Abstract of "Integrated Search and Exploration Over Large Multidimensional Data" by Alexander Kalinin, Ph.D., Brown University, May 2017.

The need for rich, ad-hoc data analysis is key for pervasive discovery. However, generic and reusable systems tools for interactive search, exploration and mining over large data sets are lacking. Exploring large data sets interactively requires advanced data-driven search techniques that go well beyond the conventional database querying capabilities, whereas state-of-the-art search technologies are not designed and optimized to work for large out-of-core data sets. These requirements force users to roll their own custom solutions, typically by gluing together existing libraries, databases and custom scripts, only to end up with a solution that is difficult to develop, scale, optimize, maintain and reuse.

To address these limitations, we propose a tight integration of data management and search technologies. This combination would not only allow users to perform search efficiently, but also offer a single, expressive framework that can support a wide variety of data-intensive search and exploration tasks. As the first step in this direction, we describe a custom search framework called *Semantic Windows*, which allows users to conveniently perform structured search via shape and content constraints over a large multidimensional data space. As the second step, we describe a general-purpose exploration framework called *Searchlight*, which allows Constraint Programming (CP) machinery to run efficiently inside a Database Management System (DBMS) without the need to extract, transform and move the data. This marriage concurrently offers the rich expressiveness and efficiency of constraint-based search and optimization provided by modern CP solvers, and the ability of DBMSs to store and query data at scale, resulting in an enriched functionality that can effectively support data- and search-intensive applications. As such, Searchlight is the first system to support generic search, exploration and mining over large multidimensional data collections, going beyond point algorithms designed for point search and mining tasks.

Fast, interactive query evaluation 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. In the third part of this work, we study the modification of initial query parameters at run-time: we describe how Searchlight can dynamically relax or constrain the parameters of a query, based on its progress, to offer more or fewer results to the user. This feature allows users to iterate over the data sets faster and without having to make accurate guesses on what parameters to use.

Integrated Search and Exploration Over Large Multidimensional Data

by
Alexander Kalinin
Sc. M., Lomonosov Moscow State University, 2007
Sc. M., Brown University, 2012

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2017

This dissertation by Alexander Kalinin is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Ŭgur Çetintemel, Ph.D., Director

Recommended to the Graduate Council

Date _____ _____

Stan Zdonik, Ph.D., Reader
Brown University

Date _____ _____

Pascal Van Hentenryck, Ph.D., Reader
University of Michigan

Date _____ _____

Rodrigo Fonseca, Ph.D., Reader
Brown University

Approved by the Graduate Council

Date _____ _____

Andrew G. Campbell
Dean of the Graduate School

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Efficient execution of constrained search queries remains an important problem in the presence of big data. There are constrained search solutions where the data first has to be extracted and put into memory. There are separate solutions for efficiently managing data, which at the same time lack search functionality. There are no existing solutions combining the two technologies in a single system. This research strives to remedy this inconsistency by introducing a new type of search framework, where search and data management facilities work closely with each other, providing users with a general search solution suitable for vast amounts of data.

At the same time, even when search queries can be executed efficiently, there still remains the issue of improving user's experience during the search. Users, at least initially, might have limited knowledge about the data set itself and the results their queries might return. That makes the query specification tedious. Users basically have to "guess" the correct queries in the attempt to receive a suitable number of meaningful results. The proposed system addresses this issue via the notion of query relaxation and constraining, which works at the query engine level and modifies the query constraints depending on its progress.

1.1 Motivation

The need for rich, ad-hoc data analysis is key for pervasive discovery. However, generic and reusable systems tools for interactive search, exploration and mining over large data sets are lacking. Exploring large data sets interactively requires advanced data-driven search techniques that go well beyond the conventional database querying capabilities, whereas state-of-the-art search technologies are not designed and optimized to work for large out-of-core data sets. These requirements force users to roll their own custom solutions, typically by gluing together existing libraries, databases and custom scripts, only to end up with a solution that is difficult to develop, scale, optimize, maintain and reuse.

In data exploration, users often have some idea about *what* they want to find, but have no idea *where* to look. This is fundamentally a *search* problem in which the user interacts with the data

in an ad-hoc fashion to identify objects, events, regions, patterns of interest for further, more detailed analysis. Let us consider simplified, but representative examples from the astronomy domain: consider an astronomer working with SDSS [4], a popular data set containing information about different celestial objects.

An object of interest might be a hyper-rectangular region in the sky (i.e., a rectangle with coordinates corresponding to right ascension and declination), and the properties might include average/min/max values of magnitudes of stars within a region, the area of the region, the lengths of its sides, etc. The user might be interested in the following types of search queries:

- *First-order queries* look for a single region satisfying the search properties. For example, Q1: "find all $[2, 5]^\circ$ by $[3, 10]^\circ$ regions with the average r-magnitude of stars within it less than 12". Note that the region dimensions are *not* fixed and can take any value within the given range. The search can include more advanced properties; e.g., "additionally, the difference between the average magnitudes of the region and its 3°-neighborhood must be greater than 5".

- *High-order queries* search for sets of regions. For example, Q2:"find a pair of celestial regions with the difference between the magnitudes not exceeding 1, located in different sectors of the sky".

- *Optimization queries* assign an objective function to all regions, and search for regions maximizing/minimizing the function. For example, Q3: "find all 2° by 3° regions that contain sky objects with the minimal r-magnitude".

Another example is performing search for time-series data. For example, MIMIC [2] contains waveform data (e.g., arterial blood pressure, ECG, etc.) for a number of patients admitted to ICU. An object of interest here might be a time interval (e.g., a temporal region) with specific properties. The user might search for time intervals that exhibited abnormal values or are similar to the specified interval (first-order queries); she might also search for sets of similar intervals to detect patterns in the patient's medical history (high-order queries).

In data exploration applications, the user should start getting results as quickly as possible. For optimization queries the system might report the running maximum/minimum. The user might not even be interested in the final result at all. She might quickly switch between queries, interrupt their execution, refine and restart them. This is sometimes called "human-in-the-loop" exploration. Additionally, the user should not be constrained to a particular type of object. She might want to search for interesting spheres instead of rectangles or add additional types of properties, e.g., a new complex aggregate or distance measure.

Perhaps surprisingly, traditional DBMSs offer very limited support for search queries, even for the most basic first-order ones. We will discuss this in more detail in context of relational and array-based DBMSs in Chapters 3 and 4. Traditional SQL constructs such as `OVER` and `GROUP BY` are not expressive enough. Even a seemingly simple query such as Q1 is thus very cumbersome to specify and difficult to automatically optimize. Fundamentally, search requires the ability to

enumerate sets of objects and identify the qualifying ones. Most DBMSs do not provide such *power-set* enumeration operations. Even if the enumeration can be done, the number of (sub)sets can be so large that sophisticated pruning and search techniques have to be employed to get results in a timely fashion.

Take SciDB [3, 10], a state-of-the-art array DBMS. Revisiting the astronomical example above, the objects of interest, regions, can be seen as equal to sub-arrays. SciDB has a powerful `window()` operator that can be used to compute aggregates for *every* possible sub-array of the specified size. This might allow users to find interesting sub-arrays by enumerating all of them and then performing filtering based on the query constraints. However, the operator processes sub-arrays exhaustively, in a specific order, without any concerns for interactivity. Moreover, since it has to compute every possible sub-array, the query becomes impractical for large search spaces — for a two-dimensional array of size 10,000x10,000, finding even a *fixed* region of size 10x10 would result in exploring approximately 10^8 regions and computing one or more aggregates for each of them. Looking for flexible-size regions significantly exacerbates the problem due to increased search space size. Moreover, for more complex search problems users might have to write several queries and perform some form of a "join" or concatenation of the intermediate results.

We argue about the necessity of introducing integrated search functionality within a DBMS. Such functionality would not only allow users to perform search efficiently, but also offer a single, expressive framework that can support a wide variety of data-intensive search and exploration tasks.

1.2 Integrated Search and Exploration Over Large Multidimensional Data

We will explore the problem of performing search queries over large multidimensional datasets. Such data can often be represented as an array with multiple *dimensions* (e.g., location coordinates, time stamps, etc.), where each array cell contains a number of *attributes* (e.g., astronomical objects magnitudes, signal amplitude measurements, etc.). A typical search query would look for sub-arrays (regions) with interesting properties. The properties are typically expressed by the user via a number of *constraints*. While we do not impose any specific restrictions on the nature of such constraints in general, in this research we concentrate our effort on *aggregate constraints*, containing aggregate functions. An aggregate function takes a sub-array as the input and produces a single scalar value. For example, $avg(R, magnitude) < 10$ is an algebraic constraint that might allow the user to search for a celestial region with average magnitude less than 10. While seemingly simple, such constraints have considerable expression power. For example, similarity between two regions can often be expressed via a distance between them (e.g., Euclidean), which can be treated as an aggregate function.

As the first step in our effort of integrating search with DBMSs query processing, we discuss a specialized framework called *Semantic Windows* (SW) [27]. In this framework users can search for multidimensional rectangular regions (windows) satisfying constraints on shape (e.g., the size

of the window) and content (e.g., the average magnitude of stars within the window). Thus, it partially covers the "First-order queries" class discussed above. SW treats an exploration query as a data-driven search task, with the search space consisting of all possible windows. Since windows can overlap and can be located anywhere in the data, this makes the search space quite large. Thus, to deliver results to the user quickly, this search space is explored in the order of *utilities*, which measure each window in terms of cost and benefit. The cost shows how expensive it is to read the window data from disk, and the benefit measures how close the window is to satisfying the constraints. Exploring more promising windows first allows the framework to find and start presenting results quickly to the user.

We implemented SW as a distributed layer on top of a relational DBMS PostgreSQL, which communicates with the DBMS via TCP/IP. The SW layer performs the search itself and delegates all computations requiring data to PostgreSQL by using SQL queries over a network connection. We conducted an extensive experimental evaluation on synthetic and real data to study the benefits and drawbacks of the approach. The experimental results showed that in many cases our framework offered online results quickly and continuously, outperforming the traditional DBMS approach significantly. For the comparison we used a complex SQL query, which is the only way to specify the studied search queries in a relational DBMS. Due to the complexity of the query, the query processor in PostgreSQL was not able to plan and execute it efficiently, so not only intermediate (online) result times, but also query completion times for SW were often better. In Chapter 3 we argue that this is an inherent limitation of query processing in SQL-based systems, which renders them incapable of efficient execution of search queries.

While SW showed the applicability of the proposed approach of closely integrating search processing within DBMS, at the same time it has a number of certain limitations:

- The framework is specialized for single-region search with specific aggregate functions. Users cannot easily express more complex search queries. For example, it might be logical to look at the difference between a region and its neighborhood to detect anomalies. The neighborhood can be expressed as a region as well. Ideally, users should be able to define such *search objects* easily with the appropriate constraints linking them together with other constraints. However, due to a specialized search algorithm, such extensibility is not trivial for SW.

- While SW can find results quickly, eventually it has to read the whole data set. This is due to the fact that SW uses sampling to perform cost-benefit analysis. Since sampling cannot give 100% guarantees that a given window violates the constraints, it is impossible to *prune* unneeded windows, which poses a significant problem for large data sets. This highlighted for us the need in sophisticated pruning techniques to exclude parts of the data that cannot contain results from exploration.

- SW exposed the necessity of carefully managing not only the search process itself, but also the data access part of it. Each time a query constraint is checked, an aggregate for the window may need to be computed, which results in disk access from the DBMS. Due to the

size of the search space, the number of such disk requests might also be quite significant. Both the search and data access should be balanced across all nodes participating in the query. While SW performs static search space distribution before the query begins, there is need in dynamic distribution as well, since some nodes might complete their job much faster. The same argument can be applied to the data access, with the need of distributing data requests across multiple nodes and possibly redistributing some data during the query execution.

To address these issues, we devised another, more general, search framework, called *Searchlight* [28], which we discuss in Chapter 4. The gist of the new approach is employing existing Constraint Programming (CP) [45] techniques to perform the search part of the query. First of all, not only CP allows users to naturally and easily express search problems from all three query classes discussed above, but it also has no significant limitations on extensibility, i.e., defining more sophisticated search objects and constraints. Users express their search queries by using a number of *decision variables* to define search objects and constraints referencing the variables to describe the desired search criteria. Variables and constraints compose the *model* of the problem. Secondly, CP solvers are very proficient at exploring large search spaces using such models. Solvers incorporate sophisticated search heuristics and pruning techniques, which helps quickly find results and efficiently remove parts of the search space that do not contain any. Search heuristics are highly customizable, which allows users to make exploration for new kinds of search problems more efficient.

These features render CP solvers a very powerful tool for interactive, human-in-the-loop data exploration and mining. Unfortunately, solvers commonly assume that all the required data fits into main memory and thus only optimize for the compute bottleneck. While this assumption has served well for many traditional CP problems, in which the primary challenge is to deal with very large search spaces, it has recently become obsolete with the availability of very large data sets in many disciplines.

Searchlight uses a two-stage *Solve-Validate* approach. At the solving stage, CP Solvers perform speculative search on main-memory synopsis structures, instead of the real data. A synopsis is a condensed representation of the data, containing information needed to perform pruning and to verify query constraints. The results are guaranteed to contain all the real results, but possibly include *false positives*. At validation, Validators efficiently check the candidates over the original data, eliminating the false positives and producing the final solutions while optimizing I/O. This two-stage approach is transparent to the CP solver internals and the users.

Searchlight can also transparently parallelize query execution across multiple nodes in a modern cluster of multi-core machines. Searchlight supports both static and dynamic balancing. During the static phase, the search space and the data space are distributed between Solvers and Validators before the query begins. During execution, Searchlight redistributes work between idle and busy Solvers to address hot spots. In times of high I/O, load it redistributes data between Validators at different nodes and starts more Validators at the same node, if the cluster has the resources available.

Our experimental results, both synthetic and SDSS-based, quantify the remarkable potential

of Searchlight for data- and search-intensive queries, for which Searchlight often performs orders of magnitude faster than the next best solution (DBMS-only or CP-solver-only) in terms of end response time and time to first result.

1.3 Enhancing User Experience: Query Relaxation and Constraining

Efficient search query execution is just the first step towards improving data exploration support in traditional systems. There still remains the issue of correctly specifying the query and its constraints. Since the user, at least initially, might have very limited knowledge of the data set, properly formulating constraints might be a complex task. Even assuming the user knows the types of constraints she wants to put into the query, getting the constraint parameters right is not trivial. For example, consider query $Q1$ from Section 1.1. The user has to specify the threshold for r-magnitude for the resulting region. Depending on the specified threshold, the query might result in the following outcomes:

- The query outputs the desired number of results, which is easy to explore. This is the desired outcome.
- The query does not output any results, since the query is over-constrained. This is the so-called "empty-answer problem". The query might also output too few results, e.g., 1-2, while the user desires at least 10. In this case the user would have to re-formulate the query, hoping for a better outcome. At this point this becomes a cumbersome guessing game in which the user tries to guess the correct constraint parameters to receive the desired number of results.
- The query returns too many results. For example, 1000 instead of 10. Such a number of results might be hard to represent in a meaningful way, and for the user to analyze. Since Searchlight outputs results interactively, minimizing delays, it is possible for the user to interrupt the query after she has received too many answers and try a new one. However, interrupting the query does not necessarily make the result any more meaningful, since the results found so far might not carefully represent the whole result set. At the same time, reformulating the query becomes the same guessing game we saw for the empty-answer problem.

The second and third outcomes have been well-studied in the research literature, especially in the context of relational queries. A common solution for the empty-answer problem is to *relax* the query [34, 36, 38, 37, 31]. This allows the system to provide "close enough" results to the user. The process can be either automatic, based on a suitable distance function, or interactive, where the user is guided through a number of possible options. As for the problem of too many results, the possible solutions include *contracting* the query, modifying its constraints to reduce the number of results [36], or ranking results with some user-provided function and outputting the top-k results [16, 42, 26, 11, 17, 52]. Unfortunately, these approaches have limited applicability for the

types of queries Searchlight addresses. In general, they assume either available statistics for efficient result cardinality estimations or indexes over all possible query objects (e.g., tuples) for efficient traversing and pruning. We discuss the related work in more detail in Chapter 2.

Since Searchlight mainly addresses ad-hoc exploratory queries with user-defined regions and constraints, it has to deal with the aforementioned problems dynamically, during the query evaluation. The main reason for this is that the objects of interest belonging to the result have to be discovered first and only then relaxation or contraction can take place. Searchlight detects the too-few/too-many results problem during the query execution and applies either *relaxation* to bring in more results or *constraining* to filter out results dominated with respect to the user-specified ranking function. At the same time, Searchlight keeps the overhead of this approach at minimum, so not to hamper the interactivity of the queries that do not need these techniques.

If during the search Searchlight cannot find the required number of results (specified by the user), it relaxes the original query constraints and revisits parts of the search space previously pruned by the CP solver. It stops when the user result cardinality requirement is satisfied, at the same time guaranteeing that the relaxed results have the minimum *distance* from the original constraints. By default, Searchlight uses a built-in function to measure this distance, which is suitable for interval-based algebraic constraints. For example, for constraint $avg(R, magnitude) < 10$, a result having the average value 12 has the distance 2 from the original query. For multiple constraints, the maximum or average distance can be taken. Alternatively, the user can specify her own distance function.

Alternatively, if Searchlight detects too many results (again, as per the user-specified cardinality requirement), it introduces new constraints into the original query to prune results dominated by already found ones and admit only better (or non-dominated) results to the current result set. The "better" is determined by the ranking function, which can be specified by the user. The user can also use the Searchlight's built-in ranking function by specifying her preferences. For example, if a query has two constraints: for the average brightness of a region and the distance from another region, the user might prefer *brighter* regions, but with *smaller* distance. Another option for the user is to use the common *skyline* computation. However, in this case Searchlight cannot guarantee the cardinality of the final result.

Our experimental results showed that the dynamic relaxation-constraining approach does not bring any significant overhead to the queries that do not need modification. Relaxation, when required, performs extremely better than the query guessing approach discussed above (i.e., a series of queries), and quite comparable with a hypothetical"oracle-based" approach, in which the user can somehow guess the relaxed query correctly on the first try (after the completion of the original query, of course). Constraining in many cases perform orders of magnitudes better than the original query, when comparing with running the query until completion and ranking results at the client side.

Chapter 2

Related Work

There has been extensive research in the areas of search queries processing and management of data. However, there has not been much research in combining the two. When it comes to search queries, it is usually assumed that the data can fit into main memory and the complexity lies with exploring the large search space. One notable example is Constraint Programming (CP) [45]. However, these days a lot of data is stored and managed in DBMSs. While there are methods available to support specialized search queries in DBMSs [12, 18, 19, 53, 13], they target particular problems that can be solved either with the help of existing indexes or after precomputing additional information first. Often, each query type needs its own precomputed structures, and changing query constraints leads to rebuilding these structures. Such solutions lack generality, limit users to particular types of query constraints, and offer poor support for ad-hoc exploration.

2.1 Constraint Programming

There is a considerable body of research directed to making search efficient for large search space queries with different types of constraints [45]. Traditional Constraint Programming (CP) solvers (e.g., Gecode, IBM ILOG, Comet) support a large variety of constraints and are highly extendable, both in terms of new types of constraints and search heuristics. A typical solver builds a search tree for the problem and traverses it attempting extensive pruning at the tree nodes. Pruning allows solvers to provably cut entire sub-trees from the search space, dramatically decreasing its size. Introducing new search heuristics allows users to tune the search process by customizing the tree traversal strategy, which might improve performance for specific and new types of problems without modifying the solver internals. To efficiently traverse very large search spaces most CP solvers support parallel search, including work stealing [35, 15] and over-partitioning [46]. Work stealing involves dividing the search space between solvers at run-time, where idle solvers take away parts of the search space from currently busy solvers. Over-partitioning treats a CP problem as embarrassingly parallel, dividing the search space into a very large number of pieces before the query starts. Then, at run-time, each solver traverses a single piece until completion and takes

another one until all pieces are exhausted. We give a background on CP solvers and discuss parallel search in more detail in Chapter 4.

Our search framework, Searchlight (Chapter 4), employs an existing CP solver to perform the search. We have not modified the solver's logic, so Searchlight does not introduce any new constraint programming techniques. The main theme of our research and, thus, the main novelty behind Searchlight is efficiently executing CP-based search queries over *DBMS data* in-place, without the need to extract it and load in memory. Our experiments (Section 4.5) showed that traditional CP solvers are ill suited for searching large volumes of data or handling DBMS-resident data natively, i.e., without extracting, transforming and partitioning it. Solvers assume that the data fits into main memory, and query constraints can be verified relatively quickly. When taking secondary storage latency and performance into consideration, this assumption does not hold in practice. Solvers perform a large number of constraints checks during the search, which makes secondary storage a major bottleneck. Thus, efficiently mediating solver's access to the DBMS-resident data becomes an important problem. To address this issue Searchlight uses a novel two-stage *Solve-Validate* approach. CP solvers perform the search using in-memory *synopsis* structures. This, however, might result in producing *false-positive* solutions. At the second stage solutions are verified on the *original* data to ensure correct query results. We describe this approach in more detail in Chapter 4.

2.2 Interactivity and Online Answering

Another important area of research is interactivity of DBMS queries, which often means producing *online* results [24, 22, 44]. Interactivity does not necessarily mean completing the entire query in a matter of seconds. If a query contains complex computations (e.g., multiple aggregate functions accessing large portions of data) and produces multiple results, outputting them all quickly might not be physically possible. However, it might be possible to severely decrease delays between subsequent results, when introducing them one-by-one to the user. Another common way to support interactivity is approximate answering, where the system outputs approximate result and possibly keeps updating it during the execution. Interactivity allows systems to support "human-in-the-loop" exploration, in which users do not necessarily need to see the entire and/or exact result.

One of the most interesting topics in online answering for DBMS queries is *online aggregation* [24, 22], which strives to bring approximate results to the user when computing common aggregate functions over large data sets. The result contains a *confidence interval* based on the required probability of containing the real result, so the user better understands its guarantees. It is also constantly updated as the computation evolves and more data tuples are explored.

We designed both our frameworks, Semantic Windows and Searchlight, with the goal of being as interactive as possible. Rather than support approximate answering, we decided to provide *exact* online results to the users. Our frameworks support this by steering the search in the direction of possible results (solutions) using sampling or synopses, and then validating such (approximate) results over the original data as quickly as possible. One might argue about the applicability of

online aggregation techniques for approximate answering of search queries. As we extensively argue in Chapter 3, search queries in general deal with complex objects, which makes the direct application of online aggregation with confidence guarantees to such queries non-trivial. Thus, approximate answering for such queries remains an open problem.

Online results have been studied for more complex queries as well. One example is skyline-over-join queries [44], which compute non-dominated points for queries involving joins. In some cases it is possible to carefully examine input and output distribution of the join operator to determine input tuple ranges that might produce dominating results needed for the skyline. Such ranges are processed first. In this case online results are guaranteed to be exact. Note, however, that such queries not only are highly specialized, but also do not involve search. Determining the output distribution for a search query is naturally impossible, since it would require knowing the location of the results beforehand. This defeats the very purpose of search queries. Such complexity makes most dominate-based online answering techniques applicable for traditional queries unsuitable for search queries.

2.3 Database Queries Processing

We want to explicitly contrast our work with aggregate query processing and data cube exploration. The first one involves efficient computation of aggregates for the specified query region (e.g., via the SQL `WHERE` clause). While constraints in Searchlight might contain aggregates, its queries are much more complex in nature, since they involve search. In general, a search query cannot be easily mapped to an aggregate query, since the former does not have a query region. The complexity lies with finding this region, based on the specified constraints.

To compute aggregates, additional structures (e.g., R-/B-trees) are often used to find the tuples belonging to the region and perform the computation. These structures can be extended to include more information about the data. For example, Multi-Resolution Aggregate (MRA) trees [32], where nodes are annotated with aggregate information (e.g., sum/min/max/count). The query's aggregates can be estimated at every level of the tree with progressively better intervals, guaranteed to contain the exact result. As we discuss in Chapter 4, Searchlight uses similar techniques as the basis for synopsis structures, which allow the search process to quickly estimate functions and prune the search space.

Sampling-based methods [24, 8, 40] are also commonly used to approximate aggregates, and often have lower costs. Our first framework, Semantic Windows [27], which we present in 3, uses stratified sampling techniques for aggregate estimation to steer the search in the direction of promising candidates. However, sampling does not provide 100% confidence guarantees, which makes them unsuitable for *provably* pruning search sub-trees. This is the main reason why Searchlight uses synopsis structures. Synopses provide answers in form of intervals, which are guaranteed to contain the real result. As we show in Chapter 4, this technique allows Searchlight to severely reduce the amount of data that have to be read.

Data cube exploration [21] involves computing one or more aggregates over `GROUP BY`s of subsets of attributes. For a particular `GROUP BY` users can perform operations like roll-up (expanding an attribute from the `GROUP BY` set) or drill-down (adding an attribute to the `GROUP BY` set). Thus, data cube exploration is essentially a series of related aggregate queries. Different techniques have been explored to speed-up the exploration, e.g., materializing parts of the cube [23], compressing it for general [50] and spatio-temporal cases [33]. Additional information can be stored in the cube to provide more information to the user, e.g., the degree of abnormality for values [47]. The important distinction with our work is that data cube exploration does not involve search. When the user specifies the `GROUP BY` attributes and the aggregates, the problem lies in efficiently computing the corresponding aggregate queries.

2.4 Search Queries in Databases

While DBMSs lack general search methods, there is a large body of research dedicated to specific problems [12, 18, 19, 53, 13] involving search. For example, Top-K and Skyline queries [12, 18] look for dominating tuples, e.g., "find top-3 companies by revenue" or "find best houses based on the number of bathrooms and the total area". Such queries, however, are usually solved by first precomputing and indexing additional information first. Then, the index can be used to retrieve the answers efficiently with little additional effort. For example, the Onion technique [13] builds convex hulls for required DBMS relations and then uses the hulls to answer optimization (e.g., min/max) queries with linear constraints. Another interesting approach involves building a specialized graph-like index for optimization problems and using the A* algorithm to traverse it [53].

Another approach to search queries involves some form of nearest neighbors search. For example, it might be possible to use histograms [11] to estimate the required linear distance from the query point (e.g., the "ideal" house, as defined by the user) to other possible candidates. This defines a region for tuples that need to be checked. Then, the region's tuples are retrieved and sorted based on the distance function. The function must be defined by the user and reflects her search preferences (e.g., "the house area is two times more important than the number of bathrooms").

We want to emphasize two specific issues with the approaches discussed above. First of all, they target specific problems with specific constraints. Additional indexes are built with the assumption that all queries and constraints are known in advance, which might not hold true for ad-hoc exploration. In case no estimations can be made or no index is available, the DBMS would have to resort to a sequential scan of the whole data, which is prohibitively expensive for large data sets. Secondly, such queries generally search for tuples. If the user wants to search for more complex objects, like rectangular regions with flexible sizes, not only traditional indexes (e.g., R-/B-trees) are not suitable for the task, but also even an exhaustive sequential scan ceases to be an option due to the severely increased search space. Indexing all possible objects (e.g., regions) the user might search for is also infeasible due to a very large number of them. These issues clearly indicate the need for general search methods available within the DBMS and integrated with its query processing.

2.5 Spatial Databases

Spatial DBMSs [48] allow users to manage and query spatial data. These systems are often built on top of traditional DBMSs, and can efficiently retrieve particular objects (e.g., buildings, rivers, etc.), find all features inside a specified query region and perform nearest-neighbors search. Such queries, however, do not generally involve search. Objects can be easily indexed and then retrieved by using common index structures, like pyramids [51, 7] or R-trees. For nearest-neighbors search, the point of reference is given (e.g., find the nearest ATM to the *current location*).

The most interesting type of spatial queries related to search is Content-Based Retrieval (CBR) [48], which explores relationships between objects (e.g., topological, directional, metric). For example, the user might search for a building near a lake with a grove nearby. One common way to process CBRs is to precompute a complete graph of all objects, containing the relationship information [6, 41]. Then, the search can be performed using this graph. The constraints are generally easy to check, since they involve looking up the corresponding graph edge, and the search space is small. In contrast, for search queries we target in our research we assume a potentially large search space of objects, which is infeasible to precompute, index and maintain. The constraints are also more expensive, since they might involve multiple computations that involve accessing a lot of data.

