteristics. A business may have a discount budget that is sufficient only to a specific number of customers within a given campaign. An advancement officer of a university may have a limited time to call a certain number of alumni on a given day. An astronomy researcher may have the time to study a limited number of celestial objects for a research project. In all these cases, the ability to have the system automatically tweak the query to produce a result set of a desirable size is very helpful.

Beyond these generic use scenarios, we have anecdotal evidence from a demonstration of waveform data exploration [11]. In this demo, the users were given the ability to fill in the parameters for template queries, such as signal amplitude, interval length, etc. However, since they had very limited knowledge about the presented dataset, their queries often output either too many results (sometimes thousands) or nothing at all. Both of the outcomes were frustrating, and the users had to go through a number of trial-and-error runs to identify a small set of results that is amenable to manual deeper inspection.

1.2 Query Refinement Approaches

Ideally, any system should perform any refinements automatically by detecting if the query needs to be modified during its evaluation without specific directives from the user. If there is a need for relaxing the query, it should choose the constraints and the degree of modification such that the final results are *closest* to the original constraint parameters according to the provided distance function. In the case of tightening, it should rank the results and output the top ones according to the specified ranking function. For performance optimization, it should reuse as much computation as possible to minimize the overhead from multiple searches. At the same time, if there is no need to modify the query, the automatic approach should not incur any significant overhead.

Query refinements have been studied in the context of relational DBMSs [4, 6–8, 12–18]. Similar approaches can be extended to array DBMSs (such as SciDB) as well. However, the query described in the example belongs to a different kind of *search* queries, which is the reason why an engine such as Searchlight that supported CP was used in the first place instead of the original array DBMS. As we argue in Section 6, existing methods cannot be easily applied to constraint-based search queries. These methods generally assume that either the result cardinality can be easily estimated for the query, or appropriate range-based indexes (e.g., B-trees or R-trees) exist over the objects of interest (which are time intervals for the example above), so that the search space of all possible objects could be traversed efficiently. Due to the ad-hoc nature of search queries, however, the result cardinality is hard to estimate. At the same time, since objects of interest are defined by the query itself, as part of the constraint specification, indexing becomes infeasible [10].

Another approach, possibly applicable to some simple search queries, would be to rewrite such queries in SQL and execute them on a relational (or array) engine, opening the possibility of using the existing query refinement techniques. However, previous research [9, 10] suggests that executing such search queries in a traditional engine, while possible, incurs significant performance penalties. More importantly, such conversion results in very complex SQL queries, for which the applicability of existing refinement methods would be quite limited.

1.3 Overview of Dynamic Query Refinement

In a nutshell, Searchlight uses Constraint Programming (CP) to perform the search over an in-memory *synopsis* of the original data. The solver dynamically builds the search tree, possibly pruning parts of the tree that cannot satisfy the constraints. We observe that the main reason the original query fails and, thus, produces an empty result is search pruning. If the query needs relaxation, only the previously pruned parts of the search space need to be revisited. Thus, we track these parts and later "replay" the search over them, if needed. The replaying is guided by a user-provided distance function — more promising replays are explored first. It is important to mention that this replay happens automatically, when we detect the absence of the desired number of results. At the same time, replaying is performed as part of the original query, thus removing the need to re-explore previously finished parts of the search space, resulting in considerable time savings. From the logical perspective, this approach can be seen as introducing an objective function and searching for results that minimize it. This results in a very natural extension of the traditional CP model that already supports objective function-based search. The only piece of information required from the user is the desired cardinality of the result k and, possibly, the distance function $d()$ [1]. Our approach guarantees to produce top-k results closest to the original constraints (i.e., minimizing $d()$).

Query tightening can be seen as a dual problem for query relaxation. Thus, the two work closely together. If the number of results at some point during the search exceeds the desired result cardinality, we switch the approach to query tightening, which essentially ranks all results according to the specified ranking function $r()$ [2]. The main idea behind our approach involves introducing a *dynamic* ranking constraint at that particular moment during the search that restricts the final result to top-k objects maximizing $r()$. As in the case of relaxation, it can logically be seen as introducing an objective function with the maximization

[1]We provide built-in distance functions by default.
[2]Some built-in ranking functions are provided by default as well.

goal. We call such a constraint dynamic since its parameters are constantly updating depending on the current result.

1.4 Contributions

Our main contributions are as follows:

- We introduce a novel distributed relaxation and tightening framework for search queries, which automatically detects the need for modifying the query at run-time and performs query refinements without the need for any user intervention. The user just needs to specify the target result cardinality and, optionally, the distance and ranking functions.
- The framework is general and applicable to different types of constraints. It works for any constraint of form $a \leq f_c() \leq b$, where $f_c()$ is an arbitrary expression, including User-Defined Functions (UDFs). It can be extended to other types of constraints, with the introduction of a meaningful distance and/or ranking measure. If desired, it can be even applied to other engines beyond Searchlight, provided they follow the CP execution model.
- Our implementation of this framework operates at the DBMS engine level and extends it in a natural way that is compatible with a generic CP model. Due to such a seamless integration, it does not interfere with the existing query processing, does not impact performance of queries that do not require relaxation/tightening, and leverages existing DBMS engine features.

We performed an extensive experimental evaluation over synthetic and real (MIMIC II) data, which we discuss in Section 5. Our results reveal tremendous time savings compared to manual refinements and very low overhead.

The rest of the paper is organized as follows. In Section 2 we describe Searchlight as the DBMS/CP platform we use to explore our solutions. Section 3 presents the formal model behind our relaxation/tightening framework. Section 4 describes design and implementation of our solutions. Section 5 presents the experimental evaluation Section 6 discusses the related work and Section 7 concludes the paper.

2 SEARCHLIGHT BACKGROUND

Searchlight is an extension of a traditional query processing engine for efficient execution of search queries. It introduces Constraint Programming (CP) methods to the query processing and operates *inside* the engine. The user submits a search query in form of a constraint program (decision variables and constraints over them).

Searchlight processes the query by creating a number of CP *Solvers* that perform the search on an in-memory *synopsis* of the original data. Since synopsis is a lossy compression of the original data, the CP-provided results (solutions) might contain false positives. Thus, the results need to be verified over the original data. This is done by another Searchlight component called *Validator*. The two components work concurrently to facilitate online answering. When Validator confirms a solution, it is immediately output, while false positives are filtered out. While Searchlight does not have the notion of a query plan, each execution can be imagined as a pipeline between two "operators": Solver and Validator. The Solver receives the query, outputs a stream of solutions (tuples) to the Validator, while the latter filters out false positives and outputs the final solutions to the user. Searchlight is implemented as part of the SciDB query engine and uses its infrastructure to distribute both search space and data across the cluster. Thus, Solvers and Validators exhibit both multi-node and multi-core parallelism. The details of query processing are out of scope of this paper, and the in-depth discussion can be found in the Searchlight paper [10].

There are no specific limitations on types of queries Searchlight can work with, since CP solvers are quite general. Users can use any of the constraints generally found in CP solvers. At the same time, most useful types of queries tend to use aggregate functions to assess regions in the data. For example, the query presented in the introduction section can be represented in Searchlight as follows:

- Decision variable x defines the start of the resulting interval. Its domain is the entire length of the recorded data, since the interval can start anywhere.
- Variable lx defines the length of the interval, $lx \in [8, 16]$.
- The amplitude constraint: $avg(x, x + lx, ABP) \in [150, 200]$, where $avg()$ is a built-in Searchlight aggregate. We will denote it c_1 for future reference.
- The left neighborhood constraint: $|max(x, x + lx, ABP) - max(x - 8, x, ABP)| \geq 80$. The right neighborhood is similar. We will denote them as c_2 and c_3 respectively.

