Searchlight: Enabling Integrated Search and Exploration over Large Multidimensional Data

Alexander Kalinin
Brown University
akalinin@cs.brown.edu

Ugur Cetintemel
Brown University
ugur@cs.brown.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

We present a new system, called *Searchlight*, that uniquely integrates constraint solving and data management techniques. It allows Constraint Programming (CP) machinery to run efficiently inside a 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 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.

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 search distribution.

We built a prototype implementation of Searchlight on Google's Or-Tools, an open-source suite of operations research tools, and the array DBMS SciDB. Extensive experimental results show that Searchlight often performs orders of magnitude faster than the next best approach (SciDB-only or CP-solver-only) in terms of end response time and time to first result.

Proceedings of the VLDB Endowment, Vol. 8, No. 10

1. INTRODUCTION

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 solutions, typically by gluing together existing libraries, databases and custom scripts, only to end up with a solution that is difficult to scale, optimize, maintain and reuse.

Our system, Searchlight, addresses these limitations by uniquely combining the ability of modern array-based DBMSs to store and query data at scale with the rich expressiveness and efficiency of modern CP solvers to search and optimize over large search spaces.

Exploration via search queries In data exploration, users often have 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 SDSS [2], a data set containing information about celestial objects. An object of interest might be a rectangular region in the sky (with coordinates corresponding to right ascension and declination) with the properties including average/min/max star magnitudes within the region, the region area, 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". The region dimensions are *not* fixed and can take any value within the given range. The search can include more advanced properties; e.g., "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".

Perhaps surprisingly, traditional DBMSs offer very limited support for search queries, even for the most basic first-order ones, as we extensively argued in [13]. 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. For example, SciDB has a powerful 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 filtering them. 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* window of size 10x10 would result in exploring approximately 10^8 windows and computing one or more aggregates for each of them. Looking for flexible-size windows significantly exacerbates the problem (Section 5). 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.

These limitations motivate the use of an integrated solver functionality within a DBMS. Such functionality not only allows us to perform search efficiently, but also offers a single, expressive framework that can support a wide variety of data-intensive search and exploration tasks.

Through CP, Searchlight can support many other common types of search, such as similarity queries over time-series data (e.g., stock trading data, medical measurements). Such queries allow users to look for a time interval (i.e., a temporal region) satisfying the specified "similarity distance" and possibly other constraints. While we focus only on aggregate search queries here, Searchlight and its underlying techniques are general and apply to a broad range of search tasks. We make additional comments about extensions for time-series data in Section 3.2.3.

Why Constraint Programming for Search? While DBMSs struggle with queries such as Q1-Q3, these can be compactly expressed and efficiently answered by the CP approach [21]. In CP, users first specify *decision variables* over some domains, defining the search space. They then specify *constraints* between the variables, e.g., algebraic, "all variables must have different values", etc. Finally, a CP *solver* finds *solutions*, the variable assignments satisfying the constraints. For Q1, the 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 region 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 way. As for optimization queries, such as Q3, CP solvers 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, they incorporate a collection of sophisticated optimizations including pruning, symmetry breaking, etc. The first is a core technique that safely and quickly removes parts of the search space that cannot contain any results.
- **Interactivity**: CP solvers are designed to identify individual solutions fast and incrementally. Users can pause or stop the search, or ask for more solutions. This is fundamentally different from conventional query optimization that aims to minimize the completion time of queries.
- **Extensibility**: CP solvers are designed to be modular and extensible with new constraint types and functions. Moreover, users can introduce their own *search heuristics* that govern the search process.

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. Our experimental results indeed show that solvers become unacceptably slow when operating over large, out-of-core data sets (Section 5.1).

CP solvers are commonly used for NP-hard search problems, as they go beyond straightforward enumeration and can effectively navigate through 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 Overview 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 with very little modification. Similarly, the underlying DBMS engine will not require major modifications. This allows Searchlight to gracefully and automatically evolve as the underlying systems evolve. Moreover, the same framework can be applied to compatible systems.
- **Optimized data-intensive search and exploration**: Searchlight provides optimized execution strategies for CP, 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 solvers 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, producing *candidate* results. A synopsis is a condensed representation of the data, containing information sufficient to perform pruning and to verify query constraints. The candidates are guaranteed to contain all real results, but possibly include false positives. Validators efficiently check the candidates over the 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; 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.

Searchlight can also transparently parallelize query execution across 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.