2.6 Query Relaxation, Contraction and Top-K Answering

Query relaxation [38, 37, 31, 36] deals with the empty-answer and too-few-answers query problems by relaxing the original query constraints to include more answers. The user usually provides a cardinality requirement for the number of answer she wishes to obtain. If the query is empty or does not provide enough answers, the original constraints are modified to include more answers so that the cardinality requirement is satisfied as close as possible (i.e., without overflowing the user with too many results) and the answers are "close" to the original constraints. This "closeness" is usually measured by a distance function, in which case the query outputs relaxed results with the minimal distance from the original constraints. Another option is to guide the user through the relaxation interactively, via multiple steps, by providing possible relaxation options at each step.

The existing body of work on query relaxation can be roughly put into two broad categories. The first category includes relaxation based on some statistics readily available in the database. For example, Stretch-and-Shrink (SnS) framework [36] uses query cardinality estimations via a precomputed sample to check if the original query satisfies the cardinality requirement and then use the estimations to find relaxed ranges for each range-based query constraint independently. In such way the cardinality requirement can be fulfilled as close as possible by relaxing only a single constraint. Then the user is guided through a series of steps, where at each step she chooses the next constraint to relax and its value. After each step SnS recomputes the relaxed ranges to accommodate the user choices. This way SnS can actually handle query contraction as well, if the original query's cardinality exceeds the one required by the user. The framework heavily relies on fast cardinality estimations. Multiple estimations might have to be made at every step of the

interactive refinement. Another framework [11] uses histograms to produce cardinality estimations and derive proper constraint ranges for the query. In this example 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 [37] and machine learning [38] frameworks to produce relaxations and guide the user through the process. 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. Such solutions would not be suitable for Searchlight, since, as we discussed earlier in this chapter, our framework 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 available in the database to relax constraints during the query execution. One approach [31] is to relax join and selection predicates, and obtain the relaxation *skyline*. The framework targets range-based constraints as well, so each result can be represented as a vector of distances from the original constraint ranges. The results with non-dominated distance vectors are output to the user. The proposed method uses R-trees to traverse the tuple space, and use MBRs (Minimum Bounding Rectangles) provided by R-tree nodes to prune the nodes dominated by the running skyline. Actually, using R-trees as means of hierarchical representation of the tuple space (i.e., as the search space tree) and the MBR-based domination pruning is a common technique for relaxing (and contracting) queries in relational databases. In general such methods have limited applicability for Searchlight, since results are not known and, hence, cannot be indexed beforehand. Searchlight uses search space traversal and pruning techniques as part of the query processing and the relaxation processes. However, the structure of the search space differs significantly from the relational one, since Searchlight generally works with regions instead of tuples, and the search space itself depends on the query constraints. Additionally, query constraints might be more complex than ranges over attributes, potentially referencing data outside of regions. This makes R-trees (or other traditional index trees) unsuitable for traversal and pruning.

The too-many-results problem, when the number of results exceeds the user's requirement, creates the dual problem of *contracting* the query. The system might try to modify the query constraints [11, 36] so that the query outputs the required number of results. Such methods generally use precomputed statistics to make fast cardinality estimations and find suitable ranges for query constraints. Thus, the corresponding frameworks usually handle both relaxation and contraction at the same time. Another approach is to get rid of excessive answers by ranking them and outputting only the best few. The "best" can be based on a scalar ranking function (top-k queries) or vector domination (skyline [12] queries), where vectors are produced by ranking each constraint (attribute) independently. These methods commonly rely on traditional precomputed structures, such as views [16, 26] and R-trees [42, 52]. These structures are then used for efficient traversal and possible result pruning. The view-based approaches additionally require to know at least part of the workload beforehand to materialize proper views. The R-tree approaches follow the familiar pattern

of traversing the tree and performing MBR-based pruning. For some top-k queries (e.g., where the tuple rank equals the number of other tuples it dominates) the R-tree can be used to provide fast rank estimations. 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. Examples include sorting [17, 9], batch computation [9] or building structures optimized for particular queries [42]. As we pointed out before, for Searchlight queries no indexes are available beforehand. The only option is to perform the search over the data and perform query contraction during the query execution. This might seem similar to the sequential scan-based approaches, however, as we have discussed in this section, the nature of the queries is different, which requires us to introduce new approaches.

Chapter 3

Interactive Data Exploration Using Semantic Windows

As the first step in developing an easy-to-use, interactive approach for human-in-the-loop exploratory analysis of data at scale, we discuss our novel search framework called *Semantic Windows* [27]. It allows users to conveniently perform structured search via shape and content constraints over a multidimensional data space.

3.1 Semantic Windows Overview

Consider the following data exploration framework. A user examines a multidimensional data space by posing a number of queries that find rectangular regions of the data space the user is interested in. We call such regions *windows*. After getting some results, the user might decide to stop the current query and move to the next one. Or she might want to study some of the results more closely by making any of them the new search area and asking for more details. Let us look at two illustrative examples in Figure 3.1.

Example 3.1.1. A user wants to study a data set containing information about stars and other objects in the sky (e.g., SDSS [4]). She wants to find a rectangular region in the sky satisfying the following properties:

- The shape of the region must be 3° by 2°, assuming angular coordinates (e.g., right ascension and declination).
- The average brightness of all stars within the region must be greater than 0.8.

The example above describes a spatial exploration case, where windows of interest correspond to two-dimensional regions of the sky. However, the framework can be used for other cases, e.g., for one-dimensional time-series data:

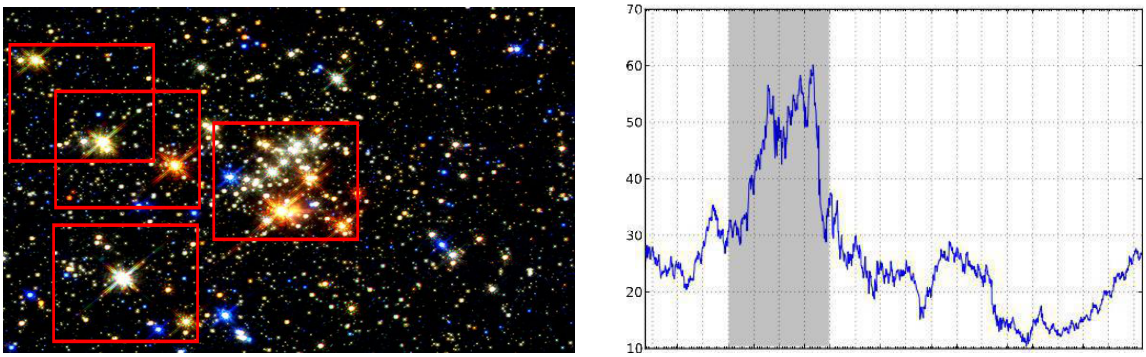

Figure 3.1: Exploration queries searching for bright star clusters (left) and periods of high average stock prices (right).

Example 3.1.2. A user studies trading data for some company. The search area represents stock values over a period of time. The user wants to find a time interval (i.e., a one-dimensional region) having the following properties:

- The interval must be of length from 1 to 3 years.
- The average of stock prices within the interval must be greater than 50.

By defining windows of interest in terms of their desired properties, users can express a variety of exploration queries. Such properties, which we call *conditions*, can be specified based on the shape of a window (e.g., the length of a time interval) and an aggregate function of the data contained within (e.g., the average stock value within an interval).

Another important requirement in data exploration is interactivity. Since the amount of data users have to explore is generally large, it is important to provide online results. This allows the user to interrupt the query and modify it to better reflect her interests. Moreover, many applications can be satisfied with a subset of results, without the need to compute all of them. Most SQL implementations do not allow such functionality, making users wait for results until the entire query is finished.

Semantic Windows (SW) treats an exploration query as a data-driven search task. The search space consists of all possible windows, which can vary in size and overlap. We use a cost-benefit analysis to quantify a utility measure to rank the windows and decide on the order by which we will explore them. We use sampling to estimate values for conditions on the data, which allows us to compute a distance from these values to the values specified in the query. We then guide the search using a *utility* estimate, which is a combination of this distance, called *benefit*, and the *cost* of reading the corresponding data from disk. We use shape-based conditions to prune parts of the search space.

For experimentation purposes, we implemented a *distributed* version of the framework on top of PostgreSQL. The computation is done by multiple *workers* residing on different nodes, which

interact between themselves and with the DBMS via TCP/IP. To explore windows efficiently, we require efficient execution of multidimensional range queries, which can be achieved via common index structures, like B-trees or R-trees. To estimate utilities we collect a sample of the data offline and store it in the DBMS.

In summary, the proposed SW framework:

- Allows users to interactively explore data in terms of multidimensional windows and select interesting ones via shape- and content-based predicates.
- Provides online results quickly, facilitating human-in-the-loop processing needed for interactive data exploration.
- Uses a data-driven search algorithm integrated with stratified sampling, adaptive prefetching and data placement to offer interactive online performance without sacrificing query completion times.
- Offers a diversification approach that allows users to direct the search for qualifying SWs in unexplored regions of the search space, thus allowing the user to control the classical exploration vs. exploitation trade-off.
- Provides an efficient distributed execution framework that partitions the data space to allow parallelism while effectively dealing with partition boundaries.

3.2 SW Model and Queries

Assume a data set S containing objects with attributes $a_1, \ldots, a_m$ (e.g., brightness, price, etc.) and coordinates $x_1, \ldots, x_n$. Thus, S constitutes an n-dimensional search area with dimensions $d_1, \ldots, d_n$. We will often specify S in terms of the intervals it spans (e.g., $S = [L_1, U_1) \times [L_2, U_2)$ for a two-dimensional data set). Next, we define a *grid* on top of S. The grid G_S is defined as a vector of *steps*: $(s_1, s_2, \ldots, s_n)$. It divides each interval $[L_i, U_i)$ into disjoint sub-intervals of size s_i, starting from L_i: $[L_i, L_i + s_i) \cup [L_i + s_i, L_i + 2s_i) \cup \cdots \cup [L_i + k \cdot s_i, U_i)$. The last sub-interval might have size less than s_i, which has no impact on the discussion. Thus, S is divided into a number of disjoint *cells*. The search space of an SW query consists of all possible *windows*. A window is a union of adjacent cells that constitutes an n-dimensional rectangle. Since the grid determines the windows available for exploration, the user can specify a particular grid for every query.

Let us revisit examples presented in Section 3.1. Example 3.1.1 could be represented as follows:

- $d_1 = ra$, $d_2 = dec$, $a_1 = brightness$ [1].
- $S = [100°, 300°) \times [5°, 40°)$, $G_S = (1°, 1°)$.

and Example 3.1.2 as:

[1] The original SDSS data does not contain a brightness attribute. However, this attribute or a similar one can be computed from other attributes using an appropriate function.

- $d_1 = time$, $a_1 = price$.
- $G_S = (1\,year)$, $S = [1999/02/01, 2003/11/30)$

An *objective function* $f(w)$ is a scalar function, defined for window w. There are two types of objective functions:

- *Content-based.* They are computed over objects belonging to the window. Since the value must be scalar, this type is restricted to aggregates (average, sum, etc). We further restrict them to be *distributive* and *algebraic* [21]. This is done for efficiency purposes — the value of $f(w)$ must be computable from the corresponding values of the cells in w. We discuss this in more detail in Section 3.5.
- *Shape-based.* They describe the shape of a window and do not depend on data within the window. We restrict ourselves to the following functions: $card(w)$ and $len_{d_i}(w)$. The former defines the number of cells in w, which we call the *cardinality* of w. The latter is the length of w in dimension d_i in cells. Assuming w spans interval $[l_i, u_i)$ in d_i, $len_{d_i}(w) = \frac{u_i - l_i}{s_i}$. Other functions, for example computing the perimeter or area, are possible.

A *condition* c is a predicate involving an objective function. The result of computing condition c for window w is denoted as w_c. The framework is restricted to algebraic comparisons, for example $f(w) > 50$.

The conditions for Example 3.1.1 can be defined as: $len_{ra}(w) = 3$, $len_{dec}(w) = 2$ and $avg\_brightness(w) > 0.8$. For Example 3.1.2, the conditions can be expressed as: $len_{time}(w) \geq 1$, $len_{time}(w) \leq 3$ and $avg\_price(w) > 50$.

An SW query can now be defined as $Q_{SW} = \{S, G_S, C\}$, where C is a set of conditions. The result of the query is defined as: $RES_Q = \{w \in W_S | \forall c \in C : w_c = true\}$, where W_S is a set of all windows, defined by G_S.

3.3 Existing SQL Extensions for Data Exploration

As a first option, we look into expressing SW queries using SQL. While SQL has constructs for working with groups of tuples, such as `GROUP BY` and `OVER`, they are insufficient for expressing all possible windows. `GROUP BY` does not allow overlapping groups, which makes it impossible to express overlapping windows. `OVER` allows users to study a group of tuples (also called window in SQL) in the context of the current tuple via `PARTITION BY` clause. However, it allows only one such a group for every tuple, not a series of groups with different shapes. Thus, only a subset of possible windows can be expressed this way. The standard SQL is even more restrictive and does not allow functions to be used in `PARTITION BY`. This makes it difficult to express multidimensional windows at all.

One general way, which we implemented, is to express an SW query as follows:

1. Compute objective function values for every cell of the grid via `GROUP BY`.

```
WITH RECURSIVE
-- computing cells
cells(cell_id, lx, rx, ly, ry, val) AS (
    SELECT cell_num(x, y),
        (cell_coords(cell_num(x, y))).lx,
        (cell_coords(cell_num(x, y))).rx,
        (cell_coords(cell_num(x, y))).ly,
        (cell_coords(cell_num(x, y))).ry,
        (sum(val), count(val))::sc_val
    FROM data
    WHERE x >= start_x() AND x < end_x()
        AND y >= start_y() AND y < end_y()
    GROUP BY cell_num(x, y)
),
-- all windows built by expanding to the "right"
windows_int(lx, rx, ly, ry, val) AS (
    SELECT lx, rx, ly, ry, val
    FROM cells
    UNION ALL
    SELECT r.lx, r.rx + step_x(), r.ly, r.ry,
        (SELECT ((r.val).s + sum((c.val).s),
            (r.val).c + sum((c.val).c))::sc_val
         FROM cells AS c
         WHERE (c.lx = r.rx AND c.ly >= r.ly AND
             c.ry <= r.ry))
    FROM windows_int AS r
    WHERE EXISTS(SELECT 1
                 FROM cells AS c
                 WHERE (c.lx = r.rx AND c.ly >= r.ly
                     AND c.ry <= r.ry))
),
-- all windows built by expanding "up"
windows(lx, rx, ly, ry, val) AS (
    SELECT lx, rx, ly, ry, val
    FROM windows_int
    UNION ALL
    SELECT r.lx, r.rx, r.ly, r.ry + step_y(),
        (SELECT ((r.val).s + sum((c.val).s),
            (r.val).c + sum((c.val).c))::sc_val
         FROM cells AS c
         WHERE (c.ly = r.ry AND c.lx >= r.lx AND
             c.rx <= r.rx))
    FROM windows AS r
    WHERE EXISTS(SELECT 1
                 FROM cells AS c
                 WHERE (c.ly = r.ry AND c.lx >= r.lx
                 AND c.rx <= r.rx))
)
-- main query
SELECT lx, rx, ly, ry, (val).s / (val).c AS val
FROM windows
WHERE (val).c <> 0 AND (val).s / (val).c > 200
```

Figure 3.2: An example of SQL query expressing a simple exploration SW query.

2. Generate every possible window by combining cells, using recursive Common Table Expressions (CTEs). Values for the objective functions are computed by combining the values of the cells.

3. Filter windows that do not satisfy the conditions.

Figure 3.2 gives an example of a two-dimensional SW query expressed this way. It searches for all windows (without any other restrictions) for which $avg(val) > 200$. Note that this query is even less complex than the one from Example 3.1.1. Some parts of the query are omitted for the sake of clarity, since they are not required for understanding the query. This includes:

- Helper functions and types, which are also written in SQL and define simple expressions and constants. For example, `cell_num()` computes the number of the cell a tuple belongs to in the row-major order.

- A small sub-query that computes $avg(val)$ via `GROUP BY` for every possible cell, including empty ones. One caveat of `GROUP BY` is that it does not output cells that contain no tuples. Since such cells are still required to compute the windows, they are explicitly assigned values of 0.

This type of query leads to two major problems. First, due to its complexity most traditional query optimizers would likely have hard time executing the query efficiently. As an example, we provide experimental results for PostgreSQL in Section 3.6. More importantly, the query performs an aggregation in the beginning. This means the computation is blocked until all cells have been computed. Such a query would not be able to output online results. Techniques like *online aggregation* [24] are very limited here, since exact, not approximate, results are required. Also, applying online aggregation to such a complex query is challenging at best.

To address these problems we propose to extend SQL to directly express SW queries. Our extensions are as follows:

- The new `GRID BY` clause for defining the search space. This clause replaces `GROUP BY` (both cannot be used at the same time).

- New functions that can be used for describing windows. Namely, $LB(d_i), UB(d_i)$ to compute the lower and upper boundaries of a window in dimension d_i. Other functions are possible depending on the user's needs.

- New functions for defining shape-based conditions. Namely, $LEN(d_i)$, which is equivalent to $len_{d_i}(w)$. This function is syntactic sugar, since it is possible to compute it by using boundary functions introduced above.

Additionally, we reuse the existing SQL `HAVING` clause to define conditions for the query. Figure 3.3 shows how Example 3.1.1 can be expressed with the proposed extensions.

In the `GRID BY` clause, `BETWEEN` defines the boundaries of the search area for every dimension and `STEP` defines steps of the grid (`ra`, `dec` are attributes of `sdss` and serve as dimensions). The query is

```
SELECT LB(ra), UB(ra), LB(dec), UB(dec),
       AVG(brightness)
FROM sdss
GRID BY ra BETWEEN 100 AND 300 STEP 1,
        dec BETWEEN 5 AND 40 STEP 1
HAVING AVG(brightness) > 0.8 AND
       LEN(ra)  = 3 AND
       LEN(dec) = 2
```

Figure 3.3: The SW query from Figure 3.2 written with the proposed SQL extensions.

processed in the same way as a `GROUP BY` query in SQL, except that instead of groups it works with windows defined via the `GRID BY` clause. `HAVING` has the same meaning — filtering windows that do not satisfy conditions. Since `SELECT` outputs windows, only functions describing a window can be used there: the ones describing the shape and the ones that were used for defining conditions. Similar restrictions are imposed in the SQL standard for `GROUP BY` queries.

3.4 The SW Framework

3.4.1 Data-Driven Online Search

We first describe our basic algorithm for executing SW queries with online results. Logically, the search process can be described as follows:

1. Data set S is divided into cells c_i as specified by grid G_S.
2. All possible windows w are enumerated and explored one by one in an arbitrary order.
3. If window w satisfies all conditions (i.e., $\{\forall c \in C : w_c = true\}$), the window belongs to the result.

This suggests a naive algorithm to compute an SW query. The algorithm presented in this section is designed to provide online results in an efficient way. The main idea is to dynamically generate promising candidate windows as the search progresses and explore them in a particular order.

The search space of all possible windows is structured as a graph. First we define relationships between windows. Giving a window w, an *extension* of w is a window w', which is constructed by combining cells of w with a number of adjacent cells from its neighborhood. If w' is extended in a single dimension and direction from w, it is called a *neighbor*. An example can be seen in Figure 3.4. A two-dimensional search area is divided into four cells, labeled 1 through 4. Window $1|2|3|4$ [2] is an extension of window 1, since it is produced by adding adjacent cells 2 through 4. At the same time, $1|2|3|4$ is a neighbor of $1|2$, since $1|2$ is extended only in the vertical dimension and the only

[2] $c_1|\ldots|c_k$ labels a window consisting of cells c_1 through c_k

direction — "down". The resulting search graph consists of vertices representing windows and edges connecting neighbors.

With the search graph defined, we use a heuristic best-first search approach to traverse it. The heuristic is based on the utility of a window, which will be discussed in more detail in Section 3.4.2. In a nutshell, utility is a combination of the cost of reading a window from disk and the potential benefit of the window, which is a measure of how likely the window is to satisfy the query conditions. The resulting algorithm is presented as pseudo-code below:

Algorithm 1 Heuristic Online Search Algorithm

```
Input: search space S, grid G_S, conditions C
Output: resulting windows RES_Q

procedure HeuristicSearch(S, G_S, C)
    PQ ← ∅, RES_Q ← ∅                                  ▷ PQ — priority queue
    StartW ← StartWindows(S, G_S, C)
    for all w ∈ StartW do
        EstimateUtility(w)
        insert(PQ, w)                                   ▷ utility as priority
    while ¬empty(PQ) do
        w ← pop(PQ)
        UpdateUtility(w)
        if Utility(w) ≥ Utility(top(PQ)) then
            Read(w)                                     ▷ read from disk if needed
            UpdateResult(RES_Q, C, w)
            N ← GetNeighbors(w, S, G_S)
            for all n ∈ N do
                EstimateUtility(n)
                insert(PQ, n)
        else
            insert(PQ, w)
```

The algorithm uses a priority queue to explore candidate windows according to their utilities. The utility is estimated before a window is read from disk via a precomputed sample. Since estimations might improve while new data is read from disk during the search, when the next window is popped from the queue, we update its utility via the *UpdateUtility*() function. This can be seen as lazy utility update. If the window still has the highest utility, it is explored (i.e., read from disk and checked for satisfying the conditions). Otherwise, it is returned to the queue to be explored later. Additionally, we periodically update the whole queue to avoid stale estimations. In this case only the windows for which new data is available are actually updated.

The procedure begins by determining the initial windows via the *StartWindows*() function. Since the search space might be large, it is important to aggressively prune windows. Suppose the user specifies a shape-based condition that defines the minimum length n for resulting windows in some dimension (e.g., $len_{d_i}(w) \geq n$). *StartWindows*() does not generate windows that cannot satisfy this condition, effectively pruning initial parts of the graph. Otherwise, the search starts

from cells. A similar check is made in the $GetNeighbors()$ function, which generates all neighbors of the current window. It checks for conditions that specify the maximum length in a dimension (e.g., $len_{d_j}(w) \leq m$) and does not generate neighbors that violate such conditions.

Since the number of windows can be very large, at some point the priority queue may not fit into memory. It is possible to spill the tail of the queue into disk and keep only its head in memory. When a new window has a low utility, it is appended to the tail (in any order). When the head becomes small enough, part of the tail is loaded into memory and priority-sorted, if needed. For efficiency, the tail can be separated into several buckets of different utility ranges where windows inside a bucket have an arbitrary ordering.

While additional pruning, based on other conditions, might further increase the efficiency of the algorithm, it is not safe to apply in general. Since providing approximate results is not an option we consider, it is not possible to discard a window or its extensions solely on the basis of estimations. Content-based conditions must be checked on the exact data. In general, extensions have to be explored as well, since they too can produce valid results. However, in some restricted cases, it is possible to prune them.

One example is the so-called *anti-monotone* constraints [39]. In case of a content-based condition $sum(w) < 10$ and assuming $sum()$ can produce only non-negative values, it is possible to prune all windows that contain the current window w' if $sum(w') \geq 10$, since $sum()$ is monotonic on the size of a window. The length and cardinality of a window are other examples of such functions. Since they are data-independent, the corresponding conditions are always safe to use for pruning, which our algorithm supports. Such anti-monotone pruning, however, would not necessarily decrease the amount of data that has to be read. Windows that just overlap with w' might still be valid candidates for the result.

3.4.2 Computing Window Utilities

The utility of a window is a combination of its *cost* and *benefit*. The cost determines how expensive it is to read a window from disk. Since the grid is defined at the logical level without considering the underlying data distribution, some windows may be more expensive to read than others, if the data distribution is skewed. Also, since windows overlap, the cost of a window may decrease during the execution if some of its cells have already been read as parts of other windows. We assume that the system caches objective function values for every cell it reads, so it is not necessary to read a cell multiple times. The cost is computed as follows. Let $|S| = n, |G_S| = m$, where $|S|$ is the total number of objects in the data set and $|G_S|$ is the total number of cells. Let the number of objects belonging to non-cached cells of window w be $|w|_{nc}$. Then the cost C_w of the window is computed as: $C_w = \frac{|w|_{nc}}{n/m} = \frac{|w|_{nc} m}{n}$.

In case data does not have considerable skew, the cost is approximately equal to the number of non-cached cells belonging to the window. However, in general the cost might differ significantly, depending on the skew. To compute the cost accurately, it is necessary to estimate the number of objects in a window. We use a precomputed sample for the initial estimations and update these

estimations during the execution as we read data. When reading a window, we not only compute the objective function values, but also the number of objects for every cell. This results in computing an additional aggregate for the same cells, which does not incur any overhead.

The second part of the utility computation is benefit estimation. Here, we determine how "close" a window is to satisfying the user-defined conditions. First, we compute the benefit for every condition independently. The framework currently supports only comparisons as conditions, but the approach can be generalized for other types of predicates. Assume an objective function $f(w)$ and the corresponding condition in the form: $f(w)$ *op* val, where *op* is a comparison operator. We assume $f(w)$ can be estimated for window w via the precomputed sample. Let the estimated value be f_w. Then the benefit b_w^f for condition f for window w is computed as follows:

$$b_w^f = \begin{cases} \max\left\{0, 1 - \frac{|f_w - val|}{eps}\right\} & \text{if } f_w \; op \; val = false \\ 1 & \text{if } f_w \; op \; val = true \end{cases}$$

The value *eps* determines the precision of the estimation and is introduced to normalize the benefit value to be between 0 and 1. It often can be determined for a particular function based on the distance of the corresponding attribute values from val. For example, if $f = avg(a_i)$, then $\max\{|val - min(a_i)|, |val - max(a_i)|\}$ can serve as *eps*. Alternatively, a value of the magnitude of val can be taken initially and then updated as the search progresses.

The total benefit B_w of window w is computed as the minimum of the individual benefits, since a resulting window must satisfy all conditions: $B_w = \min_{f \in C} b_w^f$.

Since $b_w^f \in [0, 1]$, it follows that $B_w \in [0, 1]$. The utility of window w is a combination of the benefit and cost:

$$U_w = sB_w + (1 - s)\left(1 - \min\left\{\frac{C_w}{k}, 1\right\}\right)$$

The cost is divided by k to normalize it to $[0, 1]$. In case the user did not provide any restrictions on the maximum cardinality for windows, k is equal to m. Otherwise, it is equal to the maximum possible cardinality inferred from shape-based conditions. The parameter s ($s \in [0, 1]$) is the *weight* of the benefit. Lowering the value allows exploring cheaper, but promising, windows first, while increasing it prioritizes windows with higher benefits at the expense of the cost. Intuitively, it is better to first explore windows with high benefits and use the cost as a tie-breaker, picking the cheaper window when benefits are close to each other.