When the Solver receives a CP specification similar to the above, it dynamically builds a *search tree* for the query. An example of such a tree for the query above can be seen in the left part of Figure 2. The nodes represent the current variable domains (search states), while edges represent Solver decisions that lead to the corresponding search states. Children's variable domains are subsets of the parent's ones, and leaves have the variables *bound* (i.e., the domains are scalars). The decision process (search heuristic) is tunable, can be selected and modified by the user [10]. The search process itself is performed in a traditional backtracking way. If, while building the tree, the Solver establishes a violation of query constraints (by relying on estimations from the synopsis), the entire corresponding sub-tree is pruned from the search (marked with the red "no" symbols in the figure), and is never built or visited. When a leaf of the tree is reached (i.e., the variables are bound to scalar values), the corresponding variable assignment (solution) is passed to the Validator.

3 QUERY REFINEMENT MODEL

In this section we discuss our relaxation/tightening framework. While described in the context of Searchlight, the model does not depend on its implementation details and is primarily driven by well established CP concepts. Throughout the paper, we use query "constraining" as a synonym for "tightening".

Each CP search query consists of a number of decision variables X and constraints C. Given such a query the search framework either outputs all results or proves there is none. However, if the user specifies the desired *cardinality* of the result k, the framework behaves differently depending on the outcome of the original query:

- The query outputs exactly k results. In this case the behavior stays the same.
- The query outputs a set of results $R = r, |R| < k$. In this case it is automatically modified to produce additional $k - |R|$ results. The additional results are guaranteed to minimize the specified $RD(r)$ function (discussed below). This is query relaxation.

- The query outputs $|R| > k$ results. In this case the framework effectively ranks the results based on the $RK(r)$ function (discussed below), and the query returns top-k results according to $RK(r)$. This is query constraining.

When relaxing or constraining a query, we consider only range-based constraints of form $a \leq f(X) \leq b$. The nature of $f()$ is not important. It might be an arbitrary algebraic expression containing User-Defined Functions (UDFs), built-in functions and variables from X. In our running example $f()$ is the $avg()$ function for constraint c_1 and the $max(x, x + lx) - max(x - 8, x)$ expression for c_2. We generally treat $f()$ as black boxes, but we assume the knowledge of values a', b' at every node of the search tree, such that $a' \leq f(X) \leq b'$ for all possible values of X at that node. This information is readily available as part of the search process and is used to make search decisions (for building the tree) and prune parts of the tree. In Searchlight a', b' are derived from the each node's variable domains by using the synopses to estimate constraint functions [10]. Thus, no modification to the query engine should be required in this regard. By default, we consider all range-based constraints for relaxation/constraining, but the user can exclude any of them from the process. We denote constraints considered for relaxation (constraining) as C^r (C^c). C^r does not necessarily equal C^c.

The relaxation and constraining processes are based on result ranking via separate relaxation and constraining ranking functions. In the two following sections we describe the default functions we use. Then we discuss the custom ranking functions requirements.

3.1 Query Relaxation Model

Assume a constraint $c \in C^r : a \leq f_c(X) \leq b$ and a result r such that $f_c(r) = t$. We define the relaxation distance $RD_c(r)$ as follows:

$$RD_c(r) = \begin{cases} 0 & \text{if } a \leq t \leq b \\ \frac{t-b}{max(f_c(X))-b} & \text{if } t > b \\ \frac{a-t}{a-min(f_c(X))} & \text{if } t < a \end{cases}$$

Then, the total relaxation distance $RD(r)$ is:

$$RD(r) = \max_{c \in C^r} w_c RD_c(r),$$

where $w_c \in [0, 1]$ are constraint weights, which can be defined by the user. By default, $w_c = 1$.

We selected the $RD()$ definition above just as a suitable *default*, aiming at providing reasonable out-of-box experience. As we discuss in Section 3.3, users can choose their own functions, provided they respect certain requirements. In general, p-norm is a logical choice when some distance between query points needs to be measured [4]. We chose max ($p = \infty$) to penalize results where some outlier constraints have large $RD_c()$ values. This allows us to limit the distances of final results. A weighted sum ($p = 1$) or Euclidean distance ($p = 2$) are other viable choices.

The denominators in $RD_c(r)$ require further explanation. In general, $f_c()$ from different constraints might have different scales. For example, one constraint might deal with ages (e.g., $f_{c_1}() \in [0, 150]$), while another with similarities (e.g., $f_{c_2}() \in [0, 1]$). That is why we perform $[0, 1]$-normalization of each $RD_c(r)$ by dividing it by the maximum possible difference in values. Min/max values for $f_c()$ can be usually derived from the obvious domain restrictions (e.g., age cannot exceed 150). We additionally allow users to specify the max/min values with the query, giving them more control over relaxation: we will not relax the corresponding constraints beyond the specified min/max values. We will use $RD_c(r)$ to denote the normalized distances, as well as the original ones, where appropriate.

In addition to $RD(r)$ for each result r we define $VC(r)$ as the number of constraints from C^r violated by r divided by $|C^r|$. Thus, $VC(r) \in [0, 1]$ is the normalized number of violated constraints. Then, we define the total relaxation penalty for r as:

$$RP(r) = \alpha RD(r) + (1 - \alpha)VC(r)$$

The $RP()$ allows users to prefer results with smaller number of violated constraints, which is especially important in cases when $RD(r)$ "looses" information about individual constraints (e.g., as in our choice of max). α controls the degree of the preference (the default is 0.5). We picked weighted sum as a suitable default, giving the user the choice between two criteria, $RD()$ and $VC()$. However, this default is not essential for the proposed framework. $RP()$ can be changed, and other criteria can be incorporated in the formulation, if required.

The model provides the following *relaxation guarantee*: if the user submits query Q with the cardinality requirement k, the query outputs at least k results r with the lowest $RP(r)$ values possible.

The definition above naturally incorporates queries in no need of relaxation, with $\geq k$ results. If r satisfies the original constraints, $RP(r) = 0$. Thus, the guarantee is automatically fulfilled. If the user chose to specify tight min/max bounds for some $f_c()$, we might not be able to find k results in case there are not enough results satisfying even *maximally* relaxed constraints (i.e., $\min f_c() \leq f_c() \leq \max f_c()$). This is because we effectively treat such $f_c()$ specification as a "hard" constraint.

Let us revisit the running MIMIC example. Assume the user wants $k = 3$ results and $C^r = \{c_1, c_2, c_3\}$. Additionally, $avg()$ and $max()$ values for the ABP signal lie within $[50, 250]$. Then a possible search might progress as follows:

1. A result $r_1 = (180, 85, 85)$ is found (we write a result as a tuple of $f_i()$ values). Since it satisfies the constraints, $RP(r_1) = 0$ and it is output to the user.
2. $r_2 = (190, 80, 90)$. Since, $RP(r_2) = 0$, it is output.
3. Searchlight cannot find any more results, so it starts relaxing the query.
4. $r_3 = (160, 70, 60)$ is found. It violates c_2 and c_3. For r_3: $RD_{c_1} = 0, RD_{c_2} = \frac{10}{80} = 0.125, RD_{c_3} = \frac{20}{80} = 0.25$. Thus, $RD(r_3) = 0.25$, and $RP(r_3) = \frac{1}{2}(0.25 + \frac{2}{3}) = 0.458$.
5. $r_4 = (130, 80, 80)$ is found. For r_4: $RD_{c_1} = \frac{20}{100} = 0.2, RD_{c_2} = 0, RD_{c_3} = 0$. Thus, $RD(r_4) = 0.2$, and $RP(r_4) = \frac{1}{2}(0.2 + \frac{1}{3}) = 0.267$. Since $RP(r_4) < RP(r_3)$, r_3 is discarded, and r_4 is put into the result.