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.

Paper layout. The rest of the paper is organized as follows. Section 2 gives a survey of constraint programming and discusses how data exploration is expressed in Searchlight. Section 3 describes the two-level search query processing. Section 4 discusses distributed search. Section 5 contains experimental evaluation. Section 6 discusses related work. Section 7 concludes the paper.

2. DBMS-INTEGRATED SEARCH

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

2.1 CP Background and Or-Tools

Let us revisit the astronomy example from the Introduction. 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 these are: $x \in [0, 10], y \in [20, 50], lx \in [2, 5], ly \in [3, 10]$. 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 by 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 subarray (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* of 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 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 current *search heuristic* of the solver, which can be specified by the user. A search heuristic typically first chooses an unbound variable. The choice can be based on the size of its domain, its minimum/maximum value or made 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 $x = v$, in which case the right branch becomes $x \neq v$. 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 a solution.

After the solver chooses a branch, and the variable's 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*, since no solutions are possible in the sub-tree corresponding 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 explores the branch. Otherwise, the search ends, since no alternatives are possible. If the solver reaches a leaf, the corresponding assignment is a solution, since all constraints are satisfied. If the user wants to obtain 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 the CP solver logic can be found in the related books [21].

In Searchlight we use the backtracking CP solver from Google's Or-Tools [1], which follows the execution model described above. Or-Tools is highly customizable. Users can add new constraints, functions, define their own search heuristics, perform nested search 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 control the search (e.g., search restarts, switching branches, etc.).

2.2 Search Queries

Searchlight supports any type of queries a typical CP solver can process. However, to access the DBMS data users have to use User-Defined Functions (UDFs). Revisiting the astronomy example, in the constraint $avg() < 12$, $avg()$ is a UDF, since it requires accessing 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, UDFs are not able to return single values. A UDF usually returns an interval $[m, M]$ containing all possible values of the function for all currently possible assignments of the variables. Such an interval is sufficient to check constraints and attempt pruning at internal nodes. For the example constraint $avg() < 12$, the interval $13 \leq avg() \leq 17$ would allow the solver to prune the current sub-tree, however $11 \leq avg() \leq 17$ would not.

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 ops 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. 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.

Let us describe briefly how the UDF $A = avg(x, y, lx, ly, r)$ could be implemented. A returns interval $[m, M]$ containing all possible values of $avg(r)$ 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 (M) is equal to the minimum (maximum) value 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, their average values can be computed with $agg()$ calls, and m (M) will be equal to the minimum (maximum) value just computed.

The search is guided by a search heuristic. Searchlight supports Or-Tools heuristics (e.g., random, impact, split, etc.) without any modifications. It can also be extended with new ones. This allows users to customize search for particular problems, which is common in CP. New 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 evaluating the impact of the heuristic's decisions.

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

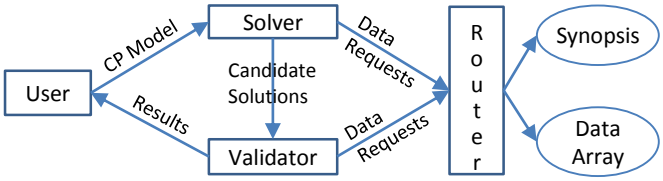

Figure 1: Two-level search query processing.

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 might call UDFs many times during the search, since 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 5.1.

Searchlight uses two-level query processing instead, which combines *speculative execution* over a *synopsis* of the array with *validating* the results. This architecture is aimed at:

- **Interactivity**: Searchlight should start outputting results quickly and minimize delays between subsequent 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 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 the CP solver, e.g., constraints, heuristics, etc. Searchlight does not modify the Or-Tools engine. Instead, it uses the customization features available in Or-Tools to merge it with the DBMS.

3.1 Two-level Query Processing

The two-level query processing is illustrated in Figure 1. When the user submits a search query in the CP form, Searchlight starts the CP *Solver* for processing it. As we discussed in Section 2.2, a search query accessing the 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 used to approximately answer the 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 are expected to return intervals as well, this does not complicate their implementation.