We illustrate the algorithm with the search graph from the previous section, presented in Figure 3.4. The exact parameters for the search area are irrelevant. Assume the whole data set contains $n = 200$ objects and the number of objects in cells 1, 2, 3 and 4 ($m = 4$) is 50, 20, 30 and 100 respectively. We assume that $eps = 10, k = 4$ and $s = \frac{1}{2}$. The only condition is $f(w) > 13$ and the estimated values f_w are specified on top of windows. The numbers to the left (bottom) of windows specify benefits, costs and utilities as b, c and u respectively. Initially, only the values for cells are computed, and the values for other windows are computed when they are explored. The search progresses as follows:

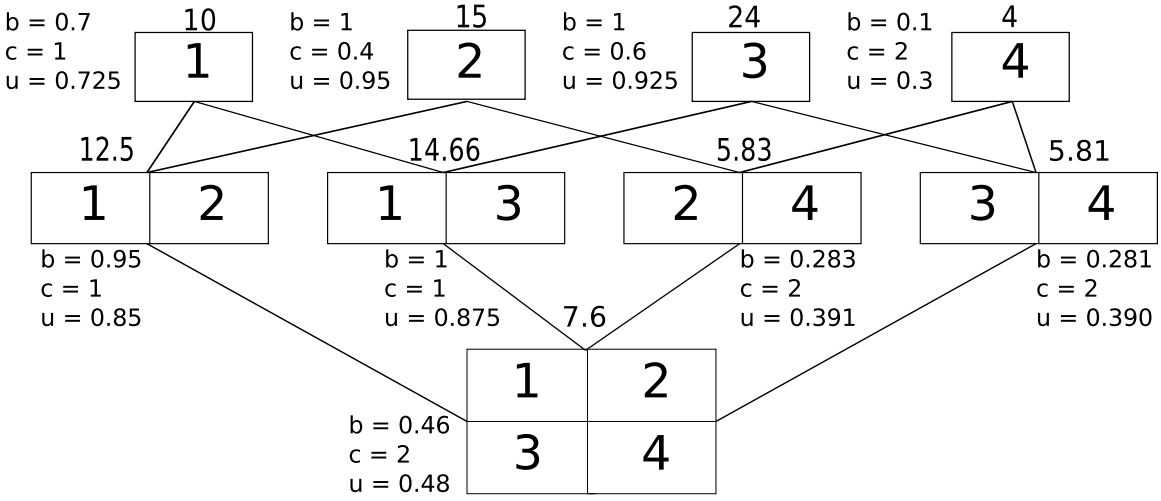

Figure 3.4: The search graph for an SW query, annotated with *b*enefits, *c*osts, *u*tilities and objective function estimations

1. Since there are no shape-based conditions, the search starts with cells. Initially, $PQ = [2, 3, 1, 4]$, and the algorithm picks cell 2, which is read from disk. It generates two neighbors: 1|2 and 2|4. $f(w)$ is estimated from the sample, and the windows are put into the queue. Note that $C_{1|2} = C_1 = 1, C_{2|4} = C_4 = 2$, since cell 2 has already been read.

2. $PQ = [3, 1|2, 1, 2|4, 4]$. The algorithm picks cell 3 and reads it from disk. It generates its neighbors: 1|3 and 3|4.

3. $PQ = [1|3, 1|2, 1, 2|4, 3|4, 4]$. The next window to explore is 1|3. It is processed in the same way and generates the only neighbor possible — 1|2|3|4.

4. $PQ = [1|2, 1, 1|2|3|4, 2|4, 3|4, 4]$. The search moves to window 1|2. Since cells 1 and 2 have already been read, the window is not read from disk and explored in memory. Its only extension, 1|2|3|4, was generated already and is skipped.

5. $PQ = [1, 1|2|3|4, 2|4, 3|4, 4]$. The search then goes through 1, 1|2|3|4, 2|4, 3|4 and, finally, 4 in the order of their utilities. Due to caching, all windows except 1|2|3|4 are checked in memory, and when 1|2|3|4 is explored, only cell 4 is read from disk.

At every step, after a window is read from disk, the condition is checked. If the window satisfies the condition, it is output to the user. Otherwise, it is filtered.

3.4.3 Progress-driven Prefetching

Reconsider the heuristic search algorithm presented in Section 3.4.1. Every time a window is explored, all of its cells that do not reside in the cache have to be read from disk. If the grid contains a large number of cells, the search process might perform a large number of reads, which might incur

considerable overhead. If the data placement on disk does not correspond to the locality implied by windows, reading even a single window might result in touching a large number of pages dispersed throughout the data file. The problem goes beyond disk seeks. If only a small portion of objects from each page belongs to the window, these pages have to be re-read later when other windows touch them. This might create thrashing. An implementation of the algorithm that does not modify the database engine cannot deal with this problem. The problem might be partially remedied by clustering the data, yet there are times when users cannot change the placement of data easily. The ordering might be dictated by other types of queries users run on the same data. Another possible way is to materialize the grid by precomputing all cells. However, this requires that the query parameters (i.e., the search area, the grid and functions) be known in advance. Since exploration can often be an unpredictable ad hoc process, we assume the parameters may vary between queries. In this case, materialization performed for every query will make online answering impossible.

To address this problem, we use an adaptive prefetching strategy. Our framework explores windows in the same way, but it prefetches additional cells with every read. The window is extended to a new window, according to the definition of the extension from Section 3.4.1. When the size of prefetching increases, intermediate delays might become more pronounced due to the additional data read with every window. At the same time, due to the decreased number of reads, this reduces the overhead and the query completion times. Decreasing the size of prefetching has the opposite effect. While this approach can be seen as offering a trade-off between the online performance and query completion time, we show that, in some cases, prefetching can be beneficial for both, as we demonstrate through experiments.

During the search it is beneficial to dynamically change the size of prefetching. A read can have two outcomes. *Positive* reads result in reading cells that belong to the resulting windows. It might be the window just read or windows overlapping with it. While new results keep coming, the framework prefetches a constant *default* amount, controlled by a parameter. By setting this parameter users can favor a particular side of the trade-off. On the other hand, a *false positive* read does not contribute to the result, reading cells that do not belong to the resulting windows. A false positive can happen for two reasons:

- The remaining data does not contain any more results. To confirm this, the search process still has to finish reading the data. All remaining reads are going to be false positives. In this case the best strategy would be to read it in as few requests as possible.
- Due to sampling errors, utilities might be estimated incorrectly. Since new results are still possible, it is better to continue reading data via select, short requests.

Since it is basically impossible to distinguish between the two cases without seeing all the data, we made the prefetching strategy adapt to the current situation. In this new technique, which we refer to as *progress-driven prefetching*, the size of prefetching increases with every new consecutive false positive read, which addresses the first case. When a positive read is encountered, the size is reset to the default value to switch back at providing online results. The size of prefetching, p, is

Algorithm 2 Prefetch Algorithm

Input: window w, prefetch size p
Output: window to read w'

procedure PREFETCH(w, p)
  $w' \leftarrow w$
  for $i = 1 \rightarrow n$ **do** ▷ n — number of dimensions
    for $dir \in \{left, right\}$ **do**
      $max \leftarrow C_{w'} + p \prod_{k \neq i} len_{d_k}(w')$
      repeat
        $ext \leftarrow GetNeighbor(w', d_i, dir)$
        if $C_{ext} \leq max$ **then**
          $w' \leftarrow ext$
      until $C_{ext} > max$

computed as:

$$p = (1 + \alpha)^{\alpha + fp\_reads} - 1$$

$\alpha \geq 0$ is the parameter that controls the default prefetching size. We call it the *aggressiveness* of prefetching. In case $\alpha = 0$, $p = 0$ and no additional data is read. Increasing α results in increasing the default prefetching size, which favors the query completion time. fp_reads is the number of consecutive false positives. When it increases, p increases exponentially. If a new result is discovered, fp_reads is set to 0, and p is automatically reset. This corresponds to the adaptable strategy described above. The exponential increase was chosen so that the size would grow quickly in case no results can be found. This allows us to finish the query faster. We assume that the number of consecutive false positives due to sampling errors is not going to be large, so online performance will not suffer.

The pseudo-code of Algorithm 2 describes how a window is extended according to the value of p. The size defines a "cost budget" for the extension. This budget is applied as a number of possible extension cells independently for every direction in every dimension. The budget-based approach allows us to address a possible data skew. Since for a fixed dimension a window can be extended in two directions, we denote them as $left$ and $right$.

3.4.4 Diversifying Results: Exploration vs. Exploitation

Our basic SW strategy is designed for "exploitation" in that it is optimized to produce qualifying SWs as quickly as possible without taking into account what parts of the underlying data space these results may come from. In many data exploration scenarios, however, a user may want to get a quick sense of the overall data space, requiring the underlying algorithm to efficiently "explore" all regions of the data, even though this may slow down query execution.

With the basic algorithm, when a number of resulting windows is output, other promising windows that overlap with them will have reduced costs and higher utilities due to the cached cells. Such windows will be explored first, which might favor results located close to each other. In some

cases it might be desirable to jump to another part of the search space that contains promising, but possibly more expensive, windows. This way the user might get a better understanding of the search area and results it contains. We considered two approaches to achieve that.

The first approach is to include a notion of *diversity* of results within the utility computation. We define a *cluster* of results as an MBR (Minimum Bounding Rectangle) containing all resulting windows that overlap each other. Discovering all final clusters faster might give a better understanding of the whole result. When a new window is explored we compute the minimum Euclidean distance *dist* from the window to the clusters already found. The distance is normalized to $[0, 1]$ and included as a part of the window's benefit: $B'_w = \frac{B_w + dist}{2}$, resulting in the modified utility U'_w. If the window being explored belongs to a cluster, we find the next highest-utility window w' with a non-zero *dist* and compare utilities U'_w and $U'_{w'}$. If w' has a higher utility, it is explored first. We call this a "jump". With this approach, promising windows might be stifled by consecutive jumps. To avoid this problem, the jumping is turned off at the current step if the last jump resulted in a false positive.

Another approach is to divide the whole search area into sub-areas, according to a user's specification. For example, a time-series data might be divided into years. Each sub-area has its own queue of windows and the search alternates between them at every step. If a window spans multiple sub-areas, it belongs to the sub-area containing its left-most coordinate point, which we call the window's *anchor*. This approach is similar to the online aggregation over a `GROUP BY` query [24], where different groups are explored at the (approximately) same pace. Since some sub-areas might not contain results, the approach may cause large delays. At the same time, it makes the exploration more uniform.

3.5 Architecture and Implementation

We implemented the framework as a distributed layer on top of PostgreSQL, which is used as a data back-end. The algorithm is contained within a client that interacts with the DBMS. To perform distributed computation, the clients, which we call *workers*, can work in parallel under the supervision of a *coordinator*. The coordinator is responsible for starting workers, collecting all results and presenting them to the user. The search area is partitioned into disjoint sub-areas among the workers. A window belongs to the worker responsible for the sub-area containing the window's *anchor*, its leftmost point. Since some windows span multiple partitions, the worker is responsible for requesting the corresponding cell data from other workers. An overview of the distributed architecture is shown in Figure 3.5.

The worker consists of the Query Executor, which implements the search algorithm, including various optimizations such as prefetching. The Window Processor is responsible for computing utilities and objective function values (exact and estimated), based on the information about cells provided by the Data Manager. The Data Manager implements all the logic related to reading cells from disk, maintaining additional cell meta-data, and requesting cell data from other workers. This

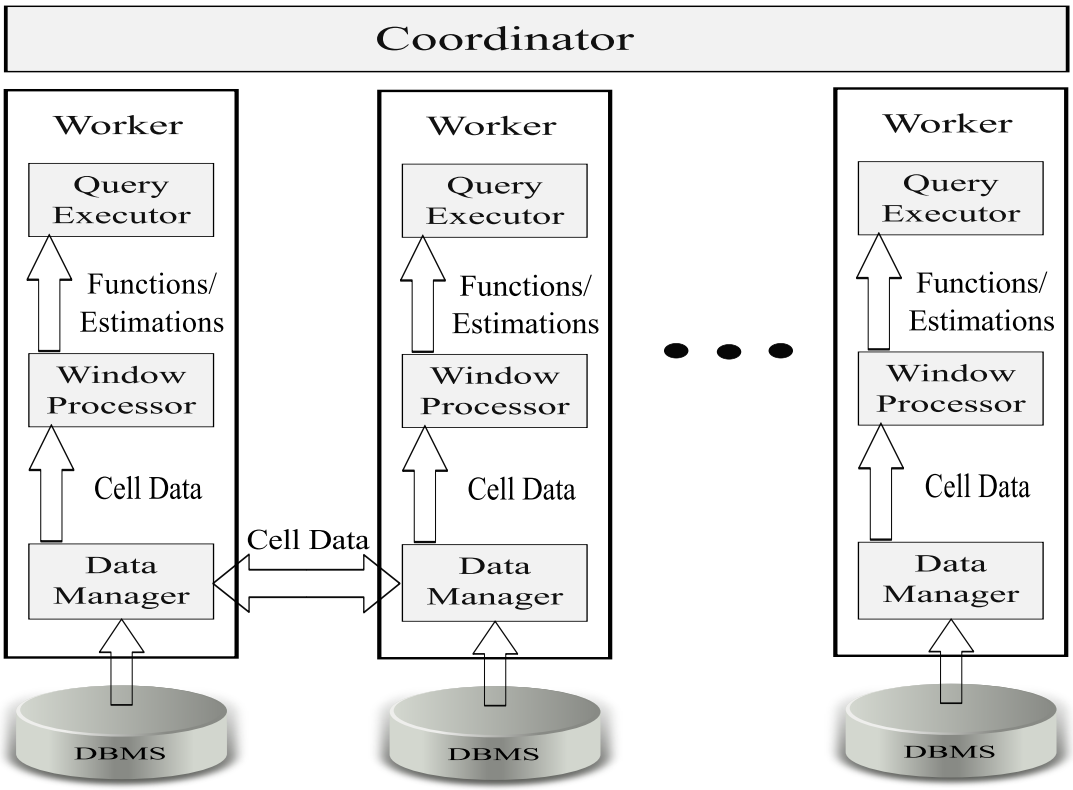

Figure 3.5: Distributed SW architecture

includes:

Caching. The cache contains objective function values for all cells read from disk and requested from other workers. When a new window is explored, only the cells that are not in the cache are read. We assume that the objective values for all cells can fit into memory. This is a fair assumption, since data objects themselves do not have to be cached.

Sample Maintenance. The Data Manager maintains a sample that is used for estimating objective function values and the number of objects for every cell that has not been read from disk. We assume that a precomputed sample is available in the beginning of query execution. The parts of the sample belonging to other partitions are requested from the corresponding workers.

DBMS Interaction and I/O. When a request for a window is made, the Data Manager performs the read via a query to the underlying DBMS. It requests objective function values for non-cached cells belonging to the window in a single query. These values are combined with the cached ones at the Window Processor producing the objective value for the window. Windows consisting entirely of cached cells are processed without the DBMS. We assume that the value of an objective function for a window can be combined from the values of cells the window consists of. All common aggregates (e.g., $min()$, $sum()$, $avg()$, etc.) support this property.

Remote Requests. When a window spans multiple partitions, the Data Manager requests objective

values for the cells belonging to other partitions from other workers. If a remote worker does not have the data in cache, it delays the request until the data becomes available. Eventually it is going to read all its local data and, thus, will be able to answer all requests. After every disk read, the worker checks if it can fulfill more requests. If so, it returns the data to the requester. At the same time, the requester continues to explore other windows. When the remote data comes, the corresponding windows are computed and reinserted into the queue. The only way a worker may block is when it has finished exploring its entire sub-area and is waiting for remote data. Thus, the total query time is essentially dominated by the total disk time of the slowest worker.

3.6 Experimental Evaluation

For the experiments we used the prototype implementation described in Section 3.5, written in C++. Single-node experiments were performed on a Linux machine (kernel 3.8) with an Intel Q6600 CPU, 4GB of memory and WD 750GB HDD. All experiments with the distributed version presented in Section 3.6.4 were performed using EBS-optimized m1.large Amazon EC2 instances (Amazon Linux, kernel 3.4).

Data Sets For the experiments, we used three synthetic data sets and SDSS [4]. Each synthetic data set was generated according to a predefined grid. The number of tuples within each cell was generated using a normal distribution with a fixed expectation. Each synthetic data set contains eight clusters of tuples, where a cluster is defined as a union of non-empty adjacent cells. We generated three queries, one for each data set, which select four clusters. These target clusters differ in their distance from each other, which we call *spread*. Essentially, each data set contains the same clusters (and tuples), although their coordinates differ. Thus, each query has the same conditions, which allowed us to measure the effect of the spread in a clean way. The parameters of the query are: $S = [0, 1000000) \times [0, 1000000), s_1 = s_2 = 10000, card() \in (5, 10), avg() \in (20, 30)$.

For the SDSS data set, the situation is different. It is hard to ensure the same cleanliness of the experiment when dealing with different spreads of the result. We thus chose three queries with (approximately) the same selectivity and different spreads. However, in this case the conditions for each query differ and the search process explores different candidate windows in every case. The parameters of the queries are: $S = [113, 229) \times [8, 34), s_1 = s_2 = 0.5, card() \in (10, 20)/(5, 10)/(15, 20), avg(\sqrt{rowv^2 + colv^2}) \in (95, 96)/(100, 101)/(181, 182)$, where ra, dec (from the relational SDSS) are dimensions, / divides high, medium and low spread respectively and $rowv, colv$ are velocity attributes.

Each of the data sets takes approximately 35GB of space, as reported by the DBMS. PostgreSQL's shared buffer size was set at 2GB. Computing an objective function for a window is transformed into a SQL prepared statement call. The statement is basically a range query, defining the window, with a `GROUP BY` clause to compute individual cells. For the efficient execution of range queries, we created a GiST [25] index for each data set [3] The size of each index is approximately 1GB. Each

[3] In PostgreSQL, GiST indexes are used instead of R-trees, which would be a logical choice for SW queries.

query results in a bitmap index scan, reading the data pages determined during the scan and an aggregation to compute the objective function. PostgreSQL performed all aggregates in memory, without spilling to disk.

Data Placement Alternatives As described in Section 3.4.3, the placement of data on disk has a profound impact on the performance of the algorithm. In the experimental evaluation we considered three options:

- Ordering tuples by one of the coordinates (e.g., order by the x-axis). In this case windows generally contain tuples heavily dispersed around the data file. We denote this option as "Synth/SDSS-*axis*" in text (e.g., "SDSS-ra").
- Clustering the tuples according to the GiST index. This reduces the dispersion of tuples, but since R-trees do not guarantee efficient ordering, there still might be considerable overhead for each range query. This option is denoted with suffix "-ind".
- Clustering the tuples by their coordinates in the data file. One common way to do this is to use a space-filling curve. Since we used a Hilbert curve, we denoted this option with suffix "-H". Another way is to cluster together tuples from the same part of the search area. For example, tuples from each of the eight synthetic clusters are placed together on disk, but no locality is enforced between the clusters. This is a convenient option in case more data is added and the search area is extended, since it does not destroy the Hilbert ordering. We define this option with suffix "-clust".

Stratified Sampling We used a stratified sampling approach to estimate utilities. Assuming the total number of cells is m and the total sample budget is n tuples, each cell is sampled with $t = \frac{n}{m}$ tuples. If a cell contains fewer tuples than t, all its tuples are included in the sample and the remaining cell budget is distributed among other cells. The idea is similar to the idea of *fundamental regions* [14] and congressional sampling [5]. A cell is a logical choice for a fundamental region. Each cell is independently sampled with SRS (Simple Random Sampling). Since cells effectively have different sampling ratios, we store the ratio with each sampled tuple to correctly estimate the required values, which is the common way to do this. All experiments reported were performed using a 1% stratified sample.

3.6.1 Query Completion Times

In this experiment we studied the effect of the data placement and prefetching on the query completion time. Table 3.1 provides the results for the high spread query. Other queries exhibited the same trend. α denotes the aggressiveness of prefetching, where "No pref" means no prefetching.

To establish a baseline for the comparison, we ran a corresponding SQL query (as described in Section 3.3) in PostgreSQL and measured the total and I/O (disk) time. Due to the nature of the SQL query, PostgreSQL did a single read of the data file, and then aggregated and processed all windows in memory. For the synthetic data set, the query resulted in 1,457.84s total and 677.94s

Table 3.1: Query completion times for different aggressiveness values (in seconds)

Dataset	**No pref**	$\alpha = 0.5$	$\alpha = 1.0$	$\alpha = 2.0$
Synth-x	28,206.84	13,521.55	8,602.45	6,957.33
Synth-clust	1,123.12	859.08	886.01	817.59
SDSS-dec	26,725.05	4,542.17	3,145.15	2,109.76
SDSS-clust	1,510.59	1,145.37	1,130	1,158.29

I/O time. For SDSS the query resulted in 3,589.93s total and 849.70s I/O time. The difference between the synthetic and SDSS times was due to small differences in the size of the data sets and the parameters of the queries (SDSS selected more windows, which resulted in more CPU overhead for PostgreSQL).

As Table 3.1 shows, in case when data is physically clustered on disk (-clust), the framework is able to outperform PostgreSQL even without using prefetching. This is due to a very small CPU overhead for the framework. When using prefetching, the difference becomes even more pronounced. In the "SDSS-clust" case, using $\alpha = 2.0$ resulted in 30% less completion time. It is important to mention that the framework starts outputting results from the beginning, while the SQL approach outputs all results only at the end.

In case the data is physically dispersed on disk (-x and -dec ordering), prefetching allowed us to reduce the completion time significantly, i.e., by an order of magnitude. In the case of SDSS, the framework eventually started outperforming PostgreSQL, while for the synthetic data a considerable overhead remained. In general, the performance improvement depends on the properties of the data set, such as the degree of skew or the number of clusters. Despite the remaining overhead for the synthetic data, we believe the framework remains very useful even in such cases, since it starts outputting results quickly.

3.6.2 Online Performance

This experiment studies the effect of prefetching on online performance (i.e., delays with which results are output). As the previous experiment showed, increasing the prefetching size reduces the query completion time. At the same time, it should increase delays for results since the amount of data read with each window increases. To present the experiment more clearly, instead of showing individual result times, we show times to deliver a portion of the entire result (a percentage of the total number of answers). The time to find all the answers (i.e., 100% in the figures) and the completion time of the query generally differ. The former might be smaller, since even when all answers are found, the search process has to read the remaining data to confirm this. This means users can be sure the result is final only when the query finishes completely. We provide the PostgreSQL baseline, explained in the previous section, as a dotted line in the figures.

Figure 3.6 shows the online performance of all three queries with different spreads for the synthetic data set (sorted by the axis x). All queries behaved approximately the same, since at every step the algorithm considers windows from the whole search space. The differences were due to the physical

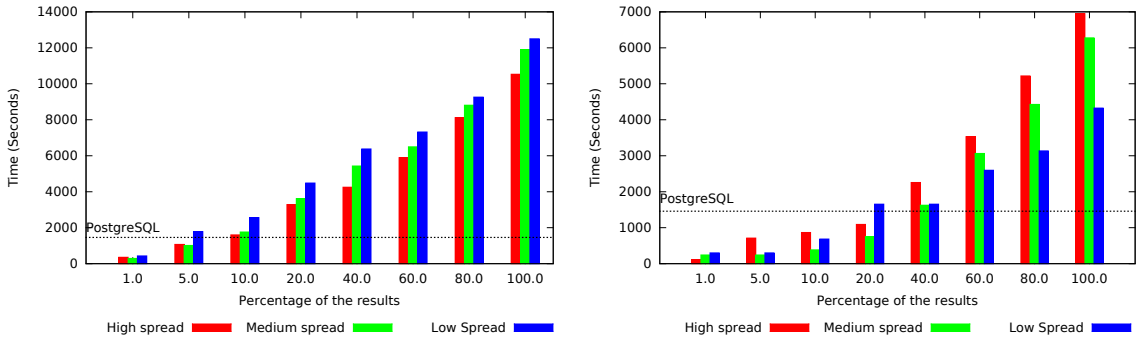

Figure 3.6: Online performance of the synthetic queries (Synth-x, left: $\alpha = 0.5$, right: $\alpha = 2.0$)

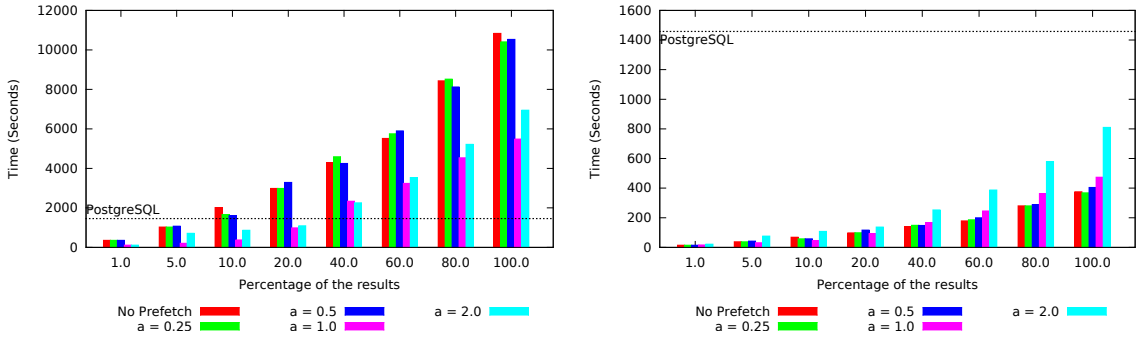

Figure 3.7: Online performance of the high-spread synthetic query (left: Synth-x, right: Synth-clust)

placement of data. For the case of $\alpha = 2.0$ the final result was found faster for the low spread query. Since in this case clusters were situated close to each other, prefetching large amounts of data around one cluster allowed the algorithm to "touch" another, nearby, cluster as well. Other orderings of the data set resulted in the same behavior.

Figure 3.7 shows the online performance of the high-spread query for the "Synth-x" and "Synth-clust". For "Synth-x" larger aggressiveness values resulted in much better online performance during the whole execution, although $\alpha = 2.0$ created longer delays at some points. The situation changes with the beneficial clustered ordering. While values up to $\alpha = 1.0$ behaved approximately the same, $\alpha = 2.0$ created much longer delays. This exposes a trade-off between the query completion time savings coming from prefetching data and the delays for online results. Figure 3.8, which shows the results for the high spread SDSS query, demonstrates the same trend. If the user is not aware of the ordering, $\alpha = 1.0$ might be considered a "safe" value on average, which both provides considerable savings and does not cause large initial delays. For advanced usage, the aggressiveness should be made available to users to control. If the user is satisfied with the current online results, she can increase the value to finish the query faster. If the ordering is not beneficial (e.g., axis-based), the value should not be set to less than $\alpha = 1.0$.

Other queries exhibited the same trend and are not shown. We do not show the results similar

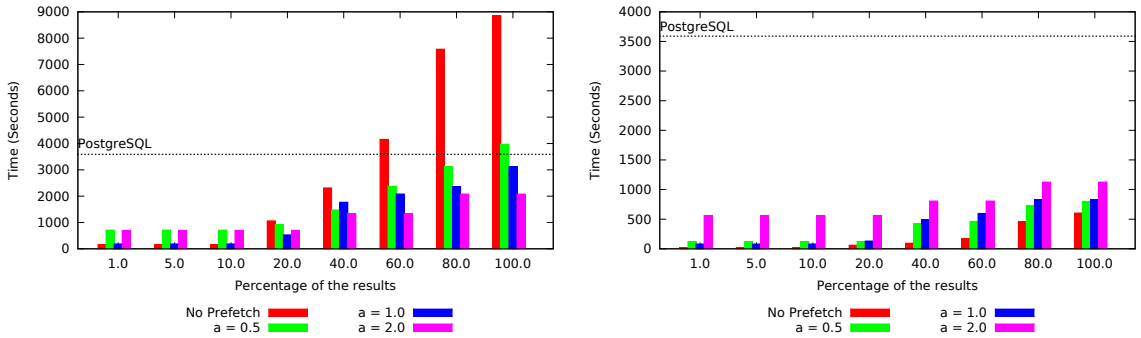

Figure 3.8: Online performance of the high-spread SDSS query (left: SDSS-dec, right: SDSS-clust)

to Figure 3.6 for SDSS. Since different spread queries for SDSS differ in the number of results and query parameters, the algorithm behaved different each time (e.g., considered different candidate windows and used different sizes for prefetching), which made the direct comparison of different queries meaningless.

We should mention that first results are output in 10 seconds in most cases and in 80 seconds in worst cases. This fulfills one of the most important goals: start delivering online results fast.

3.6.3 Physical Ordering of Data

Here we studied the effect of the physical data placement, described in Section 3.4.3, in more detail. Table 3.2 presents statistics of data file reads for three different ordering options described at the beginning of Section 3.6. We ran a single query (one per data set) and used `systemtap` probes in PostgreSQL to collect the statistics. In the table, "Total" refers to the total time of reading all blocks, without considering the time to process the tuples.

