Multivariate Probabilistic Range Queries for Scalable Interactive 3D Visualization

Amani Ageeli, Alberto Jaspe-Villanueva, Ronell Sicat, Florian Mannuss, Peter Rautek, Markus Hadwiger

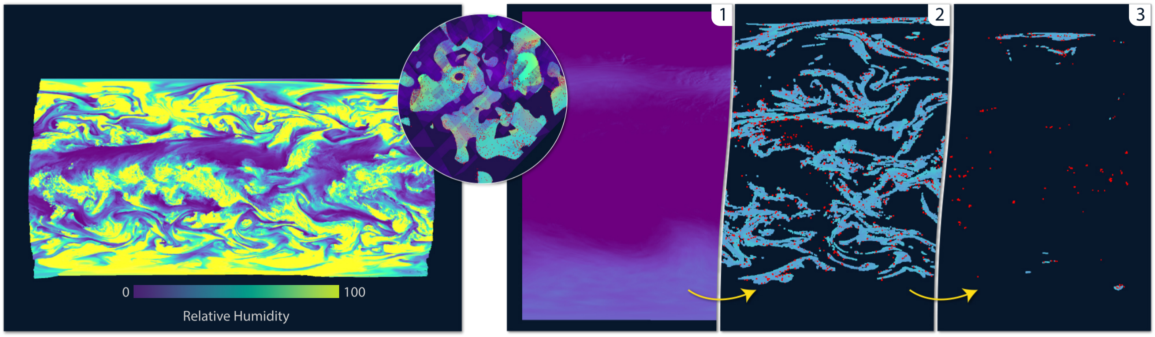


Fig. 1. **Probabilistic queries.** For the meteorological data set shown on the left ($N > 1$G cells), on the right we show three consecutive query results. (1) Without filtering the data; (2) Filtering for temperature in $[-24,-8]$ and relative humidity in $[15,20]$; 1% false positives (shown in red). (3) A $d = 5$-dimensional query, in addition filtering height in $[0,4000]$ and two vector field components. (Inset) We improve both the query performance and the false positive rate via a hierarchical early-out strategy using a novel concept of supercells.

Abstract—Large-scale scientific data, such as weather and climate simulations, often comprise a large number of attributes for each data sample, like temperature, pressure, humidity, and many more. Interactive visualization and analysis require filtering according to any desired combination of attributes, in particular logical *AND* operations, which is challenging for large data and many attributes. Many general data structures for this problem are built for and scale with a fixed number of attributes, and scalability of joint queries with arbitrary attribute subsets remains a significant problem. We propose a flexible probabilistic framework for multivariate range queries that decouples all attribute dimensions via *projection*, allowing any subset of attributes to be queried with full efficiency. Moreover, our approach is *output-sensitive*, mainly scaling with the cardinality of the query result rather than with the input data size. This is particularly important for joint attribute queries, where the query output is usually much smaller than the whole data set. Additionally, our approach can split query evaluation between user interaction and rendering, achieving much better scalability for interactive visualization than the previous state of the art. Furthermore, even when a multi-resolution strategy is used for visualization, queries are jointly evaluated at the finest data granularity, because our framework does not limit query accuracy to a fixed spatial subdivision.

Index Terms—High-dimensional filtering, multivariate filtering, output-sensitivity, multivariate attribute queries, progressive culling

1 Introduction

Large scientific data sets, whether from simulations or measurements, are a core focus of visualization. However, their size and complexity make interactive visualization and analysis challenging. Fast rendering and query evaluation (or *filtering*) require efficient, scalable data structures and algorithms. In addition to large sizes due to a large number of data samples or grid cells, scientific data often comprise many scalar, vector, and tensor attributes per sample or cell. For example, scalars such as temperature, pressure, humidity, and salinity in large-scale climate or weather simulations, or porosity and permeability in subsurface simulations. Visual analysis requires the efficient *filtering* of data, often based on multivariate range queries, to locate and visualize all cells with a certain combination of attribute values. We refer to data with many attributes as multivariate data living in a high-dimensional *attribute space*. We say that this space comprises D *attribute dimensions*, with our work targeting scalable visualization of data with large D, e.g., from $D = 6$ to $D = 20$ or more. While logical *OR* operations in range queries over multiple dimensions are simple because they are separable, i.e., each attribute can be queried independently, logical *AND* operations require checking all attributes whether they *jointly* fulfill the desired properties. This is an inherently D-dimensional problem: *AND* queries must be evaluated—at least conceptually—in a D-dimensional space. However, for large D, the *curse of dimensionality* [69] can quickly become a problem for query performance as well as for efficient storage.

For visualization purposes, the spatial position and corresponding visibility of cells,[1] as well as their corresponding "locations" in attribute space are both crucial properties. We distinguish two major conceptual strategies for evaluating D-dimensional queries for visualization:

1. Evaluate queries in *attribute space first:* Determine a *query result* set, containing all cells for which the query evaluates to true. Only afterward determine the (potentially) visible subset of cells.
2. Evaluate queries in *spatial domain first:* Traverse data in a spatial data structure, such as an octree; often in visibility order. For each encountered cell (or node with many cells), evaluate the query.

For strategy #1, common solutions that scale well to large D are, e.g., R-trees [22] or kd-trees [6]. For a data set consisting of N cells, these scale roughly between $D \cdot \log N$ and $D \cdot N^a$, $a = 1 - \frac{1}{D}$ [29]. However, for large D several scalability problems remain for interactive visualization:

- *Amani Ageeli, Alberto Jaspe-Villanueva, Ronell Sicat, Peter Rautek, Markus Hadwiger are with King Abdullah University of Science and Technology (KAUST), Visual Computing Center, Thuwal, 23955-6900, Saudi Arabia. E-mail: {amani.ageeli, alberto.jaspe, ronell.sicat, peter.rautek, markus.hadwiger}@kaust.edu.sa.*
- *Florian Mannuss is with Saudi Aramco, Dhahran, Saudi Arabia. E-mail: florian.mannuss@aramco.com.*

Manuscript received 31 Mar. 2022; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

[1] We will refer to items to query mainly as *cells*, with D attributes each. In addition to geometric grid cells, the same applies to voxels and point samples.

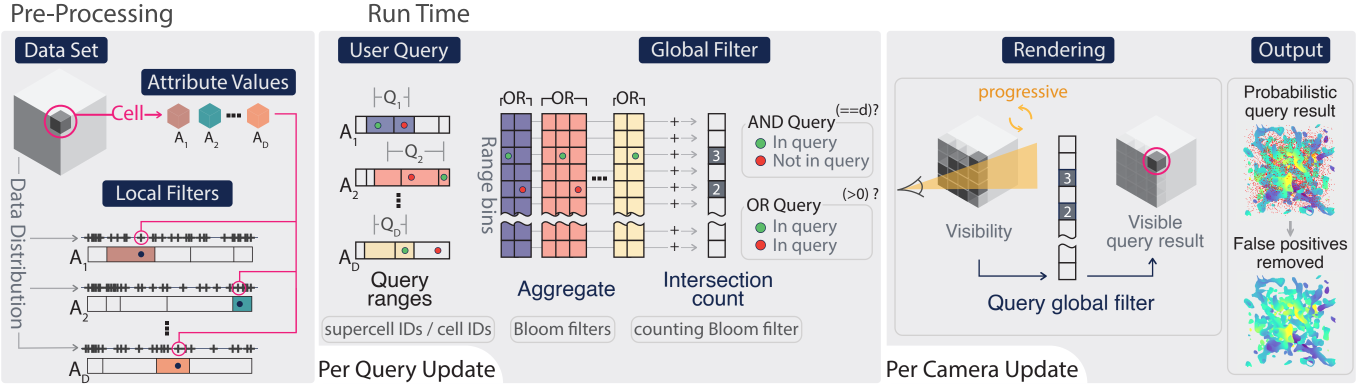


Fig. 2. **Overview.** Our approach consists of three main stages: (1) Given data with D attribute dimensions A_i, a pre-processing stage computes D one-dimensional *projections* of the set of all cells into *local filters*. No D-dimensional data structure is used. Each local filter comprises multiple *range bins*, where each bin is implemented as a *Bloom filter*. (2) d-dimensional ($d \leq D$) query evaluation at run time aggregates range bins of d local filters into a single *global filter* implemented as a *counting Bloom filter*. (3) During rendering, octree nodes query the global filter in visibility order to obtain a binary *occupancy state* per node. We use a *supercell ID* per node to speed up this process, and all states are cached for progressive query updates.

- For large D, trees in attribute space, such as R-trees, become very large, which often prevents storage in GPU memory, and each single tree traversal for query evaluation is hard to parallelize.
- Many queries only touch a *subset* $d \leq D$ of all attributes. The entire D-dimensional tree must be in memory, and it is not feasible to load or stream a lower-dimensional subset of the tree. Query time is also similar or slower than querying all D dimensions [55].

Alternatives avoiding these drawbacks build on bitmap indices [57,67]. However, these scale roughly with $D \cdot N$ (or $d \cdot N$), which can become a major problem for large data due to linear scaling with the data size N.

Strategy #2 above enables leveraging *view-dependence* to query attributes only for the potentially visible set (PVS) of cells inside the view frustum and not occluded, as determined by occlusion culling. We denote the number of cells in the PVS by $\tilde{N}$ (see Table 1). While this helps, since often $\tilde{N} \ll N$, *scanning*, i.e., checking all $d \leq D$ attributes of each cell, scales with $d \cdot \tilde{N}$ (worst case still $\tilde{N} = N$), accessing actual attribute data in memory.[2] Using spatially aggregated attribute information, e.g., per octree node, can help. However, for coarse approaches, such as a (min, max) value pair per attribute, queries become very inaccurate for nodes with many cells. Aggregating more accurate information, such as a *joint D-dimensional histogram* per node, scales exponentially with D in memory and in query time. Potential solutions are exploiting histogram sparsity [36]; or using integral histograms [48], which improve query time, but use more memory and still scale exponentially.

Combining both strategies above is not straightforward. For example, storing an attribute space tree, such as an R-tree, in every node of a spatial hierarchy is not feasible. In this paper, we propose a novel approach for scalable multivariate query evaluation (Fig. 1) that achieves an efficient combination of both strategies. Our approach mainly scales with the *expected output cardinality* $|Q|$, of any query Q, and the *visible output (PVS)*, instead of with the input size N or the dimensionality D.

First, we use strategy #1: Evaluating a query Q starts in attribute space (Fig. 2, center), by using $d \leq D$ pre-computed *1D projections*[3] (Fig. 2, left). The idea of projections is similar to bitmap indices [57,67]. However, we achieve scalability *independent of the input size* N, via fixed-size[4] *probabilistic hash tables* (Bloom filters). To also incorporate strategy #2, during rendering a single hash table *independent of both* N *and* $d \leq D$ is queried during PVS determination (see Fig. 2, right).[5]

The major properties and contributions of our method are:

- Although we evaluate D-dimensional *joint* (logical AND) queries, we store and aggregate only D separate 1D *projections*. This also enables including or excluding any subset of d dimensions from a query, facilitating streaming and out-of-core approaches.
- Our method is *output-sensitive* with respect to both the cardinality of the query result as well as to the on-screen visibility (the PVS).
- We query *full-resolution* data, but introduce *supercells* for efficient *early-out* during spatial hierarchy traversal, significantly reducing both query time and the false positives of probabilistic hashing.

2 Related Work

Multivariate and multifaceted data are very common in visualization applications [27], requiring corresponding data structures and algorithms.

Multivariate histograms facilitate query-driven visualization [57] of such data, enabling users to select interesting subsets by specifying multi-attribute range queries, such as (30 < temperature < 50) AND (15 < humidity < 30). However, multivariate or multi-dimensional histograms for joint distributions of multiple attributes often suffer from the curse of dimensionality [69], because they grow exponentially as B^D, for D attribute dimensions with B histogram bins each. Lu and Shen [36] introduce a multivariate histogram representation for query-driven visualization on local data blocks. They apply a data space transformation to large yet sparse multivariate histograms, transforming them into smaller multi-dimensional arrays, which are encoded using a dictionary-based approach. Wei et al. [62,63] use compressed bitmap indices [65] to perform local histogram matching in multi-field datasets. Sparse PDF maps [24] and sparse PDF volumes [53] also compactly encode joint probability density functions of data, by fitting high-dimensional Gaussian basis functions. Histogram (or PDF) encodings are used in various applications, such as fuzzy isosurfacing [59,61], uncertainty visualization [33], range distribution queries [15], or to reduce artifacts in multi-resolution volume rendering [53,68].

Table 1. **Terminology** used in this paper.

term	explanation
D	# attribute dimensions A_i (e.g., temp., pressure, ...).
d	dimensionality $d \leq D$ of current query Q.
A_i	attribute dimension i, with $i \leq D$. One value per cell.
cell	individual grid cell (regular grid: voxel), with cell ID.
local filter	one for each attribute A_i; comprises several range bins.
range bin	cells in subrange of specific A_i; Bloom filter per bin.
global filter	one counting Bloom filter for all $d \leq D$ queried A_i.
N	# cells in input data. Uniform 3D volume: $N := M^3$.
$\tilde{N}$	# cells in view-dependent potentially visible set (PVS).
Q	query / query result set (an arbitrary set of cell IDs).
$\|Q\|$	# cells in query: cardinality of query result set.
Q_i	attribute range $[Q_i^{\min}, Q_i^{\max}]$ of query for attribute A_i.
m	# bits/counts in a Bloom filter (size of bit/count vector).
k	# hash functions for a Bloom filter.
node	node of spatial 3D subdivision (e.g., octree node).
supercell	all cells in a node are jointly assigned *one* supercell ID.

[2] We note that accurate filtering does not permit an easy multi-resolution solution to reduce $\tilde{N}$ further: Ultimately, queries must refer to individual cells.