While the Solver processes the model it produces *candidate solutions* (*candidates*). A candidate is a CP solution

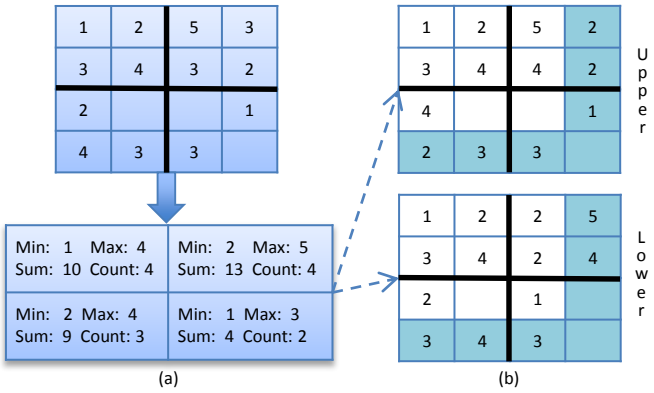

Figure 2: Synopsis example. (a) 2x2 synopsis grid. (b) Upper and lower bound arrays for the $avg()$ of the highlighted region (in white).

(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.

The Solver sends each 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. This heuristic simply assigns the values from the candidate to the variables. Since at the Validator all API calls are answered using the original data, this results in proper validation of query constraints. Thus, if the Validator's solver fails, the candidate is a false positive.

The validation is CP-based for the sake of generality. Since the Validator clones the model from the Solver, it does not assume anything about the constraints. New constraints, UDFs and API calls can be added without modifying the Validator. The two-level processing is completely transparent to users. They call the API from UDFs using the Router, which directs the calls to the appropriate data.

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.

3.2 Synopsis

Synopsis is a lossy compression of data, which is used to give approximate answers to the API calls in form of intervals guaranteed to contain the real result. We assume it fits into memory, so the Solver, which accesses it during the search, can execute its model efficiently. During distributed processing, when the Solvers operate on parts of the search space, the synopsis might partially reside on disk.

Different API calls might require different synopsis types, which we discuss in Section 3.2.3. Searchlight can use multiple synopses in a single query, if required by constraints (API calls). Since this paper concentrates on aggregates, the main discussion follows the aggregate grid synopsis.

3.2.1 Synopsis for Aggregate Estimations

An example of the synopsis is presented in Figure 2(a). The original 4x4 array is divided into four 2x2 *synopsis cells*. For each cell we keep information needed to answer the API calls. For aggregates these are min/max, the sum and count of all elements. Cells might store other information, e.g., the distribution of a cell, bitmaps for sparse cells, etc. The synopsis is a 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*.

The synopsis provides answers to the API calls in form of intervals $[m, M]$, guaranteed to contain the exact value. For the $elem()$ call, m (M) is simply the min (max) value stored in the corresponding cell. The $agg()$ call is different.

Let us assume the user calls $A = avg(R)$, where R is the white rectangular region from Figure 2(b). R intersects 3 out of 4 synopsis cells partially. Since cells might correspond to different distributions, this implies multiple possible values of A. The main idea is to find the distributions that reach the lower bound m and the upper bound M, which is illustrated in Figure 2(b). Both arrays might have produced the synopsis. Their A values are 2.125 and 3.286, so the call will return $[2.125, 3.286]$. For the original array, $A = 2.857$.

Estimating bounds for aggregates over the synopsis is done in a similar way as over Multi-Resolution Aggregate (MRA) tree structures [14]. For example, to compute the interval for $avg()$, the upper (lower) bound array must contain as many high (small) value elements as possible to increase (decrease) the average without violating the synopsis information. Possible elements from all cells are considered in the descending (ascending) order and added one by one until the average cannot increase (decrease).

The detailed algorithm with proofs can be found in the MRA paper [14]. For an MRA tree (e.g., an R- or quad-tree) each intersection of the query region with an MRA node might contain any number of objects referenced by the node. However, for arrays the minimum and maximum number of elements in the intersection is known. Thus, for each cell there is the minimum and maximum number of elements that must be included in the estimation. We modified the MRA estimation algorithm accordingly to account for this.

3.2.2 Synopsis Layers

The aggregate synopsis has a parameter — the cell size (e.g., 2x2), which we call its *resolution*. There is a trade-off when choosing the correct resolution for a query. Estimations are computationally more efficient with coarser synopses (i.e., with larger cells) due to a smaller number 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 an array form 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 2 there are four pieces.
2. Each piece covered by the cell more than 75% (a parameter) is estimated from this cell. The coverage is defined as the ratio of the areas of the piece and the cell.
3. The remaining pieces are left for the next layer, finer, synopsis. In Figure 2, 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 synopses in such a way as to avoid small region-cell

intersections, which decrease the estimation quality. At the same time, we cover large parts of the region with a small number of cells, which improves performance. To further speedup the 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 2, the region is covered $\frac{9}{16} \times 100\% = 56\%$. So the algorithm would use the next synopsis layer, if available.