Table 3.2: Disk statistics for the synthetic dataset

Data set	**Total (s)**	**Mean/Dev read (ms)**	**Reads (blks)**	**Re-reads (blks)**
Synth-x	24,987	2.4/2.5	10,476,601	6,477,523
Synth-ind	3,053	0.7/1.7	4,217,096	218,018
Synth-clust	738	0.2/0.8	4,001,263	2,185
Synth-H	747	0.2/0.8	4,000,592	1,514

When the physical ordering did not work well with range queries (i.e., -x), the DBMS effectively had to read the same data file more than twice (see the "Re-reads" column), which supports the thrashing claim made in Section 3.4.3. Moreover, since each range query resulted in multiple dispersed reads, the time of a single read grew considerably and became more unpredictable (see the "Mean/Dev" column), because of seeks. SDSS showed the same results.

3.6.4 Distributed Processing

We now present experimental results for the same queries over distributed data. When a search area is partitioned among the workers, there is flexibility in partitioning the data itself, which we studied along three cases. In the first one, the data partitioning corresponds to area partitioning, which requires some workers to perform remote data requests. We call this *no-overlap*. Another case, *full-overlap*, involves creating overlapping partitions so that workers have all data available locally and no remote requests are made. This case is possible only if shape-based conditions (e.g., the cardinality of windows) are known in advance or if the data is fully replicated. Otherwise, the overlap will be partial and workers will be making remote requests, albeit a smaller number of them. We call this third case *part-overlap*.

Table 3.3: Results for the distributed synthetic high-spread query (Synth-clust, $\alpha = 1.0$)

Nodes, Overlap	**First result**	**All results**	**Total time**
1 node, no	6	820	1820
2 nodes, no	6	470	1050
4 nodes, no	5	360	580
8 nodes, no	7	200	350
1 node, full	6	820	1820
2 nodes, full	6	490	1250
4 nodes, full	5	350	790
8 nodes, full	7	255	650
8 nodes, part	7	300	540

Table 3.3 presents the results for the synthetic high-spread query. All times are in seconds and rounded to the nearest integer. Since the I/O performance of Amazon EBS may vary, the number of runs was at least 10. For the no-overlap case, the reason behind the sub-linear scalability is primarily the skew, since the distribution of results is uneven, especially for the case of 4 and 8 nodes, which may likely be common in practice. Moreover, the pattern of reads heavily depends on a worker's area, which creates another difference in result times. As for the total time, we tried to make the partitioning as even as possible, so that each worker has approximately the same amount of data. However, since partitions must be aligned with cells, this is not completely possible. Total disk times differed for up to 100 seconds between workers. Supporting the claim of Section 3.5, the total time was dominated by the slowest worker's total disk time. The additional overhead was kept under 10 seconds.

The full-overlap case performed worse than the no-overlap case in general. For the total times, this can be explained by the additional disk reads overhead. The overlapping parts were read more than once independently by workers that needed them, so some workers had more data to read. In the no-overlap case, such data was read on a single node and served to other nodes from cache. Result times did not become better, since the number of resulting windows spanning the overlap was small. Moreover, these windows had to be read from disk, which introduced delays. One could argue that reading all data locally might reduce delays when the number of resulting windows spanning

the overlap is large. However, there are times when such promising data would be read by other workers sooner and served faster during remote requests. Overall, the full-overlap case does not consistently offer improvements over the no-overlap case.

For the part-overlap case, we present only the result for 8 nodes, since it is consistent with the same trend. The total time is between the times for other cases, which was expected. The time to get all results was worse, due to a different read pattern: the required remote data came in the middle of a large local read and the corresponding windows were explored later. The prefetching pattern was different as well.

Since the total completion time is determined by the total disk time, we performed another experiment where we varied the amount of data read by each worker. With increasing the size skew the total time grew to 390 and 455 seconds for the no-overlap case of 8 nodes. This shows the need to carefully balance workers, which is common for any distributed algorithm. This can be done before running the queries or by automatically assigning sub-areas to workers while estimating their data sizes from the sample.

Chapter 4

Searchlight: Integrated Search and Exploration

As we discussed in Chapter 3, introducing advanced search methods into traditional query processing significantly improves performance and usability of large scale data exploration. While Semantic Windows framework showed the applicability of this approach, at the same time it indicated the need in a more general solution, which would:

- Allow users to execute a variety of ad-hoc queries, rather than using a predefined query set. Users should be able to define and customize their queries based on the exploration problem at hand, using existing constraints or adding new constraint types if required.
- Severely decrease the amount of data that needs to be read for answering the query. It should perform extensive pruning not only for the search space, but for the data as well.
- Control query execution dynamically by balancing search space, data and the available computation resources between nodes participating in the query to provide better interactivity and query completion times.

In this chapter we present a novel framework called *Searchlight* [28], which addresses these requirements. Searchlight is a marriage between Constraint Programming (CP) and data management techniques, which concurrently offers the rich expressiveness and efficiency of constraint-based search and optimization provided by modern CP solvers, and the ability of DBMSs to store and query data at scale. This results in an enriched functionality that can effectively support both data- *and* search-intensive applications. As such, Searchlight is the first system to support *generic* search, exploration and mining over large multi-dimensional data collections, going beyond point algorithms designed for point search and mining tasks.

4.1 Overview of Searchlight

First, let us revisit typical search queries types discussed in Section 1.1. While DBMSs struggle with queries such as Q1-Q3, these can be compactly expressed and efficiently answered by a CP approach [45]. In CP, users first specify *decision variables* over some domains, defining the search space. They then specify constraints between the variables, e.g., algebraic ones, "all variables must have different values", etc. Finally, a CP *solver* finds *solutions*, which are the variable assignments satisfying the constraints. For Q1, the decision variables would simply define the sky region (i.e., the corners of a rectangle, or the center and radius of a sphere) with domains restricting the regions to a particular sky sector and constraints specifying the properties. To express high-order queries such as Q2, more variables and constraints can be introduced in a straightforward manner. As for optimization queries, such as Q3, CP solvers natively support min/max searches with objective functions.

Several salient features make CP solvers particularly well suited for generic search and exploration:

- **Efficiency**: CP solvers are very proficient at exploring large search spaces. Similar to query optimizers, solvers also incorporate a collection of sophisticated optimizations including pruning, symmetry breaking, etc. The first is a core technique that safely and quickly remove from consideration parts of the search space that cannot contain any results.

- **Interactivity**: CP solvers are designed to identify individual solutions fast and incrementally. User can pause or stop the search or ask for more solutions at any time. This is fundamentally different from conventional query optimization that aims to minimize the total completion time of queries.

- **Extensibility**: CP solvers are designed in a modular and extensible manner; they can be extended to include new constraints types and functions. Moreover, users can easily introduce their own *search heuristics*, tuning the search process.

CP solvers are commonly used for NP-hard search problems as they go beyond straightforward enumeration and can effectively navigate large search spaces through a variety of effective pruning techniques and search heuristics that leverage the structure of the search space. Note also that expressing the original search problem via CP does not increase the original problem's complexity. Take as an example the problem of finding a fixed-size sub-array satisfying some constraints. The number of possible sub-arrays is polynomial on the size of the array. Exhaustive search would produce a polynomial, albeit a very inefficient, algorithm. A standard CP solver is going to construct and explore a search tree, where leaves correspond to possible sub-arrays. Thus, the complexity remains the same, but the search space can be explored in a much more efficient way.

Searchlight supports *data- and search-intensive applications* at large scale by integrating CP and DBMS functionality to operate on multidimensional data. Its design is guided by the following goals:

- **Integrated search and query**: Searchlight offers both standard DBMS functionality (e.g., storage and query processing) and sophisticated search. This integrated functionality simplifies application development and reduces data movement between the systems.
- **Modularity and extensibility**: Our design is minimally invasive and work with different solver implementations without any or very little modification. Similarly, the underlying DBMS engine will not require major modifications. This goal allows Searchlight to gracefully and automatically evolve as the underlying systems evolve. This also allows the same framework to be applied to compatible systems.
- **Optimized data-intensive search and exploration**: Searchlight provides optimized execution strategies for CP logic, integrated with regular queries, on large data sets.
- **Distributed, scalable operation on modern hardware**: Searchlight supports efficient distributed and parallel operation on modern clusters of multi-core machines.

Users can invoke existing solvers along with their built-in search heuristics using an array DBMS language. Under the hood, Searchlight seamlessly connects the solver logic with query execution and optimization, allowing the former to enjoy the benefits of DBMS features such as efficient data storage, retrieval and buffering, as well as access to DBMS query operators. Searchlight runs as a distributed system, in which multiple solvers will work in parallel on different parts of the data- and search-space.

An important challenge is to enable existing CP solver logic (without any modifications) to work efficiently on large data. To achieve this goal, Searchlight uses a two-stage *Solve-Validate* approach. At the solving stage, Solvers perform speculative search on main-memory synopsis structures, instead of the real data. A synopsis is a condensed representation of the data, containing information needed to perform pruning and to verify query constraints. The results are guaranteed to contain all the real results, but possibly include false positives. At validation, Validators efficiently check the candidates over real data, eliminating the false positives and producing the final solutions while optimizing I/O. Solvers and Validators invoke different instances of the same unmodified CP solver logic; yet the former will be directed to work on the synopses and the latter on the real data through an internal API that encapsulates all array accesses. This two-stage approach is transparent to the CP solvers and the users.

We present an implementation of Searchlight as a fusion between two open-source software, Google's Or-Tools [1], a suite of operations research tools that contains a powerful CP solver, and SciDB, a multidimensional array DBMS. Our experimental results quantify the remarkable potential of Searchlight for data- and search-intensive queries, for which Searchlight often performs orders of magnitude faster than the next best solution (SciDB-only or CP-solver-only) in terms of end response time and time to first result.

4.2 DBMS-Integrated Search

In Searchlight users pose search queries in form of constraint programs, which reference data in a DBMS. While conceptually Searchlight is not restricted to a particular data model, in this paper we target array data. An array consists of elements with *dimensions* serving as array indexes. Each element has its own tuple of properties, called *attributes*. For example, for an astronomy data set, right ascension and declination might serve as dimensions, while magnitudes or velocities as attributes.

4.2.1 CP Background and Or-Tools

Let us revisit the astronomy example from Section 1.1. Assume the user decides to search for all rectangular regions in the sky sector $[0, 10]^\circ \times [20, 50]^\circ$, where coordinates are defined using right ascension and declination. The regions must be of size $[2, 5]^\circ$ by $[3, 10]^\circ$ and have the average r-magnitude of objects contained within less than 12.

To form a constraint program the user first defines *decision variables* over *domains*, which correspond to objects of interest, the regions. For this example it is sufficient to define four variables $x \in [0, 10], y \in [20, 50], lx \in [2, 5], ly \in [3, 10]$. x, y describe the leftmost corner of the region and lx, ly — the lengths of the sides. CP has limited support for non-integer domains, so in this example the "resolution" of search is 1°. If higher precision is required, real values can be converted to integers, e.g., by multiplying on 1,000.

The next step is to define constraints. The size constraint is expressed via domains of lx and ly. There are two left:

- A region must fit into the sector: $x + lx - 1 \leq 10$ and $y + ly - 1 \leq 50$.
- The r-magnitude constraint: $avg(x, y, lx, ly, r) < 12$. $avg()$ computes the average value of attribute r over the sub-array (x, y, lx, ly). We assume it is readily available to use in constraints and elaborate on this in the next section.

Decision variables and constraints constitute the *model* for the problem. A common way to obtain solutions in CP is to perform *backtracking search*. Other methods exist, e.g., local search, but they might not guarantee the exact result. A typical backtracking solver organizes the search as a binary tree. At every non-leaf node at least one decision variable is *unbound* (i.e., its current domain contains multiple values). At every such a node, the solver makes a *decision* consisting of two branches. The decision depends on the *search heuristic* the solver is using, which can be specified by the user. A search heuristic typically first chooses an unbound variable. The choice might be based on the size of its domain, the minimum/maximum value of the domain or just randomly. After the variable is selected, the two branches of the decision correspond to two opposite domain modifications. For example, the left branch might assign the value v to the variable x, in which case the right branch would remove v from the domain of x. Another common decision is to split the domain: $x \in [0, 9] \rightarrow x \in [0, 4] \vee x \in [5, 9]$. Then the solver chooses a branch, and repeats the

process until a leaf of the tree is reached. At a leaf all variables are bound, and the solver can report them to the user as a solution.

After the solver walks into a branch, and the variable domain changes, constraints get notified of the change, and can check for violations. Additionally, some constraints might be able to reason about the domains and modify them further. For example, if $x = 9$, lx becomes 2, because of the $x + lx - 1 \leq 10$ constraint. Thus, the constraint sets $lx = 2$, and the process is repeated until no further changes to the domains can be made, which is called *local consistency*. The process itself is called *constraint propagation*.

If during the propagation a constraint cannot be satisfied (e.g., $avg() > 14$ for $x = 9, lx = 2$), the search *fails* at the current state, since no solutions are possible in the sub-tree rooted to the state. The solver *prunes* the sub-tree and backtracks, rolling back all changes, until an ancestor with an unexplored branch is found. If such an ancestor exists, the solver goes into the branch. Otherwise, the search ends, since no alternatives are possible. If the solver does not fail when it reaches a leaf, the corresponding assignment is a solution, since all constraints are satisfied. If the user wants to continue searching for more solutions or the problem is an optimization one (e.g., $max(avg(x, y, lx, ly, r))$), the solver backtracks from the leaf, and explores the rest of the search space. More thorough description of CP solver details can be found in the related books [45].

Searchlight uses the backtracking CP solver from Google's Or-Tools [1], which follows the execution model described above. Users can add new constraints, functions, define their own search heuristics, including performing nested searches at any node of the main search tree. They can also define monitors to track the search progress, limit the time of the execution or perform modifications (e.g., search restarts, switching branches, etc.). This makes Or-Tools highly customizable, and Searchlight extensively uses these features to efficiently merge the solver with the DBMS.

4.2.2 Search Queries

Searchlight essentially supports any type of queries a typical CP solver can process. However, to use additional Searchlight features, e.g., working with array data, users have to employ User-Defined Functions (UDFs).

UDFs allow users to define additional concepts to use in constraints, where built-in functions are not enough. Revisiting the astronomy example, in the constraint $avg() < 12$, $avg()$ must be defined as a UDF, since it requires the array data. UDFs are essentially treated as black boxes by the solver [1]. It periodically calls them to obtain their values at the current search state. In general, since variables might be unbound, it is not possible to return a scalar value. So a UDF usually returns an interval $[m, M]$ containing all possible values of the function for all possible assignments of the variables from the current domains. Such an interval is enough to check constraints and perform pruning at internal nodes. A scalar value can be represented as $m = M$. Usually, tighter intervals mean more efficient pruning. For the example constraint, $avg() < 12$, the interval $13 \leq avg() \leq 17$ would allow the solver to prune the current sub-tree, while $11 \leq avg() \leq 17$ would not.

[1] In Or-Tools UDFs can be written in C++, which gives users a large degree of freedom.

For performance, array operations are restricted to the Searchlight API, which at present consists of two calls:

- $elem(X, a)$, which returns the value of attribute a at coordinates $X = (x_1, \ldots, x_n)$.
- $agg(X_1, X_2, a, op)$, which computes the aggregate *op* over attribute a for the sub-array bounded by X_1 and X_2. *op* can be any of the common aggregates (i.e., min, sum, etc), and it is easy to add support for others. Users can specify multiple *op*s in a single call.

These calls are useful for array exploration, since individual elements and sub-arrays are natural entities of interest. The API is not fixed and can be easily extended for future applications. It is meant to provide building blocks for constraints, and a UDF can contain any number of API calls. For example, a user might want to compute the average value of some attribute for a sub-array and its neighborhood and compare the two to detect anomalies. This can be implemented with several API calls in a single UDF, which would return the difference between the averages. Searchlight provides built-in functions for common aggregates, and the library of functions can be easily extended.

Let us describe briefly how the UDF $A = avg(x, y, lx, ly, r)$ could be implemented. A must return interval $[m, M]$ containing all possible values of r for every sub-array $[x', x'+lx'-1] \times [y', y'+ly'-1]$, where x', y', lx', ly' are values from the current domains. When the number of possible sub-arrays (i.e., the product of the cardinalities of the domains) is greater than a threshold, then m and M are equal to the minimum and maximum values of r in the minimum bounding sub-array. This requires a single $agg()$ call. If the number of the sub-arrays is less than the threshold, the average for each of them can be computed with an $agg()$ call. m and M then will be equal to the minimum and maximum values just computed. For a single sub-array (i.e., in a leaf of the search tree), $m = M$.

4.2.3 Search Heuristics

After the user poses a query, the search process is guided by a search heuristic. Searchlight can run native Or-Tools search heuristics (e.g., random, impact-based, split, etc.) without any modifications. It can also be further extended by defining new ones. This allows users to customize search for particular problems, which is very common in CP. New search heuristics are defined in the same way as in plain Or-Tools. Users can also use API calls to access array data, e.g., for estimating the impact of the heuristic decisions.

Let us show an example of a custom search heuristic, which also allows us to illustrate the concepts introduced in the previous section. This heuristic is part of Searchlight, and we found it useful for Semantic Windows [27] queries. An example search tree that might have been produced for the astronomy example is shown in Figure 4.1. The heuristic is *utility-based*. First, it divides the search space into non-overlapping areas by splitting each variable's domain into several intervals (the number of intervals is a parameter). An area is a Cartesian product of intervals, one per variable. Then, the heuristic takes a number of random variable assignments (*probes*), from each interval. A

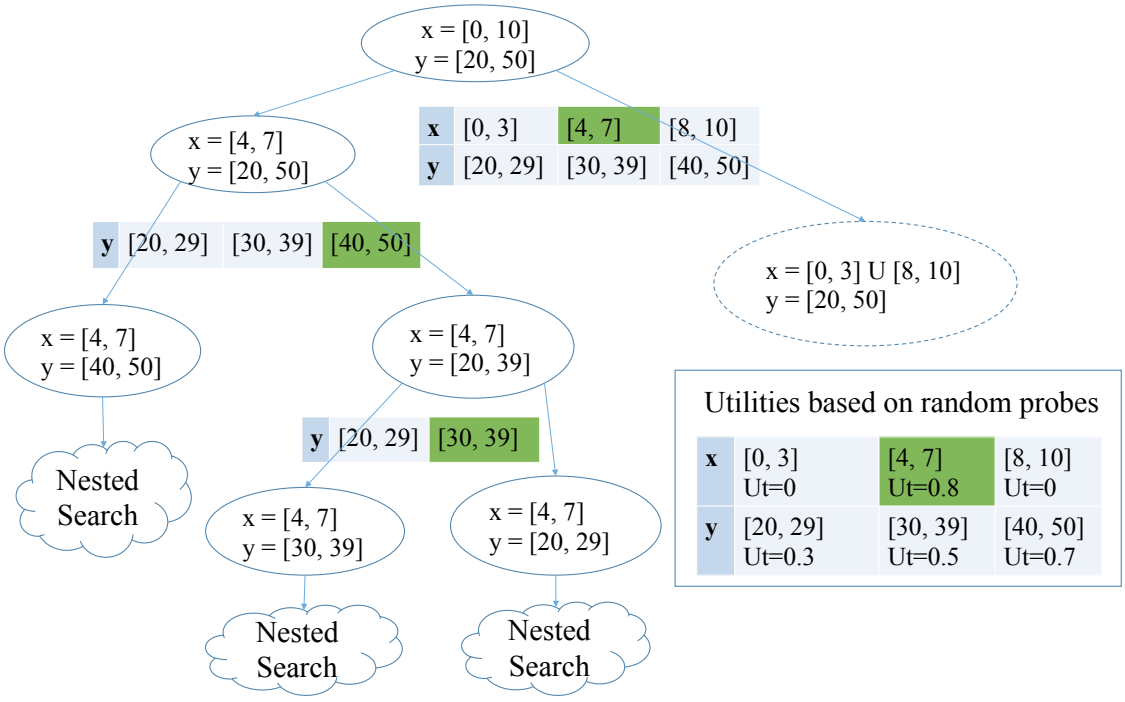

Figure 4.1: Search tree of a utility-based heuristic. Intervals chosen at each step are highlighted. Utilities are provided in the table as Ut=. The state within the dashed ellipse is pruned.

probe might be successful (i.e., a solution) or a failure (i.e., it violates constraints). The utility of an interval is the ratio of successful to total number of probes. The probing itself is done by using nested random search, which makes it very general and applicable to other types of problems. Even if probes cannot catch interesting areas (e.g., in case the search space is very large), the heuristic still can prune some areas immediately when they violate constraints.

For the astronomy example the areas are defined based on variables x, y, since they define the location of a region. After the probing is completed, at every step the heuristic chooses the variable and interval with the maximum utility. When domains of x and y become intervals (i.e., the current state is an area), a nested search of the area is started with some heuristic. We found random and split heuristics to be quite efficient, however users might specify others. Figure 4.1 also shows an example of pruning. When the solver explores the right branch of the root, it detects a violation of a constraint and prunes the entire sub-tree.

4.3 Searchlight Query Processing

The naive way to process Searchlight queries would be to run a traditional CP solver without any changes and transform Searchlight API calls into DBMS queries. This, however, results in a very poor interactive and total performance. The solver can call UDFs many times during the search, since

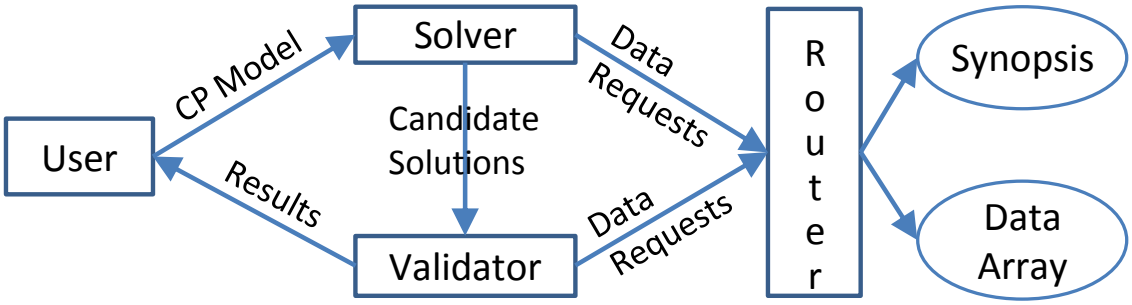

Figure 4.2: Two-level search query processing.

it assumes them to be relatively cheap. More importantly, pruning cannot work without requesting data, since UDF values are required for provable pruning. In practice, the naive approach results in reading the same data multiple times by arbitrarily ordered data requests, which causes DBMS memory buffer thrashing. This is supported by our experimental evaluation, presented in Section 4.5.1.

Searchlight uses two-level query processing instead, which combines *speculative execution* over a *synopsis* of the array with *validating* the results. We designed this architecture with the following goals in mind:

- Interactivity. Searchlight should start outputting results quickly and minimize delays between results. Some overhead is impossible to avoid, e.g., computing an aggregate for a large sub-array might be expensive. However, the time to *discover* the next result should be minimized. This is the main task of speculative execution.
- Total performance. Parts of the data not containing any results should be eliminated efficiently with few data accesses. Searchlight uses extensive pruning at the speculative level to achieve this.
- Expressiveness. Users should be able to use tools available in a plain CP solver, e.g., constraints, heuristics, etc. We did not modify the solver's engine directly, but rather used the customization features available in Or-Tools to merge the solver with the DBMS.

4.3.1 Two-level Query Processing

The two-level query processing is illustrated in Figure 4.2. When the user submits a search query in form of a CP model, Searchlight starts a CP solver for processing it. As we discussed in Section 4.2.2, a search query that reads DBMS data makes Searchlight API calls. When such a call is made, it is processed by the *Router*, which sends it to a synopsis. Synopsis is lossy compression of data, which is able to give approximate answers to API calls. For each call it returns an interval guaranteed to contain the exact answer. For example, $elem(X, a)$ might return $[5, 10]$, while the real value is 7.

Since in general UDFs return intervals as well, it does not create complications for users. In most cases manipulating intervals is not harder than manipulating scalar values.

While the Solver processes the model by using the synopsis instead of the original data, it produces *candidate solutions* (*candidates*). A candidate is a CP solution (i.e., a complete variable assignment), and does not violate any constraints. However, since candidates are synopsis-based, they might contain false positives. Since synopsis intervals are guaranteed to contain exact answers, candidates are guaranteed to include all real solutions.

When the Solver produces a candidate, it sends the candidate to the *Validator*, which checks it over the original data. At the beginning, the Validator clones the initial model from the Solver and starts its own CP solver, which, however, runs a different search heuristic. When a new candidate is received by the Validator, the heuristic assigns the values from the candidate to the variables. However, this time all API calls are answered using the original DBMS data, which results in proper validation of query constraints. Thus, if the Validator's solver fails, the candidate is a false positive.

We made the Validator CP-based for the sake of generality. Since it executes the same model as the Solver, it does not assume anything about the nature of constraints. New constraints, UDFs and API calls can easily be added without modifying the Validator. Moreover, the users do not have to be aware about the two-level processing at all.

Since the Validator uses its own CP solver, it works in parallel with the main Solver. Potentially expensive validations do not hamper the search and are made concurrently, which greatly improves interactive performance.

The Router directs API calls to the appropriate data depending on the context. Users write API calls using the Router, and Searchlight takes care of the details internally.

4.3.2 Synopsis

Synopsis is lossy compression of data, which gives approximate answers to the Searchlight API calls. We assume it fits into memory, so the Solver, which uses it during the search, can execute its model efficiently. It might partially reside on disk, especially during distributed processing, where the Solvers operate on parts of the search space.

Synopsis is more of a concept than a particular structure. Different API calls might require different synopsis types, which we discuss later in this section. Searchlight can use multiple synopses in a single query, if required by constraints (API calls).

Synopsis for Aggregate Estimations

An example of the synopsis is presented in Figure 4.3(a). The original 4x4 array is divided into four *synopsis cells* of size 2x2. For each cell we keep information needed to answer API calls. For aggregates these are min/max, the total sum and count of all elements. Cells might store other information, e.g., distribution of a cell, bitmaps for very sparse cells, etc. The synopsis introduces lossy compression. For example, the top-right cell in the figure could be produced by sub-arrays $(5, 3, 3, 2)$ and $(5, 4, 2, 2)$. We will call such sub-arrays *cell distributions*.

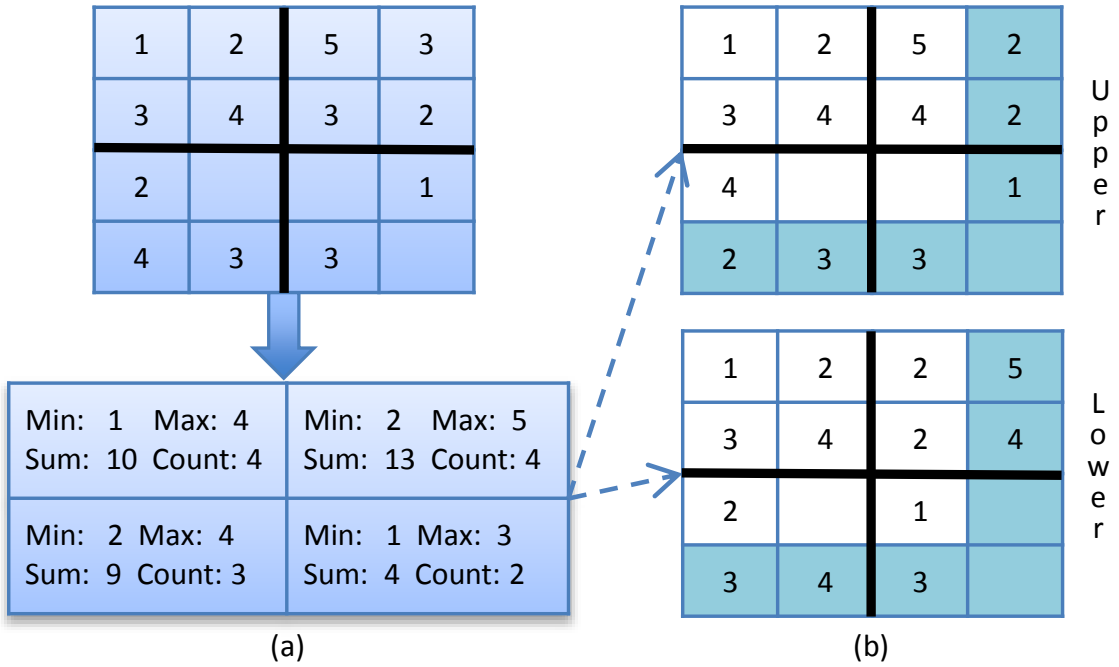

Figure 4.3: Synopsis example. (a) 2x2 synopsis grid. (b) Upper and lower bound arrays for the $avg()$ of the highlighted region.