[3] There are D *separate* projections; each with a size independent of N.

[4] The size is chosen for expected query cardinality $|Q|$, not for input size N.

[5] It is a *single* global hash table, irrespective of the dimensionality d or D.

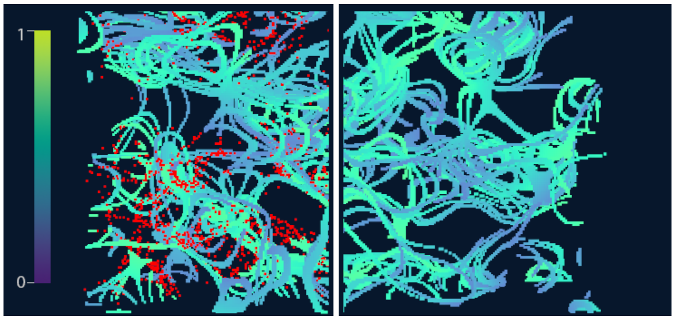


Fig. 3. **False positives.** The global filter representing the query result set is queried using probabilistic hashing. While this leads to some false positives (left; shown in red), they can be removed by *scanning* the already significantly reduced number of cells in the query result (right).

Bitmap indices [14,43,54,57,66,67] have been used widely for performing multivariate range queries. Given N data items with V possible values, a bitmap index will have one bitmap with N bits for each possible value v. In each bitmap, the n-th bit is set if the n-th data item's value is equal to that of the bitmap's associated value v. Range queries can be performed by applying a logical OR operation on the bitmaps of the values within the queried range. Multivariate data with D attributes require $D \cdot V$ bitmaps, each of size N. Range queries can be performed by applying a logical AND operation to the intermediate results of individual attribute range queries. To reduce storage cost of bitmap indices, several compression methods have been proposed such as Byte-aligned Bitmap Code [1,3] and Word-Aligned Hybrid [66], to name just a few. Stockinger et al. [58] used binning to reduce the size of the bitmap index when applied to attributes with high cardinality, e.g., floating point scientific data. Dextrous data explorer (DEX) [57] is an implementation of bitmap indexing specifically applied to query-driven visualization, e.g., for isosurface extraction and multi-attribute filtering.

Span space [34,35] methods can be used to optimize isosurface extraction from scalar volumes, or to quickly search for cells that fall within a given min-max range. Near optimal isosurface extraction (NOISE) [35] uses kd-trees [6], and isosurfacing in span space with utmost efficiency (ISSUE) [51] uses 2D regular lattices for fast span space search. Subsequent work uses octrees [52,64] and max-trees [60] in order to extract the isosurface and perform spatial filtering of volumetric data. Fiber surfaces [12] extract surfaces from bivariate data.

R-trees and similar data structures are widely used in database management systems for multivariate range queries [4, 22, 50, 56]. The R-tree, introduced by Guttman [22], is a hierarchy of nested D-dimensional minimum bounding regions. Range queries are performed via recursive traversal of the tree to find intersections with the bounding region of the query. Kratky et al. [28], demonstrated the inefficiency of R-trees for narrow range queries, i.e., for small min-max differences, due to a high probability of false positive intersections. They further discuss how for such queries the R-tree efficiency decreases as the dimension of the indexed space increases. Representations based on R-trees often also suffer from the curse of dimensionality, with memory requirements and query times often becoming unfeasible with higher dimensions [55]. The BB-tree [56] is a main memory index structure for performing multi-dimensional range queries. It has been shown to be faster than R*-tree [4], kd-tree [6], PH-tree [70], and brute force scanning, for multivariate range queries up to a selectivity of 20%.

Range trees and similar data structures [7,38] also enable efficient multivariate range queries, although they can be prohibitive in terms of pre-processing and storage requirements [8]. Similarly, kd-trees [6] and quadtrees have been used for minimizing the search space of 2D range queries in spatial data [49]. Kd-trees have also been used to interactively query and explore time series [71], or 3D curve data sets [37].

Approximate techniques can greatly speed up range queries, often in the form of approximate nearest neighbor search [2,26] and selectivity estimation [45,47]. The former techniques often employ random projections [40,41] of data points to lower dimensions to address the curse of dimensionality. The latter selectivity estimation techniques [45,47] aim to estimate the cardinality or size of the results of a multivariate query to optimize query execution plans in databases. These methods often employ variants of multi-dimensional histograms [21,42,47], which can be encoded as multi-resolution wavelets [39].

Bloom filters are a compact, probabilistic data structure for testing the membership of an item in a set [10]. They can report false positive results, but no false negatives. Bloom filters are widely used for database indexing [11], but have also enabled efficient culling of segmented volume data [9]. There exist many variants, such as counting Bloom filters [46], spectral bloom filters [16], and tree-structured Bloom filters [17,20]. Similar data structures with deletions [18] and data locality [5] have also been proposed. An example for performing range queries is the multi-dimensional segment Bloom filter (MDSBF) of Hua et al. [25]. Similar to bitmap indices [67], Bloom filters can also benefit from bit string compression, for example using roaring bitmaps [13,31,32]. In our work, we use a recent compression algorithm [30] leveraging modern SIMD processors for fast de/compression.

3 Algorithm

Our approach consists of the three main conceptual stages depicted in Fig. 2. The first stage is a pre-computation for the entire input data set. Only the subsequent two stages are executed at run time. The second stage is view-independent, and only needs to be executed whenever a new query is specified by the user. Only the third stage is executed during rendering, e.g., for each new camera transformation. However, even in the third stage all query computations are cached and do not need to be re-computed for any new view unless the query changes.

3.1 Overview and Processing Stages

The major goals, data structures, and algorithms of each stage of our approach are as follows. Details are described in subsequent sections.

Pre-processing: Local filters. For a given data set, we pre-compute a representation of the D-dimensional attribute space. However, instead of computing a D-dimensional data structure, we only compute D one-dimensional *projections*, one for each attribute $A_1, \ldots, A_D$ (Fig. 2, left). We call the data structure for each attribute A_i a *local filter*, each of which spanning the range of the corresponding attribute A_i, from the minimum value to the maximum value of A_i. For each attribute A_i we adaptively subdivide its range into b_i *range bins* $B_i^1, \ldots B_i^{b_i}$. Each bin represents a *set* of cell IDs, whose attribute values map to the corresponding range. However, instead of storing an actual set or a bitmap for each bin, which would scale with the data size N, we hash cell IDs into a Bloom filter [10] for each bin. A Bloom filter is a hash table comprising a bit vector of m bits for *probabilistic* membership queries. In our approach, m is chosen according to the *expected cardinality* $|Q|$ of queries Q, instead of according to the input data size N.

View-independent query evaluation: Global filter creation. To evaluate a query Q for an attribute range $\{Q_i = [Q_i^{\text{min}}, Q_i^{\text{max}}]\}_{i=1}^{d}$, in $d \leq D$ dimensions, we first proceed as follows. For each attribute A_i, we determine the range bins of the corresponding local filter overlapping the range $[Q_i^{\text{min}}, Q_i^{\text{max}}]$ (Fig. 2, center), and compute a *bitwise OR* of all m Bloom filter hash table bits. Then we *sum* the result over all attributes A_i into a single *global filter* for all attributes (Fig. 2, center). The global filter is also implemented as a probabilistic hash table. However, in contrast to the local filters, the global filter is a *counting Bloom filter* [19], with each count $\leq d$ (Fig. 4). The crucial insight here is that the resulting counts allow determining the set of cell IDs that are in the d-dimensional *intersection* of all $d \leq D$ query ranges, corresponding to a *logical AND*, although no d- or D-dimensional data structure or computation is used. Fig. 4 illustrates this approach.

View-dependent query evaluation: Global filter querying. The global filter computed in the previous stage is view-independent and contains all information about the query result irrespective of the spatial hierarchy used for rendering, such as an octree.[6] Therefore, in the rendering stage we need to (selectively) query and propagate the information from the global filter into the spatial hierarchy. We do this

[6] Without restricting generality, we use and refer to an octree as the spatial hierarchy used for rendering. Any other spatial subdivision would also work.

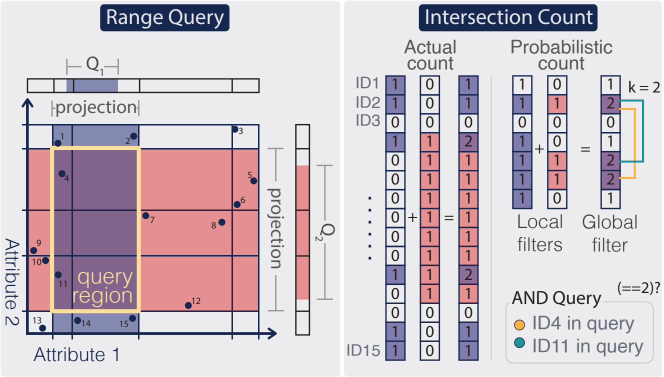


Fig. 4. **Joint query as intersection of projections.** Each cell in the input data (here, $N = 15$ cells) is a point in a D-dimensional (here, $D = 2$) attribute space (left). Our *local filters* store 1D projections, which are intersected by *counting* how often each cell is contained in a 1D projection (center right). For a d-dim. AND query ($d \leq D$), only cells of total count d are in the query: They are in the *intersection* of d projections. Instead of storing N cell counts, we obtain a *probabilistic count* from the *global filter*, which is a *counting Bloom filter* of size $m \ll N$ (right; $k = 2$).

in a *view-dependent* and *progressive* manner. (However, note that our local and global filters do not depend on any specifics of the spatial hierarchy or rendering.) In essence, every octree node needs a binary *occupancy state* whether the current query Q evaluates to true for *any* cell[7] located within the spatial extent of the node. We compute the occupancy state of octree nodes in two major ways (for interior nodes, the state accurately reflects entire subtrees; see also Algorithm 1):

1. First, we employ an *early-out* strategy during octree traversal, using the concept of supercell IDs (see below) that are queried against the *global filter* in the same way as individual cell IDs are. In many cases, this already determines an occupancy state of *false* for an entire subtree, without having to continue tree traversal.
2. Where early-out fails, ultimately an octree leaf node is reached. We then iterate over all cell IDs contained in the leaf node, and for each ID query the hash table comprising the *global filter* to obtain the occupancy state of the cell. This is an operation independent of D, and solely depends on the number k of hash functions used by the Bloom filter (see below). The occupancy state of an octree leaf node is then the *bitwise OR* of the states of the individual cells. The occupancy states of interior octree nodes are then computed by propagating the state of the leaf nodes up the tree, again performing a *bitwise OR* operation in each step.

The above two strategies are combined with standard view frustum and occlusion culling: Only subtrees/nodes that are within the view frustum and are not culled according to occlusion culling are visited. In total, this amounts to a *lazy evaluation strategy* combing *two* output-sensitive aspects: Individual cells are only queried if they are (1) potentially visible; *and* (2) early-out (depending on the query output) has failed.

Supercells. The essential idea of supercells is that each corresponding supercell ID represents all *full-resolution* cells within the spatial extent of a given octree node, without corresponding to any multi-resolution down-sampling. Despite this, supercell IDs are inserted into the *local filters* in almost the same way that regular cell IDs are. Nevertheless, this simple addition of supercell IDs to the filter hash tables allows a very efficient, conservative *early-out* from octree traversal, skipping processing for whole subtrees of the spatial hierarchy, *without any false negatives* in the query evaluation. See Sec. 4.2 for the details.

Eliminating false positives. For a completely accurate occupancy state, we have one remaining problem: The implementation of the global filter as a Bloom filter can produce false positives (see Sec. 3.3). That is, the state of cells, and due to propagation also that of octree nodes, can be *true* although it should be *false*. Moreover, due to binning in each local filter, false positives also arise from cells that are in a range bin that overlaps the query range, but that are in fact outside the query range (Fig. 4). However, it also depends on the application whether this problem actually needs to be addressed explicitly:

- If our method is essentially used as a fast culling strategy, because cells are rendered via a transfer function that maps any cell outside the query range to full transparency, nothing needs to be done.
- In the case where the query must be completely accurate, e.g., because no transfer function is used, or it is only applied to some but not all attributes in a query, and therefore all cells in the query result might be rendered, false positives need to be eliminated.

Fig. 2 depicts a visualization with false positives (Fig. 2, top right image), and with false positives removed (Fig. 2, bottom right image). We eliminate false positives by simply iterating over all cells contained in a visible octree leaf node with an occupancy state of *true*. We re-evaluate the query by *scanning* each cell to accurately update the node's occupancy state. If no transfer function is used, at this time a *bitmask* containing the occupancy state of each individual cell in the leaf node can also be computed to *cache* which cells should not be rendered.

Because the set of these cells is already reduced drastically compared to the whole data set, this update strategy is usually fast. Furthermore, we can again employ a lazy evaluation strategy: We eliminate false positives progressively node by node, until all nodes have been updated. Each node also needs to be updated only once. Once the occupancy state is final, it is cached and re-used in all subsequent rendering frames.

For fast interaction, we can also consider the following property: It is interesting to note that, due to the random hashing that we employ, the spatial pattern of where cells corresponding to false positives are located appears similar to a white noise distribution. This makes it quite easy for the user to ignore visible false positives during interaction, while still being able to understand the essential characteristics of the data set without confusion. See Fig. 5 for an illustration of this effect.

Caching of occupancy state. Although we compute the occupancy states in the spatial hierarchy in a view-dependent, output-sensitive manner, once the final state of any node has been computed, it does not change for different views until the query itself, and thus the global filter, are changed. We therefore cache the occupancy state in each octree node, avoiding unnecessary re-computation. See Algorithm 1.

