Culling for Extreme-Scale Segmentation Volumes: A Hybrid Deterministic and Probabilistic Approach

Johanna Beyer, Haneen Mohammed, Marco Agus, Ali K. Al-Awami, Hanspeter Pfister, and Markus Hadwiger

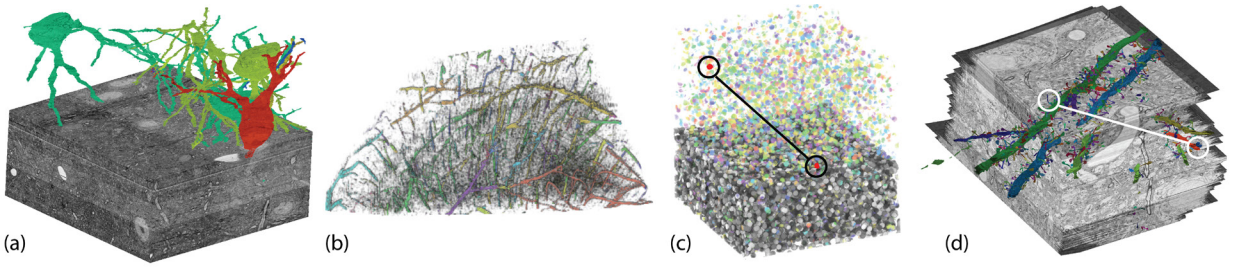


Fig. 1. ***Adaptive hybrid culling of millions of labeled segments***, for applications such as volume rendering and interactive spatial queries. (a,b) Empty space skipping for volume rendering: (a) Mouse Cortex 2 (13.25 million segments); (b) KESM Mouse Brain (224,436 segments). (c,d) Visual representations of spatial queries: (c) Phantom Spheres 2, distance computation between two user-selected segments (4.9 million segments); (d) SEM Mouse Cortex, distance computation between two segments (4,107 segments).

Abstract—With the rapid increase in raw volume data sizes, such as terabyte-sized microscopy volumes, the corresponding segmentation label volumes have become extremely large as well. We focus on integer label data, whose efficient representation in memory, as well as fast random data access, pose an even greater challenge than the raw image data. Often, it is crucial to be able to rapidly identify which segments are located where, whether for empty space skipping for fast rendering, or for spatial proximity queries. We refer to this process as *culling*. In order to enable efficient culling of millions of labeled segments, we present a novel hybrid approach that combines deterministic and probabilistic representations of label data in a *data-adaptive* hierarchical data structure that we call the label list tree. In each node, we adaptively encode label data using either a probabilistic constant-time access representation for fast conservative culling, or a deterministic logarithmic-time access representation for exact queries. We choose the best data structures for representing the labels of each spatial region while building the label list tree. At run time, we further employ a novel *query-adaptive* culling strategy. While filtering a query down the tree, we prune it successively, and in each node adaptively select the representation that is best suited for evaluating the pruned query, depending on its size. We show an analysis of the efficiency of our approach with several large data sets from connectomics, including a brain scan with more than 13 million labeled segments, and compare our method to conventional culling approaches. Our approach achieves significant reductions in storage size as well as faster query times.

Index Terms—Hierarchical Culling, Segmented Volume Data, Bloom Filter, Volume Rendering, Spatial Queries

1 Introduction

In recent years, huge advances in imaging technology have led to a massive increase in raw image and volume data sizes. Today, large volumes, such as those from high-resolution electron microscopy, can reach hundreds of teravoxels in size. However, in order for scientists to be able to perform analysis tasks, the raw image data are not even sufficient. The next crucial step is to *label* every voxel to identify it as belonging to a particular *segment*, which refers to different structures (e.g., dendrites), or parts of structures (e.g., spines of dendrites). Labeling voxels of nanometer-scale data was a very time-consuming and labor-intensive task, often resulting in only sparsely labeled data with a few thousand segments [4]. However, current state-of-the-art techniques for automatic segmentation now create densely labeled volumes with millions of segments within a few hours or days [25, 26]. In this paper, we focus on volumes where each voxel has an additional 24-bit or 32-bit integer label, which increases the overall data size (compared to 8-bit image data) significantly. However, it is crucial for scientists to be able to efficiently visualize and query the raw data together with the label data, for example for checking segmentation accuracy and performing proof-reading, and for a variety of analysis tasks.