Synopsis provides answers to Searchlight API calls in form of intervals $[m, M]$, guaranteed to contain the exact value. Answering $elem()$ calls is easy: m and M are just the minimum and maximum values stored in the corresponding cell. The $agg()$ call requires more work.

Let us assume the user called $avg(R)$, where R is a highlighted region from Figure 4.3(b). R intersects 3 out of 4 synopsis cells partially. Since cells might correspond to different distributions, this implies multiple possible values of $avg(R)$. The main idea is to find the cell distributions that reach the lower bound m and the upper bound M of the interval, which is illustrated in the part (b) of the figure. It is easy to verify that both arrays might have produced the synopsis. Their $avg(R)$ values are 2.125 and 3.286, so the API call will return the interval $[2.125, 3.286]$. For the original array, $avg(R) = 2.857$.

Estimating upper and lower bounds for aggregates over a synopsis is similar to aggregate estimations for Multi-Resolution Aggregate (MRA) tree structures [32]. The algorithm essentially follows the same path. For example, to compute the interval for $avg()$, the upper (lower) bound array must contain as many large (small) elements as possible to bring the average up (down), without violating the synopsis information. The possible elements from all cells are considered in the decreasing (increasing) order and added one by one, until the average cannot go up (down).

The detailed algorithm with proofs can be found in the MRA paper [32]. The difference for arrays is due to intersections of the query region and cells (MRA tree nodes). For an MRA tree (e.g., an annotated R-tree), each intersection might contain any number of database objects referenced by the tree node, while for arrays the minimum/maximum number of elements for each intersection is

known. Thus, each cell has a budget of minimum and maximum number of elements it must provide for an estimation, and we account for this budget when selecting elements during the computation as described above.

Synopsis Layers

Synopsis array described above has a parameter — the size of the cell (e.g., 2x2), which we call its *resolution*. There is a trade-off when choosing the correct resolution for a query. Estimating aggregates is computationally more efficient with coarser synopses (i.e., with larger cells) due to smaller numbers of cells participating in estimations. At the same time, they might provide poor quality estimations, especially for small regions. Finer synopses, on the other hand, provide better quality estimations, albeit at higher cost. Synopses for a single array can be thought of as a hierarchy of layers, low to high resolution.

Given a synopsis hierarchy for an array, Searchlight starts from the lowest-resolution layer and proceeds as follows:

1. The query region is divided into disjoint pieces: intersections with the synopsis cells. For example, in Figure 4.3 there are four pieces.

2. Each piece covered by the corresponding cell more than 75% (a parameter) is estimated from this cell. The coverage is defined as the area of the piece divided by the area of the cell.

3. The remaining pieces are left for the next layer, finer, synopsis. In Figure 4.3, the bottom-right piece is covered 25% and the top-right one 50%. If there are no more layers, the pieces are estimated from the current one.

The basic idea behind the algorithm is to cover each region with the synopsis such that to avoid small region-cell intersections, which taint the estimation quality, and at the same time cover large parts of the region with a small number of cells, which improves performance. To further speed-up computation the algorithm employs the following heuristic: if the region is covered by the cells of the current synopsis layer more than 75% (a parameter), the algorithm stops at this layer, ignoring individual cell coverage. This heuristic allows us to avoid cases where large regions have a small number of poorly covered pieces. In Figure 4.3, the region is covered $\frac{9}{16} \times 100\% = 56\%$. So the algorithm will use the next synopsis layer, if available.

Synopsis layers are also considered when validating candidates. If one of the layers is aligned with the UDF's query region, that synopsis is used to compute the UDF instead of the original data array. To be aligned with a synopsis, the region must intersect all its cells completely, which guarantees the exact answer. This optimization severely improves performance, since synopses are either fit in memory or, at least, are much more compact than data arrays.

Synopsis Alternatives

The choice of the synopsis as an aggregate grid was dictated by using arrays for the original data. There is no need in a tree (e.g., R-tree), since grid cell coordinates can be easily computed from the original array ones. Synopsis layers can be seen as a slice of the well-known pyramid structure [51]. In general, Searchlight does not use the complete pyramid due to memory restrictions. Several in-memory layers are used for estimations, and the lowest pyramid layer, the original array, is used for validations. In case of other DBMS types, other structures might be more suitable. For example, for an RDBMS, MRA-trees [32] would be a better choice.

We want again to emphasize the fact that the synopsis concept is not exclusive to aggregate structures. Other types of constraints might necessitate other types of structures. One example is sub-sequence similarity matching for time-series data [19, 20]. A common way to answer such queries is first to compute the Discrete Fourier Transform (DFT) for all sub-sequences of the time-series, take several components of each DFT as points and produce a *trace* [19] by combining the points from adjacent sub-sequences. A trace can be seen as a set of points in a multi-dimensional space. Instead of DFT another suitable transformation can be used, such as Discrete Wavelet Transformation (DWT) [43], Piecewise Aggregate Approximation (PAA) [30] or indexable Symbolic Aggregate approXimation (iSAX) [49]. If the traces become large, they can be further covered by a number of MBRs, which then can be indexed (e.g., by an R-tree).

Such an index obviously fits in the synopsis concept described above. Each MBR can be used to estimate the similarity distance between the query sequence and all time-series sub-sequences represented by the MBR, which is exactly what a CP solver needs. We have implemented the new synopsis type, based on the PAA transformation and MBA coverage, as part of Searchlight. To interact with the synopsis on the low level, we added API call $dist(x_l, x_r, Q)$, which computes the maximum Euclidean distance between the user's query sequence Q and any sub-sequence of size $|Q|$ lying within the $[x_l, x_r]$. We also introduced the user level function $dist(x, Q)$, which allows the user to specify the similarity constraint in the obvious way, e.g., $dist(x, Q) \leq 10$, where Q is the query sequence, as before, and x is a decision variable denoting the start of the resulting sub-sequence in the data.

We demonstrated the applicability of the approach using the waveform data from the MIMIC data set [29]. The performance for the similarity queries depends mostly on the quality of the index estimation, which Searchlight does not have control over. At the same time the search overhead is kept at minimum. It is important to mention that the overall implementation effort was quite small, and was heavily dominated by the index implementation. At the same time, the main search engine and architecture did not require any changes at all, which underlines great extensibility potential for Searchlight.

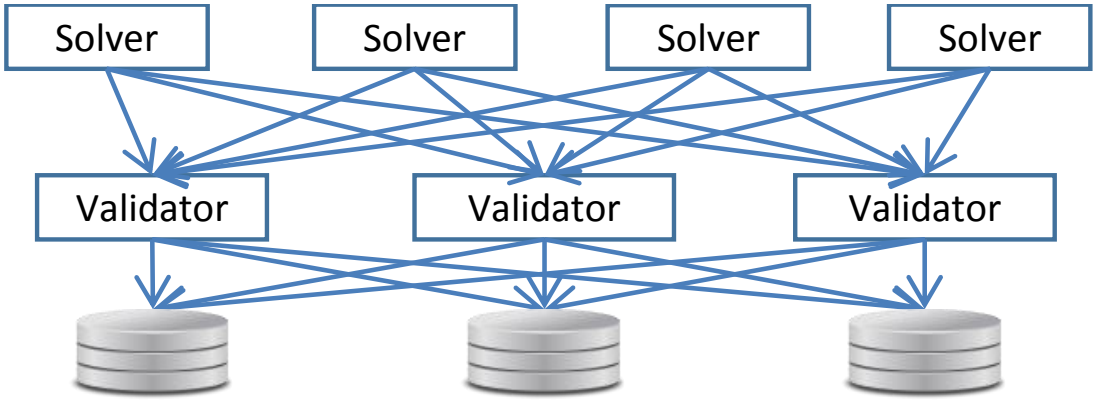

Figure 4.4: Distributed Searchlight with Solver and Validator layers.

4.4 Distributed Searchlight

Searchlight supports distribution on both levels of query processing. At the first level, the search space is distributed among Solvers in a cluster. At the second level, Validators are assigned to nodes responsible for different data partitions, and validate candidates sent to their corresponding nodes. There can be an arbitrary number of Solvers and Validators, and they do not have to reside on the same nodes. This gives users considerable freedom in managing cluster resources. Solvers can be put on CPU-optimized machines, while Validators can be moved closer to the data. Moreover, there might be multiple Solvers and Validators at each node at the same time, which explores multi-core parallelism. The architecture is illustrated in Figure 4.4.

4.4.1 Searchlight in SciDB

Searchlight uses SciDB [3, 10] as the array DBMS. A typical SciDB cluster consists of *instances*, and each array is distributed among all instances. During query execution one instance serves as a *coordinator*. Coordinator combines partial results from other, *worker*, instances and returns the final result to the user.

Each array is divided into *chunks*, possibly overlapping tiles of fixed size. SciDB computes a hash function over the leftmost corner of a chunk, which produces the instance number that will own the chunk. One of the important features of SciDB is attribute partitioning, which means different attributes are stored in different sets of chunks. This is similar to vertical partitioning in columnar RDBMSs.

In addition to built-in query operators, SciDB allows users to write their own (User-Defined Operators, UDOs). We implemented Searchlight as a UDO. Thus, Searchlight directly participates in query execution inside the DBMS engine and has access to the internal DBMS facilities. We use SciDB networking to pass all control and data messages. Validators use the temporary LRU cache to store chunks pulled from other instances. When computing API calls, Searchlight accesses the arrays in the buffer memory, without the need to serialize and transfer the data to a client. In summary,

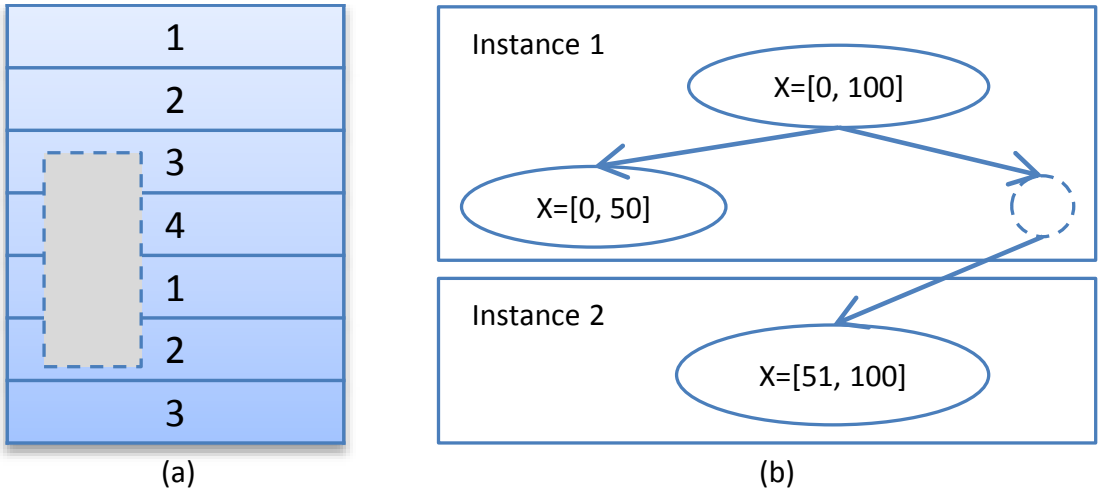

Figure 4.5: Search balancing. (a) Static round-robin balancing. The highlighted region is a hot-spot. (b) Dynamic balancing with transferring a sub-tree.

working inside the DBMS engine significantly decreases the overhead and avoids duplication of the same functionality.

4.4.2 Search Distribution and Balancing

Initially, the search space is distributed among the Solvers statically. Since the search space is a Cartesian product of the variable's domains, it can be represented as a hyper-rectangle. We slice this hyper-rectangle along one of its dimensions into even pieces and assign each slice to a Solver in the round-robin fashion. This is illustrated in Figure 4.5(a).

The static scheme targets search hot-spots, parts of the search space containing large numbers of candidates. A Solver might get "stuck" in a hot-spot, while others finish their parts quickly. This creates an imbalance, where some Solvers do most of the work, while others sit idly. The round-robin method distributes continuous hot-spots among multiple solvers, as shown in Figure 4.5(a).

The static partitioning depends on the total number of slices. Too few slices might result in a hot-spot not being covered by multiple Solvers, which brings back the bias. Too many slices might reduce the quality of synopsis estimations for some heuristics, which hurts pruning. It might also create additional burden for Solvers when maintaining domains.

When the static balancing falls through, Searchlight uses dynamic balancing as a fall-back strategy. When a Solver becomes idle (i.e., when it has finished its part), it reports itself to the coordinator as a *helper*. Coordinator pops a busy Solver from a queue, dispatches a helper to it, and pushes it back to the queue. Thus, a busy Solver might receive multiple helpers. The busy Solver cuts a part of its search tree and sends it to the helper. A Solver might reject help, e.g., due to the heuristic, in which case the helper is dispatched to another busy Solver. This process is highly dynamic. A helper becomes a busy Solver itself and might receive helpers in the future. Such balancing is similar

to *work stealing*, which is common in distributed CP.

Dynamic balancing is illustrated in Figure 4.5(b), where a Solver at Instance 1 cuts its right sub-tree and sends it to the helper at Instance 2. The helper starts the search by treating it as a root node. Transferring a sub-tree is cheap, since all Solvers run the same model. A sub-tree is a set of domain intervals, and can be serialized into a message efficiently.

4.4.3 Validating and Forwarding Candidates

In SciDB, array chunks are hash-partitioned across the entire cluster and DBMS queries are broadcast across all nodes. That means at the validation search UDFs might expose large latency: UDFs make multiple API calls, and each call has to be broadcast across the cluster, all answers must be collected and combined together. Only then the Validator can continue, until the next API call. Additionally, API functions would have to be distributive or algebraic [21]. This is true for many common functions (including the API aggregates). However, we wanted the API to be extendable with other type of functions, including holistic ones, which cannot be easily distributed. Considering this, we decided to investigate another approach.

The array is sliced into multiple partitions in a similar way as shown in Figure 4.5(a). In this case, however, each node running Validators gets exactly one slice. During the validation Searchlight transparently pulls the required array chunks from other instances and caches them in a temporary LRU buffer. The buffer is disk-backed, so no remote chunks are ever re-fetched. Chunks are pulled only on demand, when an API call needs them. If the search does not touch some parts of the array, which is common due to pruning, the corresponding data is neither fetched nor read from disk at all. With this approach UDFs can make multiple API calls during a validation without worrying about latency. Each call will be served from memory, after the chunks are fetched only once. Arbitrary complex functions can be added to the API, since any sub-array can be iterated over locally.

In some cases such data re-distribution might hurt performance. For queries that read the majority of the array, transferring a lot of chunks might saturate the network. However, we assume this to be rare in practice due to the nature of search queries. Even in such cases Searchlight does not stop to provide great interactive performance, as supported by experiments. Moreover, as an optimization, such redistribution can be done during the execution of the first query, which would create a chunks cache persistent between queries.

When a Solver finds a candidate, it has to send it to the appropriate node for validation. In general, solvers have no knowledge about which API calls will be made during the validation. These calls, however, determine the array chunks the validation needs. We solve this problem by first *simulating* the validation, which is performed by a same-node Validator. The Validator switches its Router (Figure 4.2) to the "dumb" mode. In this mode, the Router answers API calls with $[-\infty, +\infty]$ intervals, satisfying any constraint. At the same time, it logs all API calls. After the simulation is finished, the Validator retrieves the log and determines the chunks needed by the candidate. After that, it forwards the candidate to the node responsible for most of the chunks required (it might keep the candidate at the local node). The simulation is very lightweight, since it does not require

data access, disk or memory.

Some Validators might get flooded with candidates. This becomes a CPU, not data, problem, since the data will be quickly pulled from remote instances and mostly residing in memory. In such cases Searchlight dynamically starts more Validators at the struggling nodes. Validations are performed concurrently by reading data from the same shared buffer, so no chunks are duplicated. The number of additional threads depends on the current state of the search process, which we discuss in the next section.

In case a node keeps getting flooded with candidates and there are no additional CPU resources (threads) available to run more Validators, Searchlight performs *rebalancing*. It takes a portion of candidates and sends them to another node. This effectively might result in that new node fetching more data, since candidates do not belong to its partition. While this process might be expensive in practice, at this point during the execution it is assumed to be beneficial for both interactive and total query performance. When performing rebalancing, the Validator first tries to find a new node that has been used for rebalancing from the same node before. That might mean the new node might have already fetched some of the required data and would not have to start from scratch. Secondly, the Validator will rebalance only to a node that does not have any of its own candidates, choosing an idle node instead.

4.4.4 Additional Performance Improvements

Systems usually restrict the number of simultaneous active jobs (e.g., via thread pools), either on per node or per query basis. Each Searchlight node participating in a query has a number of active jobs corresponding to the Solvers and Validators. The challenge is to put more CPU resources to either of them depending on the current state of the search. While it is possible to do this in the static way, before the query begins, that might be highly inefficient. For example, given 8 threads per node, 4 might be given to the Solvers and 4 to the Validators. However, it is hard to predict if a particular node is going to experience high load in the number of candidates or if the search is going to have a lot of candidates at all. To address this issue Searchlight uses a *dynamic* resource allocation scheme, where it monitors the current state of the search process for each node. Initially, given n available threads per node, it starts $n - 1$ Solvers and a single Validator at each node. If the number of candidates at a node increases over time, Searchlight stops Solvers when they finish exploring their parts of the search space and starts more Validators instead. Ultimately, nodes overwhelmed with candidates might become Validator-only. In this case, the remaining search space is automatically distributed between Solvers at other nodes. When the number of candidates decreases, Searchlight stops Validators and releases the Solvers allowing them to continue the exploration.

Searchlight optimizes resource utilization even further. Recall, when a Solver finishes its part of the search space, it reports itself to the coordinator as a helper. While it is waiting, Searchlight starts another Validator to avoid wasting resources. When a new search space load arrives, Searchlight stops a Validator and wakes up the Solver. This also ensures that when the Solvers have finished the search entirely, all resources go to Validators.

Another optimization involves batching candidates for more efficient I/O performance. Some queries produce a large number of candidates dispersed around the data array. If these candidates are validated in an arbitrary order, that might result in *thrashing*: DBMS repeatedly reads chunks from disk, evicts them and reads again for other candidates. To address this issue Searchlight first divides each data partition into multiple continuous *zones*. Each candidate is assigned to the zone containing most of the chunks required for its validation. This information is determined by the simulation process described before. At each step Validators take candidates from the same zone, enforcing locality. The zones are sorted in the most-recently-used order, so recent zones are checked first. By default Searchlight ensures that at least two zones can fit in memory at the same time. Firstly, candidates corresponding to continuous objects (windows) often span neighboring zones. Secondly, this allows us to avoid the case where two frequently-used zones evict each other from memory.

4.5 Experimental Evaluation

We performed extensive experimental evaluation of Searchlight for clusters of sizes 1 through 8. We used EBS-optimized Amazon EC2 instances c3.xlarge (4 virtual CPU cores, 7.5GB memory each). Storage per each instance varied in size, however each disk volume was an EBS general purpose SSD (no provisioned IOPS). The OS was Debian 7.6 (kernel 3.2.60). We used SciDB 14.3 as the DBMS and Or-Tools 1.0.0 (the current SVN trunk at the time) as the CP solver.

The main data array was generated using Gaussian distribution with varying mean (from 0 to 200) and small variance. At places we injected small sub-arrays with means of 300, 350 and 500, which introduced natural zones of interest (clusters). The data array size was $100,000 \times 100,000$ elements with a single attribute and took 120GB of space (the binary size of the SciDB file). We used a 100×100 synopsis, which took approximately 400KB. We performed the evaluation varying the size of the search space by choosing more or less restrictive constraints and variable domains. For each search space we also chose queries that produced different number of candidates to vary Validator loads. The queries were as follows:

HSS (Huge Search Space). The query looked for 800×800 sub-arrays with $avg() \in [330, 332]$. While results were situated around the clusters, the search space size was 10^{10} sub-arrays, which was impossible to finish in a reasonable time. We used HSS to explore time-limited execution.

SSS-HS, SSS-LS (Small Search Space, where HS/LS stands for High/Low selectivity). The left-most corners of the sub-arrays were restricted to coordinates divisible by 330. LS searched for $2,000 \times 2,000$ sub-arrays with $avg() \in [95, 120]$. For HS the sizes varied from 500 to 2,000 with step 100, and $avg() \in [330, 332]$.

SSS-ANO (ANOmaly). This query checked the ability of Searchlight to handle more elaborate constraints. In addition to searching for $1,000 \times 1,000$ sub-arrays with $avg() \in [200, 600]$, it computed the maximum element of the sub-array's neighborhood of size 500. The query selected only sub-arrays for which the difference between their maximum elements and the neighborhoods' was greater

than 100, which can be seen as detecting anomalies. The query was expressed via CP constraints with the only UDFs being $avg(), max()$.

LSS-HS, LSS-LS (Large Search Space). These were similar to SSS queries, except the left-most coordinates of sub-arrays were divisible by 10. Thus, the search space was much larger. The sub-array sizes varied from 500 to 2,000 with step 100. For HS $avg() \in [495, 505]$, for LS $avg \in [330, 332]$.

LSS-ANO. The query was similar to SSS-ANO: looking for 500×500 sub-arrays with $avg() \in [200, 600]$. The neighborhood size was 200, and an additional constraint was made: not only the difference between the maximums was at least 250, but also the difference between the region's minimum and the neighborhood's maximum was at least 200.

4.5.1 Exploring Alternatives

We studied two alternatives to Searchlight. The first, CP, ran a traditional CP solver on DBMS data. Data requests were made via UDFs, but no synopses were used, only the original data. This allowed us to explore the applicability of state-of-the-art CP solvers for exploring large data sets. The second alternative performed search by using the SciDB `window()` operator, which computes aggregates for every possible fixed-size sub-array. Then, the `filter()` operator was used to select the required sub-arrays. Since the operators belong to the Array Functional Language (AFL), we called this approach AFL. AFL does not allow full richness of constraints supported by Searchlight, but it allowed us comparison with a native DBMS solution. For all approaches we specify the size of the cluster via a hyphen, e.g., CP-8 means running CP in an 8-node cluster.

While the number of alternatives might seem scarce, these are the only ones available to the users today. It might be possible to use SciDB with complex client scripts or extend it with specialized indexes. However, this would require non-trivial effort, and would result in comparison with, basically, a different system. The main goal of this experiment was to explore existing alternatives.

Table 4.1: HSS times (secs), #results found in 1 hour

Approach	**First result**	**Delays**	**Results**
SL-1	13	0.001/3.8/101.1	981
SL-8	5	0.001/5.9/21.9	6,336

Results for HSS are presented in Table 4.1. Since the search space was very large, it was infeasible to run the query until completion. Thus, we limited the time to 1 hour. We see it as a common use-case, when users want to get some results within a time limit to get an idea about the content. Nor CP, nor AFL found anything. Both performed too many data reads, which destroyed the performance. Running the query in an 8-node cluster did not help. SL (Searchlight), found first result in 13/5 seconds and kept providing them during the execution with small delays. We provide delay statistics as min/avg/max delays between subsequent results. We believe this to be a good measure of the interactivity. We also provide the time to find the first result as a measure of the initial response time.

Table 4.2: SSS-HS-modified query times, seconds

Approach	First result	Delays	Total time
SL-1/8	6.19/4.8	6.19/4.8	6.52/5.13
CP-1/8	3,360/91	3,360/91	3,611/304

Even for a small search space, CP was not able to finish SSS-HS in 13 hours, and no results were found. Thus, we decided to decrease the search space by running SSS-HS inside a $10,000 \times 10,000$ sub-array of the original array (Table 4.2). Even in this case, CP was no match for SL. This is due to a large number of API calls produced by CP. While this was true for SL as well, SL was able to utilize the synopsis, greatly improving its performance. Moreover, for CP pruning did not make much difference, since it requires reading the data first. In a cluster the performance of CP increased significantly, while still remaining well below the SL. When we tried a larger sub-array of size $30,000 \times 30,000$, Searchlight was done in 11 seconds, whereas CP could not find any results within 3 hours, and is thus not a competitor.

Table 4.3: Times for SSS-HS(top)/-LS(bottom), seconds

Approach	First result	Delays	Total time
SL-1	8.2	0.09/2.4/8.2	31.2
SL-8	6.12	0.02/1.2/6.9	13.9
AFL-1	2,105	2,105	2,105
AFL-8	301.3	1.1/3.3/7.1	945.3
SL-1	16.7	0.001/0.14/16.7	2,198
SL-8	6.1	0.001/0.05/6.1	563.3
AFL-1	1,852	1,852	1,852
AFL-8	295	295	295

AFL, however, was able to complete the SSS queries within reasonable time. Unfortunately, expressing the constraints in AFL was not entirely possible, since `window()` does not support variable sizes. We tried to use the `concat()` operator to combine results of several window operators, which gave very poor performance due to subpar implementation. Moreover, SSS-HS required 256 `concat()` operators for a single query, which is hard to optimize. We decided to use another approach. First, since SSS-LS/HS require window coordinates divisible by 330, we created a temporary array via the `regrid()` operator, which divides the array into tiles and computes aggregates for them. This actually gives an I/O performance boost to SciDB, since such an array can be seen as an index. Then, we ran several filter-window queries over the same connection (one for each possible size) and measured the total time of all queries as well as times for intermediate results. The regrid time was added to the time of the first result, since it was the essential part of the AFL query. It was impossible to express SSS-HS exactly (after the `regrid()`, window sizes had to be divisible by 330 as well), so we modified the query preserving the high selectivity. For SSS-LS we ran a single regrid-window-filter query. Results are presented in Table 4.3.

While AFL required computing every sub-array, Searchlight was able to prune most of the data

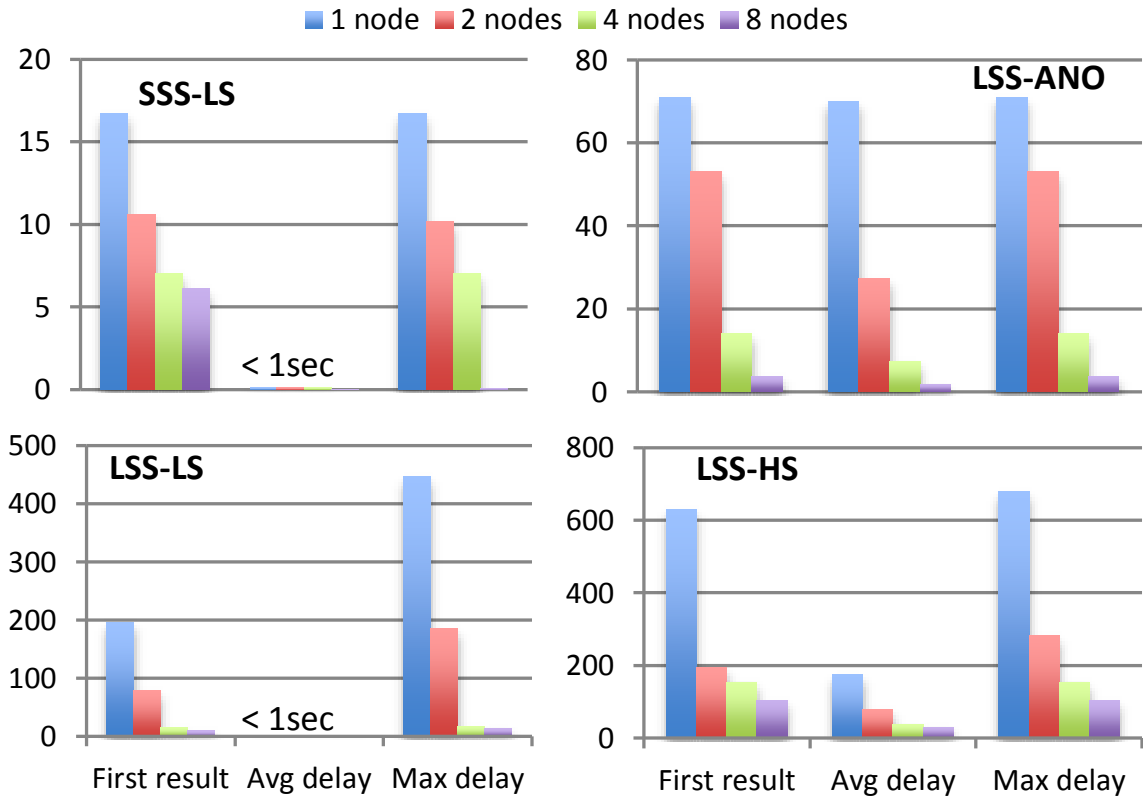

Figure 4.6: Delays for subsequent results for SL (secs).

and provide much better performance. SS-LS query, however, is different. While pruning was possible, the candidates touched the majority of the array, which created significant I/O load. As we discussed in Section 4.4.3, we consider such queries rare. In the cluster the performance of AFL improved significantly, and it was able to beat SL by utilizing the `regrid()` array. Note, SL remained the best interactive solution in all cases.