3.2 Query Representation and Characteristics

We target general d-dimensional range queries ($d \leq D$) consisting of a combination of specified ranges in each of d relevant attributes A_i, where a given expression consisting of logical AND, OR, and NOT operations should be evaluated to determine which cells are members of the output result (set) of a query. Any more complicated expression can be built from simple building blocks. However, the major computational challenge is efficiently evaluating logical AND queries, because they (conceptually) must be evaluated *jointly* in d-dimensional attribute space, whereas logical OR queries are intrinsically one-dimensional.

Query result. The result of any query Q is a *subset* of cell IDs, of the total set of N possible cell IDs comprising the input data set. That is, N is the cardinality of the input, and we denote the cardinality of the subset comprising the query result by $|Q| \leq N$. We target *full-resolution* query evaluation. That is, no down-sampled multi-resolution

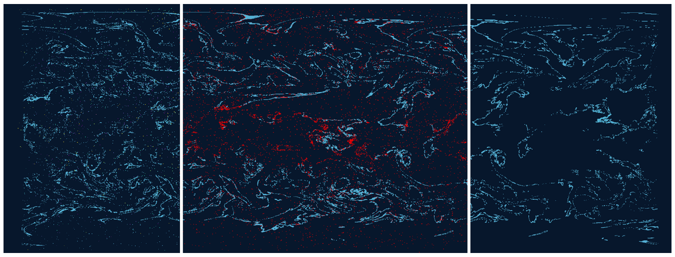


Fig. 5. **Spatial distribution of false positives.** Due to probabilistic hashing, false positives occur randomly throughout space, making them easy to ignore during interaction. (Left) True and false positives; (center) false positives highlighted in red; (right) false positives eliminated.

[7] This always refers to individual cells, not to a less-accurate down-sampled (multi-resolution) representation. Our approach targets full-resolution accuracy.

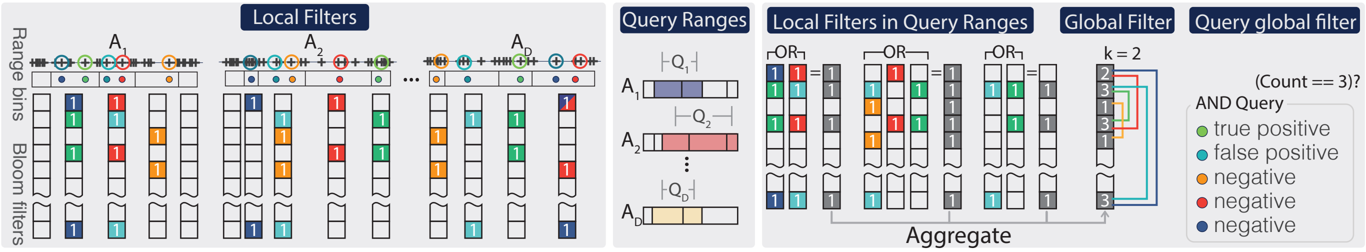


Fig. 6. **Bloom filter usage in local and global filters.** (Left) Five example cell IDs are inserted into the Bloom filters comprising the *range bins* of the *local filters*. Each Bloom filter is a bit vector of size $m \ll N$. (Center) At run time, only the bit vectors of bins in the query ranges Q_i need to be accessed. (Right) Over each attribute A_i in the query Q, a bitwise OR gives the *probabilistic* union of cell IDs in Q_i. Finally, all Q_i are *summed* into the *global filter*, a single *counting* Bloom filter of size $m \ll N$. False positives can occur with probabilities determined by the Bloom filter configuration.

aggregation of "lower-resolution" cells is used for query evaluation or to increase performance at a loss of accuracy. (For rendering, any multi-resolution approach can be used at the same time, if desired.)

AND queries. A cell ID is in the query result (set) of an AND query, as given by the *global filter*, when the count values of the k hash table entries (see below) corresponding to the cell ID *all* are *equal to* d, for a d-dimensional query in attribute space, with $d \leq D$. (See Algorithm 1.)

OR queries. A cell ID is in the query result (set) of an OR query, as given by the *global filter*, when the count values of the k hash table entries (see below) corresponding to the cell ID *all* are *non-zero*.

NOT queries can be evaluated by inverting the desired range of a query, and building a composite query for the resulting attribute ranges.

Query characteristics and scalability. Although the query cardinality is only really limited by the data size, i.e., $|Q| \leq N$, most meaningful queries result in a much smaller $|Q| \ll N$. Moreover, $|Q|$ typically grows slower than N with increasing data size. In particular, we especially target high-dimensional queries, and since logical AND operations correspond to *set intersections*, the higher the dimensionality d of a query is, the smaller the cardinality $|Q|$ usually becomes. In our method this is reflected by the use of Bloom filters. Their size is determined by a fixed length m, which we choose according to the expected query size $|Q|$. Our approach therefore scales with $|Q|$ instead of with N, essentially making it largely independent of the data size N.

3.3 Bloom Filters: Probabilistic Set Membership

A Bloom filter [10] is a probabilistic representation of a *set* of elements, with probabilistic membership queries. Set elements are hashed into a hash table bit vector of length m bits.[8] A Bloom filter requires the choice of k hash functions, e.g., $k = 2$ or $k = 3$. Inserting an element into the filter is done by setting k bits, determined by applying the k hash functions to the element, to one. Set membership is queried by again hashing an element k times, and checking whether *all* corresponding k bits in the Bloom filter are set to one. Due to possible collisions, false positives can occur with a certain probability, depending on the choice of m and k, as well as on the number of inserted elements n (Eq. 2). A crucial property of Bloom filters in our context is that false negatives (an element is reported as not in the set, although it in fact is) are guaranteed to not occur. This makes Bloom filters very well suited for *conservative culling*: Skipping computation when set membership is *guaranteed to be false*, and at worst performing an unnecessary computation when set membership as given by the filter is *true*, although it in fact is *false*.

Bloom filters are very space efficient when the "universe" U of all possible elements—in our case all N possible cell IDs—is large, and the cardinality n of the set actually in the Bloom filter—in our case in particular the query result cardinality $|Q|$—is small. That is, when

$$n = |Q| \ll N = |U|. \tag{1}$$

The second case of importance in our context is the number of cell IDs in a given *range bin* of a *local filter* (see below). Probabilistically, the expected false positive rate of Bloom filters can be estimated by [11]

$$r_{\text{fp}} = \left(1 - e^{-kn/m}\right)^k. \tag{2}$$

Here, m is the length of the hash table bit vector,[9] k is the number of hash functions, and n is the number of inserted elements, e.g., $n = |Q|$. It is crucial that Eq. 2 is *completely independent of the universe size* $|U|$, which gives our approach the desired property that it depends on and scales with the expected query cardinality $|Q|$, instead of data size N.

We note that the expected false positive rate for set intersections is higher when Bloom filters of individual sets are combined, as we also do, compared to a direct Bloom filter computation from the exact (but for us unknown) intersected set [11]. Thus, in our approach the probabilistic estimate for increasing d, and correspondingly decreasing $|Q|$, decreases less than given by Eq. 2 with $n = |Q|$. The false positive rate is bounded by the smallest set [11]. The worst case thus is $n = |Q_i|$, for the smallest query $|Q_i|$. However, in practice we observe significantly decreasing false positive rates when d grows (see Table 3 and Fig. 12).

3.4 Local Filters: 1D Attribute Space Projections

Our approach employs 1D projections, where all D attribute dimensions are treated separately and independently. For each attribute A_i, we build a *local filter* comprising multiple Bloom filters (Fig. 6, left). In the pre-processing stage, for each local filter we split the value range of each attribute A_i into a number of *range bins*, covering the entire range from minimum to maximum value of A_i occurring in the data set. The width of each range bin is adaptively chosen such that the resulting cardinality of cells with an attribute value A_i within the bin's range is roughly constant. We compute this in a hierarchical fashion, subdividing the query range at the median position approximated via cumulative histograms; then proceeding recursively for the subranges below and above the median, respectively. This approach seamlessly adapts to non-uniform cell attribute value distributions. In addition, we avoid exceeding a pre-scribed percentage of the total range for any bin.

After the bin boundaries have been chosen, one Bloom filter of size m bits is constructed for each range bin. For each attribute A_i, all range bins together comprise the *local filter* of that attribute. In total, we pre-compute D local filters, each comprising b_i Bloom filters ($i \leq D$).

3.5 Global Filter: Set Intersection via Counting

Our global filter is a single *counting Bloom filter*. That is, the global filter comprises a single vector of m integer *counts*, irrespective of the query or data attribute dimensionalities d ($\leq D$) and D, respectively.

To compute the global filter for a given query Q, we sum the d values of each of the m *local* Bloom filter bits that result from the logical OR of all range bin bit vectors within the query range Q_i, *for all d* attributes A_i ($i \leq d$), see Fig. 6. That is, we compute the global filter as

$$\text{global\_filter}\left[j\right] = \sum_{i=1}^{d} \text{local\_filter}_i\left[j\right], \quad \text{for all } j \in [1,m], \tag{3}$$

where

$$\text{local\_filter}_i\left[j\right] = \text{OR}_{b=b\_min}^{b\_max}\, \text{local\_filter\_range\_bin}_{i,b}\left[j\right]. \tag{4}$$

We note that the size m is required to be the same for all our Bloom filters (global and local): The number of vector entries must match.

[8] For brevity, here we refer to a vector of m bits. In the case of a *counting* Bloom filter, such as our *global filter*, the vector consists of m integer *counts*.

[9] In our *global filter*, the vector comprises m integer *counts* instead of bits. However, all other arguments, in particular the false positive rate, stay the same.

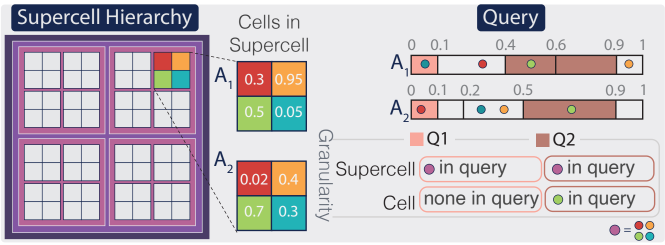


Fig. 7. **Attribute co-occurrence granularity.** Because joint queries depend on *spatial* locations where attribute values occur *jointly*, they depend on the *granularity* at which spatial "co-occurrence" is determined. Queries may give overly *conservative* answers depending on granularity: From single *cell*, to *supercell* granularity of $2^{3l}n^3$ cells, $l \geq 0$. Here, $n = 2$. (Also note that we refer to 3D cell arrangements, but show only 2D here.)

Membership query. The summation above results in set membership information from which both AND as well as OR query results can be obtained: (1) A cell is in the joint d-dimensional query result of an AND query, if its count in the global filter *at all k corresponding hash positions* is equal to the dimensionality d: *Set intersection* results from counting dimensionalities (Fig. 4). Counting also enables incremental filter updates (see below). (2) A cell ID is in the query result of an OR query, if its count in the global filter *at all k corresponding hash positions* is non-zero, i.e., when each count is at least one.

Scalability. Because the number of hash functions k is a small constant (we use $k = 2$ or $k = 3$), this method is faster than scanning over d actual attribute values. The number of accesses is restricted to the constant number k, instead of increasing with query dimensionality $d \leq D$. For $\tilde{N}$ cells in the PVS, only $k \cdot \tilde{N}$ instead of $D \cdot \tilde{N}$ checks are required. Thus, our scalability is independent of the dimensionality D. Moreover, checking d actual attributes requires accessing full attribute data in memory, whereas only querying the global filter *repeatedly* accesses the same hash table, which is beneficial for automatic memory cache utilization. All operations are also trivial to parallelize.

Query updates. Whenever the query is changed, the global filter is either computed from scratch or *updated incrementally*, by subtracting or adding the corresponding vector local_filter$_i$[] (Eq. 4) from the previous vector global_filter[] (Eq. 3), i.e., the previous vector of counts.

It is important to note that our usage of a counting Bloom filter results in the same major properties as those of a regular Bloom filter (in particular, the false positive rate in Eq. 2), except that m integer *counts* are allocated instead of m bits, in contrast to the standard usage of counting Bloom filters [46]. The major reason for this is that we *increase or decrease* counts by *adding or subtracting an entire Bloom filter bit vector* of some attribute A_i to or from the global filter. In contrast, a standard counting Bloom filter adds or subtracts individual elements. Our "element" to add or subtract is an entire bit vector.

Counting global filter vs. counting Bloom filter. Instead of a regular Bloom filter, as used in all local filter range bins, for the single global filter we essentially employ a counting Bloom filter [46], but in detail with a different usage and corresponding properties. Counting enables us to perform *incremental query updates*: Any attribute dimension A_i that has already been inserted into the global filter can also be *subtracted out* again, by simply subtracting the bit vector of an entire attribute A_i. This subtraction is not possible in a standard Bloom filter, but, unlike in our global filter, can also incur problems such as false negatives in standard counting Bloom filters, where individual elements are removed. Together with the trivial addition of attributes (by adding the bit vector of an entire A_i), this enables arbitrary incremental changes to an already computed global filter for faster interactive updates.

Overall, our global filter has four crucial differences to the standard usage and corresponding properties of counting Bloom filters:

- Counting is done such that each attribute A_i is treated together: Individual elements (or range bins) of one A_i *cannot* be removed from the global filter, but entire attributes can be removed exactly.
- Related to this fact, unlike standard counting Bloom filters, deletion of any attribute A_i *cannot result in false negatives* to occur.