3.2 Query Constraining Model

The query performs constraining only when the number of results exceeds the k required by the user. That means during constraining each result r_i satisfies all constraints in C^c. For each function $f_c(X)$ from constraints in C^c the user can specify her preference in form of maximization or minimization of the function. For example, if some constraint's $f_c(X)$ is a property like the amplitude of a signal, the user might prefer large values of $f_c(X)$. On the other hand, if $f_c(x)$ is some spatial distance, the user might prefer smaller values. For each constraint $c \in C^c : c = a \leq f_c(X) \leq b$ and result $r : f_c(r) = t$ we define the ranking function $RK_c(r)$ as follows:

$$RK_c(r) = \begin{cases} \frac{b-t}{b-a} & \text{if } c \text{ is being maximized} \\ \frac{a-t}{b-a} & \text{if } c \text{ is being minimized} \end{cases}$$

Since r satisfies the constraints, $a \leq t \leq b$. As in the case of query relaxation, we normalize $RK_c()$ to $[0, 1]$ to account for the possibility of different scales for $f_c()$. If the interval is half-open, i.e., a or b is not specified, a suitable domain boundary for $f_c()$ can be used instead.

We define the full rank of result r as:

$$RK(r) = 1 - \sum_{c \in C^c} w_c RK_c(r),$$

where $w_c, 0 \leq w_c \leq 1 \wedge \sum_c w_c = 1$ represent the constraint weights to prioritize some constraints over others. By default, $w_c = \frac{1}{|C^c|}$. Note, $RK(r)$ assigns higher ranks to better results, which is more natural to the user. As in the case of $RD()$, any p-norm could be a reasonable choice for $RK()$. We saw the weighted sum providing meaningful results in practice. In addition, it allows us to demonstrate that the approach is flexible to the choice of distance functions.

Assuming these definitions, the model provides the following *constraining guarantee*: when the user submits query Q with the cardinality requirement k, if Q has at least k results r, the query outputs at most k results with highest $RK(r)$ possible. Note, if the query does not have at least k results, the relaxation will be performed instead.

Revisiting the running MIMIC example, let $C^c = \{c_1, c_2, c_3\}$, $w_{c_i} = \frac{1}{3}$. Let us assume the user prefers maximization for all constraints and wants a single result. Note, that f_{c_1} and f_{c_2} has the maximum value, 200, derived from the domain. The search might progress as follows:

(1) A result $r_1 = (160, 100, 100)$ is found. Its rank is $RK(r_1) = 1 - \frac{1}{3}(\frac{40}{50} + \frac{100}{120} + \frac{100}{120}) = 0.178$.
(2) A result $r_2 = (150, 80, 85)$ is found. Its $RK(r_2) = 0.014$. Since $RK(r_2) < RK(r_1)$, r_2 is discarded.
(3) The next result is $r_3 = (190, 120, 120)$. Its $RK(r_3) = 0.289$. Since $RK(r_3) > RK(r_1)$, r_1 is discarded, and r_3 becomes the new top-1.

In addition to the scalar ranking approach just described we support another popular vector-based ranking called *skyline* [3]. In that case the query simply outputs non-dominated results, where a result is a vector of values of $f_c(), c \in C^c$ (as in the example above). By definition, V dominates W, iff $\forall i : v_i \geq w_i \wedge \exists i : v_i > w_i$. The meaning of $>$ for each $f_c()$ is defined by the user's minimization/maximization preference, as in the scalar case. For skyline, however, we cannot guarantee that the number of results will not exceed k, since non-dominated vectors are not comparable.

3.3 Approach Customization

In addition to the default penalty and ranking functions discussed above, the user can add their own custom functions. The functions may be called by the query engine during the search at any search tree node, where some variables may still be unbound. This means the custom $RP()/RK()$ functions must be able to output the penalty/rank interval for all possible solutions contained in the corresponding sub-tree. Our implementation provides the user with the current variable domains and synopsis-based intervals for all constraint functions at any node. This information should be enough to compute the required bounds.

The functions must conform to the certain requirements to ensure the relaxation/constraining guarantees and proper performance. For a custom RP returning ranges $[lp, hp]$, the requirements are:

- $RP() \geq 0$, with larger values corresponding to worse relaxation. All results satisfying the original query must have $RP() = 0$.
- $lp = hp$ at solutions (leaves) of the tree, since the variables are bound there.
- $RP()$ cannot underestimate lp, which means it cannot be greater than the minimum of penalties for all possible solutions at the corresponding sub-tree.

The requirements for the RK functions are similar, with the only differences that $RK()$ assigns larger values to better candidates and, thus, should not underestimate hp. The user can define her own dominance measure for the skyline ranking as well. The corresponding function is periodically called over the current top-k results to determine if any of the current sub-tree solutions might enter the top-k.

In principle, customization can go beyond static ranking function. *Dynamic* functions is an interesting extension, where the ranking functions may change depending on the results already found, e.g., the user might want to prefer "diversity" among the relaxed results so that their relaxation distances differ by at least the specified amount. This can be accomplished by introducing new constraints depending on the results already discovered. Other dynamic modifications might be possible as well, depending on the user's preference. Studying such richer functionality is left for future work.

4 QUERY RELAXATION AND CONSTRAINING

In this section we describe the implementation of the relaxation/constraining model. Our approach is general and can be applied to other CP-based engines.

In general, a CP solver dynamically builds and traverses the search tree. The dynamic nature of the search process allows for modification of existing constraints and the addition of new ones. Thus, if a query does not produce enough results satisfying the original constraints, we can revisit some parts of the search tree with modified (relaxed) constraints. If a query produces too many results, we can introduce new dynamic constraints to prune results having smaller ranks than the already found ones.

The efficiency of the relaxation and constraining heavily relies on effective pruning. We exploit the following cases:

- During the main search at the Solver. This is the most effective point. It allows us to prune parts of the search tree with possibly large number of candidates and avoid unnecessary validations later.
- Just before validating the candidate at the Validator. This is effective in case Validators lag behind the Solvers, and their candidate queues grow large. In that case new ranking/penalty information about the current result might allow us to perform additional pruning and avoid expensive I/O.
- After validating the candidate at the Validator. Even if the candidate passes the validation over the real data, it is necessary to check them again with the up-to-date penalty/ranking values. However, this is is done only for correctness, with no performance benefits. These checks do not require any additional I/O.

4.1 Query Relaxation

A CP solver dynamically builds the search tree and validates query constraints at every node of the tree. A node either satisfies the constraints or *fails*. Successful nodes eventually lead to leaves, which produce candidate solutions. When the search is finished, if we have not found k results (the user's desired cardinality), we start revisiting parts of the search tree with modified, relaxed constraints. We do not have to revisit successful search nodes, since these nodes satisfied the *original* constraints and cannot yield any new solutions. Previously failed search nodes, on the other hand, could lead to new candidates satisfying the *relaxed* constraints. We call this process *fail replaying*.

When we encounter a failed search node, we prune the node as usual. However, we also record the current search state of the node:

- Current decision variable domains. This information is crucial when the fail is replayed later to resume the search from this exact point without revisiting any extraneous search nodes.
- The ranges $[a', b']$ for every function $f_c(), c \in C^r$. As we discussed in Section 3, these are available as part of the normal search process.