Synopsis layers are also considered when validating candidates. If one of the layers is aligned with the candidate, that synopsis is used for the validation instead of the original data array. To be aligned with a synopsis, the region must intersect all its cells completely, 100%, 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.

3.2.3 *Synopsis Alternatives*

The choice of the aggregate synopsis as a grid was dictated by using the array data representation. There is no need in an index tree, such as R-tree, since the grid cell coordinates can be easily computed from the array coordinates. Synopsis layers can be seen as a slice of the well-known pyramid structure [26]. Searchlight does not use the complete pyramid due to memory restrictions. Several in-memory layers are sufficient for estimations, and the lowest layer, the data array, is used for validations. For other DBMS types, other structures might be more suitable. For example, for an RDBMS, MRA-trees [14] would be a better choice.

We want 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 [8, 9]. 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* [8] by combining the points from adjacent sub-sequences. A trace can be seen as a set of points in a multidimensional space. Since traces are quite large, they can be further covered by a number of MBRs, which then can be indexed (e.g., via 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. This notion can be encapsulated by a new Searchlight API call: $dist(x_l, x_r, Q)$, which computes the distance between the query sequence Q and any subsequence of size $|Q|$ lying within the $[x_l, x_r]$.

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,

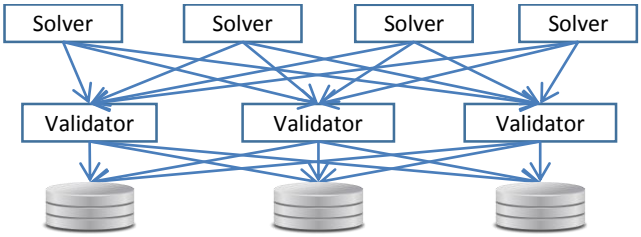

Figure 3: Distributed Searchlight with Solver and Validator layers.

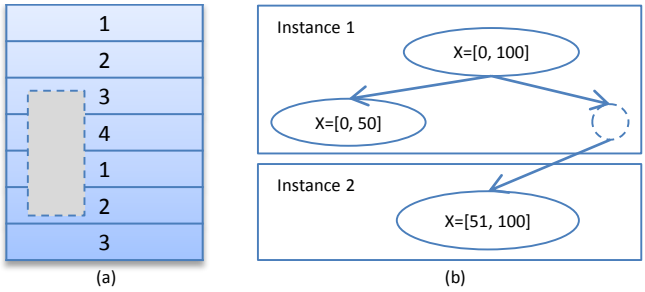

Figure 4: Search balancing. (a) Static round-robin balancing. The highlighted region is a hot-spot. (b) Dynamic balancing with transferring a sub-tree.

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 3.

4.1 Searchlight in SciDB

Searchlight uses SciDB [6] as the array DBMS. A SciDB cluster consists of *instances*, and each array is distributed among all instances. During query execution one instance serves as a *coordinator*, which combines partial results from other, *worker*, instances and returns the final result.

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 User-Defined Operators (UDOs). We implemented Searchlight as a UDO. Thus, it 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 answering API calls Searchlight accesses data internally, in the DBMS buffer. In summary, working inside the DBMS engine significantly decreases the overhead and avoids duplication of the same functionality.

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(a).

The static scheme targets search hot-spots, parts of the search space containing a lot of candidates. A Solver might get "stuck" in a hot-spot, while others finish quickly and sit idle. This creates an imbalance. The round-robin method distributes continuous hot-spots among multiple Solvers, as shown in Figure 4(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 the estimations for some heuristics, which hurts pruning. It might also hurt Solvers performance due to increased maintenance costs.

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*. The coordinator takes a busy Solver from a queue, dispatches the 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(b), where the Solver at Instance 1 cuts its right sub-tree and sends it to Instance 2. The helper treats it as the root of a new search tree. 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.