Naturally, interactive visualization and analysis of terabyte-sized label volumes requires efficient data structures, together with hierarchical algorithms, that are able to efficiently prune the amount of data that needs to be processed interactively. The major approaches employ multi-resolution data structures, and many techniques have been proposed for their out-of-core management, processing, and visualization [6]. However, most existing techniques are limited to continuous data, e.g., grayscale pixel intensity. Creating efficient multi-resolution hierarchies for segmented volumes with integer label data poses significant challenges. In contrast to continuous pixel intensities, the correct down-sampling of labels is an additional inherent obstacle. Integer label data cannot be down-sampled by performing low-pass filtering followed by sub-sampling, because this would introduce non-existent labels. Therefore, the typical approach is to choose some *subset* of labels when computing a lower-resolution representation [6]. This, however, leads to missing label data. Nevertheless, for rendering purposes this is often seen as acceptable. However, for accurately identifying where specific segments are located (e.g., for distance computations), it is essential to be able to access the full-resolution label information.

- *Johanna Beyer and Hanspeter Pfister are with Harvard University, Cambridge, MA, USA. E-mail: {jbeyer, pfister}@seas.harvard.edu*
- *Marco Agus and Markus Hadwiger are with King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia. E-mail: {markus.hadwiger, marco.agus}@kaust.edu.sa.*
- *Haneen Mohammed is with Harvard University and KAUST. E-mail: haneen.mohammed@kaust.edu.sa*
- *Ali K. Al-Awami is with KAUST and Saudi Aramco, Dhahran, Saudi Arabia. E-mail: awami.ali@gmail.com*

Manuscript received 31 Mar. 2018; accepted 1 Aug. 2018. Date of publication 16 Aug. 2018; date of current version 21 Oct. 2018. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TVCG.2018.2864847

Note that we do not need to access the entire *voxel* data for such spatial queries. Rather, we need a full-resolution *list* (or set) of labels that occur within a certain region to use as a compact search structure. However, naively propagating such label data throughout a spatial hierarchy such as a tree leads to prohibitively large amounts of data. For example, the accurate list of labels of the root node, which corresponds to the lowest-resolution down-sampled image data, comprises a list of *all* labels of the entire volume, even if the root's image data are very small.

Efficient culling of label data. In addition to pure storage considerations, it is essential that the lists of labels in a spatial hierarchy are stored in a way that allows for efficient, random access for *culling* computations. We use the term culling to refer to the process of efficiently *locating* spatial regions in a volume that contain a certain set of segments, e.g., all the segments currently selected by the user for visualization, whereas all other segments are currently disabled. (Strictly speaking, culling refers to *culling away* irrelevant volume regions.) Culling is a fundamental operation that is essential for both:

(1) Rendering purposes, where, for example, empty space skipping requires determining which parts of the volume are empty, i.e., do not contain any segments that are currently enabled for visualization;

(2) Accurate spatial queries, such as locating all nodes containing some set of segments, or computing spatial distances between segments.

Main goals and properties. Our main goals are therefore to (1) design a novel hierarchical data structure for compact storage of very large integer label data; which also does (2) enable efficient hierarchical traversal for both kinds of culling computations described above.

Our approach combines a *deterministic, logarithmic-time access* data structure with a *probabilistic, constant-time access* data structure in a *data-adaptive* manner. For each node in the hierarchy, we choose the best representation for label data, taking into account memory size as well as expected run time query performance. Furthermore, our approach is also *query-adaptive*. We adapt run time query evaluation to both the characteristics of the current query, such as the number of labels in the query, as well as to the availability of the different possible label data representations in each node encountered during traversal.