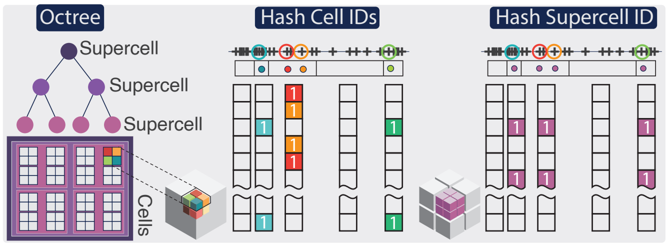


Fig. 8. **Supercells.** Each is assigned one *supercell ID*, but represents all $2^l n \times 2^l n \times 2^l n$ cells ($l \geq 0$) in the spatial extent of a given octree node, for octree leaf nodes ($l = 0$) of n^3 cells (here, $n = 2$). Inserting supercell IDs into all *local filters* enables an early-out strategy during octree traversal, by performing conservative culling queries against the *global filter.* (Here: 3D data illustrated in 1D/2D/3D; only one supercell ID shown hashed.)

- The counts in the global filter are *guaranteed* to be bounded by the number $d \leq D$ (Fig. 6, right). This allows a simple deterministic allocation of $\lceil \log D \rceil$ bits for storing each count value.
- The false positive rate in our global filter is *exactly the same* as that of a regular Bloom filter (Eq. 2). This is not the case for the standard use of a counting Bloom filter [46].

Alternative non-counting global filter. If the capability for incremental global filter updates is not desired, each count could be substituted by a single bit, i.e., a regular Bloom filter could be used. In this case, the *summation* of local filter bits into counts becomes a simple *bitwise AND*. However, if this is done AND and OR queries cannot be fulfilled from the same global filter: Each such query requires a re-computation of the global filter. For these reasons, and because the storage corresponding to the count values of a single global filter is usually not a critical issue, we always employ a counting global filter.

Choice of Bloom filter size. We choose m (the number of counts in the global filter) according to the expected query cardinality $|Q|$, but always choose a size such that $m \ll N$. Table 3 illustrates the impact of different ratios between query size and Bloom filter size, in particular on the false positive rates. We note again that the same size m must also be used for all Bloom filters comprising the local filter range bins.

4 Spatial Hierarchy

Although the first two stages (Fig. 2, left and center) of our approach can be used without visualization, our design is very much targeted toward the characteristics and requirements of visualization. That is, in particular, the traversal of a hierarchical space subdivision such as an octree, and the integration and interaction with visibility determination via view frustum and (progressive) occlusion culling. The third stage of our method therefore comprises octree traversal strategies that interact efficiently with query evaluation (checking the global filter hash table), and performing *conservative* culling, i.e., not visiting octree nodes that are guaranteed to be empty, but maybe visiting some nodes that turn out to be empty after all. In particular, recursive octree traversal must compute and update octree node *occupancy states*. See Algorithm 1.

4.1 Attribute Co-Occurrence Granularity

A crucial issue of *joint* attribute queries is the *spatial granularity* at which the co-occurrence of $1 < d \leq D$ attribute values is determined. Exact co-occurrence would have to be determined for (infinitesimal) spatial points, as to some extent done by several methods, such as fiber surfaces [12]. If, however, we consider it to be sufficient for each attribute value (range) to appear *anywhere* within a spatial region of finite extent, then the accuracy of co-occurrence determination depends on the region size (Fig. 7). For example, two isosurfaces might come as close as the size of the region, but they might not actually intersect. We refer to this phenomenon as *attribute co-occurrence granularity*.[10]

A crucial fact to note is that, even with finite granularity, queries can be evaluated *conservatively* at *any* granularity: If there is no co-occurrence within the region at some granularity, it is *guaranteed* that

[10] It is, however, crucial to note that this is the granularity of "grouping" *full-resolution* data. We do not consider down-sampled multi-resolution data.

attribute values cannot co-occur down to the individual point level. (Interpolation issues are discussed in Sec. 5.3.) As illustrated in Fig. 7, this means that *conservative* query answers can be exploited at different granularities: Because there can be no false negatives, we can use coarse-granularity checks for efficient *early-out* during octree traversal. We do this via regions of $2^{3l}n^3$ cells ($l \geq 0$, n const.) we call *supercells*.

4.2 Supercells

For *conservative culling* with *reduced spatial granularity* (Fig. 7), we introduce the concept of *supercells* to enable an efficient *early-out* strategy in the recursive computation of spatial occupancy state. See Figs. 8, 9, 11, and Algorithm 1. For a regular grid, we assign a *supercell ID* to spatial regions of $2^l n \times 2^l n \times 2^l n$ cells, where $l \geq 0$ determines the *spatial granularity* of the supercell. $l = 0$ refers to the finest granularity. In an octree, each supercell corresponds to one octree node.[11] Thus, the number of supercells equals the number of octree nodes. It is crucial that, although each node is assigned only a single supercell ID, the supercell corresponds to *all attribute values* of the $2^{3l}n^3$ cells within the spatial extent of the node, where $l = 0$ for leaf nodes of size n^3.

Projecting supercell IDs into the local filters. In the pre-processing stage, we insert all supercell IDs into all *local filters* as follows: Denoting the number of octree levels by ℓ, every cell corresponds to additional ℓ supercell IDs. (These IDs are not exclusive: Many other cells share the same supercell IDs.) We now simply compute, for every cell out of N cells, $\ell + 1$ IDs (one cell ID, plus ℓ supercell IDs), and insert all of these IDs into all local filters, according to the cell's attribute values.

Different cells with the same supercell IDs will be inserted multiple times, but the bit vectors of the local Bloom filters will not have duplicates (there is no counting). In the worst case, a single supercell ID can end up in all bins of all local filters (but only once per bin at most). For real data, the "spread" of supercell IDs over attribute ranges will be much less. The supercell ID of the octree root node will end up in the largest number of bins of all supercells (typically in all bins). This approach essentially "links" the spatial domain and the attribute space.

Overhead of supercells. Any disadvantage of supercells during run time is negligible. For example, for a volume that is a cube of $N = M^3$ cells, with a power-of-two side length $M = 2^L$, $L \geq 0$, i.e., M cells in each of the three spatial dimensions, and a leaf node size of n^3 cells (n a power of two), for the total number of supercells we obtain the bound

$$\#\text{supercells} < \frac{8M^3}{7n^3} = \frac{8N}{7n^3}. \tag{5}$$

Considering false positives in Bloom filters, the false positive rate (Eq. 2) corresponds to the number of inserted IDs, and therefore inserting additional IDs increases the false positive rate. However, the above formula shows that the number of supercell IDs that will be inserted in addition to the N cell IDs is very low. For example, for octree leaf nodes with n^3 cells, for $n = 4$ the number of additional supercell IDs is $1.79\% \cdot N$, for $n = 8$ it is $0.22\% \cdot N$, and for $n = 16$ it is $0.028\% \cdot N$.

4.3 Occupancy State Propagation

At run time, we can perform query evaluation and the corresponding determination of a binary *occupancy state* per node during recursive traversal of the spatial hierarchy (e.g., an octree). See Algorithm 1.

Every visited octree node first checks whether its occupancy state is already valid, i.e., whether it has been computed and cached before. If this is not the case, then we perform a *conservative* test for a guaranteed *false* occupancy state update using the node's supercell ID. This is always correct: (1) Supercells cannot give false negatives (see above and Fig. 7). (2) The global filter also has guaranteed no false negatives. However, the test can be overly conservative due to supercell regions of $2^{3l}n^3$ cells. Despite this, accuracy naturally increases exponentially, as supercell granularity decreases during octree traversal (l decreases).

Early-out with supercell IDs. Because there cannot be false negatives, for every node visited during octree traversal, before we visit child nodes or individual cells (for leaf nodes), we first query the supercell ID against the global filter for a conservative test. If the ID is not in the global filter, we know that the occupancy state of the entire subtree starting with the current node is *false*. See Fig. 9 and Algorithm 1.

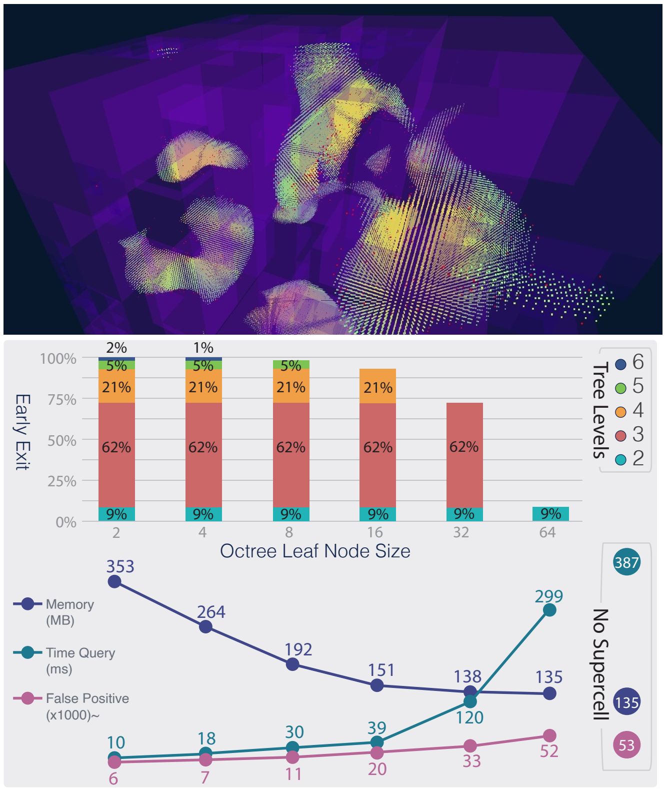


Fig. 9. **Early-out with supercells.** (Top) Octree nodes that exit tree traversal early are shown in purple. (Bottom) Efficiency of early-out with supercells for different octree leaf node sizes n^3 ($n = 2$ to $n = 64$): Percentages of nodes exiting on a given tree level (root is level 0). With increasing node size, memory consumption decreases, but query times and false positive rates increase. Here, the sweet spot is around $n = 16$.

Algorithm 1: Spatial Traversal and Occupancy Updates ($k = 2$)

```
function queryGlobalFilter ( $cellID$, $d$ )
$hash1 = $ getHash1($cellID$);
$hash2 = $ getHash2($cellID$);
if ($globalFilter$.getCount($hash1$) $= d$) and
  ($globalFilter$.getCount($hash2$) $= d$) then
    return $true$;
else
    return $false$;
end
function traverseTreeGetOccupancy ( $node$, $d$ )
if $node$.isCachedOccupancyValid() then
    return $node.occupancyState$;
end
if !queryGlobalFilter($node.superCellID$) then
    $node.occupancyState = false$;
    return $node.occupancyState$;
end
if !$node$.isLeafNode() then
    $node.occupancyState =$
      $node$.visitAllChildNodesGetAggregateOccupancy($d$);
else
    $node.occupancyState =$
      $node$.visitAllCellsGetAggregateOccupancy($d$);
end
return $node.occupancyState$;
```

[11] We emphasize that this has nothing to do with *multi-resolution* rendering using octrees: For us, the octree just provides a hierarchical "spatial grouping."

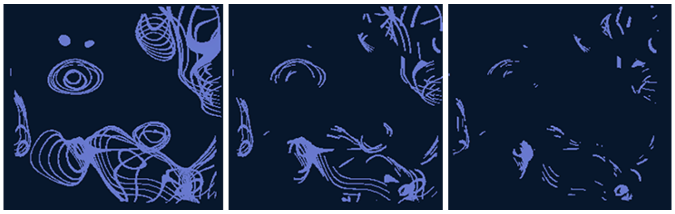


Fig. 10. **Multi-dimensional intersections.** With growing number of attribute dimensions in an AND query, fewer and fewer cells are selected, i.e., the query cardinality $|Q|$ gets smaller. From left to right, $d = 1,2,3$.

Propagation up the tree. If the conservative check using the supercell ID did not provide a guaranteed negative answer, we need to traverse further. If the node is not a leaf node, all child nodes need to be visited, and the occupancy state of the current node is the logical OR of the states of all child nodes. If the node is a leaf node, recursive traversal stops, and all cells within the spatial extent of the node are queried against the global filter. The occupancy state of the node is then the logical OR of all individual cell states. See Algorithm 1.

Asynchronous cell queries. The individual cell queries for leaf nodes can be scheduled to be computed asynchronously. Once the asynchronous computation of a leaf node finishes, the node's occupancy state is known and cached. On the next octree traversal, the node's occupancy state will then be propagated up the tree.

Eliminating false positives. If a completely accurate (i.e., guaranteed no false positives) result is required, we can eliminate false positives by visiting leaf nodes and for each of the contained cells evaluate the exact query using scanning, i.e., by checking all attributes individually. This operation can also be scheduled to be computed asynchronously, progressively updating the occupancy state of tree nodes to eliminate false positives. We note that cached occupancy states in the tree that were already set to valid need to be invalidated for the part of the tree above each leaf node that is being updated (and whose occupancy state in fact changed due to false positive elimination).

5 Results and Evaluation

We evaluate our method using the data sets listed in Table 2, consisting of two real data sets from different scientific domains (meteorology and oceanography), which use physical attributes such as temperature, humidity, pressure; and a synthetic test data set based on Perlin Noise [44] with different frequencies and seed numbers per attribute. We also compare our results with other techniques for multivariate filtering.

5.1 Implementation

Our framework is implemented in C++ and OpenGL. All tests rely on our own implementations, using the C++ Standard Template Library, except for the R-tree, for which we used the fast template library by Guttman et al. [23], and the compression of Bloom filter bit vectors, for which we used the libraries by Lemire et al. [30]. Our system supports data coming from different kinds of sources, and can be easily integrated into larger or more complex visualization systems.

All our results were measured on an Intel Xeon 6230R with 128 GB, and a NVIDIA RTX 3090 GPU, running Windows 10. To facilitate fair comparisons, for all performance tests we used only a single CPU core and no GPU implementation, except for standard OpenGL rendering.