After this information has been obtained, we compute the best (*BRP*) and worst (*WRP*) relaxation penalties possible for the saved failed node (i.e., for the solutions in its sub-tree). For the built-in *RP* function this is straightforward from the definition. The custom function, as discussed in Section 3.3, must compute these values itself. After the relaxation penalties are computed, the fail is inserted in the priority queue ranked by the *BRP*.

If during the search at least k results are found, we just stop recording the fails. At the same time the constraining mechanism turns on, which we discuss further in Section 4.3. If the main search completes with less than k results, the relaxation is needed. We start replaying fails from the priority queue (i.e., minimizing *BRP*). To replay a fail, we do the following:

(1) A new CP search is initiated with the decision variables assigned the domains recorded at the fail. This can be seen as traveling back in time to the moment just before the fail.
(2) Each violated constraint is modified: $a \leq f_c() \leq b \rightarrow a' \leq f_c() \leq b'$, where $[a', b']$ is the recorded interval. This guarantees the search will not fail again when resumed.

The new search is handled exactly the same as the original one. Thus, it might fail again, at other search nodes further in the search tree. One example is when a previously valid constraint becomes violated due to more accurate synopsis estimations (those tend to become better closer to leaves). Such *repeated* fails are caught as the original ones and might be replayed later. During replays we explore only previously untouched parts of the search tree. No search nodes are ever revisited, which improves performance and guarantees the absence of duplicate results.

The discussed replay mechanism can be seen as naïve, since it does not allow any additional pruning at the search level. When we replay a fail, we relax previously failed constraints maximally, so they cannot fail again. Thus, we just relax constraints until the search reaches the leaves. The new candidates, however, do not necessarily belong to the best-k results. To improve the efficacy of pruning we take into account the Maximum Relaxation Penalty (*MRP*) among the already found results.

$MRP \in [0, 1]$ effectively defines the worst penalty a result can have to belong to the best-k, and it constantly changes during the execution. Let us first assume the built-in *RP* function. While the number of results found is less than k, the $MRP = 1$. When at least k results have been found, *MRP* might decrease. We modify the process as follows:

- When a fail is recorded, its *BRP* is compared with *MRP*. If $BRP > MRP$, the fail is discarded completely, since its search sub-tree cannot provide any useful candidates (i.e., with $RP(r) \leq MRP$).
- When a fail is selected for replaying, its recorded intervals $[a', b']$ are tightened according to the current *MRP* to improve pruning, since the constraints will be relaxed as minimally as possible. In addition, we repeat the *BRP* and *MRP* comparison, since *MRP* might have changed.

Let us discuss the interval tightening in more detail. Assume the built-in *RP*() function discussed in Section 3.1 with $\alpha \neq 0$. If $\alpha = 0$, the relaxation distance does not influence *RP*(), so no tightening is possible. Otherwise, to qualify for the result, all candidate solutions r must have $RP(r) \leq MRP$. Since $RP(r) = \alpha RD(r) + (1 - \alpha)VC(r)$,

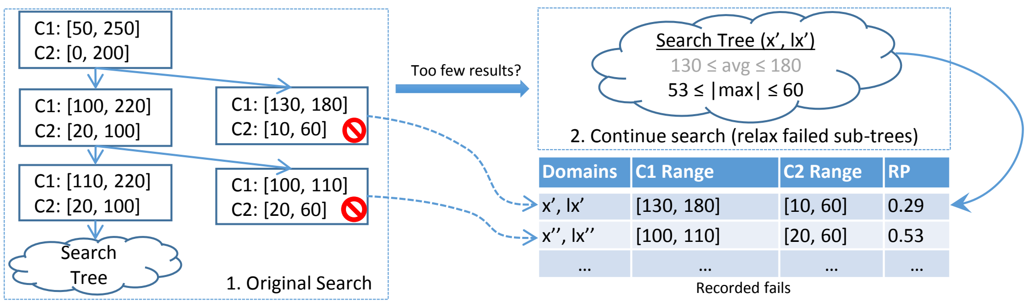
$$RD(r) \leq \frac{MRP - (1 - \alpha)VC(r)}{\alpha}.$$

Let us revisit the example from Section 3.1 and illustrate the above algorithm with Figure 2. In this figure the search nodes are rectangles with the synopsis values for the c_1 and c_2 functions (we do not show c_3). The search encounters two fails. The first one in the search order is the lower one, for which both c_1 and c_2 are violated, since $110 < 150$ and $60 < 80$. Its $BRP = \frac{1}{2}(max(\frac{40}{100}, \frac{20}{80})) + \frac{1}{2}\frac{2}{3} = 0.53$. The fail is recorded into the table. Then the search encounters the upper fail, for which only c_2 is violated. Its $BRP = \frac{1}{2}(\frac{20}{80}) + \frac{1}{2}\frac{1}{3} = 0.29$. Assume $MRP = 0.5$ at some point during the relaxation, and the next fail is being taken from the table. According to the formula above, $RD(r) \leq 0.33$. Thus, the $[10, 60]$ is tightened to $[80 - 0.33 * 80, 60] = [53, 60]$, which is used as the relaxed c_2.

For the custom *RP*() function we cannot apply the same logic for tightening intervals, since the custom *RP*() is effectively a black box. In this case the constraints are relaxed to the $[a', b']$ intervals. However, at each search node we call the custom *RP*() function to check against the *MRP*. If the search node does not pass the check, we fail the node and prune the sub-tree completely.

After the search tree with relaxed constraints reaches a leaf, the corresponding solution is submitted to the Validator in the same way as in the original search. Validator is aware of the relaxation and validates the relaxed candidate accordingly, by taking into account the current *MRP* value. As we discussed in the beginning of Section 4, it performs two checks: one before and one after the validation. At the first one it compares the candidate's *BRP*(r) value (supplied by the Solver) with the current *MRP* and discards the candidate if $BRP(r) > MRP$. Otherwise, it performs the validation as usual with *all* constraints relaxed maximally with respect to *MRP*. Relaxing all constraints here is required for correctness, since even if some constraints fail during validation, the solution's penalty might still be below the *MRP*, and it will qualify for the result. That is why the second check is required after the validation.

It is the responsibility of the Validator to update the *MRP* value, since it produces the final result. If a new result decreases *MRP*, the Validator broadcasts the change to all instances in the cluster, so *MRP* is (asynchronously) updated for all Solvers/Validators



Domains	C1 Range	C2 Range	RP
x', lx'	[130, 180]	[10, 60]	0.29
x'', lx''	[100, 110]	[20, 60]	0.53
…	…	…	…

Figure 2: Example of fail recording and replaying for the running MIMIC query.

participating in the query. No special changes to the distributed query processing is required beyond that.

The correctness of the the model's relaxation guarantee is due to the correctness of the CP search itself. While the search process itself uses heuristics to guide the process, the pruning is performed in the provable way, without eliminating any valid results. Constraints modification during the search process is guided by the *MRP*, which is maintained based on the currently discovered results, so subsequent relaxation results will not be eliminated. As far as the complexity of the approach goes, it is equivalent to the complexity of the underlying CP search problem, since the relaxation is performed as in terms of CP.

4.2 Query Relaxation Optimizations

We now discuss a number of useful optimizations for query relaxation. These do not modify the main algorithm, but offer performance gains in common situations.

Computing functions at fails. When we catch a fail, we save the $[a', b']$ intervals for functions $f_c(), c \in C^r$. However, if the search fails at a search node, it does not necessarily mean all constraints have been verified yet. The fail might happen at the first violated constraint, in which case the subsequent constraints are not touched at all. This implies some $f_c()$ values might be unknown. For example, in our running MIMIC example, if c_1 fails, c_2, c_3 are not verified, and min/max ABP values are not computed.