Label list trees and multi-resolution label representations. Our main data structure is a hierarchy of integer label data that we call the *label list tree*. Each node of this tree stores label data in what we call label lists (i.e., we do not store the voxel data in this node, but the list or set of contained labels). However, for the two different culling operations described above, we in fact store two different kinds of label lists in each node. The first kind is for rendering purposes, called a *resolution-adjusted* label list, and contains the labels corresponding to the down-sampled data used for rendering. The second kind is for exact queries, called a *resolution-independent* label list, and contains all label information with respect to the exact full-resolution volume.

Label lists are our concept for storing label data. We use the term list in a general manner. In fact, each list is a *set* of labels (i.e., no duplicates, and order is irrelevant). Furthermore, we conceptually view each label list as a *bit string*. A 1-bit at a certain index means that the corresponding label is in the set, and a 0-bit means that it is not. For, e.g., 24-bit data, such a bit string is therefore 2^{24} bits long. For actual storage, we build on extensive work on efficient data structures for set membership queries in the areas of big data and database indexing, which, however, we have not yet seen used in the visualization literature. We encode each bit string using either (1) a *deterministic* representation, based on Roaring bitmaps [11], but combined with hierarchical delta encoding; or (2) a *probabilistic* representation, based on Bloom filters [7]. The former is a simple, but highly efficient, compressed encoding of bit strings that supports fast logarithmic-time random access. The latter is a probabilistic hashing method that provides constant-time random access for arbitrarily large data sizes, but results in *conservative* culling.

Application scenarios. We have implemented and evaluated two different application scenarios with different requirements: First, we support fast culling for empty space skipping for efficient volume rendering of densely segmented volumes. Second, we support accurate spatial queries between segments, which can be used for detailed analysis. For example, finding all axons in some spatial region of interest in a neuroscience volume, or computing the spatial distance between a certain dendrite and axon. Both scenarios have different requirements on the underlying label data, because spatial queries need to be evaluated with respect to the full resolution, while empty space skipping is performed with respect to the resolution currently visible on screen.

Contributions. We substantially reduce both the size and query evaluation time for label volumes with millions of segments. We achieve this via a novel hierarchical, data-adaptive data structure called the label list tree, and a novel adaptive, hierarchical approach for query evaluation. We build this tree in a pre-processing step, encoding the label data of each node for both resolution-independent (accurate) queries as well as for resolution-adjusted (rendering) queries. Label data are adaptively stored in the most-efficient data structure, depending on data characteristics and expected query performance: (1) Deterministic label storage is accurate, but is not bounded in size. However, we further reduce the size of accurate storage via hierarchical *delta encoding*. (2) Probabilistic label storage scales to arbitrary data sizes, but is conservative due to possible false positives. At run time, we dynamically evaluate queries via a hierarchical approach that incrementally prunes the input query while filtering it down the tree, culling each node probabilistically or deterministically, based on data and query characteristics.

2 Related Work

We review volume representations for large and labeled data, culling in visualization, and different data structures for representing label lists.

Representation of labeled volume data. Large volume data sets are typically subdivided into smaller bricks or blocks, to allow for out-of-core processing, and to avoid having to load and process the volume in its entirety [6]. Representations can range from a simple regular grid of volume blocks to more scalable multi-resolution hierarchies, such as k-d trees [39, 40], octrees [8, 12, 19, 29], or page table hierarchies [22].

For segmentation volumes, instead of storing a scalar intensity value per voxel, an integer segment (or label) ID is stored [21]. More compact representations for segmentation volumes have been proposed. Compresso [33], for example, stores segmentation as a separate boundary map and label list. However, most compact representations require on-the-fly decoding, and are not geared towards interactive volume rendering and fast random data access. An alternative approach to storing volume data is to extract the surface geometry of segmented objects [27], and subsequently deal with meshes instead of with volumes.