5.2 Evaluation

In Table 2, we compare our technique, both with the use of supercells (SC) and without (BF; only for comparison purposes), with other techniques for multivariate filtering in visualization: (1) Scanning; checks each cell and attribute individually (also determines ground truth); (2) MinMax octree; uses an octree with fixed-size leaf nodes of 16^3 cells, testing a $(\min, \max)$ value pair per attribute per node; and (3) an R-tree.

We evaluate three data sets with three queries each, of dimensions $d = 1$ (Q1), $d = 2$ (Q2), and $d = 3$ (Q3), respectively. The query cardinalities $|Q|$ of each of these queries are given in Table 3.

Pre-processing. Table 2 reports the times for building each basic data structure. For our method, this is the creation of the *local filters*. This pre-processing is an offline process that is done just once. For our data sets, it takes from several minutes to one or two hours. In our approach, the total *Storage Size* of all local filters, as well as the *Pre-Processing* time, depend directly on the number of range bins computed. However, query evaluation only has to load or stream the small subset of bins within the actual query range under consideration.

Query performance (global filter construction). *Create GF* in Table 2 is the time to construct the *global filter* by summing all local filter range bins within the query ranges Q_i, representing the query result set Q by the global filter. This is done only once for each new query, and takes from a few milliseconds for the Red Sea data set to a few seconds in the worst cases, which still facilitates user interaction.

Query performance (global filter querying). In Table 2, *Query Time* is the time for computing the *occupancy state* of all cells, which is the final query result. Query times include the time needed to eliminate false positives. (For comparison, SC times in parentheses are without false positive elimination.) In order to avoid confusing measurements for query evaluation with on-screen visibility, all reported numbers are for complete data set traversals without any PVS determination, i.e., for $\tilde{N} = N$. We note, however, that this eliminates one clear advantage

Table 2. **Method comparisons.** We compare our method without (BF) and with supercells (SC), respectively, with (1) Scanning d attributes, (2) MinMax (octree with 16^3 leaves; test d out of D min-max per node, (3) D-dimensional R-tree. Our results (BF, SC) use $m = 25$ %N, $k = 2$. Our query times include false positive elimination (SC times in parentheses without). BF query times are for comparison only; no hierarchy is used.

	Scanning	MinMax	R-tree	BF	**SC**
GFS Meteorology Prediction. 2,048x1,024x512 (N = 1,073,741,824), D = 8.					
Pre-Processing [min]	-	8.75	194.76	25.33	231.66
Storage Size [MB]	-	31.99	79,591	18,814	20,363
Global Filter [MB]	-	-	-	256	256
Q1 Create GF [s]	-	-	-	3.96	4.11
(*d*=1) Query Time [s]	39.76	15.60	59.48	11.21	**4.25** (3.44)
FPR [%]	-	-	-	43.02	10.07
Q2 Create GF [s]	-	-	-	7.13	7.04
(*d*=2) Query Time [s]	77.46	8.00	41.95	11.26	**3.65** (2.19)
FPR [%]	-	-	-	23.22	4.33
Q3 Create GF [s]	-	-	-	7.24	7.77
(*d*=3) Query Time [s]	110.01	5.38	34.84	11.22	**2.91** (1.43)
FPR [%]	-	-	-	1.10	0.17
Red Sea. 500x500x50 (N = 12,500,000), D = 6.					
Pre-Processing [min]	-	0.08	1.08	0.08	0.28
Storage Size [MB]	-	0.37	757.48	164	166
Global Filter [MB]	-	-	-	2.98	2.98
Q1 Create GF [s]	-	-	-	0.003	0.002
(*d*=1) Query Time [s]	0.48	0.08	0.31	0.14	**0.04** (0.01)
FPR [%]	-	-	-	2.76	0.18
Q2 Create GF [s]	-	-	-	0.004	0.004
(*d*=2) Query Time [s]	0.91	0.07	0.37	0.16	**0.03** (0.007)
FPR [%]	-	-	-	1.48	0.07
Q3 Create GF [s]	-	-	-	0.006	0.006
(*d*=3) Query Time [s]	1.24	0.07	0.26	0.12	**0.02** (0.006)
FPR [%]	-	-	-	0.19	0.02
Synthetic Perlin Noise. 512x512x512 (N = 134,217,728), D = 15.					
Pre-Processing [min]	-	2.17	28.50	4.14	12.94
Storage Size [MB]	-	7.49	18,486	4,490	4,842
Global Filter [MB]	-	-	-	32	32
Q1 Create GF [s]	-	-	-	0.28	0.23
(*d*=1) Query Time [s]	5.14	4.53	6.32	2.02	**1.25** (1.11)
FPR [%]	-	-	-	29.82	20.96
Q2 Create GF [s]	-	-	-	0.16	0.17
(*d*=2) Query Time [s]	9.59	0.40	2.68	1.44	**0.23** (0.02)
FPR [%]	-	-	-	0.21	0.10
Q3 Create GF [s]	-	-	-	0.35	0.34
(*d*=3) Query Time [s]	14.36	0.30	1.73	1.40	**0.23** (0.002)
FPR [%]	-	-	-	0.08	0.005

Table 3. **Evaluation of false positives for different Bloom filter sizes.** Bloom filter size m, false positives (FP), true negatives (TN), and true negatives skipped by supercell early-out (SC-Skip) are given in % of the data size N. The false positive rate (FPR; Eq. 6) is in %. We compare our method without (BF) and with (SC) the use of supercells. See Fig. 12.

		m [%N]	FP [%N]	TN [%N]	FPR [%]	SC-Skip [%N]
GFS Meteorology Prediction. 2,048x1,024x512 (N = 1,073,741,824), D = 8.						
Q1 (d=1) $\lvert Q\rvert$ = TP = 13.3 %N	BF	50	14.82	71.86	17.10	
	SC		**2.16**	**84.51**	**2.50**	**74.06**
	BF	25	37.29	49.39	43.02	
	SC		**8.73**	**77.95**	**10.07**	**66.41**
	BF	15	59.87	26.80	69.07	
	SC		**27.79**	**58.88**	**32.07**	**46.45**
Q2 (d=2) $\lvert Q\rvert$ = TP = 2.12 %N	BF	50	6.11	91.78	6.24	
	SC		**1.05**	**96.83**	**1.08**	**88.03**
	BF	25	22.73	75.15	23.22	
	SC		**4.24**	**93.64**	**4.33**	**83.86**
	BF	15	49.01	48.87	50.07	
	SC		**15.43**	**82.46**	**15.76**	**69.22**
Q3 (d=3) $\lvert Q\rvert$ = TP = 0.03 %N	BF	50	0.19	99.77	0.19	
	SC		**0.03**	**99.93**	**0.03**	**93.28**
	BF	25	1.10	98.86	1.10	
	SC		**0.16**	**99.79**	**0.17**	**90.92**
	BF	15	3.97	95.99	3.97	
	SC		**0.85**	**99.10**	**0.86**	**83.17**

of our method: The spatial hierarchy traversal for query evaluation can be the same as the one that is already used for rendering and visibility culling, efficiently skipping all subtrees outside the PVS completely.

Memory footprint. MinMax (with fixed-size blocks of 16^3 cells) stores only a small amount of data ($2 \cdot D$ floats per block), whereas an R-tree needs much more space. In contrast, our probabilistic approach is more than three times smaller than the R-tree (in terms of pre-computed local filter storage), and it scales better with the number of attribute dimensions D. The memory overhead of supercell IDs in (compressed) local filter storage is around 10%. In Table 2, *Global Filter* is the (uncompressed) size of the global filter, which is independent of D and only depends on the chosen Bloom filter hash table size m. Since the global filter is quite small, it is not compressed and therefore has the same size whether or not supercells are used. Although we have not implemented it, the global filter could easily be stored in GPU memory.

False positive rate evaluation. For concrete results, here we report *measured* false positive rates (Table 3, Fig. 12) instead of probabilistic estimates (Eq. 2). The measured false positive rate is defined as

$$FPR = \frac{FP}{FP + TN}. \qquad (6)$$

FP is the number of false positives, TN the number of true negatives, and $FP + TN$ the total number of ground truth negatives; see Fig. 11.

Table 3 reports how the Bloom filter size m influences the false positive rates for different queries, as well as how much the use of supercells decreases both the number of true negatives that are tested individually, as well as the false positive rate (see below). Fig. 12 visualizes these results. A more complete evaluation is provided in Table 6 and in Figs. 13, 14, and 15 in the supplementary appendices.

It is crucial to note that the use of our supercells with hierarchical traversal and early-out usually results in a *significant decrease* of the false positive rate. The reason for this is that in our early-out strategy a single supercell ID represents many cell IDs, incurring the false positive probability only once per supercell instead of once for each cell.

5.3 Discussion

Interaction use cases. In a typical user interaction scenario, only a small number of nodes will become visible whose occupancy state has not already been cached before. For this reason, while the timings given in Table 2 are given for whole data sets, our approach is usually fully interactive. The creation of a new global filter is not always interactive, but can be made much faster by incremental changes from a previous query. However, even a full update is still faster than previous work.

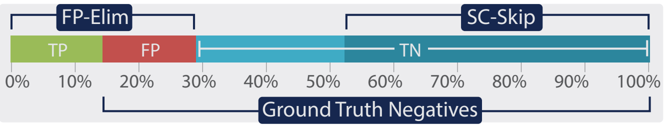


Fig. 11. **True positives (TP), false positives (FP), true negatives (TN).** The false positive rate is $FPR = FP/(FP + TN)$, for $FP + TN$ ground truth negatives. To eliminate all false positives (FP-Elim.), we only need to scan $TP + FP$ cells, instead of all $N = TP + FP + TN$ cells. Moreover, we use supercells with early-out to skip a significant part of TN (SC-Skip).

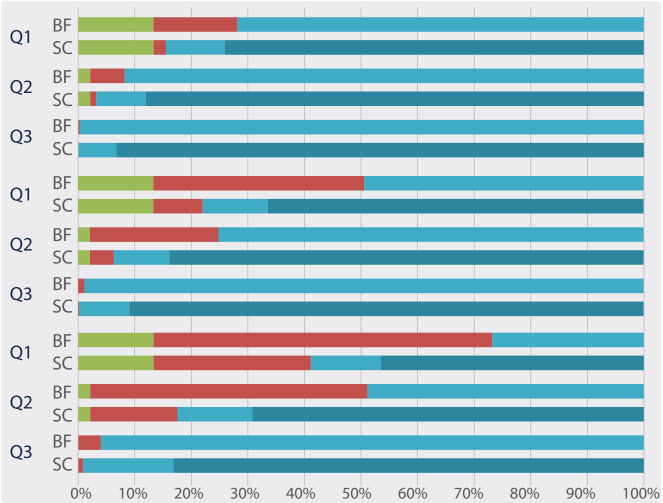


Fig. 12. **Evaluation of false positives for different Bloom filter sizes.** Stacked bar charts corresponding to Table 3; but here the results are grouped according to $m = 50$ %N, $m = 25$ %N, $m = 15$ %N (top to bottom).

Data interpolation. In this paper, we have focused on data attributes that are not interpolated. When attribute values should be interpolated to reconstruct a continuous field (e.g., interpolating from grid cell vertices), this has to be taken into account for accurate query results. One way to integrate interpolation into our method would be to treat this case similarly to how we compute supercells: Each cell with interpolated data corresponds to a (min, max) pair of values, and the cell's ID would have to be projected into all local filter range bins that overlap its (min, max) range. However, we leave this for future work.

Structured vs. unstructured grids. Since we treat individual cells as data points with a set of attribute values per point, our approach does not depend on the type of grid (structured or unstructured) that is used, and can also directly be applied to meshless (e.g., point-based) data.

6 Conclusions

Our probabilistic data structure and output-sensitive query method have several important benefits regarding scalability with data size and, in particular, the number of attribute dimensions. The main target of our framework are logical AND queries in high-dimensional attribute spaces. The major idea of our query method is that query evaluation should be as *output-sensitive* as possible. Regarding the query evaluation by itself, we achieve this by targeting scalability with the cardinality $|Q|$ of the query result set Q, which is particularly useful for *joint* queries of many attributes. Moreover, our approach is also output-sensitive regarding the visible part of the data set in the visualization, by integrating directly with standard octree traversal or other hierarchical space subdivisions. The use of our concept of supercells is particularly useful to speed up query time, both for view-independent queries—where the supercells allow for efficient early-out during hierarchical query evaluation—as well as during rendering, where standard view frustum and occlusion culling can integrate directly with supercell-based skipping of whole subtrees of the hierarchical space subdivision.

Acknowledgments

This work was supported by King Abdullah University of Science and Technology (KAUST), and was also supported in part by a grant from Saudi Aramco (#3879).

References

[1] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. In *International Conference on Very Large Data Bases (VLDB)*, pp. 329–338, 2000.

[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 459–468, 2006.

[3] G. Antoshenkov. Byte-aligned bitmap compression. In *Data Compression Conference (DCC)*, p. 476, 1995.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. *SIGMOD Record*, 19(2):322–331, 1990.

[5] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *VLDB Endowment*, 5(11):1627–1637, 2012.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[7] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[8] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[9] J. Beyer, H. Mohammed, M. Agus, A. K. Al-Awami, H. Pfister, and M. Hadwiger. Culling for extreme-scale segmentation volumes: A hybrid deterministic and probabilistic approach. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):1132–1141, 2019.

[10] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[11] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2007.

[12] H. Carr, Z. Geng, J. Tierny, A. Chattopadhyay, and A. Knoll. Fiber Surfaces: Generalizing isosurfaces to bivariate data. *Computer Graphics Forum*, 34(3):241–250, 2015.

[13] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719, 2016.

[14] C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. *SIGMOD Record*, 28(2):215–226, 1999.

[15] A. Chaudhuri, T. H. Wei, T. Y. Lee, H.-W. Shen, and T. Peterka. Efficient range distribution query for visualizing scientific data. In *IEEE Pacific Visualization Symposium (PacificVis)*, pp. 201–208, 2014.