We tried to compare Searchlight and AFL for the LSS queries. However, even running a version of LSS-LS with fixed size windows resulted in a very poor AFL performance. We simplified the query and ran it on a `regrid()` array, similar to SSS. Modified LSS-LS did not find any results within 13 hours. All CPU cores were saturated and the query itself had a simple query plan. Such poor performance can be explained by the necessity of checking 10^8 windows. Thus, AFL cannot handle larger-than-trivial search spaces. SL finished the modified LSS-LS in under 2 minutes and output the first result in 18 seconds.

4.5.2 Online and Total Performance

We studied interactive and total performance of distributed Searchlight by varying the number of nodes in the cluster. The results are shown in Figure 4.6. We do not provide minimum delays, since

they were mostly under 1 second.

We decided to omit SSS-ANO and SSS-HS, since the former demonstrated the same trend as LSS-ANO, and the latter finished in 30 seconds even for a single node with the first result delivered in 8 seconds. Searchlight provides great interactivity even on a single node. Running it in a cluster brings most delays down even further. One notable exception is LSS-HS, for which the first result time remained around 2 minutes and averages around 30 seconds even in the 8-node cluster. However, this was a very hard query with a large number of candidates and limited pruning possible.

Table 4.4: Total times for queries, in seconds

Query	**1 node**	**2 nodes**	**4 nodes**	**8 nodes**
SSS-LS	2,138	1,799	1,572	563
SSS-HS	31	19	14	13.9
SSS-ANO	163	60	27	16
LSS-LS	2,830	1,271	646	491
LSS-HS	1,381	663	332	225
LSS-ANO	443	215	59	39

Total times for all queries are presented in Table 4.4. For some queries it was hard to improve performance beyond 4 nodes, since the granularity of the distribution on both levels of the execution is a slice (a sub-tree for dynamic balancing). One exception is SSS-LS, which scaled poorly as it touched almost every chunk of the array. Thus, Validators saturated the network (Amazon EBS is network-based as well). The results can be partially explained by bursty performance of EC2. When we were running the query in a 2-node cluster, the total time varied from 30 to 90 minutes. When we ran a similar query in a *local* 4-node cluster, the completion time went down from 2,008 (1 node) to 700 seconds (4 nodes). Note, LSS-LS also has low selectivity. However, since it did not touch as many chunks, it scaled much better.

4.5.3 SDSS Experiment

We experimented with the entire Sloan Digital Sky Survey (SDSS) [4] catalog, which contains information about objects in the surveyed portion of the sky. We used right ascension (ra) and declination (dec) as coordinates. The catalog provides a large number of attributes, and we chose the model magnitudes: u, g, r, i, z. These are spectrum measures that can be used to analyze the "brightness" of objects. The binary size of the data in the DBMS was approximately 80GB. We performed the experiment in an 8-node cluster.

Table 4.5: SDSS queries times (Q_i given in text), secs

Query	**First result**	**Delays**	**Total time**
Q_1	10	0.001/2/54	300
Q_2	17	17	132
Q_3	24	0.004/6/45	331
Q_4	29	0.21/13/29	134

We fed Searchlight a variety of queries searching for sky regions (sub-arrays) with average magnitudes belonging to a range. Similar queries are often used in SDSS workloads when filtering individual objects. Some of the results are presented in Table 4.5. The queries are given below in the format: "rX,mY,lenZ,stT", where X means the resolution (e.g., 1° for sub-arrays with leftmost coordinates divisible by 1°), Y means ranges for magnitudes (we used all five of them in the same query), Z means the range of lengths (e.g., 4° to 5°) and T means the step for the lengths. The delays, as before, are given as min/avg/max. If the delay is a single number, that means there was only one result. The queries were: Q_1 = res1,m10-20,len4-5,st0.05; Q_2 = res1,m5-15,len0.4-0.5,st0.001; Q_3 = res0.1,m0-20,10-30,10-15,0-50,0-40,len5; Q_4 = res0.1,m5-15,len0.1.

SDSS was a big challenge for Searchlight. Objects with different magnitudes are dispersed around the data set, which makes pruning difficult. Thus, the completion times were worse than for the synthetic data. Even so, Searchlight was able to provide great interactive performance.

Chapter 5

Query Relaxation and Constraining

As we discussed in Section 1.3, initially the user might have little knowledge about the dataset. Even if she has understanding of the nature of constraints she wants to query the data with, it might be hard for her to come up with proper constraint parameters. As an example, consider a simple range constraint $avg(x) < 10$. The user might want to obtain at least 10 results from the query. If the query returns no results or 1-2 results, she might want to consider broader range for the $avg(x)$ function. Unfortunately, there is no way for her to know which range value will work. Setting the value to 12 might still not return enough results, while setting it to 20 might return too many of them, e.g., a hundred or more. This guessing game becomes much more cumbersome for the user, when the number of constraints increases. In that case she would need to try different combinations of constraints and possibly stitch together results from multiple modified queries to obtain results close enough to the original constraints.

Searchlight aspires to improve the user's experience by automatically modifying the query during the execution. The user specifies the cardinality requirement in addition to the query constraints. If the search produces too few results, Searchlight automatically relaxes the query to include more results that minimize the distance from the original constraints. At the same time, if the query starts producing too many results, Searchlight introduces new constraints to the query to rank the results and output the top ranking ones, filtering all the others. Thus, Searchlight is able to adapt the query execution dynamically to either relaxation or constraining. The user does not have to choose the strategy when specifying the query.

As we discussed in Chapter 2, the methods for such query *relaxation* and *contraction* techniques are well-researched in the context of relational databases. However, since Searchlight touches upon the different realm of search queries, the previous approaches cannot be directly applied to our system. In this chapter we discuss how Searchlight handles query relaxation and constraining[1],

[1]We prefer the term *constraining* to a more common term *contraction* due to the way the contraction is handled

challenges that these techniques bring to our architecture and our solutions that deal with these challenges.

5.1 Query Relaxation and Constraining Model

Each Searchlight query consists of a number of decision variables X and constraints C. In addition to the query itself the user gives Searchlight the desired cardinality for the result k. The original query might have the following outcomes:

- The query outputs exactly k results. In this case it is processed in the usual way, without any modifications.
- The query outputs a set of results $R = r$ such that $|R| < k$. In this case it is automatically modified by Searchlight to include additional $k - |R|$ results. The additional results are guaranteed to minimize the specified $RD(r)$ function (discussed below)[2] This is called query relaxation.
- The query outputs $|R| > k$ results. In this case Searchlight ranks the results based on the provided $RK(r)$ function (discussed below). The query is guaranteed to return top-k results according to $RK(r)$. This is called query constraining.

When relaxing or constraining the query Searchlight considers 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 UDFs (see Section 4.2.1) and variables from X. For the purpose of relaxation and constraining Searchlight treats $f()$ as black boxes. As we pointed out in Section 4.2.1, we assume at every node of the search tree the function produces the range of all possible values for all the corresponding values of X, i.e., $a' \leq f(X) \leq b'$. UDFs, the decision variables and their common algebraic expressions satisfy this requirement. By default, Searchlight considers all range-based constraints, but the user can exclude any of them from the relaxation and/or contraction process. Let us denote all constraint Searchlight considers for relaxation as C^r and for constraining as 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 5.1.1 and 5.1.2 we describe the out-of-box ranking functions Searchlight uses. Then, in Section 5.1.3 we discuss the custom ranking functions requirements.

in our system — by introducing additional constraints.

[2]Strictly speaking, the number of additional results might exceed $k-|R|$ in case many results have the same $RD(r)$ value.

5.1.1 Query Relaxation Model

For each constraint $c \in C^r$ we define the relaxation distance $RD_c()$ as follows. Assuming constraint $a \leq f_c(X) \leq b$ and a result r, s.t. $f_c(r) = t$:

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

Then the total relaxation distance for 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, $w_c = 1$. The more the weight of the constraint, the faster its penalty grows as part of the relaxation distance.

One problem exposed by the definition above is the possible difference in scale between $f_c()$ for different constraints. For example, one constraint might deal with ages (e.g., $f_c() \in [0, 200]$), while another might deal with similarities (e.g., $f_c() \in [0, 1]$). To account for this we use normalized $RD_c()$ values by dividing each $RD_c()$ by the maximum possible difference for each $f_c()$. Such maximum differences can be usually derived from the obvious domain restrictions (e.g., similarity cannot exceed 1). We additionally allow users to specify maximum and minimum values for each $f_c()$, giving them more control over relaxation. Searchlight will not relax the corresponding constraints beyond the specified values. From this point on we will assume $RD_c(r), RD(r) \in [0, 1]$ for any r.

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 $|C^r|$. Obviously, $VC(r) \in [0, 1]$. Then, the total relaxation penalty for result r is defined as:

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

α is a parameter that allows the user to put more importance in the distance of the results from the original constraints or in the number of violated constraints. By default $\alpha = 0.5$.

Assuming these definitions, Searchlight provides the following *relaxation guarantee*: if the user submits query Q with the cardinality requirement k, Searchlight outputs at least k results r with the lowest $RP()$ values among all possible r. There are two important points to consider:

- The guarantee above naturally incorporates queries that do not need relaxation. If the query has at least k results, all of them will have $RP = 0$ by definition, which is the minimal possible value. In that case no relaxed results are possible, since for relaxed results $RP > 0$.

- If the user specifies relaxation restrictions for some $f_c()$ (see above), Searchlight cannot output k results, if the number of results satisfying the *maximally relaxed* constraints is less than k. In that case even the maximally relaxed query is still over-constrained and does not contain enough result. Searchlight will just output all of them.

Revisiting the astronomical example from Section 4.2.1, the query has variables $x \in [0, 10], y \in [20, 50], lx \in [2, 5], ly \in [3, 10]$ describing the region of interest the user is searching for (i.e., the top-left coordinates and lengths in the two-dimensional right ascension/declination search space). It also has the following constraints:

- $x + lx - 1 \leq 10 \wedge y + ly - 1 \leq 50$ for the region to fit inside the whole search region.
- $C_1 = avg(x, y, lx, ly, r) \leq 11$ defining the user preference for the average value of the r-magnitude for objects inside the region.
- $C_2 = avg(x, y, lx, ly, g) \leq 10$, which is analogous to the r-magnitude constraint above, but for the g-magnitude.

Let us assume that the user wants $k = 3$ results and $C^r = \{C_1, C_2\}$. Additionally, $avg()$ values for each magnitude belong to the $[-20, 20]$ domain interval, $\alpha = \frac{1}{2}$ and $w_{C_i} = 1$. The results have the form (r, g), where $r = avg(r)$ and $g = avg(g)$. Then the search might progress as follows:

1. A result $r_1 = (11, 10)$ is found. It satisfies both constraints, so $RP(r_1) = 0$.
2. A result $r_2 = (9, 9)$ is found. Again, it satisfies both constraints, so $RP(r_2) = 0$.
3. Searchlight cannot find any more results satisfying the original constraints, so it starts relaxing the query.
4. A result $r_3 = (18, 18)$ is found. It violates both constraints: $RD_{C_1}(r_3) = \frac{18-11}{20-11} = 0.777$, $RD_{C_2}(r_3) = \frac{18-10}{20-10} = 0.8$. Thus, $RD(r_3) = \max\{0.777, 0.8\} = 0.8$, and $RP(r_3) = \frac{1}{2}(RD(r_3) + VC(r_3)) = \frac{1}{2}(0.8 + \frac{2}{2}) = 0.9$.
5. A result $r_4 = (11, 13)$ is found. It violates only C_2, thus $RD_{C_1}(r_4) = 0$, $RD_{C_2}(r_4) = \frac{13-10}{20-10} = 0.3$, $RD(r_4) = \max\{0, 0.3\} = 0.3$, $RP(r_4) = \frac{1}{2}(0.3 + \frac{1}{2}) = 0.4$. Since $RP(r_4) < RP(r_3)$, r_4 is inserted into the top-3 results, and r_3 is discarded.

Depending on the user preference, r_3 might be output during the search and then indicated as obsolete by the system when r_4 is found. If the user does not wish to receive intermediate results, r_3 is found and discarded without the user's knowledge.

5.1.2 Query Constraining Model

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

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

Note, by the nature of constraining it is guaranteed $a \leq t \leq b$. As we discussed in the section about query relaxation, if the constraints contain functions with the values over different scales, we normalize the $RK_c(r)$ to be in $[0, 1]$ by dividing the value to $b - a$. If the interval is half-open, i.e., a or b is not specified for the constraint, a suitable domain boundary for the function can be used instead.

We define the full rank of result r as follows:

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

where $w_c, 0 \leq w_c \leq 1 \wedge \sum_c w_c = 1$ represent the constraint's weights. They allow the user to prioritize some constraints over the others. By default $w_c = \frac{1}{|C^c|}$. We subtract the weighted sum of individual ranks from 1 so that the better results get higher ranks, which is a more natural definition.

Assuming these definitions, Searchlight provides the following *constraining guarantee*: when the user submits query Q with the result cardinality requirement k, if Q has at least k results r, Searchlights outputs at most k results with highest $RK()$ ranks among all possible r. Note, if the query does not have at least k results, Searchlight will either perform the relaxation or just output all results, as per the user's preference. In both cases the constraining will not be activated.

Revisiting the astronomical example from Section 5.1.1, let $C^c = \{C_1, C_2\}, w_{C_i} = \frac{1}{2}$. We assume the user prefers maximization for both functions and wants only one result (i.e., $k = 1$). The search might progress as follows:

1. A result $r_1 = (11, 10)$ is found. Its rank is $RK(r_1) = 1 - \frac{1}{2}(\frac{20-11}{40} + \frac{20-10}{40}) = 1 - 0.24 = 0.76$.
2. A result $r_2 = (9, 9)$ is found. Its rank is $RK(r_2) = 1 - \frac{1}{2}(\frac{20-9}{40} + \frac{20-9}{40}) = 0.73$. Since $RK(r_2) < RK(r_3)$ r_2 is discarded.
3. A result $r_3 = (20, 15)$ is found. Its rank is $RK(r_3) = 1 - \frac{1}{2}(\frac{20-20}{40} + \frac{20-15}{40}) = 0.94$. Since $RK(r_3) > RK(r_1)$, r_1 is discarded, and r_3 becomes the next top-1 result.

As in the case of query relaxation, if the users wishes to receive intermediate results, Searchlight outputs all of them, but might obsolete some if better results are found at some point (e.g., r_3 makes r_1 obsolete in the example above).

In addition to the scalar value ranking approach just described Searchlight supports another popular ranking paradigm called *skyline*, which is vector-based. In that case Searchlight simply outputs non-dominated results, where a result is a vector of values of $f_c(), c \in C^c$. By definition, V dominates W, iff $\forall i : v_i \geq w_i \wedge \exists i : v_i > w_i$. The user can choose the meaning of $>$ for each component by specifying if she wants to minimize or maximize the corresponding function. In case the user chooses the skyline, however, Searchlight cannot guarantee that the number of results will not exceed k. There is no way to directly compare non-comparable vectors.

5.1.3 Custom Ranking Functions

In addition to the out-of-box ranking of results we discussed in the two previous sections, Searchlight can be extended to support custom ranking functions, if needed. These function must satisfy the following requirements to ensure correct search tree pruning and provide maximum efficiency during the query execution.

First, we discuss custom relaxation ranking. As in the built-in approach, it is based on the distance of a relaxed result from the original constraints. The user has to define a custom penalty function $RP()$ with the following requirements:

- $RP() \geq 0$, with larger values corresponding to worse relaxation. All results satisfying the original query must have $RP() = 0$.
- The user has to define the $RP()$ function as a range function for all possible search states (search tree node). She is provided with the current decision variable domains (i.e., the subset of the original search space) and *ranges* of values for $f_c(), c \in C^r$, corresponding to the current search tree node. The user has to return the range of possible penalties $[lp, hp]$.
- To ensure proper pruning and results, $RP()$ and its ranges must satisfy the following criteria:
  - $lp = hp$ must be exact for the bounded decision variables, when the search is at a solution. This affects result reporting in the obvious way, since $RP()$ is the ranking function for the relaxed results, and Searchlight reports the top-k of them.
  - lp cannot be underestimated (i.e., greater than the minimum of all possible lp values at all leaves corresponding to the search node's sub-tree). Otherwise, the pruning might cut valid results.
  - lp cannot decrease for the descendants of the search node, i.e., if node S' is a descendant of node S, then $lp(S') \geq lp(S)$. This guarantees the validity of pruning.

The custom query constraining is rank-based as well, so the user has to define a custom ranking function $RK()$ with the following requirements:

- Larger $RK()$ values correspond to better results.
- The user has to define $RK()$ for every possible node of the search tree, as in the case of $RP()$. She is given the current decision variable domains and the current values for $f_c(), c \in C^c$. The function must produce range $[lr, hr]$ of possible ranks for all possible solutions corresponding to the search node's sub-tree.
- To ensure correct pruning and result reporting, $RK()$ must satisfy the following requirements:
  - $lr = hr$ must be exact at a solution (as in the case of $RP()$). Since results are ranked, and Searchlight outputs the top-k results, this guarantees proper result reporting.

- hr cannot be underestimated (i.e., less than the maximum of all possible hr values at solutions corresponding to the search node's sub-tree). This is necessary for proper pruning.
- hr cannot increase with the descendants of the current search node, i.e., if node S' is a descendant of node S, then $hr(S') \leq hr(S)$. This is necessary for valid pruning.

In addition, if the user wants to perform skyline-like computation with defining her own notion of domination between result vectors, she can provide a function similar to $RK()$ above. The only exception is that she is provided with the current top-k vectors at each call, since domination requires to know the current result in total instead of just scalar rank values. The requirements specified above for $RK()$ can be naturally converted to the custom skyline function, which effectively is a constraint making pruning decisions at every node of the search tree. Additionally, this function can and must modify the current top-k results, since a new result might dominate a number of others in the top-k set.

As can be seen, the general requirements for the custom ranking functions are quite similar, which is no surprise. Searchlight uses similar techniques for both relaxation and constraining at the query execution level, which allows it to adapt the search to either of these strategies dynamically.

5.2 Query Relaxation and Constraining in Searchlight

As we discussed in Section 4.1, Searchlight accepts queries in form of Constraint Programs (CP). The search is performed by the CP solver, which dynamically builds and traverses the search tree. The dynamic nature of the query execution process allows Searchlight to naturally modify the existing constraints and add new ones. This is the main idea behind supporting query relaxation and constraining in Searchlight. If the query does not produce enough results satisfying the original constraints, Searchlight revisits some parts of the search tree with modified (relaxed) original constraints. If the query produces too many results, Searchlight introduces new constraints to prune results having smaller ranks than the already found ones.

As in the case of the original query processing, the efficiency of the relaxation and constraining heavily relies on effective pruning. There are three possible pruning points when performing relaxation or constraining[3]:

- During the main search at the Solver. This is the most effective point of pruning. Not only it allows Searchlight to prune parts of the search tree from consideration, but possibly a large number of candidate solutions as well. This is also the most natural point of pruning, and we target it with query constraints.
- Just before validating the candidate at the Validator. Note, each candidate solution is submitted by a Solver. Thus, it has passed the search-level checks. However, due to extensive disk

[3]see Section 4.3 for the description of the Searchlight two-level query processing.

accessing Validators work slower than Solvers. That means many candidates in the Validator backlog queue might have been submitted before the top ranks or relaxation penalties were updated. Rechecking the candidates using up-to-date information might allow some *pre-disk* pruning.

- After validating the candidate at the Validator. Even if the candidate passes the validation checks over the real data, it is necessary to check them again if the new ranking/penalty information became available. Note, this is the least efficient point of pruning, not allowing any significant performance gains, since the *disk* validation has just been performed. It might ease the burden for the user by filtering out inferior results at the server and possibly decreasing the amount of data transferred to the client.

In the following sections we describe the basic methods behind query relaxation and constraining in more detail. We also discuss some optimizations we perform to obtain additional performance gains.

5.2.1 Query Relaxation

As we discussed in Section 4.2.1, a CP solver dynamically builds the search tree and validates relevant constraints at every node of the tree. A node either satisfies the constraints, at least currently, or *fails*. The successful nodes eventually lead to leaves, which produce candidate solutions. When the search is finished, if Searchlight has not found k results (the user's cardinality requirement), it starts revisiting parts of the search tree with modified, relaxed constraints. It does not have to revisit successful search nodes, since these nodes satisfied the *original* constraints. Revisiting those with relaxed constraints would not introduce any new candidate solutions. Revisiting previously failed search nodes, on the other hand, could lead to candidates satisfying *relaxed* constraints. We call such revisiting *fail replaying*, and this is the main technique behind the relaxation.

When Searchlight encounters a failed search node during the main search, it prunes the node as usually. However, it records the current search state at the node before doing so. The information includes:

- Current decision variable domains. This information is crucial when the fail is replayed later, so that the fail can be instantiated as the root of its own search tree. This way we do not need to revisit the whole path from the original search root to the failed node.

- The range $[a', b']$ for values of every function $f_c, c \in C^r$. These ranges are obtained as part of the search process anyway, since all constraints, including C^r, must be verified at each search node. If f_c contains UDFs, the ranges for UDFs are obtained via the synopses and then combined to produce the range for the whole expression[4]. Note, if $[a', b']$ intersects the

[4]In terms of Or-Tools, f_c is an integer expression, which must support min/max calls. If f_c is an algebraic expression, such calls have obvious implementation.

interval $[a, b], a \leq f_c \leq b$ of the original constraint, the constraint is counted as satisfied, and the interval is not recorded.

After this information has been recorded, Searchlight computes the best and worst relaxation penalties possible for this fail. For the built-in RP function this is straightforward, since the current f_c ranges and the number of violated constraints has just been computed. As for the custom RP function, it must support the corresponding call as discussed in Section 5.1.3. After the relaxation penalties are computed, the fail is inserted in the priority queue ranked by the best fail relaxation penalties.

If during the search at least k results are found, Searchlight clears the queue and stops intercepting and recording search fails. In this case, however, the constraining turns on, which we discuss later. If the main search is finished with less than k results, the relaxation is needed. Searchlight starts replaying fails in the priority queue order (i.e., minimizing the best relaxation penalty by default). To replay the fail, Searchlight does the following:

1. A new search is instantiated with the same decision variables and constraints as in the original query. Thus, the model is not modified.

2. The variables are assigned the previously recorded domains for the fail. Thus, the root of the tree corresponds to the same search state the fail was in.

3. All violated constraints are modified as follows. The original $a \leq f_c() \leq b$ become $a' \leq f_c() \leq b'$, where $[a', b']$ is the recorded interval at the moment of fail. Note, this guarantees the fail will become a successful node, and the search can continue.

Searchlight then starts the search and treats it as the original one. No special tricks are required. As the original search, the search initiated by a fail replay can also encounter its own fails. For example, when a constraint that was not violated during the original fail becomes violated. The common reason is improved synopsis estimations for UDFs when the search deepens, in which case the intervals for $f_c()$ become gradually tighter. Such *repeated* fails are caught and recorded as the original ones. They are inserted in the priority queue and might be replayed later. It is important to notice that during replays Searchlight explores only previously untouched parts of the search tree. That means no work is repeated for already visited parts of the search tree, and there can be no duplicates in the results.

As can be seen, the mechanism described above does not allow any pruning at the search level. If we replay fails with relaxing violated constrains as shown, the relaxed constraints cannot fail again, i.e., violated constraints are "fixed" completely. The only new fails can be due to new constraints failing, as we mentioned above. Thus, no pruning will occur for such replays. To remedy that Searchlight takes into account the Maximum Relaxation Penalty (MRP) among the already found results.

Let us first assume the built-in RP function. While the number of results is less than k, the MRP is set to the maximum, $MRP = 1$. When at least k results are found, MRP might become less than 1. First of all, Searchlight modifies the fail recording and replaying process as follows.

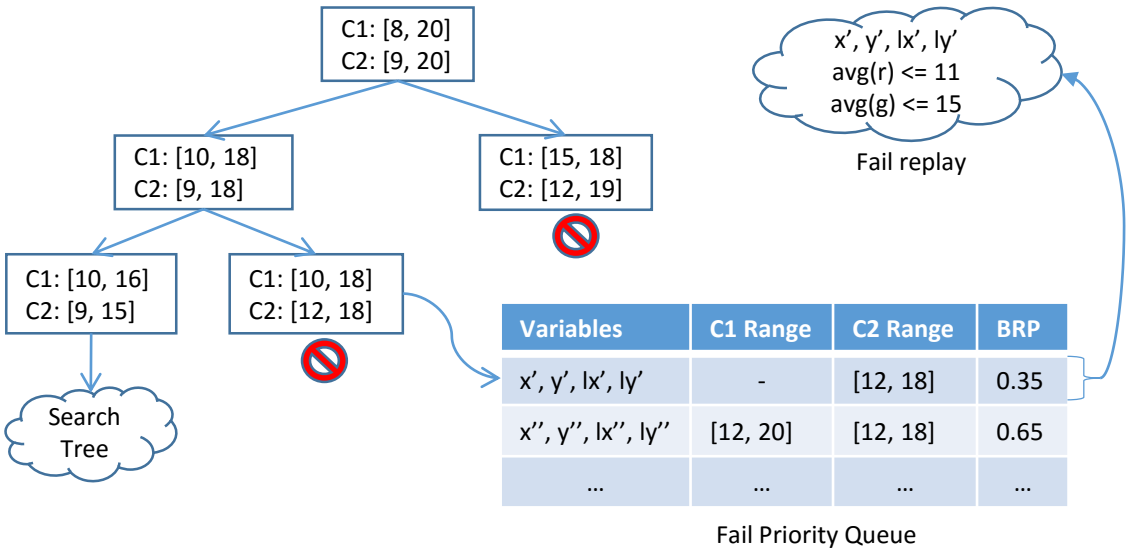

Figure 5.1: Example of fail recording and replaying for the astronomical query ($MRP = 0.5$).

- When the fail is being recorded, the $[a', b']$ interval is recorded as previously. However, if the best relaxation penalty for the fail is greater than MRP, the fail is discarded. The corresponding search tree cannot result in any candidates with $RP() \geq MRP$. This is equivalent to pruning the corresponding subtree.
- When the fail is selected for replaying, its recorded intervals $[a', b']$ are tightened by using the current value of MRP, which we discuss in more detail below. Such tightening allows Searchlight to perform more effective pruning, since constraints are relaxed as minimally as possible. Also, the best relaxation penalty for the fail is again checked against MRP. If it is greater than MRP, this fail is discarded without being replayed, and a new one is taken from the queue.