In such cases we can force the computation via the Searchlight API to obtain $[a', b']$ ranges for the remaining constraints. However, $f_c()$ might be relatively expensive, and the cumulative overhead of the fail recording might become quite large. We perform the computation in a lazy way instead, when the fail is replayed. There are compelling reasons behind the lazy evaluation. First of all, if the query does not require any relaxation (i.e., it discovers at least k results) or the fail does not pass the *MRP* check later, the fail and its intervals will not be needed at all. Thus, the total completion time might improve. Second, delays for interactive results might decrease, as we do not pay the full price of computing $f_c()$ immediately, but later, when really needed.

This does not require any significant changes to the engine. Lazy evaluated functions' values are recorded as unknown, and the constraints are considered as non-violated. During the replay, Searchlight automatically estimates unknown values and checks the constraints.

Partial relaxation at replays. When we replay fails, we relax the violated constraints according to the saved $[a', b']$ intervals (tightened with respect to the *MRP* value). However, even the tightened intervals might be quite wide, resulting in poor pruning. This is especially true for fails happening closer to the root of the tree. To avoid over-relaxation, we do not relax the violated constraints all the way, but rather use a percentage of the relaxation interval — a parameter called Replay Relaxation Distance ($0 \leq RRD \leq 1$). If, for example, a constraint $f_c() \leq 10$ needs to be relaxed to $f_c() \leq 20$, and $RRD = 0.3$, we relax the constraint to $f_c() \leq 10 + (20 - 10) \times 0.3 = 13$. The parameter exposes a trade-off. On the one hand, the relaxation becomes more conservative, avoiding potential performance drops. Too much tightening, however, might result in an increased number of fails and the cost of maintenance. On the other hand, loosing the relaxation decreases the number of replays, but might result in increased cost of the search itself. Our experiments showed that the cost of the search usually considerably outweighs the cost of the maintenance, and decreasing the value of *RRD* might considerably speed-up some queries without slowing down the others. This parameter does not change the result and is purely performance-related.

Saving function states at fails. When recording a fail, we store enough information to replay the fail later. However, some $f_c()$ functions might have additional information computed. For example, *max*() might store support coordinates for its $[a', b']$ range, which might allow us to avoid recomputing the function at other nodes of the tree. We extended the Searchlight API with the ability to serialize such information and save it when recording a fail. During the replay this information is restored. The optimization provides significant performance benefits in the presence of a large number of fails with expensive functions.

Sorting the Validator queue on ***BRP*****.** When candidate solutions are received by a Validator, they are put into its FIFO queue, which does not take candidate *BRP* values into account. However, candidates with better *BRP* values might have a better chance of belonging to the final result. They also might help to decrease the *MRP* faster, which improves pruning at Solvers. Thus, we decided to use a priority queue ranked by *BRP* at the validators instead. While a priority queue is generally more expensive than FIFO, in practice performance benefits resulting from better pruning outweigh the queue maintenance costs, as supported by our experiments.

Speculative Relaxation. Before relaxing a query, we executed the *original* query until completion. Only then the recorded fails are replayed if needed. If the user does not mind intermediate results, relaxation can start when first fails are encountered. We call this *speculative relaxation* and provide it as an option.

Speculative relaxation is done by additional CP solvers replaying fails concurrently with the main execution. This is done only when the Validators are idle, so that relaxed candidates do not interfere with the main search. Speculative Solvers still consume

CPU resources, which might slow down the main Solvers and increase query result latency for the user. At the same time, they might provide *relaxed* results much faster. The best use-case for speculation is probably when the user has little insight about the data, and expects an original query to fail.

4.3 Query Constraining

Query constraining deals with the problem of many results. That means it does not need to examine fails, since it is only interested in the results satisfying the *original* constraints, which failed sub-trees cannot contain. After the query begins execution, we consider only the query relaxation and tracks the fails. If the query produces at least k results, it turns off the relaxation, stops tracking fails and starts constraining the query to prune inferior results. The pruning is based on ranking supplied by the built-in or custom function, as described in Section 3.2. Effective pruning allows us to avoid discovering and ranking each result of the original query by eliminating entire parts of the search tree that cannot contain better results than already discovered.

Recall that we explore three possible points of pruning. The first one is Solver-based, at the search tree. When at least k results are found, Searchlight computes the Minimum result RanK (*MRK*) — the minimum rank $RK()$ among all the k results found so far. This is similar to the query relaxation's *MRP*. For a new result to belong to the top-k results its rank *RK* must exceed *MRK*. Similar to query relaxation, for each search sub-tree we compute *BRK* — the Best possible RanK, which is the maximum *RK* among all solutions that might be found in the sub-tree. This can be easily done by using Searchlight-provided synopsis estimations for constraint functions. If the *BRK* value for the sub-tree falls below *MRK*, it is pruned, since no results from that sub-tree can enter the top-k. The check is done by introducing a new *dynamic* constraint into the search: $BRK(r) \geq MRK$. The constraint is dynamic, since *MRK* is updated during the search, with new information coming from both local and remote Searchlight instances. It results in progressively better values for *BRK*, and, thus, better results. At the same time the pruning becomes progressively tighter, resulting in better performance. The *MRK* updates are performed by Validators, as in the case of *MRP*, since Validators produce final results. The cost of checking the dynamic constraint is negligent. We ensure it is done after the original constraints have been checked at the node, so all $f_c()$ functions have been already computed. Computing *BRK* itself just involves a small number of arithmetic operations.

Let us revisit the example from Section 3.2 with the same C^c and parameters. Let us assume at some search node $c_1 \in [100, 190], c_2, c_3 \in [100, 200]$. Then, for the sub-tree $BRK = 1 - \frac{1}{3}(\frac{10}{50} + 0) = 0.933$. If, for example, $MRK = 0.8$, the search will continue for the sub-tree. However, if at some node in the sub-tree $c_1 \in [100, 180], c_2, c_3 \in [100, 150]$, the *BRK* becomes $\frac{1}{3}(\frac{20}{50} + 2\frac{50}{120}) = 0.589 < MRK$. Thus, the sub-tree is pruned.

When a leaf of the search tree is reached, the corresponding candidate is sent to the Validator, which does checks similar to the query relaxation, but without relaxing any constraints. It takes into account the $BRK(r)$ value of the candidate and updates the global MRk (if r enters the top-k).

Skyline computation (see Section 3.2) is done similarly to the scalar query constraining. It is implemented by introducing another dynamic constraint for the result. However, instead of checking the scalar *MRK* value at every node, the constraint compares the estimated $f_c()$ intervals with the current skyline. If the sub-tree is dominated by the skyline, it is pruned. Otherwise, we keep traversing the tree and passes the candidates to the Validator.

The same correctness and complexity argument as for the relaxation applies to the constraining as well, since it is performed as CP search.

5 EXPERIMENTAL RESULTS

We performed an extensive experimental evaluation of the proposed techniques. The main part of the evaluation consisted of measuring the benefits of using our approach against the only alternative available to the user — manual relaxation/constraining. Additionally, we wanted to make sure these features do not bring any significant overhead to the existing query processing. Another important part of the evaluation was to measure the benefits of our optimizations from Section 4.2. Note that existing query refinement solutions are designed for traditional database systems running SQL over relational data, and are not readily applicable to constraint-based search queries over multidimensional data, which we study here. Finally, a head-to-head comparison of Searchlight with a pure DBMS approach is available elsewhere [10] for both synthetic and a real-world data sets. The same paper [10] also argues about the prohibitive complexity of formulating CP queries in relational terms.