[16] S. Cohen and Y. Matias. Spectral Bloom filters. In *ACM SIGMOD International Conference on Management of Data*, pp. 241–252, 2003.

[17] A. Crainiceanu and D. Lemire. Bloofi: Multidimensional Bloom filters. *Information Systems*, 54:311–324, 2015.

[18] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pp. 75–88, 2014.

[19] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[20] Y. Fu and E. Biersack. False-positive probability and compression optimization for tree-structured Bloom filters. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(4):19:1–19:39, 2016.

[21] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. *SIGMOD Record*, 29(2):463–474, 2000.

[22] A. Guttman. R-Trees: A dynamic index structure for spatial searching. *SIGMOD Record*, 14(2):47–57, 1984.

[23] A. Guttman, M. Stonebraker, and G. Douglas. R-Trees: a dynamic index structure for spatial searching. `https://github.com/nushoin/RTree`, 2021.

[24] M. Hadwiger, R. Sicat, J. Beyer, J. Krüger, and T. Möller. Sparse PDF maps for non-linear multi-resolution image operations. *ACM Transactions on Graphics*, 31(6):133:1–133:12, 2012.

[25] Y. Hua, D. Feng, and T. Xie. Multi-dimensional range query for data management using Bloom filters. In *IEEE International Conference on Cluster Computing*, pp. 428–433, 2007.

[26] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ACM Symposium on Theory of Computing (STOC)*, pp. 604–613, 1998.

[27] J. Kehrer and H. Hauser. Visualization and Visual Analysis of Multifaceted Scientific Data: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):495–513, 2013.

[28] M. Kratky, V. Snasel, J. Pokorny, and P. Zezula. Efficient processing of narrow range queries in multi-dimensional data structures. In *International Database Engineering and Applications Symposium (IDEAS)*, pp. 69–79, 2006.

[29] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[30] D. Lemire, L. Boytsov, and N. Kurz. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience*, 46(6):723–749, 2016.

[31] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, 2018.

[32] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience*, 46(11):1547–1569, 2016.

[33] S. Liu, J. A. Levine, P.-T. Bremer, and V. Pascucci. Gaussian mixture model based volume visualization. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 73–77, 2012.

[34] Y. Livnat. Accelerated isosurface extraction approaches. In C. D. Hansen and C. R. Johnson, eds., *Visualization Handbook*, pp. 39–55. Butterworth-Heinemann, Burlington, 2005.

[35] Y. Livnat, H.-W. Shen, and C. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[36] K. Lu and H.-W. Shen. A compact multivariate histogram representation for query-driven visualization. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 49–56, 2015.

[37] Y. Lu, L. Cheng, T. Isenberg, C.-W. Fu, G. Chen, H. Liu, O. Deussen, and Y. Wang. Curve Complexity Heuristic KD-trees for Neighborhood-based Exploration of 3D Curves. *Computer Graphics Forum*, 40(2):461–474, 2021.

[38] G. S. Lueker. A data structure for orthogonal range queries. In *Symposium on Foundations of Computer Science (SFCS)*, pp. 28–34, 1978.

[39] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *ACM SIGMOD International Conference on Management of Data*, pp. 448–459, 1998.

[40] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[41] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[42] M. Muralikrishna and D. J. DeWitt. Equi-depth multidimensional histograms. *SIGMOD Record*, 17(3):28–36, 1988.

[43] P. E. O'Neil. Model 204 architecture and performance. In D. Gawlick, M. Haynie, and A. Reuter, eds., *High Performance Transaction Systems*, pp. 39–59. Springer Berlin Heidelberg, 1989.

[44] K. Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296, 1985.

[45] E. Pitoura. Selectivity estimation. In L. Liu and M. T. Özsu, eds., *Encyclopedia of Database Systems*, p. 2548. Springer, Boston, MA, 2009.

[46] S. Pontarelli, P. Reviriego, and J. A. Maestro. Improving counting Bloom filter performance with fingerprints. *Information Processing Letters*, 116(4):304–309, 2016.

[47] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *International Conference on Very Large Data Bases (VLDB)*, pp. 486–495, 1997.

[48] F. Porikli. Integral histogram: a fast way to extract histograms in Cartesian spaces. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 829–836, 2005.

[49] A. Sayar, S. Eken, and O. Öztürk. Kd-tree and quad-tree decompositions for declustering of 2D range queries over uncertain space. *Frontiers of Information Technology & Electronic Engineering*, 16:98–108, 2015.

[50] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-Tree: A dynamic index for multi-dimensional objects. In *International Conference on Very Large Data Bases (VLDB)*, pp. 507–518, 1987.

[51] H.-W. Shen, C. Hansen, Y. Livnat, and C. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *IEEE Visualization*, pp. 287–294, 1996.

[52] Q. Shi and J. JaJa. Isosurface extraction and spatial filtering using per-

sistent octree (POT). *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1283–1290, 2006.

[53] R. Sicat, J. Krüger, T. Möller, and M. Hadwiger. Sparse PDF volumes for consistent multi-resolution volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2417–2426, 2014.

[54] R. R. Sinha and M. Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Transactions on Database Systems*, 32(3):16–es, 2007.

[55] S. Sprenger, P. Schäfer, and U. Leser. Multidimensional range queries on modern hardware. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, pp. 1–12, 2018.

[56] S. Sprenger, P. Schäfer, and U. Leser. BB-tree: A main-memory index structure for multidimensional range queries. In *International Conference on Data Engineering (ICDE)*, pp. 1566–1569, 2019.

[57] K. Stockinger, J. Shalf, K. Wu, and E. Bethel. Query-driven visualization of large data sets. In *IEEE Visualization*, pp. 167–174, 2005.

[58] K. Stockinger, K. Wu, and A. Shoshani. Evaluation strategies for bitmap indices with binning. In F. Galindo, M. Takizawa, and R. Traunmüller, eds., *Database and Expert Systems Applications*, pp. 120–129. Springer Berlin Heidelberg, 2004.

[59] D. Thompson, J. A. Levine, J. C. Bennett, P.-T. Bremer, A. Gyulassy, V. Pascucci, and P. P. Pébay. Analysis of large-scale scalar data using hixels. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 23–30, 2011.

[60] C. Wang and Y.-J. Chiang. Isosurface extraction and view-dependent filtering from time-varying fields using persistent time-octree (PTOT). *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1367–1374, 2009.

[61] K.-C. Wang, K. Lu, T.-H. Wei, N. Shareef, and H.-W. Shen. Statistical visualization and analysis of large data using a value-based spatial distribution. In *IEEE Pacific Visualization Symposium (PacificVis)*, pp. 161–170, 2017.

[62] T.-H. Wei, C.-M. Chen, and A. Biswas. Efficient local histogram searching via bitmap indexing. *Computer Graphics Forum*, 34(3):81–90, 2015.

[63] T.-H. Wei, C.-M. Chen, J. Woodring, H. Zhang, and H.-W. Shen. Efficient distribution-based feature search in multi-field datasets. In *IEEE Pacific Visualization Symposium (PacificVis)*, pp. 121–130, 2017.

[64] M. A. Westenberg, J. B. T. M. Roerdink, and M. H. F. Wilkinson. Volumetric attribute filtering and interactive visualization using the max-tree representation. *IEEE Transactions on Image Processing*, 16(12):2943–2952, 2007.

[65] K. Wu, E. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *International Conference on Scientific and Statistical Database Management*, pp. 99–108, 2002.

[66] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *International Conference on Very Large Data Bases (VLDB)*, pp. 24–35, 2004.

[67] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[68] H. Younesy, T. Möller, and H. Carr. Improving the quality of multi-resolution volume rendering. In *Eurographics / IEEE VGTC Conference on Visualization (Eurovis)*, p. 251–258, 2006.

[69] C. Yu. *High-Dimensional Indexing: Transformational Approaches to High-Dimensional Range and Similarity Searches*. Springer Berlin Heidelberg, 2002.

[70] T. Zäschke, C. Zimmerli, and M. C. Norrie. The PH-Tree: A space-efficient storage structure and multi-dimensional index. In *ACM SIGMOD International Conference on Management of Data*, p. 397–408, 2014.

[71] Y. Zhao, Y. Wang, J. Zhang, C.-W. Fu, M. Xu, and D. Moritz. KD-Box: Line-segment-based kd-tree for interactive exploration of large-scale time-series data. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):890–900, 2022.

A Queries for Evaluation

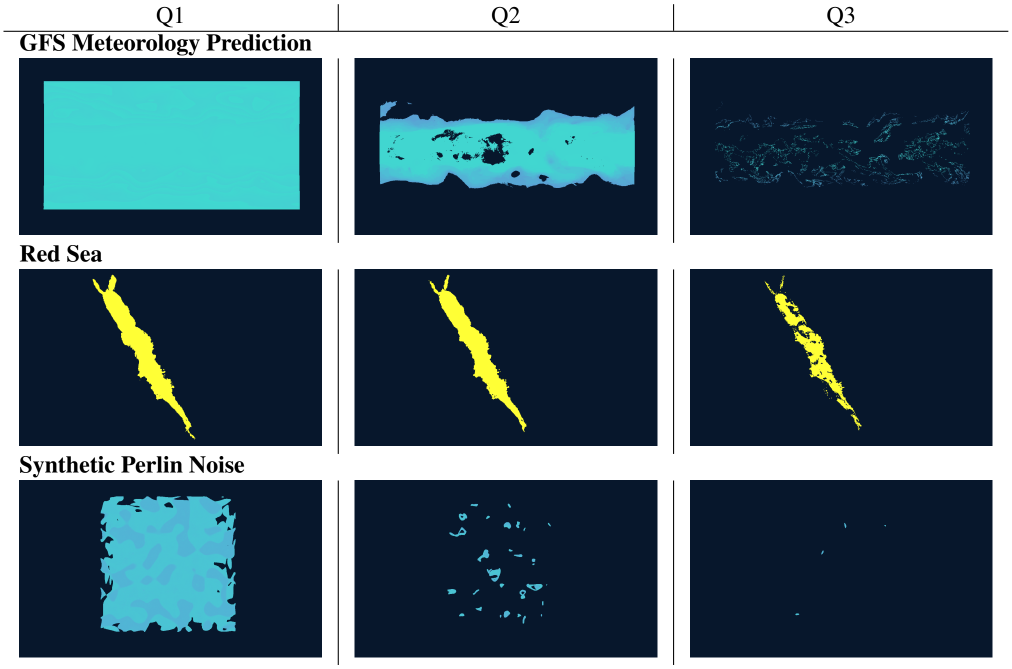
Table 4 lists the basic characteristics of each of the three queries Q1, Q2, and Q3, respectively, for each of three data sets, respectively, that we have used for the evaluation. In particular, we give the dimensionality of the query (d), i.e., the number of attributes that it is querying, and the resulting query cardinality $|Q|$, which is also the number of *true positives* (TP) of the query (see Fig. 11).

Table 5 depicts example visualizations of the results of these queries.

Table 4. **Queries used for evaluation.** For each of three data sets, we have used the following three queries with the properties given below.

Query	d (dimensionality)	$\|Q\|$ (cardinality)	$\|Q\|$ in % of data size N
GFS Meteorology Prediction			
2048x1024x512 Cells ($N = 1,073,741,824$).			
Q1	1	143,112,968	13.33 %
Q2	2	22,753,803	2.12 %
Q3	3	369,000	0.03 %
Red Sea			
500x500x50 Cells ($N = 12,500,000$).			
Q1	1	283,744	2.27 %
Q2	2	201,418	1.61 %
Q3	3	62,675	0.50 %
Synthetic Perlin Noise			
512x512x512 Cells ($N = 134,217,728$).			
Q1	1	13,235,663	9.86 %
Q2	2	76,496	0.06 %
Q3	3	1,443	0.001 %

Table 5. **Queries used for evaluation.** Example visualizations of the results of each query used for evaluation.