Let us discuss the interval tightening in more detail. We assume the built-in $RP()$ function discussed in Section 5.1.1 with $\alpha \neq 0$. If $\alpha = 0$, the relaxation distance does not influence $RP()$, and the originally recorded interval $[a', b']$ must be used. Otherwise, to qualify for the result, all candidates r must have $RP(r) \leq MRP$. This implies:

$$\begin{aligned} \alpha RD(r) + (1-\alpha)VC(r) &\leq MRP \\ \alpha RD(r) &\leq MRP - (1-\alpha)VC(r) \\ RD(r) &\leq \frac{MRP - (1-\alpha)VC(r)}{\alpha} \end{aligned} \tag{5.1}$$

This relaxation distance boundary can be used to tighten each violated interval $[a', b']$ accordingly before the fail replay starts.

Let us revisit the example from Section 5.1.1 and illustrate the above algorithm with the help of Figure 5.1. In this figure the search nodes are rectangles with the current (i.e., synopsis-estimated) values for the $avg()$ from constraints C_1 and C_2. As can be seen, the root node produces two new decisions (the search heuristic is not important for this example). The left node is processed, and produces two new decisions. For the first one, since constraints are currently satisfied (current intervals intersect constraint intervals), the search continues further (the "Search Tree" cloud in the figure). At some point the solver backtracks and explores the second decision. At this node, however, the C_2 constraint is violated, since $12 > 10$. The Best Relaxation Penalty (BRP) for the fail is: $\frac{1}{2}(\frac{2}{10} + 0) + \frac{1}{2}\frac{1}{2} = 0.35$. This is smaller than the current $MRP = 0.5$, so the fail is recorded in the priority queue (we do not record the C_1 range, since C_1 is not violated). The solver then backtracks again and explores the second child of the root. At this node both constraints are violated. Its $BRP = \frac{1}{2}(\frac{4}{10} + \frac{2}{10}) + \frac{1}{2}\frac{2}{2} = 0.8$. Since it is greater than the current $MRP = 0.5$, we discard the fail without recording it. At some point the solver starts replaying the fails and takes the first one (with the best BRP) from the queue. The C_1 constraint remains the same: $avg() \leq 11$. We can set the C_2 constraint as $avg() \leq 18$. However, since $MRP = 0.5$, $RD \leq \frac{\frac{1}{2} - \frac{1}{2}\frac{1}{2}}{\frac{1}{2}} = 0.5$. The right bound h for the constraint can be tightened to $\frac{t-10}{10} = \frac{1}{2} \implies t = 15$. Thus, C_2 is modified to $avg() \leq 15$ at this replay (search node).

For the custom $RP()$ function Searchlight cannot apply the same logic for tightening intervals, since the custom $RP()$ is basically 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 do not record the fail for further replays.

When considering MRP, the search initiated by a fail replay can still encounter other fails, which now can be of two types:

- A previously non-violated constraint is now violated. This is the same case as before. We check MRP and record the fail, if it passes the check. When replaying such a fail, the f_c intervals might be tightened even further, since $VC()$ increases.

- A previously relaxed constraint fails. This can happen due to the additional tightening described above. In this case, the fail is discarded, since the constraint has been already maximally relaxed with respect to MRP and cannot be relaxed further. This is actually caught by checking the best penalty value for the fail. In this case it will be greater than the current MRP by definition.

After the search tree with relaxed constraints reaches a leaf, it submits the corresponding candidate solution to the appropriate Validator in the same way as the original search. Since the Validator runs the same model from the original query, it needs to make sure its own constraints are properly relaxed. At the same time, it can provide additional pruning *before* the validation accesses the disk. Let us now describe how Validators handle candidate solutions in the presence of relaxation.

Validators generally work much slower than Solvers, since they need to access disk data and possibly transfer some data over the network, if data redistribution is required. That means Solvers submit candidates faster than Validators check them. When the Validator takes another batch of candidates from its pending candidates queue, some candidates in the batch might have large BRP values and might violate the current MRP value. The Validator processes the candidates as follows:

1. The Validator takes a batch of candidates from its queue of pending candidates. This is the same step as for the common search.

2. It checks the candidate's BRP value against the current MRP, and in case $BRP > MRP$ it discards the candidate. The BRP value is sent to the Validator from the Solver as additional meta-data.

3. If the candidate passes the check, the Validator relaxes constraints at its own validating CP solver. Along with the BRP value, The Solver also sends the list of violated constraints. The Validator uses this information to relax *all* constraints using the Inequality 5.1, in the same way the Solver uses it for the tightening. Note, the Validator has to relax all the constraints, since even not previously violated constraints might become violated (i.e., due to imprecise synopsis estimations). Such new violations do not necessarily fail the candidate. It might still satisfy the current MRP value. Since the tightening is not possible in case of the custom RP function, the Validator just relaxes all constraints maximally.

4. If the candidate passes the validation, the Validator computes its RP value and checks it against the current MRP value. If the RP value is greater than MRP, the solution is discarded.

5. If the solution passes the check, it is reported to the user (if the user wishes to receive intermediate results), and its RP value is broadcasted across the cluster so that all nodes can update the MRP value accordingly.

The validation itself is performed as usual, without any modifications. It can be seen that for both Solvers and Validators query relaxation does not cause any significant query processing changes. The relaxation is incorporated quite naturally inside the CP-based search and validation process, and does not require any awareness from the user. For example, if the user specifies a custom search heuristic to use, it does not interfere with the relaxation, since the search process itself remains the same.

5.2.2 Query Relaxation Optimizations

In this section we discuss a number of useful optimizations we perform for the basic query relaxation process. These optimizations do not modify the main workflow, but provide additional performance gains in some frequently occurring situations.

Computing UDF values at fails. As we described in the previous section, when Searchlight catches a fail, it computes 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. The fail happens at the first violated constraint, in which case the subsequent constraints are not touched at all. This implies some f_c values might remain unknown. For example, in our running astronomical example, if C_1 fails, C_2 is not verified, and $avg(g)$ is not computed. In this case Searchlight can force the computation for such functions to obtain proper $[a', b']$ ranges. However, f_c functions usually contain Searchlight UDFs, computing which might be relatively expensive, despite this requiring memory-only accesses. While it might seem like a negligent performance issue for a single fail, for complex queries Searchlight might catch hundreds of thousands fails. If the query contains multiple UDFs, the cumulative performance drain might become quite visible. First of all, the overall query execution time might increase significantly. Secondly, it might postpone outputting the first result, since the overhead starts accumulating before the relaxation begins, at the fail catching phase. Lastly, if the query has "real" results, satisfying the original constraints, the latency for such results increases as well, since fail recording introduces non-trivial delays. This is especially disturbing in the presence of the fact that many fail records might not be required later at all! If the query does not require the relaxation, *all* the recorded fails will be discarded after the required number of results is found, and the recording time will be spent for nothing. If the relaxation is required, the remaining fails will be discarded as soon as MRP reaches a small enough value. Thus, it is important to minimize the time spent on catching and recording the fails.

To deal with this issue, we do not force the computation of all f_c functions, but simply record the $[a', b']$ intervals already known at the moment of fail. From the implementation perspective this is a simple matter of switching the Router (see Section 4.3.1) to the "dumb" mode while recording the fail, as we do for the candidate simulation at the Validator (see Section 4.4.3). Recall that in this mode the Router intercepts all data request and answers them with trivial $[-\infty, +\infty]$ intervals. Assuming the UDFs in constraints support caching[5], this has the following impact on functions:

- Since the search state remains the same, already computed functions do not perform any more data accesses, so they just reuse the cached values. The "dumb" mode has no effect on them.
- If the function has not been computed yet, it tries to access the data, but get the trivial answers (at no cost). This generally results in the trivial $[a', b']$ intervals, where $a' = -\infty, b' = +\infty$.

Searchlight naturally treats the trivial intervals as "unknown" values, and treats the corresponding constraints as not violated. This is an optimistic approach. That means the BRP (Best Relaxation Penalty) value for such fails is overestimated. When such a fail is replayed, the chances for additional fails in the replay's search tree increase. This does not create any problems with the algorithm. Such fails do not differ from the ordinary fails, and they are recorded and processed in the same way.

While we reduce the computation overhead at the recording stage, the skipped functions will still need to be computed when we replay the fail. Thus, for the replayed fails the total savings are null.

[5]Caching is provided for built-in UDFs. For custom UDFs, the caching is provided for the current search node, and can be extended by users.

We just postpone the computation until we really need it (if ever), which is a form of lazy evaluation. The decreased overhead at the fail recording stage might result in better interactive delays (time to the first result and delays between results in the same query). The benefits for the query completion times depend on the query — if Searchlight replays (almost) all recorded fails, for example, the query completion time would remain the same, since we just pay the cost of computation later.

As we show in the experimental evaluation, while the total number of processed fails might increase, overall this optimization results in significant performance gains for queries with expensive functions. It allows us to keep the relaxation turned on for all queries without incurring any visible penalties for result delays, even for the queries that do not require the relaxation (which is not known beforehand).

Partial relaxation at replays. When Searchlight replays fails, it relaxes violated constraints within the saved $[a', b']$ interval and tightens it with respect to the MRP value. At the early replays, however, such relaxation might be too "wide". If the original query fails almost immediately, close to the root of the search tree, the violated constraints will have wide $[a', b']$ intervals. At the same time, the MRP value will be still maximal. Thus, the tightening will not be possible, and the constraints will be relaxed in the maximal possible way. This will effectively result in traversing the entire search tree with no pruning. At some point, when Validators have checked some candidates, the MRP might go down, which will allow Searchlight to perform some pruning on both Solvers and Validators (at the before-validation point). Ideally we would like the MRP value to be updated to a small value as soon as possible. To facilitate that, at a fail replay we do not relax the violated constraints all the way, but rather use a percentage of the relaxation interval. The percentage is a parameter we call Replay Relaxation Distance ($0 \leq RRD \leq 1$). If, for example, a constraint $f_c() \leq 10$ is to be relaxed to $f_c() \leq 20$, and $RRD = 0.3$, we relax the constraint to $f_c() \leq 10 + (20 - 10) \times 0.3 = 13$. This effectively results in first exploring parts of the search tree that can lead to decreasing MRP to 0.3. A potential drawback is that there might no be such candidate solutions at all for the query. In this case, the search results in a number of new fails, which are recorded and gradually relaxed further. This might create an additional delay for the results. However, our experiments showed that small values of RRD might considerably speed-up some of the queries, and, at the same time, do not hamper other queries much.

Saving UDF states at fails. Searchlight saves decision variable domains when recording fails to be able to replay the fail later from exactly the same point. Since the model essentially remains the same (relaxing constraint intervals does not change the model itself), restoring the variable domains is enough to continue the search. However, at the moment of fail, some $f_c()$ functions have been computed and some UDFs, participating in these functions, might have important caching information stored. At the very least, this information is valid at the current search node. For more advanced function implementations, this information can be used to speed up the recomputation for descendant search nodes as well. To prevent the loss of this information, Searchlight provides support for *UDF states*. At a fail, when performing the recording, Searchlight saves the state as a binary blob without interpreting it in any way. When the replay of the fail is initiated, Searchlight

restores the state for all UDFs that saved it. If the user uses her own functions, she can implement the save/load state functions. If she does not, the state is lost at fails. This optimization was found to provide significant performance gains when replaying fails, as expected. At the same time, it does not introduce any overhead, except slightly increased memory footprint.

Sorting the Validator queue on BRP. When candidate solutions are received by the Validator, they are put into a queue. Without considering the relaxation, FIFO queues work efficiently, especially when we use several zones to avoid data buffer thrashing. However, FIFO queues do not make the relaxation-specific pruning more effective. It is often the case that first fail replays produce a considerable number of candidate solutions, creating relatively long queues (recall, Solvers work much faster than Validators due to the absence of disk accesses). Subsequent fail replays might produce candidates with better relaxation penalties (BRP), however these candidates are put to the tail of the queue. At the same time, candidates with lower BRP scores might allow Searchlight to decrease MRP faster, which improves the overall pruning. To take advantage of this opportunity, instead of using a simple FIFO queue at the Validator, we use a priority queue ranked by the BRP candidate scores. While a priority queue is more expensive than a FIFO queue, in practice performance benefits resulting from better pruning outweigh the queue maintenance costs, as supported by our experiments.

Speculative Relaxation. Before relaxing the query Searchlight runs the *original* query until completion. Only then the recorded fails are replayed, if needed. It is often the case that the original query does not bring any results at all, since the constraints are too tight. Let us assume the user does not mind intermediate results, i.e., the ones that might become obsolete in the future if less relaxed ones are found. However, the first results will come only after the original query is finished. To address this issue Searchlight supports *speculative relaxation* as an option for users. If this mode is enabled, Searchlight runs an additional speculative Solver on each instance to replay fails as soon as they are caught. Technically this Solver is not different from the "original" Solvers — it runs the same model. However, it works only with fail replays and only when Validators are not heavily loaded. If Validators process other candidates, the speculative Solvers sit idle. They are completely turned off when the main Solvers finish their search trees and start replaying fails themselves. This is reasonable, since the idea behind the speculative Solvers is to *inject* some relaxed candidates while the main Solvers are struggling to produce anything. There are two important issues to consider for speculative execution:

- Speculative Solvers do not necessarily result in faster intermediate results. This is because they work with fails found by the main Solvers so far. In case of tight relaxation, when the user specifies tight relaxation limits for the constraints, such fails might be quite close to these limits. In this case the corresponding candidates might be filtered by Validators as violating even the maximally relaxed constraints. Speculative Solvers are opportunistic — if some very promising fail is found during the main search phase, it will be processed quickly and will hopefully produce some results.

- Speculative Solvers introduce a trade-off between possibly receiving some relaxed results faster

and overall query latency. A good practice is to allocate a Solver per each available CPU core. In such a case, when a speculative Solver starts replaying fails, there are no free CPU cores available, so it slows down the main Solvers. According to our experiments this might create a pronounceable latency for the query. However, this is the expected result in the presence of such resource contention. If there are freely available CPU cores, speculative Solvers do not result in any noticeable overhead.

5.2.3 Query Constraining

In contrast with the query relaxation, query constraining deals with the problem of many results. That means query constraining does not need to look at the fails, since it only works with results satisfying the *original* constraints. Actually, the two processes, relaxation and constraining, cannot be active at the same time. After the query begins execution, Searchlight considers the possibility of query relaxation and tracks the fails. If the query produces at least k results (k being the user cardinality requirement for the query), Searchlight turns off the relaxation, stops tracking fails and starts constraining the query to prune subsequent results. The pruning is based on the top-k approach with respect to the specified ranking function (built-in or custom).

Recall that Searchlight explores 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. For a new result to belong to the top-k results its Best possible RanK BRK value must not fall below MRK. Similar to the BRP for the query relaxation, BRK is the maximum possible rank among all possible results in the search sub-tree corresponding to the current search node. Thus, Searchlight puts a new constraint for the query: $BRK(r) \geq MRK$. This is similar to how traditional CP solvers handle optimization process by obtaining progressively better objective function values. There are two important properties of this constraint:

- The constraint is *dynamic*. When a new result is found, it might improve the MRK value. In this case the constraint is refined with respect to the new value to facilitate tighter pruning during the search. Note, the refining does not require the current search to halt or restart. The constraint can be updated at any search node at the moment the new MRK value is received.

- The constraint usually requires the current $[a', b']$ intervals for all functions $f_c, c \in C^c$ to compute BRK. Searchlight obtains these intervals in the same way as described for query relaxation (Section 5.2.1). Searchlight enforces this constraint to be checked only after the main query constraints have been verified. This way all f_c have been computed, the cost of computing BRK is negligent, and the additional constraint does not interfere with possible pruning performed by the main constraints.

As an illustration, consider the running astronomical example with constraints $C_1 : avg(r) \leq 11 \wedge C_2 : avg(g) \leq 10$, where both $avg()$ have domains $[-20, 20]$. We assume the user wants to

maximize both $avg()$ functions. If at the current search node the function's intervals are $avg(r) \in [0, 5]$ and $avg(g) \in [0, 15]$, then $BRK = 1 - (\frac{1}{2}\frac{6}{40} + \frac{1}{2}\frac{0}{40}) = 0.93$ (since the user is maximizing the functions, BRK is based on the right interval boundaries). If $MRK = 0.9$, for example, this node satisfies the constraint. On the other hand, if at some of its descendant the function ranges become $avg(r) \in [0, 3] \wedge avg(g) \in [0, 10]$, then $BRK = 1 - (\frac{1}{2}\frac{8}{40} + \frac{1}{2}\frac{10}{40}) = 0.78$. Thus, the node is pruned.

Searchlight does not introduce any additional constraining techniques at the Solvers. When a leaf is reached, the Solver passes the corresponding candidate solution and its BRK value to the appropriate Validator. The Validator processes the candidates in a similar way as for the query relaxation:

1. A new candidate batch is taken from the Validator's queue.
2. The candidate's BRK value is checked against MRK, in case the MRK value has changed since the candidate was put into the queue. This is similar to the query relaxation case, and provides a very effective and cheap pruning opportunity.
3. If the candidate passes the check it is then validated over the real data. In contrast with the query relaxation, the Validator does not need to relax any constraints — during the query constraining phase candidates must satisfy the *original* query constraints.
4. If the candidate passes the validation, the Validator checks its RK value against MRK. Since $f_c, c \in C^c$ have been computed over real data, the rank is the exact scalar value. If the check fails, the solution is discarded.
5. If the solution passes the check, the new MRK value is computed, and the RK value is broadcasted across the cluster, so that all Searchlight instances can update their own top-k sets and MRK values accordingly.

As can be seen, the query constraining does not require any extensive modifications to the main search process. It does not interfere with the user preferences, if, for example, the user specifies a custom search heuristic or introduces her own functions. As in the case of query relaxation, CP solvers already provide most of the required facilities to incorporate the techniques efficiently. Both relaxation and constraining naturally co-exist during the search process.

5.2.4 Query Constraining via Skylines

Searchlight provides an additional option for the query constraining — the skyline constraint. Supporting skyline does not require any additional changes, although it introduces important differences from the rank-based constraining:

- If the user's cardinality requirement k is not fulfilled (i.e., the skyline contains less than k results), the query will be relaxed as usually. However, relaxed results might violate the skyline property — they are provided just to get the user more info to work with. At the same

time, the skyline size might exceed k quite considerably, which is an important distinction from the rank-based constraining, which guarantees no more than k results. Searchlight does not have any means to compare results belonging to the skyline, since such results (vectors) are incomparable.

- The search-level pruning is still performed. It is not based on the rank, however. Instead the skyline constraint checks the current $f_c, c \in C^c$ values for the possibility of dominating results in the running skyline. This is done as traditional interval-based comparison.
- The checks at Validators are still performed, but, again, against the running skyline instead of the ranks. When a new solution is found, it is broadcasted across the cluster, and all instances update their own running skylines accordingly. This effectively updates the search-level constraint as well.

In general, skyline-based pruning is less effective than the rank-based pruning. That is because the pruning now is based on interval comparisons, for which the rank-based approach in some way is less strict than the domination-based. For example, for a search node to be pruned, it must be dominated by one of the top-k results. That means, the result must dominate *each* of the search node's f_c intervals. If the intervals are wide, which is often the case closer to the root of the search tree, such pruning is not very effective. At the same time, even for such queries we found that Searchlight in many cases provides significant performance gains comparing with running the queries without constraining and computing the skyline at the client.

5.3 Experimental Results

We performed an extensive experimental evaluation of the query relaxation and constraining features in Searchlight. The main part of the evaluation consists of measuring the benefits of using this combined query processing solution against the only thing available to the user — relaxing or constraining by hand. At the same time, we wanted to make sure these features do not bring any significant overhead to the query processing, even in the case of regular queries, which do not required modification. Another important part of the experimental evaluation is to measure the benefits of the various optimizations we described in Section 5.2.2. We do not provide comparison with, for example, the pure SciDB approach. Such a comparison is available in Section 4.5 for different kinds of queries for both synthetic and a real-world data sets,

All experiments were performed in a four-instance Searchlight cluster. The cluster consisted of Linux machines (kernel 3.16) with Intel Q6600 CPUs, 4GB of memory and WD 750GB HDD. We used two data sets. The first one was a synthetic data set, 60GB total size, which is a part of the same synthetic data set we used in Section 4.5. The second data set was a part of the MIMIC II [2] waveform data for the Arterial Blood Pressure (ABP) signal. The data set size was 50GB.

To look at different aspects of our approach we provide most of the results for two characteristic example queries for each data set. By default, we assumed the user's cardinality requirement is 10

results. The queries are as follows:

- **S-SEL** (from Synthetic SELective). If ran without modifications, this is an empty-result query for the synthetic data set. However, being maximally relaxed it becomes a non-empty, but very selective query. We used it to measure the benefits and overhead of our approach in the scenario where the user does not want to relax the query too much. Practically, it is possible for her to run the maximally relaxed version of the query in the first place, however, the user does not know that, since she cannot estimate the result cardinality.
- **S-LOS** (from Synthetic LOoSe). This query follows the same logic, but the maximally relaxed version is very loose, outputs a very large number of results, and does not allow Searchlight to perform much pruning. Thus, this is impractical for the user to run the maximally relaxed version at all. With such a query we measured the performance of Searchlight in cases when the user does not restrict relaxation in any way, for example, due to having little knowledge about the data. We see such cases as the most common and useful for our approach.
- **M-SEL** (from Mimic SELective). This is the MIMIC version of the query S-SEL.
- **M-LOS** (from Mimic LOoSe). This is the MIMIC version of the query S-LOS.

Additionally, we used maximally relaxed versions of these queries to measure the performance of the query constraining in Searchlight. This ensures a very interesting comparison from the user's perspective. If, for example, the user runs the maximally relaxed versions of the loose queries (S-LOS/M-LOS) hoping to get just enough results, she will be actually overwhelmed with the number of them. By constraining such queries, Searchlight should still allow the user to get a number of ranked results without the need to stop the query and look for a more selective version. Thus, it allows us to demonstrate the benefits of the combined approach.

5.3.1 Query Relaxation

In this experiment we measured the benefits of the automatic relaxation provided by Searchlight against the guessing-based approach, when the user would be forced to guess the correct query to get at least 10 results. We looked at the following possible user scenarios:

- **GUESS**. This is a common guessing game scenario. The original query gives an empty answer. Then the user relaxes it in a cautious way, and the new query still does not fulfill the cardinality guarantee. On the third try the user guesses the correct query — the one that outputs at least 10 results and, at the same time, does not output too many of them. We provide results with only one intermediate unsuccessful try, which is enough to demonstrate the benefits of the automatic relaxation. In the real world, the guessing game might go on for some time, since the user has a large number of options: which constraints to relax and how to modify them.
- **ORACLE**. This is the scenario in which the user guesses the query correctly from the first try (after the original query outputs empty result). Since in the real world it would be nearly

impossible for the user to guess such a query without some kind of an oracle, we call this the oracle approach.

- **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 (e.g., for selective queries) or just start outputting a large stream of results without any means of pruning (due to loosely relaxed constraints). In the latter case the user would have to stop the query and guess again, 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 5.1, where the "Auto" column corresponds to the Searchlight auto query relaxation. For the oracle scenario in the parenthesis we specify the completion time of the second, correctly relaxed, query. For these queries the oracle and max user scenarios are basically equivalent since there is no harm in relaxing the query completely.

Table 5.1: S/M-SEL query completion times (secs) for query relaxation.

Query	**Auto**	**Guess**	**Oracle**	**Max**
S-SEL	134	415	288 (165)	288
M-SEL	184	586	397 (245)	397

As can be seen, even comparing with the oracle approach Searchlight provides 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 parts are finished, it can concentrate only on the previously unexplored (i.e., failed) parts of the search tree. This is in contrast with any of the user-level approaches — when the user issues the next, more relaxed, query, Searchlight has to explore the corresponding tree entirely. There is no sharing of information between different queries.

- When relaxing the query Searchlight is able to provide additional pruning based on the best results found so far. When the user submits the "correct" query, this kind of pruning is absent. 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 in this section.

It is as interesting observation that Searchlight performed better even comparing to the correctly relaxed versions of the queries (the parenthesis times for the oracle case). This can be attributed to the distance-based pruning described in the second bullet above.

We also measured the time it takes Searchlight to obtain the first result. For the auto approach it took Searchlight 109 seconds against 125 seconds for the oracle (the best) approach. The corresponding times for the M-SEL query were 41 seconds against 202 seconds. Since Searchlight is aware of the relaxation from the beginning, it generally might find relaxed results before the relaxation phase starts. One notable case is when a candidate solution passes the checks of the original query

at the search level, but fails them at the Validator. In that case the Validator still outputs the relaxed result if it is within specified maximum relaxation bounds.

When it came to the overhead of the query relaxation approach itself, it did not exceed 10 seconds for the synthetic and 3 seconds for the MIMIC query. This overhead came from assessing and recording the fails, both suitable for future replays and discarded at the recording.

Table 5.2 provides the corresponding results for the loose queries. For the "Max" case we did not run the query until completion and stopped it after 1 hour (hence the > symbol in the table), since this was enough to demonstrate our point.

Table 5.2: S/M-LOS query completion times (secs) for query relaxation.

Query	**Auto**	**Guess**	**Oracle**	**Max**
S-LOS	147	378	250 (126)	>3600
M-LOS	80	179	119 (91)	>3600

This experiment shows the same trend as for the selective queries. Searchlight prevails for the same reasons. In contrast with the selective queries, for the loose queries the max scenario does not equal the oracle one. If the user relaxes the query maximally, it will run for a very extended period of time (we stopped after 1 hour). Note, 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. At the same time Searchlight correctly finds the top-k results and prunes the unnecessary parts of the search space at the same time. Since the maximal relaxation is out of question, the user would have to continue the guessing game incurring times potentially much larger than specified for the guess approach in the table.

Searchlight output the first result in 130 seconds for S-LOS and 44 seconds for M-LOS. The corresponding times for the best user scenario, the oracle, were 136 and 78 seconds. This is the same trend as we discussed for the selective queries. As for the auto relaxation overhead, it remained at comparably low levels: 13 seconds for S-LOS and 1 second for M-LOS.

The last experiment of this section measured the overhead of the auto relaxation for the queries that do not need it. Recall that our goal is to keep the relaxation in the "always on" mode, since the user cannot predict if the query will need the relaxation before running it. In this case, however, Searchlight incurs additional overhead coming from the following sources:

- Catching and recording fails. While it is automatically turned off when the required number of results is found, it still might be active for considerable time during the execution.

- Validation overhead for some candidates. This is a more subtle type of the overhead. If Searchlight is not relaxing the query, it might be able to filter out candidates faster, after detecting the first violated constraint. However, when considering relaxation the constraints are relaxed at the Validator as well (with respect to MRP). That makes the constraints more loose, and validations "less strict" — the Validator will admit more candidate solutions than without relaxation. Some early candidates can actually pass such a validation and be output

as relaxed results (to become obsolete later by the exact results). After the required number of results is found, the Validators will not relax constraints anymore and will return to more strict validation dictated by the original constraints.

For this experiment we took the relaxed version of the queries from the oracle scenario and ran them with relaxation turned on. The results are presented in Table 5.3.

Table 5.3: Query completion times (secs) for queries not needing relaxations.

Auto-relax	**S-LOS**	**M-LOS**	**S-SEL**	**M-SEL**
Off	126	91	165	265
On	135	106	186	332

As can be seen, in most cases the overhead was quite manageable. For the synthetic queries the overhead came mostly from maintaining the fail catching and recording. For the M-SEL query it was a mix of both. Actually, the overhead for M-SEL is the largest we have seen for the MIMIC queries. Most of the other queries, including selective ones, had the overhead on the same scale as synthetic queries. Another measure, not included in the table, is the time to get the first result, which is a reasonable measure of the system's responsiveness. For all queries turning on the relaxation did not introduce any noticeable result delays. The auto-relaxation approach resulted in increased delays no more than 1 second. One notable exception was, again, M-SEL, for which the difference was 5 seconds. Despite that, we believe the benefits of keeping the relaxation always on and getting considerable performance improvements for many queries well worth the price of quite manageable overhead.