We measure the benefit of our approach in terms of query latency. While the quality of the final results might be another measure, such a measure would be governed by the user via the ranking/penalty functions and, thus, can be considered as immutable for our purposes.

Performing a user study would be important to measure the usability of our approach. Such a study would allow us to measure user satisfaction with the quality of the refined results and to more accurately account for time savings for the user. This works was primarily directed at the design and implementation part of the framework. It also lacks a GUI component, as well as a realistic workload to perform a meaningful user study. We leave a user study for future work.

All experiments were performed on a four-instance Searchlight Amazon AWS cluster. The cluster consisted of c4.xlarge machines running Debian 8.6 (kernel 3.16) with 7.5GB of memory. We used two data sets. The first one was a synthetic data set from the original Searchlight paper, 100GB total size. The second data set was a part of the MIMIC II [1] waveform data for the Arterial Blood Pressure (ABP) signal. The data set size was 100GB as well. These data set are representative of general Searchlight workloads: the synthetic one introduces areas of varying function amplitudes, while the MIMIC provides real-world distribution. On a side note, the data sets choice plays secondary importance to the queries, since our approach relies on the efficacy of the *general* Searchlight search process and should work for all data sets that can be handled efficiently by Searchlight.

We used a variety of queries to perform the experimental evaluation. However, to provide concise and meaningful presentation, we discuss different aspects of our approach with the help of two characteristic queries for each data set. By default, we assumed the user's cardinality requirement of 10 results. The queries were as follows:

- **S-SEL** (from Synthetic SELective) is an empty-result query for the synthetic data set. Being maximally relaxed it becomes a non-empty, but very selective query.

- **S-LOS** (from Synthetic LOoSe) outputs empty result initially. However, the maximally relaxed version is very loose, outputs a very large number of results, and does not allow the search process to perform much pruning.
- **M-SEL/LOS** (from Mimic SELective/LOoSe). This are the MIMIC versions of the queries above

Semantically, the M-SEL/LOS correspond to the running example from Section 2, They contain exactly the same variables and constraints, but different parameters (domains and thresholds). Thus, we do not repeat the query constraints here. S-SEL/LOS have the same constraints (function amplitude and neighborhoods), but with synthetic attributes from the generated data. So queries basically look for certain "spikes" in the data, where a spike is determined by comparing the resulting intervals with the neighborhood.

The main idea behind choosing selective and loose types of queries is the observation that a selective query allows the user to perform manual relaxation without significant performance penalties. The user just have to relax the constraints maximally, and the query still finishes in a reasonable amount of time. The user then can choose the best 10 results. A loose query, however, being maximally relaxed outputs an avalanche of results, which results in significant latency. Such over-relaxation might be quite costly in practice. Since the user cannot easily predict the selectivity beforehand, the system should be able to handle both types of queries automatically.

We used the maximally relaxed versions of the queries above to measure the performance of the query constraining, since they output more than 10 results. As in the case of relaxation, the selective queries' results can be ranked manually. For the loose query this is infeasible.

5.1 Query Relaxation

We measured the benefits of the automatic relaxation over the manual approach, in which the user would be forced to guess the correct query, possibly in several iterations. This manual relaxation scenario is the only alternative available to the users and exactly the case we want to avoid in practice, hence it was important to compare it with our solution. The manual approach was performed using Searchlight as well, so the search engine remained the same. We will often refer to the automatic approach as just Searchlight. We studied the following scenarios:

- **USER-3**. This is a common user scenario. The original query gives an empty answer. Then the user relaxes it in a cautious way, several times, and gets the required number of results on the second try. Thus, she comes through 3 iterations (hence, the name). In practice the number of iterations might be much larger, due to a large number of relaxation possibilities, but three iterations was enough to demonstrate our point.
- **USER-2**. In this scenario the user guesses the query correctly from the first try, for the total of 2 iterations. Note, this scenario is quite infeasible in practice, and can be seen as an oracle-based approach, in which the user immediately knows the correct relaxation. This approach establishes an important baseline.
- **USER-MAX**. This is the scenario in which the user just relaxes the query maximally after the original query fails. Depending on the query, this might perform like the oracle approach above (e.g., for selective queries) or just start outputting a large stream of results without any means of pruning (for loose ones). In the latter case the user would have to stop the query and guess further, since such queries might easily take hours to finish.

First, we provide query completion times for the selective queries S-SEL and M-SEL under different scenarios described above. The results are illustrated in Table 1, where the "SL" column corresponds to the automatic Searchlight relaxation approach discussed in the paper. For the USER-2 scenario in the parenthesis we specify the completion time of the second, correctly relaxed, query. For these queries the USER-2 and -MAX scenarios are basically equivalent since there is no much penalty in relaxing the query maximally.

Table 1: S/M-SEL query completion times (secs) for query relaxation.

Query	SL	USER-3	USER-2	USER-MAX
S-SEL	97	327	210 (120)	216
M-SEL	150	544	380 (240)	380

As can be seen, even comparing with the USER-2 approach Searchlight provided considerable performance gains. They come from two sources:

- Searchlight does not need to re-explore the already traversed parts of the search tree. After the main search is finished, it can concentrate only on the previously unexplored (i.e., failed) parts. This is in contrast with any of the manual approaches, which have to start every query iteration from scratch.
- When relaxing the query, Searchlight is able to provide additional pruning based on the best results found so far. While for selective queries it is not necessarily the game changer, it has a much more pronounceable effect for loose queries, which we show later.

We also measured the time it took Searchlight to obtain the first result. For the S-SEL queries it took Searchlight 42 seconds in the SL approach versus 91s seconds for the USER-2 (the best) approach. The corresponding times for the M-SEL query were 45 and 198 seconds. We saw similar trends across all queries we ran. The results come as no surprise, since Searchlight is able to start the relaxation right away without restarting queries with new parameters.

When it came to the overhead of the query relaxation approach itself, it did not exceed 5 seconds for the synthetic and 3 seconds for the MIMIC queries. This overhead mainly came from assessing and recording the fails.

Table 2 provides the corresponding results for the loose queries. For the "Max" case, we stopped the query after 1 hour (hence the > symbol in the table), since this was enough to demonstrate our point.

Table 2: S/M-LOS query completion times (secs) for query relaxation.

Query	SL	USER-3	USER-2	USER-MAX
S-LOS	105	314	208 (106)	>3600
M-LOS	91	177	118 (83)	>3600

This experiment shows the same trend as for the selective queries with a single exception: the USER-MAX and USER-2 behaved differently. When the user relaxed the query maximally, it ran for a very long period of time (we stopped it after 1 hour) due to the very large number of results. In practice, the user cannot just stop the query and rank the currently found results, since she is not guaranteed to find the top-k among them. However, Searchlight guarantees correct top-k results. Since the maximal relaxation is out of the question, without Searchlight the user would have to continue the guessing game of scenario USER-3 but with potentially more iterations and longer times.

Searchlight outputs the first result in 92 and 45 seconds for S-LOS and M-LOS, respectively. The corresponding times for the USER-2 were 108 and 77 seconds. This is the same trend as we discussed for the selective queries. The auto relaxation overhead remained at comparably low levels: 15 seconds for S-LOS and 1 second for M-LOS.

The next experiment measured the overhead of the auto relaxation for the queries that do not need it. We wanted to explore the possibility of keeping the relaxation always on, without the user turning the knob. For this experiment we ran the second query from USER-2 with the relaxation turned on. The results are given in Table 3.