B Evaluation of False Positive Rates

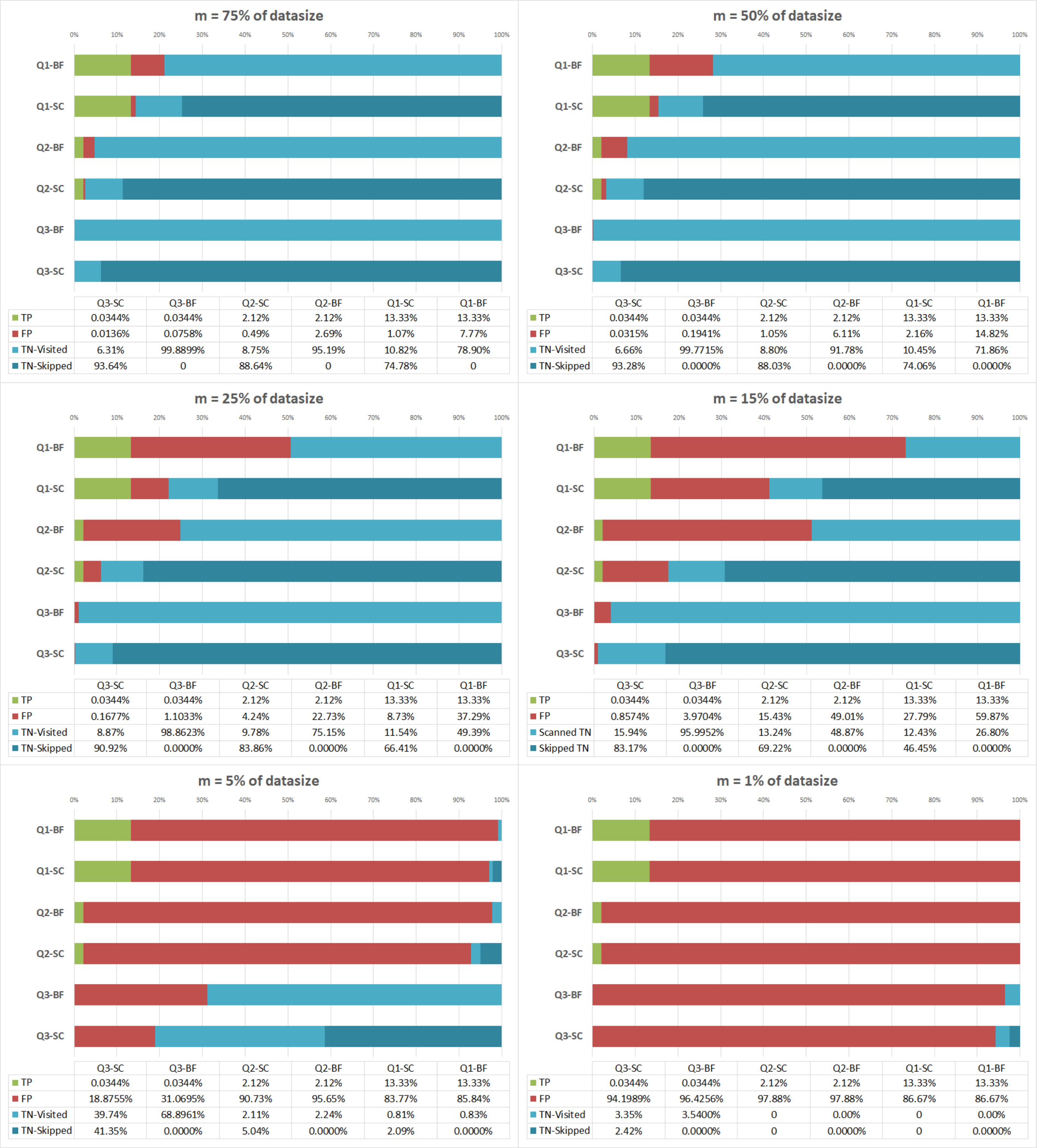
Table 6 gives a comprehensive evaluation of false positive rates and the overall impact of false positives for different Bloom filter sizes, different queries (Q1, Q2, Q3), and different data sets. Figs. 13, 14, and 15 visualize the results given in Table 6 with stacked bar charts, using the encoding described in Fig. 11. The false positive rate (FPR) is computed according to Eq. 6.

Standard scanning vs. scanning for false positive removal. We note that a standard scanning approach must scan all $N = TP + FP + TN$ cells in the data set. In our approach, we only need to perform scanning in order to eliminate false positives. Due to the fact that we do not a priori know which are the false positives, the number of cells that need to be scanned to eliminate all false positives is $TP + FP$, which for not too high false positive rates is much smaller than N. The percentage of cells eliminated from scanning is therefore $TN/N = TN/(TP + FP + TN)$. See Fig. 11.

Cost of determining *TP* and *FP*. In contrast to standard scanning, however, we need to check cell IDs against the Bloom filter, i.e., we need to hash the IDs and check against the Bloom filter bit vector. However, the number of cell IDs that needs to be checked is drastically reduced by the use of supercells. Hierarchical supercell culling *skips* many IDs contained in large subtrees for which the corresponding supercell ID gives a true negative result in the global filter. The percentage of skipped cells is reported in Table 6 (SC-Skip).

Table 6. **Evaluation of false positives for different Bloom filter sizes, different queries (Q1, Q2, Q3), and different data sets.** The Bloom filter bit vector size m, the number of true positives (TP), which is the query cardinality $|Q|$, false positives (FP), true negatives (TN), the number of cells that need to be scanned to eliminate all false positives (FP-Elim.), determined by TP+FP, and the number of true negatives that are not tested individually when hierarchical supercell culling with our early-out strategy is used (SC-Skip), are numbers of cells given in % of the data size N (denoted by [%N]). The false positive rate (FPR) is the ratio determined by $FPR = FP/(FP + TN)$, given in percent. We compare Bloom filters without (BF) and with (SC) the use of supercells, respectively. Improvements from the use of supercells (BF-SC) are given in percentage points ([p.p.]).

Query	m [%N]	$|Q| = $ TP [%N]	FP [%N]		TN [%N]		FPR [%]			FP-Elim. (TP+FP) [%N]			SC-Skip [%N]
			BF	SC	BF	SC	BF	SC	BF-SC [p.p.]	BF	SC	BF-SC [p.p.]	
GFS Meteorology Prediction. 2048x1024x512 Cells ($N = 1,073,741,824$).													
Q1	75	13.33	7.77	1.07	78.90	85.60	8.96	1.24	7.73	21.10	14.40	6.70	74.78
	50	13.33	14.82	2.16	71.86	84.51	17.10	2.50	14.60	28.14	15.49	12.65	74.06
	25	13.33	37.29	8.73	49.39	77.95	43.02	10.07	32.95	50.61	22.05	28.56	66.41
	15	13.33	59.87	27.79	26.80	58.88	69.07	32.07	37.01	73.20	41.12	32.08	46.45
	5	13.33	85.84	83.77	0.83	2.90	99.04	96.65	2.39	99.17	97.10	2.07	2.09
	1	13.33	86.67	86.67	0.00	0.00	100	100	0.00	100	100	0.00	0.00
Q2	75	2.12	2.69	0.49	95.19	97.39	2.75	0.50	2.25	4.81	2.61	2.20	88.64
	50	2.12	6.11	1.05	91.78	96.83	6.24	1.08	5.16	8.22	3.17	5.05	88.03
	25	2.12	22.73	4.24	75.15	93.64	23.22	4.33	18.89	24.85	6.36	18.49	83.86
	15	2.12	49.01	15.43	48.87	82.46	50.07	15.76	34.31	51.13	17.54	33.58	69.22
	5	2.12	95.65	90.73	2.24	7.15	97.72	92.69	5.02	97.76	92.85	4.92	5.04
	1	2.12	97.88	97.88	0.00	0.00	100	100	0.00	100	100	0.00	0.00
Q3	75	0.03	0.07	0.01	99.88	99.95	0.08	0.01	0.06	0.11	0.05	0.06	93.64
	50	0.03	0.19	0.03	99.77	99.93	0.19	0.03	0.16	0.23	0.07	0.16	93.28
	25	0.03	1.10	0.16	98.86	99.79	1.10	0.17	0.94	1.14	0.20	0.94	90.92
	15	0.03	3.97	0.85	95.99	99.10	3.97	0.86	3.11	4.00	0.89	3.11	83.17
	5	0.03	31.06	18.87	68.89	81.09	31.08	18.88	12.20	31.10	18.91	12.19	41.35
	1	0.03	96.42	94.19	3.54	5.76	96.46	94.23	2.23	96.46	94.23	2.23	2.42
Red Sea. 500x500x50 Cells ($N = 12,500,000$).													
Q1	75	2.27	0.34	0.02	97.39	97.71	0.35	0.02	0.33	2.61	2.29	0.32	91.58
	50	2.27	0.74	0.05	96.99	97.68	0.75	0.05	0.71	3.01	2.32	0.69	91.55
	25	2.27	2.70	0.17	95.03	97.56	2.76	0.18	2.58	4.97	2.44	2.52	91.48
	15	2.27	6.67	0.44	91.06	97.29	6.82	0.45	6.37	8.94	2.71	6.22	91.25
	5	2.27	34.76	3.79	62.97	62.97	35.56	3.88	31.68	37.03	6.06	30.96	87.09
	1	2.27	95.66	87.15	2.07	10.58	97.88	89.17	8.71	97.93	89.42	8.51	8.71
Q2	75	1.61	0.18	0.01	98.21	98.38	0.18	0.01	0.17	1.79	1.62	0.17	94.30
	50	1.61	0.39	0.02	98.00	98.37	0.40	0.02	0.38	2.00	1.63	0.37	94.27
	25	1.61	1.46	0.06	96.93	98.32	1.48	0.07	1.41	3.07	1.68	1.39	94.23
	15	1.61	3.74	0.16	94.65	98.23	3.80	0.17	3.63	5.35	1.77	3.58	94.20
	5	1.61	22.80	1.59	75.59	96.80	23.18	1.61	21.56	24.41	3.20	21.22	91.61
	1	1.61	91.90	74.61	6.49	23.78	93.40	75.83	17.57	93.51	76.22	17.29	18.54
Q3	75	0.50	0.02	0.00	99.48	99.50	0.02	0.00	0.02	0.52	0.50	0.02	94.82
	50	0.50	0.02	0.00	99.48	99.50	0.02	0.00	0.02	0.52	0.50	0.02	94.79
	25	0.50	0.19	0.02	99.31	99.48	0.19	0.02	0.18	0.69	0.52	0.18	94.79
	15	0.50	0.56	0.04	98.94	99.46	0.57	0.04	0.52	1.06	0.54	0.52	94.76
	5	0.50	5.68	0.44	93.82	99.06	5.71	0.44	5.27	6.18	0.94	5.25	93.35
	1	0.50	70.60	29.17	28.90	70.33	70.96	29.31	41.65	71.10	29.67	41.44	58.65
Synthetic Perlin Noise. 512x512x512 Cells ($N = 134,217,728$).													
Q1	75	9.86	4.83	2.96	85.31	87.18	5.36	3.28	2.08	14.69	12.82	1.87	35.15
	50	9.86	9.60	6.05	80.54	84.08	10.65	6.72	3.93	19.46	15.92	3.55	33.50
	25	9.86	26.88	18.89	63.26	71.25	29.82	20.96	8.86	36.74	28.75	7.98	33.83
	15	9.86	48.27	38.45	41.87	51.69	53.55	42.65	10.90	58.13	48.31	9.83	18.48
	5	9.86	86.69	85.21	3.45	4.92	96.18	94.54	1.64	96.55	95.08	1.48	1.56
	1	9.86	90.14	90.14	0.00	0.00	100	100	0.00	100	100	0.00	0.00
Q2	75	0.06	0.03	0.02	99.91	99.93	0.03	0.02	0.01	0.09	0.07	0.01	98.47
	50	0.06	0.06	0.03	99.88	99.91	0.06	0.03	0.03	0.12	0.09	0.03	98.42
	25	0.06	0.21	0.10	99.74	99.85	0.21	0.10	0.11	0.26	0.15	0.11	98.26
	15	0.06	0.52	0.20	99.42	99.74	0.52	0.20	0.32	0.58	0.26	0.32	98.10
	5	0.06	3.83	0.50	96.12	99.44	3.83	0.50	3.32	3.88	0.56	3.32	97.53
	1	0.06	41.93	5.09	58.01	94.85	41.96	5.10	36.86	41.99	5.15	36.84	88.39
Q3	75	0.001	0.01	0.00	99.99	100	0.01	0.00	0.005	0.01	0.002	0.005	99.90
	50	0.001	0.01	0.00	99.99	100	0.01	0.00	0.01	0.01	0.002	0.01	99.90
	25	0.001	0.08	0.00	99.92	99.99	0.08	0.00	0.08	0.08	0.006	0.08	99.84
	15	0.001	0.31	0.02	98.69	99.98	0.31	0.02	0.29	0.31	0.02	0.29	99.72
	5	0.001	3.70	0.44	96.30	99.56	3.70	0.44	3.26	3.70	0.44	3.26	97.90
	1	0.001	41.99	5.15	58.01	94.85	41.99	5.15	36.84	41.99	5.15	36.84	88.39



	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.0344%	0.0344%	2.12%	2.12%	13.33%	13.33%
🟥 FP	0.0136%	0.0758%	0.49%	2.69%	1.07%	7.77%
🟦 TN-Visited	6.31%	99.8899%	8.75%	95.19%	10.82%	78.90%
🟦 TN-Skipped	93.64%	0	88.64%	0	74.78%	0

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.0344%	0.0344%	2.12%	2.12%	13.33%	13.33%
🟥 FP	0.0315%	0.1941%	1.05%	6.11%	2.16%	14.82%
🟦 TN-Visited	6.66%	99.7715%	8.80%	91.78%	10.45%	71.86%
🟦 TN-Skipped	93.28%	0.0000%	88.03%	0.0000%	74.06%	0.0000%