4.3 Validating and Forwarding Candidates

In SciDB array chunks are hash-partitioned across the entire cluster, and queries are broadcast across all nodes. In such a setting search UDFs would expose large latency: UDFs make multiple API calls, and each call would have to be broadcast across the cluster. Only after the individual answers are received the Validator would be able to continue, until the next API call. Additionally, API functions would have to be distributive or algebraic [10]. This is true for many common functions (including the API aggregates). However, we wanted the API to be extendable with other types, including holistic, which cannot be easily distributed. Thus, we decided to investigate another approach.

The array is evenly sliced into multiple partitions along the longest dimension. Each node participating in the validation gets one slice. During the validation Searchlight transparently pulls the required array chunks from other instances and caches them in an LRU buffer. The buffer is disk-backed, so no chunks are ever re-fetched. Chunks are pulled only on demand, when an API call needs them. If the search prunes some parts of the array, the corresponding data is neither fetched nor read from disk at all. UDFs can make multiple API calls during the validation without worrying about latency. Each call will be served from memory, after the chunks are fetched. Arbitrary functions can be added to the API, since any sub-array is accessible locally.

In some cases such data redistribution 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 then Searchlight provides reasonable interactive performance, as supported by experiments. Moreover, as an optimization, the redistribution can be done during the execution of the first query, after which the LRU chunk cache can be reused by further 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 will need. We solve this problem by first *simulating* the validation, which is performed by the same-node Validator. The simulation is validation for which the Router (Figure 1) is switched to the "dumb" mode. In this mode, instead of reading the synopsis or data, the Router simply answers API calls with $[-\infty, +\infty]$ intervals, which satisfies any constraint. Thus, such a validation always "succeeds". At the same time, the Router logs all chunk requests made during the simulation. Then, the Validator can examine the log and forward the candidate to the node responsible for most of the chunks, possibly keeping 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 I/O, problem, since the data will be quickly pulled from remote instances and mostly residing in memory. In this case 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 Validators depends on the current state of the search process, which we discuss in the next section.

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 into either of them depending on the current state of the search. While it is possible to do this in a 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, for which it monitors the current state of the search process at each node. Initially, given n available threads per node, it starts $n-1$ Solvers and a single Validator. 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.

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 at the same node. When the new load for the Solver arrives, Search-

light stops the Validator and wakes up the Solver. This also ensures that when the Solvers have completed the entire search, all resources go to the Validators.

Another optimization involves batching candidates to improve 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*. When a candidate arrives at a node for validation, it is assigned to the zone containing most of the chunks required for its validation. This information is available from the simulation process described above. At each step Validators take candidates from the same zone, enforcing locality. Recently accessed zones are checked first. By default Searchlight ensures that at least two zones can fit in memory at the same time. First, candidates corresponding to continuous objects (regions) often span neighboring zones. Secondly, this allows us to avoid the case where two frequently-used zones evict each other from memory.

5. EXPERIMENTAL EVALUATION

We performed an extensive experimental evaluation of Searchlight for clusters of sizes 1 through 8 using EBS-optimized Amazon EC2 c3.xlarge instances (4 virtual CPU cores, 7.5GB memory) with EBS-based general purpose SSDs (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.

We generated a $100,000 \times 100,000$ data array 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 array took 120GB of space (the binary size of the SciDB file). We used a 100×100 synopsis, which took approximately 400KB of space. We varied the the search space size by choosing more or less restrictive constraints and variable domains. For each search space we chose queries that produced different number of candidates to vary the Validators loads. The queries follow:

HSS (Huge Search Space). The query searched 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, 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, with $avg() \in [330, 332]$.

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 queries were similar to SSS ones, except the left-most corners had to be divisible by 10. 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.

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 to 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 to compare Searchlight 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 1: Time of the first result, delays between subsequent results and the # of results for HSS (secs)

Approach	First	Min/avg/max delays	Results
SL-1	13	0.001/3.8/101.1	981
SL-8	5	0.001/5.9/21.9	6,336

The results for HSS are presented in Table 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 find some results within a time limit to get an idea about the content of the data. CP and AFL did not find anything. For both the I/O became a considerable bottle-neck, due to multiple data accesses. Running the query in an 8-node cluster did not change that. SL (Searchlight), found the first result in 13/5 seconds and kept outputting them during the execution with small delays. We provide the delay statistics as min/avg/max delays between subsequent results. Together with the first result time, we believe this to be a good measure of the interactivity.