Table 3: Query completion times (secs) for queries not needing relaxations.

Relax	S-LOS	M-LOS	S-SEL	M-SEL
Off	106	83	120	240
On	116	98	127	290

As can be seen, turning on the auto-relaxation does not have any significant impact on the query completion times. The notable exception is M-LOS query, for which Searchlight submitted a lot of relaxed candidates to the Validator before 10 results were actually found. However, this overhead was the largest we saw for a large number of queries we ran. Even with such an overhead the auto-method would be quite helpful for the user, since the relaxed candidates are output to her as useful feedback. At the same time, the time to first result did not change significantly for all queries, which means the interactivity was not hampered, and the overhead was limited to the total completion time.

5.2 Query Constraining

In the absence of automatic query constraining, the only option is to run the query until completion and then rank results at the client. While this might work for queries with small number of results, it is very inefficient for queries returning a lot of them. In addition, the manual approach misses significant pruning opportunities. Our main results are shown in Table 4. “Off” means no constraining, which is equivalent to the manual approach; “Rank” means scalar ranking automatic constraining, and “Skyline” — vector domination constraining. Both approaches were described in Section 3.2. By default we specify times in seconds, and we use ’h’ and ’m’ symbols to denote hours and minutes.

Table 4: Query completion times (secs) for query constraining.

Method	S-LOS	M-LOS	S-SEL	M-SEL	M-SEL’
Off	2h 8m	2h 24m	120	240	263
Rank	60	154	29	139	135
Skyline	314	13m	93	269	218

The loose S/M-LOS queries could not even finish in a reasonable time. These queries actually were outputting results with very low latency during the execution. However, since constraints were loose, they created an avalanche of such results without the ability to prune. To guarantee the top-10 results the user would have to stop the query and constrain it by hand, possibly in several iterations.

At the same time, for the same queries Searchlight provided considerable performance gains, coming from pruning both at Solvers and Validators, as we described in Section 4.3. This was especially evident for the rank-based constraining. The skyline constraining was less effective, with respect to the query completion time. However, comparing with the manual “Off” approach, the performance benefits were considerable. The reduced efficacy can be attributed to the nature of skyline — it is harder to prune interval-based search nodes at Solvers and candidates at Validators.

The selective queries S-SEL and M-SEL allowed us to measure the constraining benefits for the queries for which the client-based filtering is a viable alternative due to their reasonable completion time. In most cases our approach resulted in considerable gains. The M-SEL query is somewhat of an exception, for which the skyline approach performed worse than the “Off” approach. This can be attributed mostly to some overhead from the skyline based checks during pruning (without any benefits) and slightly different rebalancing of the candidates between Validators. The last column of the table (M-SEL’) provides results for another selective MIMIC query. It can be seen that both rank and skyline auto approaches provided significant improvements for query completion times, so the M-SEL case should not be considered a trend for selective queries.

When it comes to the overhead of the automatic approach, it is kept at the minimum. Actually, it is smaller than that for the query relaxation since it does not need any maintenance similar to tracking of failed search nodes. As for the Solver- and Validator-level checks, they are quite cheap for the rank-based constraining, being in-memory algebraic comparisons. For skyline-based constraining the checks are somewhat more expensive, and they must be active all the time, from the beginning of the query. However, the checks can be done quite efficiently using the variety of existing methods for skyline computation. This problem is well-researched, for example, for relational skylines. The overhead in this case is basically the cost of computation, which cannot be avoided for such a non-trivial constraint.

5.3 Query Relaxation Optimizations

In this section we describe experiments to measure the performance of the relaxation optimizations from Section 4.2.

Computing functions at fails. In this experiment we measured the difference between the two different strategies to compute functions $f_c()$ when catching fails. The first strategy, “Full”, corresponds to fully evaluating all functions at the failed node. The “Lazy” corresponds to the lazy evaluation. Both strategies were described in Section 4.2.

The results are presented in Table 5 where the parenthesis times specify the times to first result, which is a reasonable measure of interactivity. While times to the first result might seem large, they include the completion time of the *original* query, which found no results at all. The optimization provided benefits for more expensive synthetic queries. At the same time it did not result in any overhead for all queries. We also ran additional experiments for more expensive MIMIC queries, for which we took the same M-SEL/LOS queries and increased their cardinality requirements from 10 to 200 results. We saw the significant benefits at the fail recording stage: for some queries the fail tracking overhead decreased from 30 to 15 seconds.

Table 5: Query completion and first result times (secs) for fail recording methods.

Method	S-LOS	M-LOS	S-SEL	M-SEL
Full	120(100)	81(45)	112(46)	149(45)
Lazy	105(90)	91(45)	97(42)	150(45)

Saving UDF states at fail recording. In this experiment we measured the impact of saving additional UDF information when recording fails. In contrast with the previous optimization, which just lazily postpones computation of some UDFs, this optimization allows us to avoid re-computation of some UDFs completely. The results are presented in Table 6 for query completion and first-result times (the latter is given in parenthesis).

Table 6: Query completion and first-result times (secs) for the UDF saving optimization.

UDF saving	S-LOS	M-LOS	S-SEL	M-SEL
On	105(90)	91(45)	97(42)	150(45)
Off	113(111)	104(70)	97(40)	154(46)

The optimization was especially beneficial for the loose queries. For the selective queries the benefits were not pronounced due to the structure of those queries. The replays were relatively cheap, involving less re-computation. As in the previous case, this optimization did not result in any overhead as well, at the same time allowing better performance in many cases. The memory footprint for the saved states depends on the functions. Standard aggregate functions use about 80 bytes per save for the two-dimensional data set (16 bytes for the range itself plus 64 bytes for the support coordinates for the min and max values).

Speculative execution. As can be seen from the experiments, the time to the first result is often quite large. This is a logical result for empty-result queries, since Searchlight first finishes the main, non-relaxed, query and only then tries to relax it. We support speculative relaxation as a means to start the relaxation sooner. The corresponding query completion and first result times are presented in Table 7.

Table 7: Query completion and first result times (secs) for speculative relaxation.

Speculation	S-LOS	M-LOS	S-SEL	M-SEL
On	128(7)	90(45)	115 (2)	152(47)
Off	105(90)	91(45)	97(42)	150(45)

As can be seen from the results in many cases speculative execution significantly improved times to the first result. We could not find suitable queries to demonstrate the same trend for the MIMIC queries. While the speculative Solver for those queries replayed some of the fails, they resulted in a small number of non-perspective candidates. As we discussed in Section 4.2, speculative Solvers are restricted to fails found by main Solvers so far.

As expected, the speculative relaxation has its own overhead coming from the consumption of CPU resources by the speculative Solver. For some queries the increase in the completion time was significant. We decided to run an additional experiment (not shown here), with one additional CPU thread available. As expected, the times for the speculative relaxation turned on and off were the same, which suggests the overhead is CPU related and cannot be trivially extinguished. Basically, the decision of trading off some completion time to faster interactive results is up to the user.

Partial relaxation during replays. This optimization addresses the issue of over-relaxing the query at a fail replay, when early fails might be relaxed in a very loose way because of loose estimations. As we discussed in Section 4.2, the *RRD* parameter ($0 \le RRD \le 1$), allows us to perform the relaxation in a more controlled way by enforcing tighter relaxation. The results of changing this parameter for the loose queries are presented in Table 8. We did not see any significant effect of the parameter to first result times.

Table 8: Query completion times (secs) for different *RRD* values.