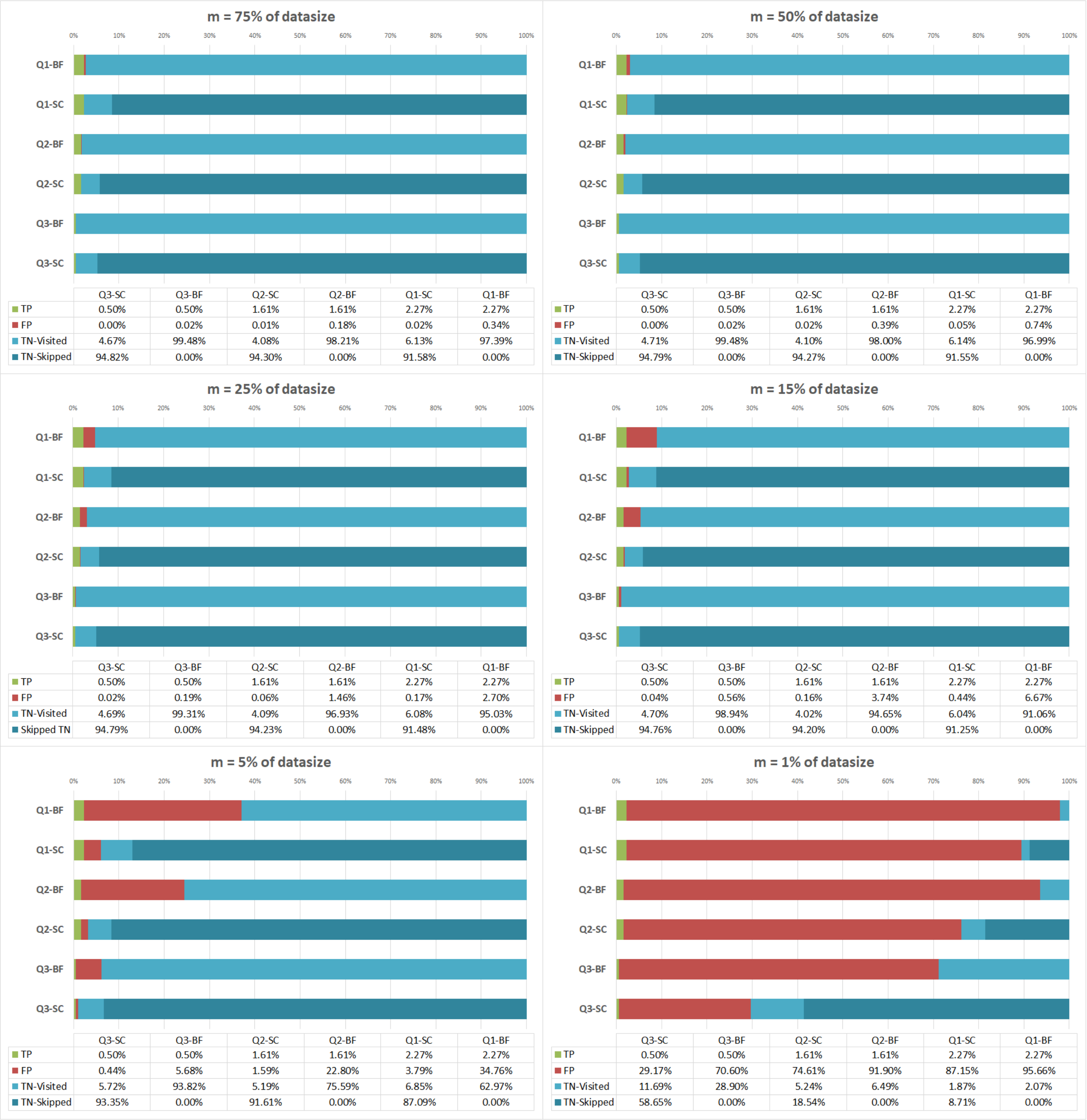
	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.0344%	0.0344%	2.12%	2.12%	13.33%	13.33%
🟥 FP	0.1677%	1.1033%	4.24%	22.73%	8.73%	37.29%
🟦 TN-Visited	8.87%	98.8623%	9.78%	75.15%	11.54%	49.39%
🟦 TN-Skipped	90.92%	0.0000%	83.86%	0.0000%	66.41%	0.0000%

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.0344%	0.0344%	2.12%	2.12%	13.33%	13.33%
🟥 FP	0.8574%	3.9704%	15.43%	49.01%	27.79%	59.87%
🟦 Scanned TN	15.94%	95.9952%	13.24%	48.87%	12.43%	26.80%
🟦 Skipped TN	83.17%	0.0000%	69.22%	0.0000%	46.45%	0.0000%

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.0344%	0.0344%	2.12%	2.12%	13.33%	13.33%
🟥 FP	18.8755%	31.0695%	90.73%	95.65%	83.77%	85.84%
🟦 TN-Visited	39.74%	68.8961%	2.11%	2.24%	0.81%	0.83%
🟦 TN-Skipped	41.35%	0.0000%	5.04%	0.0000%	2.09%	0.0000%

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.0344%	0.0344%	2.12%	2.12%	13.33%	13.33%
🟥 FP	94.1989%	96.4256%	97.88%	97.88%	86.67%	86.67%
🟦 TN-Visited	3.35%	3.5400%	0	0.00%	0	0.00%
🟦 TN-Skipped	2.42%	0.0000%	0	0.0000%	0	0.0000%

Fig. 13. **Visualization of the statistics given in Table 6** for the GFS Meteorology Prediction data set. True positives (TP) are shown in green, false positives (FP) in red, and true negatives (TN) in blue. BF denotes Bloom filters without supercells, and SC with supercells, respectively. For the stacked barcharts using supercells (SC) the two shades of blue denote TN not skipped by hierarchical supercell early-out (lighter blue) and TN skipped by hierarchical supercell early-out (darker blue), respectively. The latter is denoted by SC-Skip in Table 6.



m = 75% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.50%	0.50%	1.61%	1.61%	2.27%	2.27%
🟥 FP	0.00%	0.02%	0.01%	0.18%	0.02%	0.34%
🟦 TN-Visited	4.67%	99.48%	4.08%	98.21%	6.13%	97.39%
🟦 TN-Skipped	94.82%	0.00%	94.30%	0.00%	91.58%	0.00%

m = 50% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.50%	0.50%	1.61%	1.61%	2.27%	2.27%
🟥 FP	0.00%	0.02%	0.02%	0.39%	0.05%	0.74%
🟦 TN-Visited	4.71%	99.48%	4.10%	98.00%	6.14%	96.99%
🟦 TN-Skipped	94.79%	0.00%	94.27%	0.00%	91.55%	0.00%

m = 25% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.50%	0.50%	1.61%	1.61%	2.27%	2.27%
🟥 FP	0.02%	0.19%	0.06%	1.46%	0.17%	2.70%
🟦 TN-Visited	4.69%	99.31%	4.09%	96.93%	6.08%	95.03%
🟦 Skipped TN	94.79%	0.00%	94.23%	0.00%	91.48%	0.00%

m = 15% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.50%	0.50%	1.61%	1.61%	2.27%	2.27%
🟥 FP	0.04%	0.56%	0.16%	3.74%	0.44%	6.67%
🟦 TN-Visited	4.70%	98.94%	4.02%	94.65%	6.04%	91.06%
🟦 TN-Skipped	94.76%	0.00%	94.20%	0.00%	91.25%	0.00%

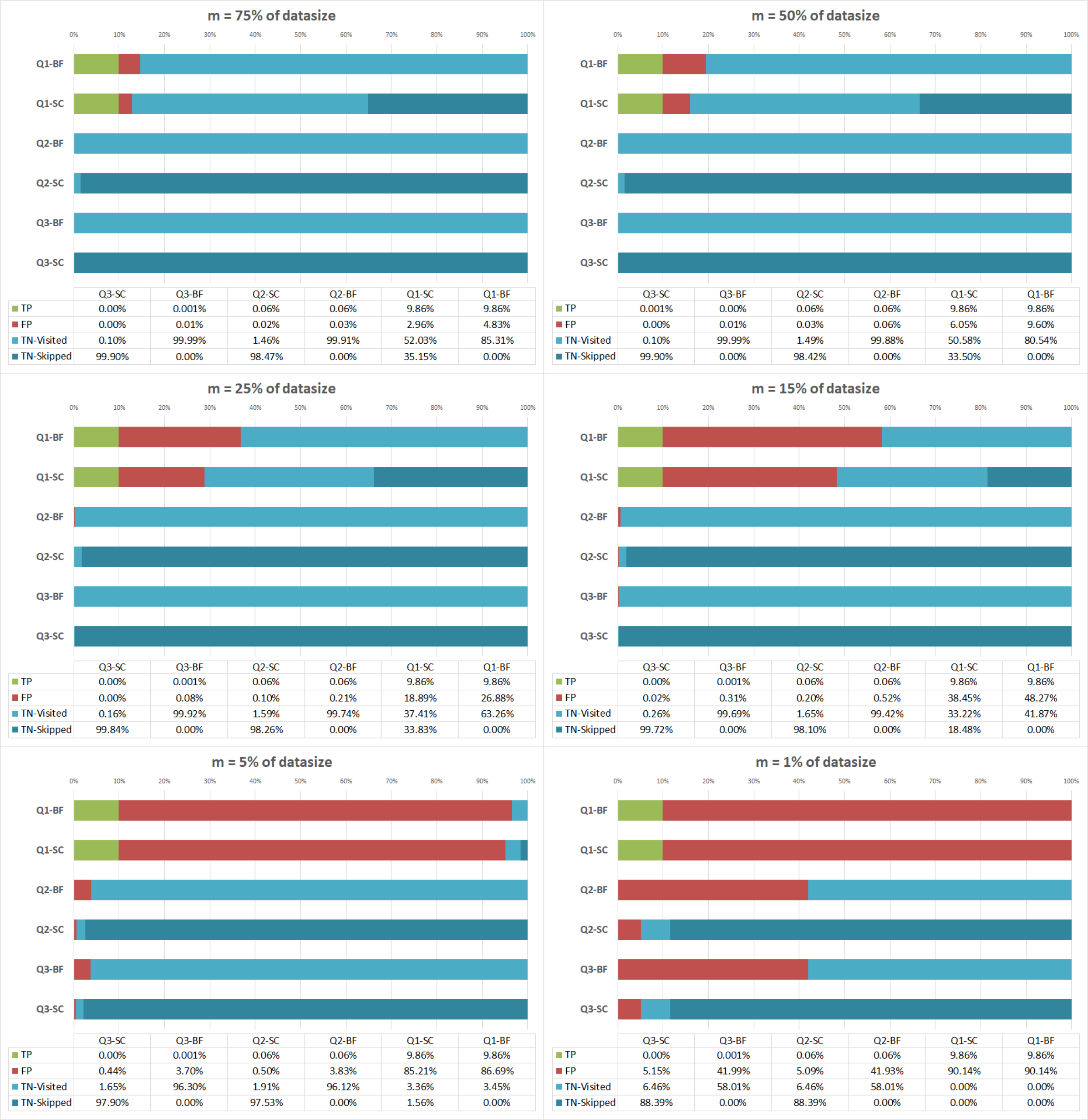
m = 5% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.50%	0.50%	1.61%	1.61%	2.27%	2.27%
🟥 FP	0.44%	5.68%	1.59%	22.80%	3.79%	34.76%
🟦 TN-Visited	5.72%	93.82%	5.19%	75.59%	6.85%	62.97%
🟦 TN-Skipped	93.35%	0.00%	91.61%	0.00%	87.09%	0.00%

m = 1% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.50%	0.50%	1.61%	1.61%	2.27%	2.27%
🟥 FP	29.17%	70.60%	74.61%	91.90%	87.15%	95.66%
🟦 TN-Visited	11.69%	28.90%	5.24%	6.49%	1.87%	2.07%
🟦 TN-Skipped	58.65%	0.00%	18.54%	0.00%	8.71%	0.00%

Fig. 14. **Visualization of the statistics given in Table 6** for the Red Sea data set. True positives (TP) are shown in green, false positives (FP) in red, and true negatives (TN) in blue. BF denotes Bloom filters without supercells, and SC with supercells, respectively. For the stacked barcharts using supercells (SC) the two shades of blue denote TN not skipped by hierarchical supercell early-out (lighter blue) and TN skipped by hierarchical supercell early-out (darker blue), respectively. The latter is denoted by SC-Skip in Table 6.



m = 75% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.00%	0.001%	0.06%	0.06%	9.86%	9.86%
🟥 FP	0.00%	0.01%	0.02%	0.03%	2.96%	4.83%
🟦 TN-Visited	0.10%	99.99%	1.46%	99.91%	52.03%	85.31%
🟦 TN-Skipped	99.90%	0.00%	98.47%	0.00%	35.15%	0.00%

m = 50% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.001%	0.00%	0.06%	0.06%	9.86%	9.86%
🟥 FP	0.00%	0.01%	0.03%	0.06%	6.05%	9.60%
🟦 TN-Visited	0.10%	99.99%	1.49%	99.88%	50.58%	80.54%
🟦 TN-Skipped	99.90%	0.00%	98.42%	0.00%	33.50%	0.00%

m = 25% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.00%	0.001%	0.06%	0.06%	9.86%	9.86%
🟥 FP	0.00%	0.08%	0.10%	0.21%	18.89%	26.88%
🟦 TN-Visited	0.16%	99.92%	1.59%	99.74%	37.41%	63.26%
🟦 TN-Skipped	99.84%	0.00%	98.26%	0.00%	33.83%	0.00%

m = 15% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.00%	0.001%	0.06%	0.06%	9.86%	9.86%
🟥 FP	0.02%	0.31%	0.20%	0.52%	38.45%	48.27%
🟦 TN-Visited	0.26%	99.69%	1.65%	99.42%	33.22%	41.87%
🟦 TN-Skipped	99.72%	0.00%	98.10%	0.00%	18.48%	0.00%

m = 5% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.00%	0.001%	0.06%	0.06%	9.86%	9.86%
🟥 FP	0.44%	3.70%	0.50%	3.83%	85.21%	86.69%
🟦 TN-Visited	1.65%	96.30%	1.91%	96.12%	3.36%	3.45%
🟦 TN-Skipped	97.90%	0.00%	97.53%	0.00%	1.56%	0.00%

m = 1% of datasize

	Q3-SC	Q3-BF	Q2-SC	Q2-BF	Q1-SC	Q1-BF
🟩 TP	0.00%	0.001%	0.06%	0.06%	9.86%	9.86%
🟥 FP	5.15%	41.99%	5.09%	41.93%	90.14%	90.14%
🟦 TN-Visited	6.46%	58.01%	6.46%	58.01%	0.00%	0.00%
🟦 TN-Skipped	88.39%	0.00%	88.39%	0.00%	0.00%	0.00%

Fig. 15. **Visualization of the statistics given in Table 6** for the Synthetic Perlin Noise data set. True positives (TP) are shown in green, false positives (FP) in red, and true negatives (TN) in blue. BF denotes Bloom filters without supercells, and SC with supercells, respectively. For the stacked barcharts using supercells (SC) the two shades of blue denote TN not skipped by hierarchical supercell early-out (lighter blue) and TN skipped by hierarchical supercell early-out (darker blue), respectively. The latter is denoted by SC-Skip in Table 6.