Table 2: The result and total times, SSS-HS-mod (secs)

Approach	Result time	Total time
SL-1/8	6.19/4.8	6.52/5.13
CP-1/8	3,240/91	4,260/304

Even for a small search space, CP was not able to finish SSS-HS in 13 hours, with no results 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 2). Even in this case, CP was no match for SL. This was 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 the performance. Moreover, for CP pruning did not make much difference, since it requires reading the data. In a cluster the performance of CP increased significantly, while still remaining well below the SL's. SL did not gain much from the cluster, since only 2 nodes were involved in validating a small number of candidates, with 6 nodes remaining idle. Impressive improvement for CP in the cluster can be explained by the nature of the solving process. A single node solver reads the data multiple times when traversing the search tree, since it does not optimize I/O or performs any intermediate caching. Dividing the search space between 8 nodes resulted in dividing the *data* as well, providing exponential I/O gains due to severely decreased data space at each node.

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 3: First result times, subsequent results delays and total times for SSS-HS(top)/-LS(bottom), secs

Approach	First	Min/avg/max delays	Total
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

Expressing the constraints of the SSS queries 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 resulted in 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 each tile. 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 3.

While AFL required computing every sub-array, Searchlight was able to prune most of the data and provide much better performance. SS-LS query, however, was different. While pruning was possible, the candidates touched the majority of the array, which created significant I/O load. As we discussed in Section 4.3, we consider such queries rare. In

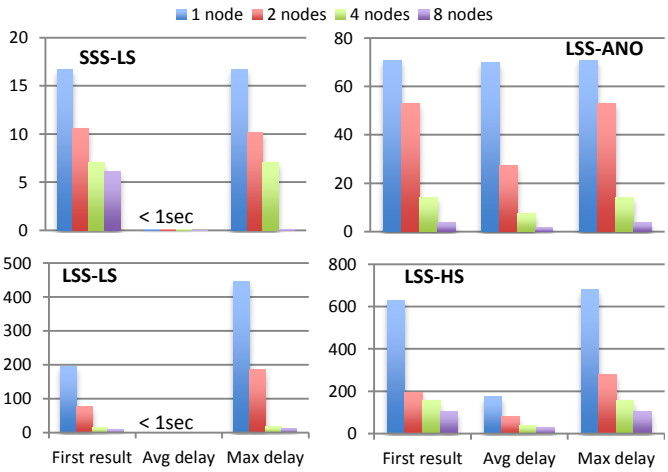

Figure 5: Delays for subsequent results for SL (secs).

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.

The problems with the queries discussed in the context of SciDB are also applicable to other systems. Query operators such as `window()`, `concat()` or `regrid()` are typical for array DBMSs. While they are useful for certain query types, they are not sufficient for search queries, since such queries require a different execution model.

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 5. We do not provide the minimum delays, since they were mostly under 1 second. We also 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. For most queries the delays significantly improved with the cluster size. 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. This was a very hard query with a large number of candidates and limited pruning opportunities.

Total times for all queries are presented in Table 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 (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

Table 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

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). Another low-selective query, LSS-LS, scaled much better since it did not touch as many chunks.

5.3 Searchlight Feature Experiments

The evaluation of the balancing strategies described in Section 4.2 is presented in Table 5, which shows the times for different numbers of slices in an 8-node cluster. First, we computed the average time for each Solver across three runs. Then, we computed the mean and standard deviation across these times. For the static case large deviation values can be explained by outliers — the times varied from 0.1 to 465 seconds. We saw the same trend for most queries. The results clearly show that the dynamic strategy significantly improves the balancing.

Table 5: Mean/StdDev of individual Solver times (top); and total times (below) for LSS-HS, secs

Balancing	8 slices	40 slices	100 slices	500 slices
Static	160/183	160/179	155/129	161/30
Dynamic	173/36	159/49	158/35	159/12
Static	466	454	338	254
Dynamic	259	284	223	235

The next experiment explores the ability of Searchlight to handle different heuristics. We compared the common Split (splits the largest domain in half) and Random (assigns a random value to a randomly chosen variable) heuristics with a custom one, called Probes. In Probes, the search space is divided into multiple cells, and the cells are explored in the utility order. The utility measures the probability of the cell to contain results, and is computed by trying a small sample of assignments from the cell. The results for two queries in an 8-node cluster are shown in Table 6. Due to the nature of LSS-HS, the Random heuristic was a very bad fit for it, demonstrating poor performance, so we aborted the execution. Split in general performed better than Probes due to the utility computation overhead of the latter.