Query *RRD*	0.1	0.3	0.5	0.7	1.0
S-LOS	106	105	106	106	106
M-LOS	87	91	112	145	54m

As can be seen from the table, the optimization resulted in gains only for some queries (we saw gains for other MIMIC queries as well). M-LOS query, for instance, fails almost immediately, and replaying its early fails with maximal relaxation effectively results in traversing most of the search tree. For S-LOS, on the other hand, search fails are relatively deep in the search tree, so maximal relaxation does not cause significant increase in the number of visited search nodes. In general, the benefits of the optimization depend on the nature of the search tree, which in turn depends on the query. At the same time, it does not introduce any overhead, which allows us to keep it always on. We saw a slightly elevated number of fail recordings and replays, but not significant enough to cause any drop in performance.

Additional experiments with the fail and candidate queues. We extended the Validator to sort the candidates on the *BRP* value. This in general might allow the Validator to identify better relaxed results faster, which in turn results in better *MRP* values and more effective pruning. We performed the experiment over a number of queries with different cardinality requirements to vary the number of candidates at runtime. For some queries, we saw 8-12% improvement in total completion times and no major impact on the first results.

We also measured the benefits of our fail-based approach presented in the paper against simply continuing the search "through" the fail. The latter would still involve relaxing constraints, but no fail recording (and later replaying) would be made — the search would be immediately resumed from the point of failing. One reason to do that would be to simplify the approach and decrease the memory overhead. However, we claim it would result in a sub-optimal approach, where the *utility* of the fails is not taken into consideration. Our experiments supported this claim. When we replayed the fails in the order they were encountered, simulating the immediate search resume, we did not see any improvements in the completion or first result times. Moreover, for some queries the completion times increased up to several orders of magnitude. For example, for S-LOS the time increased from 105 seconds to 56 minutes. We believe these results emphasize the necessity of a utility-based approach.

6 RELATED WORK

Query relaxation [12, 14–16] deals with the empty-answer and too-few-answers query problems by relaxing the original query constraints. The past work on query refinement can be studied under two broad categories. The first includes relaxation based on some statistics readily available in the database. For example, the Stretch-and-Shrink (SnS) framework [14] uses query cardinality estimations via precomputed samples and then uses the estimations to find relaxed ranges for each range-based query constraint independently. The framework heavily relies on fast cardinality estimations. Multiple estimations might have to be made at every step of the interactive refinement. Another framework [4] uses histograms to produce cardinality estimations and derive proper constraint ranges for the query. The results presented to the user are ranked based on the distance (e.g., Euclidean) from the original constraint ranges. There is also the possibility of using probabilistic [15] and machine learning [16] frameworks to produce relaxations. These methods, however, still rely on statistics to provide probabilistic estimations for the relaxation decision or the learning stage to understand the rules hidden inside the data. On the other hand, Searchlight targets queries for which the cardinality is not known beforehand. It would be also hard to estimate properly due to a large search space and possible complexity of query constraints. The results are also not known, and might be expensive to find, which makes the learning stage or probabilistic estimations infeasible.

The other category includes methods that use indexes to relax constraints. One such approach [12] is to relax join and selection predicates, and obtain the relaxation *skyline*. These methods have limited applicability for Searchlight, since results cannot be indexed beforehand. Also Searchlight generally works with regions (subsets) instead of single tuples, and the search space itself depends on the query constraints. Additionally, query constraints might be more complex than ranges, potentially referencing data outside of regions (e.g., the neighborhood constraints in the query from the Introduction). This makes R-trees (or other traditional index trees) generally ineffective for traversal and pruning.

The too-many-results problem creates the dual problem of *contracting* the query. These methods [4, 14] generally use precomputed statistics to make fast cardinality estimations and find suitable ranges for query constraints. The corresponding frameworks usually handle both relaxation and contraction at the same time. Another approach is to get rid of excessive answers by ranking and outputting only the best few. The "best" can be based on a scalar ranking function (top-k queries) or vector domination (skyline [5] queries). These methods commonly rely on traditional precomputed structures, such as views [6, 8] and R-trees [17, 18]. The view-based approaches require advance knowledge of at least a part of the workload to materialize proper views. The R-tree approaches traverse the tree and perform MBR-based pruning. If such structures are not readily available, the only option is to perform a sequential scan and either build the required structures or do the processing during the scan (e.g., sorting [3, 7], batch computation [3] or building structures optimized for particular queries [17]). For Searchlight, no indexes are available beforehand, thus we performed a comprehensive search. This might seem similar to the sequential scan-based approaches, however, as we argued, the nature of the constraint-based queries is different and, thus, requires novel approaches.

7 CONCLUSION

Fast query execution is necessary but not sufficient for effective interactive data exploration. Users often go through multiple query iterations to identify a reasonable set of results that matches their goals. It is thus critical for the underlying system to aid the users, optimizing for human labor, time and attention, to maximize user productivity.

We introduce dynamic and automatic refinement of constraint-based search queries, based on user-specified target result cardinalities. When relaxing a query, we guarantee optimal results according to user-specified distance functions. When constraining, we output the top results according to a ranking function. Unlike previous solutions, our approach does not require any pre-computed indexes nor does it require result cardinality estimations, which might be extremely hard to obtain accurately for queries with complex constraints. Our approach instead alters the constraints during the run-time. Our techniques naturally fit in and can effectively leverage the common features of CP platforms and DBMSs. Furthermore, our approach can explore more promising parts of the search space first, which considerably improves pruning and, consequently, provides better interactivity and query completion times.

Our approach provides significant performance benefits in comparison with the tedious and inefficient manual approach. At the same time, it incurs negligible performance overhead, even for queries that end up not needing any refinements.

REFERENCES

[1] [n. d.]. MIMIC II Dataset. https://mimic.physionet.org/. ([n. d.]).
[2] [n. d.]. SciDB. https://www.paradigm4.com/. ([n. d.]).
[3] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *ICDE*. 421–430.
[4] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2002. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *ACM Trans. Database Syst.* 27, 2 (June 2002), 153–187.
[5] Michael J. Carey and Donald Kossmann. 1997. On saying "Enough already!" in SQL. *SIGMOD Rec.* 26, 2 (1997), 219–230.
[6] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. 2006. Answering Top-k Queries Using Views. In *VLDB*. 451–462.
[7] Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal Aggregation Algorithms for Middleware. In *PODS*. 102–113.
[8] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. 2001. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD*. 259–270.
[9] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2014. Interactive Data Exploration Using Semantic Windows. In *SIGMOD*. 505–516.
[10] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2015. Searchlight: Enabling Integrated Search and Exploration over Large Multidimensional Data. In *VLDB*. 1094–1105.
[11] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2016. Interactive Search and Exploration of Waveform Data with Searchlight. In *SIGMOD*. 2105–2108.
[12] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. 2006. Relaxing Join and Selection Queries. In *VLDB*. 199–210.
[13] Gang Luo. 2006. Efficient Detection of Empty-result Queries. In *VLDB*. 1015–1025.
[14] Chaitanya Mishra and Nick Koudas. 2009. Interactive Query Refinement. In *EDBT*. 862–873.
[15] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. 2013. A Probabilistic Optimization Framework for the Empty-answer Problem. *VLDB* 6, 14 (2013), 1762–1773.
[16] Ion Muslea. 2004. Machine Learning for Online Query Relaxation. In *SIGKDD*. 246–255.
[17] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive Skyline Computation in Database Systems. *ACM Trans. Database Syst.* 30, 1 (March 2005), 41–82.
[18] Man Lung Yiu and Nikos Mamoulis. 2009. Multi-dimensional Top-k Dominating Queries. *The VLDB Journal* 18, 3 (2009), 695–718.