5.3.2 Query Constraining

In this experiment we measured the benefits of the automatic query constraining provided by Searchlight. Without the automatic constraining the only option available to the user is to run the query until completion and then filter results at the client. While this might work for queries returning a small number of results, it is very inefficient for queries returning a large number of them. Besides that, such a client-based approach misses significant pruning opportunities, which we wanted to demonstrate with this experiment.

The main results for the experiment are presented in the Table 5.4. As for the Table 5.2, we use the > symbol to specify we did not run the query until completion and stopped it after the specified time. By default we specify times in seconds, but we use 'h' and 'm' symbols where appropriate to denote hours and minutes.

Table 5.4: Query completion times (secs) for query constraining.

Auto-relax	**S-LOS**	**M-LOS**	**S-SEL**	**M-SEL**	**M-SEL-ALT**
Off	2h 17m	3h	150	265	314
Rank	74	242	35	258	186
Skyline	31m	45m	126	284	194

As can be seen, the loose S/M-LOS queries cannot even finish in a reasonable amount of time (for M-LOS the time is much greater due to a large search space size). If the user risks running such queries without any support from Searchlight, she will be just overwhelmed with the results without any means of meaningful interpretation. Note, these queries actually were outputting results with very low latencies during the execution. However, since constraints were loose, they created an avalanche of such results without the ability to prune. The user cannot simply stop such queries, since there are no guarantees the top-k ranked answers are among the found results.

At the same time, for the loose queries Searchlight provided considerable performance gains. They came from the pruning both at Solvers and Validators, as we described in Section 5.2.3. This is especially evident for the rank-based constraining. The skyline-based constraining was less effective, when it came to the query completion time. However, comparing with the client-based "Off" approach, the performance benefits were quite 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 — their completion time is reasonable. It can be seen that in most cases the constraining approach of Searchlight provided improvements even in this case. The M-SEL query is somewhat of an exception, for which the skyline approach performed a little 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. In other words, it was somewhat of an anomaly. The last column of the table (M-SEL-ALT) provides results for another selective MIMIC query. It can be seen that both rank and skyline auto approaches provide significant improvements for query completion times, so the M-SEL case should not be considered a trend.

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 fail catching and recording. As for the Solver-level checks, they are quite cheap for the rank-based constraining, since it *is* a very cheap constraint that does not even need access to data, synopsis or real. Moreover, the constraint is added only when Searchlight finds the number of results required by the user. Thus, for queries that do not need constraining (including the relaxation queries) there is zero overhead. The same is true for the Validator-level checks, which are insignificant comparing with the oncoming data accesses required by the validation itself, and are active only when the constraining really is needed. 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. This problem is well-researched in the literature, e.g., for relational skyline queries. We see the skyline queries overhead not as the cost of the constraining approach, but rather as the logical cost of the skyline constraint itself, which is not trivial.

5.3.3 Query Relaxation Optimizations

In this section we describe experiments we performed to measure the effect of the query relaxation optimizations described in Section 5.2.2 on query performance.

Computing UDFs at fails. In this experiment we measured the difference between the two different strategies to compute UDFs when catching fails. The first strategy, "Full", enforces computation of all function values to record the search state at the fail, even for functions that have not been computed (i.e., due to an earlier constrain violation). The rationale behind this strategy is that Searchlight can estimate the relaxation penalty more accurately. Another strategy, "Known", just considers the already computed UDF values, and assumes the other function values to be unknown (i.e., $-\infty \leq f_c() \leq +\infty$). This strategy has the benefit of lower overhead during fail recording, but at the same time might provide less information about the fails, resulting in less accurate penalty estimations and an increased number of fail replays. At the same time, we wanted to study the effect of this optimization on interactive times, in particular the time to obtain the first result of the query.

The results are presented in Table 5.5 (query completion times) and Table 5.6 (time to the first result). As can be seen from the table, for expensive synthetic queries the optimization resulted in significant benefits, both for the completion and interactive times[6] For less expensive MIMIC queries the benefits were not pronounced. At the same time, the strategy did not introduce any overhead either. We also ran additional MIMIC experiments for more expensive queries. We took the same M-SEL/LOS queries and increased their cardinality requirements from 10 to 200 results, which resulted in more extensive search. We saw the significant benefits at the fail recording stage: for some queries the overhead went down from 30 to 15 seconds. At the same time it did not have any effect at the times to the first result. For those queries first results were relatively easy to find, and at that point the overhead was not large (it was accumulating during the whole query). As for the completion times, those queries resulted in replaying most of the recorded fails, thus rendering the benefits of the recording stage computation savings mostly unnecessary.

Table 5.5: Query completion times (secs) for fail recording methods.

Record method	**S-LOS**	**M-LOS**	**S-SEL**	**M-SEL**
Full	165	81	161	183
Known	147	80	134	184

Table 5.6: Time to the first result (secs) for fail recording methods.

Record method	**S-LOS**	**M-LOS**	**S-SEL**	**M-SEL**
Full	150	43	128	41
Known	130	42	108	41

Saving UDF states at fail recording. In this experiment we measured the impact of saving

[6]While times to the first result might seem large, they include the completion time of the *original* query, which found no results at all.

known UDF states for recorded fails. In contrast with the previous optimization, which just postpones computation of UDFs whenever possible, similar to lazy evaluation, this optimization should be able to avoid re-computation of the saved UDF values completely. Logically speaking, this should have impact on both interactive and completion times, which we verified with this experiment. The results are presented in Table 5.7 (query completion times) and Table 5.8 (times to first result).

Table 5.7: Query completion times (secs) for the UDF state saving optimization.

UDF saving	**S-LOS**	**M-LOS**	**S-SEL**	**M-SEL**
On	147	80	134	184
Off	238	104	137	186

Table 5.8: Times to first result (sec) for the UDF state saving optimization.

Record method	**S-LOS**	**M-LOS**	**S-SEL**	**M-SEL**
On	130	42	108	41
Off	163	65	107	41

As can be seen from the results, the saving optimization was beneficial for most queries. For the selective M/S-SEL queries the benefits were not pronounced due to the structure of those queries. The replays were relatively cheap, involving less re-computation (which, for instance, depends on the number of constraints violated at the fail). The trend was the same for the times to the first result. To conclude, this optimization in many cases brings significant improvements in both completion in interactive times at virtually no CPU cost. 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, if the values correspond to particular coordinates in the array).

Sorting the Validator queue on BRP. This optimization tries to steer the Validator checks in the direction of promising candidates by sorting the candidate queue on the Best Relaxation Penalty (BRP) value for the candidate. It should give significant benefits for queries with large numbers of candidates. When the number of candidates is small or when most candidates contribute to the final result (i.e., selective queries) the benefits should be small to non-existent. When we ran these experiments, the original queries with the cardinality requirement $k = 10$ did not show any differences in times depending on the sorting method. This can be seen for the M-LOS/SEL queries in Table 5.9. Thus, we re-ran the experiment for the same queries with increased cardinality parameter. Those results can be seen as M-LOS-200 and M-SEL-200 ($k = 200$) in the same table. For the loose query (M-LOS-200) the benefit for the query completion time was obvious. Internally, we saw a significant decrease in the total number of candidates validated. This was due to a faster establishment of the Maximum Relaxation Penalty (MRP), which serves as the cut-off point for the potential top-k relaxed candidates. For the selective queries there were no visible benefits, which was expected as discussed above. We did not see any changes in the times to the first result. Because of that we do not provide the times here. However, due to decrease in the query completion times the delays

between intermediate results are bound to become smaller in general.

Table 5.9: Query completion times (secs) for the *BRP*-based Validator queue sorting.

Queue sorting	**M-LOS**	**M-LOS-200**	**M-SEL**	**M-SEL-200**
BRP	80	290	184	277
No-sort	81	331	183	270

Speculative execution. As can be seen from the previous experiment, often the time to find the first result is quite large. This is a logical result for the empty-result queries, since Searchlight first finishes the main, non-relaxed, query and only then tries relaxing it. So the completion time of the main query is always added to the first result time of the relaxed query. To try to remedy that Searchlight offers speculative relaxation, the goal of which is to start replaying fails before the main query finishes. This might improve the times to the first result, but might increase query completion times, since it requires additional Solvers and, thus, consumes more CPU resources. The results for the running queries are presented in Table 5.10 (query completion times) and Table 5.11 (times to the first result).

Table 5.10: Query completion times (secs) for speculative execution.

Speculative execution	**S-LOS**	**M-LOS**	**S-SEL**	**M-SEL**
On	186	94	145	185
Off	147	80	134	184

Table 5.11: Time to first result (secs) for speculative execution.

Speculative execution	**S-LOS**	**M-LOS**	**S-SEL**	**M-SEL**
On	18	45	2	42
Off	130	42	108	41

As can be seen from the results in many cases speculative execution significantly improved times to the first result. Unfortunately, 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. Note, the speculative Solver still works in the maximum relaxation boundaries, so its search tree and candidates are still subject to pruning. Moreover, the speculative Solver can only replay fails found so far by the main Solvers, which might be low quality when it comes to their best relaxation penalty. It just does not have enough information to build a search tree leading to promising candidates! Our next experiment tries to investigate this issue in more detail (see below). We still find the speculative execution a quite promising strategy for a large variety of queries.

As expected, the speculative relaxation has its own overhead coming from the additional consumption of CPU resources by the speculative Solver. This can be seen in Table 5.10. For some queries the increase in the completion time was significant despite Searchlight following a conservative approach of running the speculative Solver only at the main query stage and only while the

Validator was idle. We decided to run an additional experiment (not shown here), where we had one additional CPU thread available. As expected, the times for the speculative relaxation turned on and off were the same, which proves the overhead is CPU related and cannot be trivially extinguished. To conclude, we assume it is up to the user to decide if she wishes to exchange a portion of the total query execution time for a possibility of faster results to work with.

Sorting recorded fails by BRP. While this is not strictly an experiment exploring one of the optimizations, it still allows us to demonstrate the benefits of sorting recorded fails in the Best Relaxation Penalty (BRP) order. We compare our default BRP-based recording with the time-bases sorting, for which the fails were just replayed in the order encountered. This experiment also touches upon an interesting question: would there be any benefits if we just ignored the fails and continued exploring failed search sub-trees immediately, i.e., without first recording and then replaying the fails? While in our implementation it was impossible to implement such an approach due to the Or-Tools architecture (at least without very significant changes to the CP engine), it might still be a possibility with other CP implementations. We provide the results of the experiment in Table 5.12 (query completion times) and Table 5.13 (times to the first result).

Table 5.12: Query completion times (secs) for different fail orderings at Solver.

Sorting	**S-LOS**	**M-LOS**	**M-LOS-200**	**S-SEL**	**M-SEL**
BRP	147	80	290	134	184
Time	14m13s	80	351	144	194

Table 5.13: Time to first result (secs) for different fail orderings at Solver.

Sorting	**S-LOS**	**M-LOS**	**M-LOS-200**	**S-SEL**	**M-SEL**
BRP	130	42	42	108	41
Time	193	43	44	109	41

As can be seen from the results, for some queries the time-based sorting might have quite detrimental effect on the performance. To provide more insight for the MIMIC queries we ran M-LOS not only with the $k = 10$, but also with $k = 200$ (the M-LOS-200 query). The result was similar to the S-LOS query — a significant increase of the completion time. The time to find the first result did not change significantly for most queries, however, the result for S-LOS hinted at the possibility. S-LOS, actually, is a good example of a query for which the natural, time-based, ordering of nodes in the search tree diverges from the optimal traversal significantly. While it might be remedied by a suitable search heuristic, such a situation might be hard to predict before the query begins, and a suitable heuristic might be hard to find. On the other hand, BRP-based sorting can be seen as a kind of generally beneficial search heuristic, which steers the search tree traversal in the direction of promising relaxed candidates.

Partial relaxation during replays. This optimization addresses the issue of over-relaxing the query at a fail replay. Each fail has its own maximum relaxation based on the current MRP value, the number of violated constraints and UDF values at the moment of fail. The RRD parameter

($0 \leq RRD \leq 1$) controls the relative length of relaxation for each constraint with respect to the maximum relaxation interval for this constraint at the replay. The main idea is to prevent the Solver from building a potentially large search tree and try to find candidates with better relaxation penalty values. We provide query completion times for the two running loose queries in Table 5.14. We did not see any noticeable effect on the times to the first result.

Table 5.14: Query completion times (secs) for different RRD values.

Query	$RRD = 0.1$	$RRD = 0.3$	$RRD = 0.5$	$RRD = 0.7$	$RRD = 1.0$
S-LOS	146	147	143	149	146
M-LOS	75	80	127	289	28m

As can be seen from the results, for some queries the optimization does not have any effect. These are the queries for which the maximum relaxation intervals at the replays are relatively small already. In this case the optimization does not result in any overhead either. We saw a slightly elevated number of fail replays, but not significant enough to cause any drop in performance. Query M-LOS, on the other hand, is an example of a query for which some of the fails have very large relaxation intervals. This commonly happens when the original non-relaxed query fails relatively fast, in which case the fails correspond to "coarse", close to the root, nodes of the search tree. Relaxing them (near) maximally causes the Solver to build and traverse a large search tree, as was the case with the M-LOS query. We believe the RRD optimization can serve as a fail-safe for such situations.

To sum up the relaxation optimization results, we should say that most optimizations target particular issues that arise when the query relaxation is performed. The evaluation clearly shows the optimizations address the issues effectively for many cases, and at the same time do not result in any significant overhead for queries that do not require them. At the same time, relaxation penalty based optimizations can be seen as part of a search heuristic that steers the search in the direction of promising relaxed results. As any heuristic it produces visible benefits for some queries and struggles for other ones. In the latter case, however, the heuristic did not introduce any visible overhead either, which we consider to be a positive outcome.

Chapter 6

Conclusion

In this chapter we discuss briefly our findings, lessons learned and possible future work directions. The discussion covers three main parts of this work: the Semantic Windows framework, Searchlight and query relaxation/constraining for search queries.

6.1 Semantic Windows

We presented the Semantic Windows (SW) framework as the first step towards easy-to-use, interactive approach for human-in-the-loop exploratory analysis. SW allows users to conveniently perform structured search via shape and content constraints over a multi-dimensional data space.

The framework uses a data-driven search algorithm coupled with a variety of complementary techniques, including stratified sampling, adaptive prefetching and sophisticated data placement, to search the underlying data space quickly while providing online results. We described a prototype implementation of the framework as a distributed layer on top of PostgreSQL and conducted an experimental evaluation with real and artificial data to study the impact of various design and algorithmic decisions. The results showed that SW significantly improves online performance, relative to representative state of the art solutions. SW is not always the ideal solution when it comes to the query *completion* time, however. For some queries the traditional database approach (i.e., via a complex SQL query) might result in better query completion times. This can be seen as a benefit of batch computing — performing a sequential scan of the data with outputting results at the end. With this approach the user receives first results only when the query completes, which introduces considerable result latency. In such cases SW still provides superior interactive performance. At the same time, even the loss in the query completion time can be almost entirely eliminated or significantly reduced via the adaptive prefetching.

Despite the very promising results, we found SW approach lacking efficiency along the following directions, which we also discussed in Section 1.2:

- SW was specialized for semantic windows queries, which constitute only a small part of the

"First-order" query class discussed in Section 1.1. Users could not easily add more complex constraints. For example, to explore differences between windows and their neighborhoods, which is very useful for exploring anomalies in data. This restriction was mostly due to the custom nature of the solver, which could not easily accommodate extensions of this kind. Thus, a more general and extensible solution was required.

- SW had to eventually read the whole data set. It used sampling to steer the search in the required direction and provide results quickly. Sampling in general, however, does not give 100% guarantees. Thus, it did not allow SW to perform sophisticated pruning for the search space and data. As we noted in the experimental evaluation, even when all results had been found and output SW still had to confirm this by reading the remaining data. This made SW less suitable for large data sets. It was infeasible to execute search queries without extensively pruning parts of the data that could not contain results.

- SW did not perform balancing of any kind. While initially the search space and the data was statically distributed among the workers, if a worker became idle, it remained idle until query completion. The only way for the user to remedy this was to carefully perform pre-query distribution by hand, taking possible data skew into consideration. Obviously, such an approach was not feasible in practice. Ideally, there should have been dynamic balancing of both the search space and the data between multiple workers depending on the query progress.

6.2 Searchlight

The SW deficiencies discussed in the previous section spurred research and development of a new kind of framework that uniquely integrates constraint solving and data management techniques by allowing Constraint Programming (CP) machinery to run efficiently inside a DBMS without the need to extract, transform and move the data. We called this framework *Searchlight* [28].

Searchlight facilitates the application of modern constraint-based search methods to large data sets, while taking advantage of the data management capabilities in a modern array DBMS. It uses sophisticated techniques that combine speculative execution over a synopsis with the validation over the original data to provide interactive performance. It supports distributed computation and employs data- and search-space balancing techniques. Experimental results over real and artificial data sets showed the remarkable speedups that are possible over the state-of-the-art alternatives for data- and search-intensive queries, for both interactive and total performance.

At the same time this approach is surprisingly extensible. As we discussed in Section 4.3.2, adding new types of constraints (e.g., distance-based similarity matching for time sequences) does not require changing not only the architecture, but even the search engine. In the worst case, it is just a simple matter of adding a new synopsis type, which is similar to adding a new index type in a traditional query execution engine without changing the query execution mechanics.

In summary, Searchlight makes the following scientific contributions:

- **Constraint solvers as first-class citizens** Instead of treating solver logic as a black-box, Searchlight provides native support, incorporating the necessary APIs for its specification and transparent execution as part of query plans, as well as novel algorithms for its optimized execution and parallelization.

- **Speculative solving** Existing solvers assume that the entire data set is main-memory resident. Searchlight uses an innovative two stage *Solve-Validate* approach that allows it to operate speculatively yet safely on main-memory synopses, quickly producing candidate search results that can later be efficiently validated on real data.

- **Computation and I/O load balancing** As CP solver logic can be computationally expensive. Executing it on large search and data spaces requires novel CPU-I/O balancing approaches when performing load and data distribution.

In this part of the research we have explored the fusion between the constraint-based CP search engine and the computation-oriented traditional DBMS engine. At the same time there are directions for more in-depth research of both the CP and DBMS sides of this collaboration, which we kept as future work:

- Searchlight still requires the user to specify the search heuristic to use. In many cases an out-of-box heuristic works well. For example, the traditional split heuristic showed great performance for most queries we studied in the experimental evaluation. For some queries, however, a custom heuristic, like the probe-based one discussed in Section 4.2.3, might be more suitable. It is worth to study the possibility of automatic heuristic choice, based on the user's query. As in traditional DBMSs, this choice might be at least initially dictated by the statistics collected about the data already available from previous queries. At the same time the heuristic might be modified during the query execution when the new information becomes available. This is similar to adaptive query planning. Since Solvers are isolated entities when it comes to exploring their parts of the search tree, it is even possible to run different heuristics at different Solvers, which further improves adaptability.

- At this point we consider Searchlight-only queries, where the query is basically a CP specification with all data access restricted to Searchlight functions. It is beneficial to study more complex Searchlight-DBMS queries, especially when it comes to the derived data, e.g., when the data for the search is produced by another sub-query. In this case the query can be logically seen as consisting of two parts: the search part (Searchlight) and the data producing part (DBMS). There are great optimization opportunities between the two. Among them are the following:

  - Some constraints can be pushed down from the Searchlight part to the DBMS engine to possibly prune some data from consideration. This is similar to the predicate push-down optimization in traditional DBMSs, albeit with additional issues. There is, for example, a

question of the constraint types the DBMS can benefit from. Moreover,the predicates for pushing down might need to be *derived* from the constraints, since the objects of search are more complex (e.g., regions instead of tuples).

  - The exchange of constraints can go both ways, where some DBMS-level predicates are pushed-up to the Searchlight part of the query as new constraints. This might help the CP engine to decrease the search space size, for instance. The issue can be explored further, when the DBMS engine collects some statistics about the data during its creation and provides it to Searchlight. The statistics would depend on the search part of the query.
  - Synopsis can be created adaptively. For example, a coarse-resolution synopsis can be created at the DBMS concurrently with producing the data. The finer resolution synopses can be built at Searchlight during the search process for the data parts that need deeper exploration.

- At the present time Searchlight considers only "monolithic" search queries, where the search space and constraints define a single query specification. Basically, Searchlight can be seen as a single database query operator. It might be beneficial to explore "higher-order" queries, with multiple Searchlight operators. One notable example is similarity queries, e.g., "find two similar rectangular regions". This query implicitly contains two search sub-queries (for the first and second region) and a"join" between them (the similarity constraint). Creating different Searchlight operators for sub-queries enables different search strategies for each sub-query. In this case there is also a possibility of handling the join the DBMS itself, which puts us again into the realm of cross Searchlight-DBMS query optimization as discussed above.

6.3 Query Relaxation and Constraining

When it comes to data exploration, there is an important issue of better user support, since the users might have limited knowledge of the data. This constitutes the third major part of this work: query relaxation and constraining. Without extensive knowledge of the data it is hard for the user to assess the kind of result she is going to get. We studied the two common manifestations of this problem: the cases of too few and too many results. The first case causes the user to initiate a "guessing game" with the system — trying to guess a query that would bring enough results. The second case makes interpreting and analyzing results much harder, in which case the user has to again guess the "correct" query or rank-and-filter results after the query has been completed. At the same time Searchlight collects extensive knowledge about the data during the query progression, which can be used to either relax the constraints and bring more results (query relaxation), or to constrain the query further and output only the top results depending on the user-defined ranking function (query constraining).

Our research showed that query relaxation and constraining can be incorporated in Searchlight in an efficient and effective way. The very essence of these techniques is manipulating query constraints.

Query relaxation requires modification of the original query constraints, while query constraining requires adding new ones. Since Searchlight is a constraint-based system, which treats constraints as first-class citizens, these techniques naturally fit into its CP query execution engine. By collecting information about the query execution while it is running, Searchlight is able to make data-driven decisions about which constraints to relax and to what degree. At the same time, it is able to detect the point when the number of results becomes too large, and can add additional constraints during the search to efficiently prune results that are inferior to already found. In other words, Searchlight makes query relaxation and constraining an integral part of query processing, which allows it to explore the same optimization capabilities (e.g., pruning, speculative execution over synopses, steering the search in a specific direction) as for regular queries. This ensures the techniques can perform in the most efficient way possible during the query processing.

Our extensive experimental evaluation supports the claim. For both synthetic and real data sets Searchlight provides superior performance comparing with the only alternative available to the user — guessing the query. Moreover, even if the user could somehow guess the correct query from the first try, creating the ideal scenario, Searchlight is still able to match the performance in both interactive and query completion times.

Query relaxation and constraining might not be the only means of improving the user experience for data exploration. We believe they may be part of a more general idea of manipulating query constraints depending on the current state of the search. For Semantic Windows (SW), for example, we discussed the idea of diversifying results in Section 3.4.4. Sometimes we want to avoid being stuck at the same part of the search space. It might be desirable to "jump" to another part, thus possibly bringing results from different parts of the data set faster. This can also be seen as a type of constraints manipulating the results. The constraints might involve the search space itself (e.g., "exclude this part of the data from search") or the semantic properties of the data (e.g., "no more results from the [0, 5] average r-magnitude range"). Such constraints can be seamlessly incorporated into Searchlight in the same way as has been done for query relaxation and constraining. It should be relatively easy to give the user a significant control over creating such constraints in a way similar to UDFs, creating a very extensible run-time query modification framework. Defining such a framework in terms of API and researching the necessary query engine optimizations related to it is another very promising direction of future work.

Bibliography

[1] Google or-tools. https://code.google.com/p/or-tools/.

[2] Mimic ii dataset. https://mimic.physionet.org/.

[3] Scidb. http://www.scidb.org/.

[4] The sloan digital sky survey (sdss). http://www.sdss.org/.

[5] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, pages 487–498, 2000.

[6] Chuan-Heng Ang, Tok Wang Ling, and Xiao Ming Zhou. Qualitative spatial relationships representation io&t and its retrieval. In *DEXA*, pages 270–279, 1998.

[7] Walid G. Aref and Hanan Samet. Efficient processing of window queries in the pyramid data structure. In *PODS*, pages 265–272, 1990.

[8] Daniel BarbarÃą and Xintao Wu. Supporting online queries in rolap. In *Data Warehousing and Knowledge Discovery*, volume 1874, pages 234–243. 2000.

[9] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[10] Paul G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.

[11] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, June 2002.

[12] Michael J. Carey and Donald Kossmann. On saying "enough already!" in sql. *SIGMOD Rec.*, 26(2):219–230, June 1997.

[13] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The onion technique: Indexing for linear optimization queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 391–402, 2000.

[14] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM TODS*, 32(2), June 2007.

[15] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *Fifteenth International Conference on Principles and Practice of Constraint Programming*, pages 226–241, 2009.

[16] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.

[17] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[18] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[19] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[20] Dina Q. Goldin and Paris C. Kanellakis. On similarity queries for time-series data: Constraint specification and implementation. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 137–153, 1995.

[21] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.

[22] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. *SIGMOD Rec.*, 28(2):287–298, June 1999.

[23] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[24] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997.

[25] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

[26] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, pages 259–270, 2001.

[27] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. Interactive data exploration using semantic windows. In *SIGMOD*, pages 505–516, 2014.

[28] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. In *submission (available on demand)*, 2015.

[29] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. Interactive search and exploration of waveform data with searchlight. In *SIGMOD*, pages 2105–2108, 2016.

[30] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 3(3):263–286, 2001.

[31] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[32] Iosif Lazaridis and Sharad Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412, 2001.

[33] L. Lins, J.T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *Visualization and Computer Graphics, IEEE Transactions on*, pages 2456–2465, 2013.

[34] Gang Luo. Efficient detection of empty-result queries. In *VLDB*, pages 1015–1025, 2006.

[35] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Parallelizing constraint programs transparently. In *Principles and Practice of Constraint Programming âĂŞ CP 2007*, volume 4741, pages 514–528. 2007.

[36] Chaitanya Mishra and Nick Koudas. Interactive query refinement. In *EDBT*, pages 862–873, 2009.

[37] Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. A probabilistic optimization framework for the empty-answer problem. *VLDB*, 6(14):1762–1773, 2013.

[38] Ion Muslea. Machine learning for online query relaxation. In *SIGKDD*, pages 246–255, 2004.

[39] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD*, pages 13–24, 1998.

[40] Frank Olken and Doron Rotem. Random sampling from database files: a survey. In *Proceedings of the 5th international conference on Statistical and Scientific Database Management*, pages 92–111, 1990.

[41] Dimitris Papadias, Nikos Karacapilidis, and Dinos Arkoumanis. Processing fuzzy spatial queries: A configuration similarity approach. *International Journal of Geographic Information Science*, 13:93–118, 1998.

[42] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, March 2005.

[43] I. Popivanov and R. J. Miller. Similarity search over time-series data using wavelets. In *ICDE*, pages 212–221, 2002.

[44] Venkatesh Raghavan and Elke A. Rundensteiner. Progressive result generation for multi-criteria decision support queries. In *ICDE*, pages 733–744, 2010.

[45] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier Science, 2006.

[46] Jean-Charles RÃľgin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *Principles and Practice of Constraint Programming*, volume 8124, pages 596–610. 2013.

[47] Sunita Sarawagi, Rakesh Agrawal, and Nimrod Megiddo. Discovery-driven exploration of olap data cubes. In *EDBT*, pages 168–182, 1998.

[48] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice Hall, 2003.

[49] Jin Shieh and Eamonn Keogh. isax: Indexing and mining terabyte sized time series. In *KDD*, pages 623–631, 2008.

[50] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: Shrinking the petacube. In *SIGMOD*, pages 464–475, 2002.

[51] S. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104 – 119, 1975.

[52] Man Lung Yiu and Nikos Mamoulis. Multi-dimensional top-k dominating queries. *The VLDB Journal*, 18(3):695–718, 2009.

[53] Zhen Zhang, Seung-won Hwang, Kevin Chen-Chuan Chang, Min Wang, Christian A. Lang, and Yuan-chi Chang. Boolean + ranking: Querying a database by k-constrained optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 359–370, 2006.