To explore optimizations from Section 4.4, we performed micro-experiments using local machines to fully control CPU, network and disk. As we mentioned above, Amazon EC2 performance for network- and disk-intensive queries might vary significantly between runs. We used two local nodes (Linux kernel 3.8, Intel Q6600 CPU, 4GB of memory and WD 750GB HDD) running four SciDB instances, two instances per node. We took a 20GB subset of the original data and set the DBMS buffer size to 1GB for each instance.

Table 7 shows results for two queries, which were run with 4 threads available per instance. We distributed the threads between the Solvers and Validators as discussed in Section 4.4. In the table, "x-y" stands for the *static* distribution with x Solvers and y Validators per instance. Dynamic means dynamic threads allocation during the search.

Table 6: First result times, subsequent results delays and total times for different heuristics for SSS-ANO(top)/LSS-HS(bottom), secs

Approach	First	Min/avg/max delays	Total
Random	13.1	0.1/2.3/13.1	13.7
Split	4	0.04/0.7/4	4.3
Probes	11.7	0.1/2.7/13	16
Split	32.5	0.005/19.2/96.6	171.9
Probes	104	0.1/30/104	225

Table 7: Total times for different thread distributions

Query	Dynamic	3-1	2-2	1-3
LSS-LS	8m18s	14m3s	9m4s	9m44s
LSS-ANO	19s	19s	3m19s	3m30s

The dynamic strategy performed better than the static one, which requires the user to guess the correct allocation. In contrast, the dynamic strategy adapts to the current situation. We saw similar trends for all queries we ran.

Another experiment explores the ability of Searchlight to choose correct synopsis layers during query execution. We present the results in Table 8 for different layers that were available to Searchlight (e.g., 100 means 100x100 synopsis).

Table 8: Total times for different synopsis layers

Query	1000	100	10	1000-100-10
LSS-HS	NA	4m41s	2h28m	3m
SSS-LS	21m30s	15m	6m9s	1m10s

The choice of synopsis layers has a profound impact on query times, especially in case of LSS. When running LSS-HS with the 1000x1000 synopsis, we actually were not able to finish the query in several hours. These results show the trade-off described in Section 3.2.2, when coarse synopses produce too many candidates and fine synopses require too much CPU power. This trend might be less apparent for SSS. SSS-LS performed better for the 10x10 synopsis, since Searchlight was able to use it for validation instead of the original data. The Solver times actually increased from 6 seconds to 6 minutes between the 100 and 10 experiments. The best benefit was achieved when all synopses were used together (i.e., 1000-100-10), since Searchlight was able to choose best synopses for both estimations and validations.

The last experiment in this section shows the benefits of using candidate batching, as discussed in Section 4.4. In case of a small number of candidates or if the candidates are situated close to each other, the batching does not provide any benefits. In our experiments it did not have any significant effect, positive or negative. For one query, SSS-LS, that had a large number of candidates scattered around the whole search space, using batching decreased the total time from 21m30s to 15m.

5.4 SDSS Experiment

We experimented with the entire Sloan Digital Sky Survey (SDSS) [2] 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 9: First result times, subsequent results delays and total times for SDSS queries (Q_i given in text), secs

Query	First	Min/avg/max delays	Total
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 9. 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 reasonable interactive performance.

6. RELATED WORK

There is a considerable body of research directed at making search efficient for large search spaces for different types of constraints [21]. Traditional CP solvers (e.g., Gecode, IBM ILOG, Comet) support parallel search, including work stealing [18, 7] and over-partitioning [22]. We use similar techniques for balancing in Searchlight. However, our experiments showed that traditional CP solvers are ill suited for searching large volumes of data or handling external (e.g., DBMS-resident) data natively, i.e., without extracting, transforming and partitioning it.

The idea of using linear solving and optimization techniques has been explored for specific problems in relational DBMSs, such as *how-to* queries [17] and Quantifier-Free Linear Integer Arithmetic (QFLIA) [16]. In how-to queries the user specifies constraints for DBMS tables and allowed database modifications, and the system employs a MIP solver to produce a set of *hypothetical* tables, satisfying the specification. In QFLIA, the authors allow the model to reference DBMS tables via *membership* constraints, which tie model variables to existing tuples. The linear part is handled by the solver, and the membership is checked by a DBMS engine. The authors, however, do not explore complex search queries or address the problem of accessing the data by the solver. The latter, as we have shown, renders traditional solvers infeasible for large data sets.

We want to explicitly contrast our work with aggregate query processing, spatial 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 often contain aggregates, its queries are much more complex, since they involve search. 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 [14], where nodes are annotated with aggregate information. 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 discussed in Section 3.2.1, Searchlight uses this technique as the basis for the aggregate grid synopsis. Sampling-based methods [12, 5, 19] are also commonly used to approximate aggregates, and often have lower costs. However, they do not provide 100% confidence guarantees, which makes them unsuitable for provably pruning search sub-trees. Thus, Searchlight does not use sampling.

Data cube exploration [10] involves computing 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`) or drill-down (adding an attribute to the `GROUP BY`). 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 [11], compressing it for general [25] and spatio-temporal cases [15]. Additional information can be stored in the cube to provide more information to the user, e.g., the degree of abnormality for values [23]. 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 query.

Spatial DBMSs [24] allow users to manage and query spatial data. They are often built on top of traditional DBMSs, and can efficiently retrieve particular objects (e.g., buildings, rivers, etc.), find all features inside a window and perform nearest-neighbors search. Such queries do not generally involve search. Objects can be retrieved by using common index structures, like pyramids [26, 4] 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 Searchlight is Content-Based Retrieval (CBR) [24], 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 [3, 20]. Then, search can be performed using this graph. The constraints are generally easy to check (e.g., look up the edge), and the search space is small. In contrast, for Searchlight we assume a large search space of objects, which is infeasible to precompute and maintain. The constraints are also more expensive, since they might involve multiple complex computations.

Semantic Windows (SW) [13] is aimed at searching for multi-dimensional rectangular regions with specified aggre-

gate properties. The framework runs on top of an RDBMS and supports distributed computation. However, this work has two major limitations. First, it uses a custom priority-based search method tailored to a conjunction of aggregate constraints for a single window (which can vary in sizes). It would be hard to extend SW for arbitrary constraints or multiple windows in the same query (e.g., "anomaly" queries discussed in the paper). Searchlight can easily handle SW queries with additional constraints without any modifications, which corresponds to our vision of a unified search engine. More importantly, SW uses sample-based estimations to steer the search in the direction of candidates. Thus, SW has to eventually read the whole data set to produce the final result, since it cannot provably prune the search space. In contrast, Searchlight uses provable pruning, which reduces the amount of data to be processed and greatly increases performance, as supported by the experiments.

We note that our system differs in the kind of problems it targets in relation to the so-called "big data" systems, most of which aim to efficiently perform analytical queries, machine learning tasks, stream processing, etc. Our framework targets a fundamentally different problem.

7. CONCLUSIONS

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. Our experimental results over real and artificial data sets show 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.

8. ACKNOWLEDGEMENTS

This research was supported in part by the Intel Science and Technology Center for Big Data and NSF IIS-1111423.

9. REFERENCES

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

[2] The sloan digital sky survey. http://www.sdss.org/.

[3] C.-H. Ang, T. W. Ling, and X. M. Zhou. Qualitative spatial relationships representation io&t and its retrieval. In *DEXA*, pages 270–279, 1998.

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

[5] D. Barbar and X. Wu. Supporting online queries in rolap. In *Data Warehousing and Knowledge Discovery*, volume 1874, pages 234–243. 2000.

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

[7] G. Chu, C. Schulte, and P. 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.

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

[9] D. Q. Goldin and P. 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.

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

[11] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[12] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, 1997.

[13] A. Kalinin, U. Cetintemel, and S. Zdonik. Interactive data exploration using semantic windows. In *SIGMOD*, pages 505–516, 2014.

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

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

[16] P. Manolios, V. Papavasileiou, and M. Riedewald. Ilp modulo data. In *FMCAD*, pages 28:171–28:178, 2014.

[17] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.

[18] L. Michel, A. See, and P. Van Hentenryck. Parallelizing constraint programs transparently. In *Principles and Practice of Constraint Programming CP 2007*, volume 4741, pages 514–528. 2007.

[19] F. Olken and D. 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.

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

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

[22] J.-C. Rgin, M. Rezgui, and A. Malapert. Embarrassingly parallel search. In *Principles and Practice of Constraint Programming*, volume 8124, pages 596–610. 2013.

[23] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of olap data cubes. In *EDBT*, pages 168–182, 1998.

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

[25] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the petacube. In *SIGMOD*, pages 464–475, 2002.

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