Culling. In visualization and graphics, the term *culling* refers to methods that aim to quickly reject (and avoid processing) those parts of the input data that do not contribute to an algorithm's output [18]. For example, a widely used approach for speeding up volume rendering is to cull all *empty space* surrounding the actual region of interest in a volume. This is done by checking the min/max values of a volume block against the currently set transfer function, to determine if this block will be invisible or "empty" after rendering. Empty blocks are then discarded and do not need to be rendered [15] or even downloaded to the GPU. Culling can be performed hierarchically using a multi-resolution hierarchy, which allows quickly discarding large regions.

We focus on culling for segmented volumes, i.e., we want to quickly discards those parts of the volume that do not contain certain label IDs.

Culling for empty space skipping. Empty space skipping is an efficient way to speed up volume rendering. Most recent scalable volume rendering approaches for large data sets use GPU ray-casting [12, 19, 22, 28]. In all these approaches, volume data are represented in a multi-resolution data structure that is traversed and sampled on the GPU during ray-casting [6]. Volume rendering of segmented volumes is done similarly. However, it typically requires two volumes to be present: the original data volume, as well as the labeled data [5]. Rendering can then access both volumes per sample and use the current label to decide which render mode or transfer function to use [21]. Empty space skipping in volume rendering reduces the amount of data that needs to be loaded as well as sampled during the ray traversal step and therefore can significantly reduce the memory footprint and rendering times [20]. Different methods for empty space skipping have been proposed [2, 12, 19, 20, 23, 37, 41]. However, the actual culling step (i.e., determining which parts of the volume are empty), always requires some meta-data for each volume block. For image volumes,

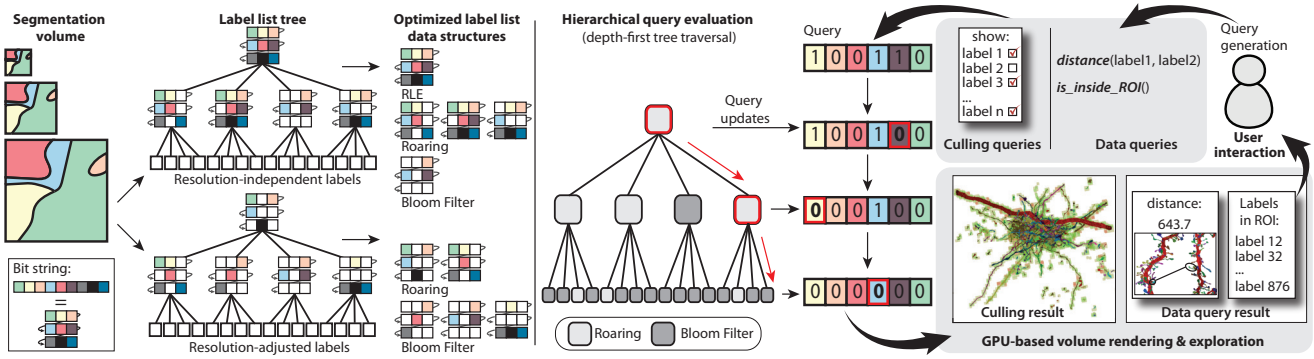


Fig. 2. ***System overview.*** (Left) *Data-adaptive* part: Label data are stored hierarchically in the *label list tree*. Each tree node stores two types of label lists. The first type is a *resolution-independent* label list, corresponding to the spatial region of each label list tree node. The second type is a *resolution-adjusted* label list, corresponding to the (down-sampled) resolution level of each node. For both, we data-adaptively determine the optimal representation. (Right) *Query-adaptive* part: At run time, we support two different types of queries: Approximate culling queries for volume rendering, and exact data queries for analysis. Queries are evaluated hierarchically, and filtered through each step of depth-first traversal of the label list tree. The label list representation used in each node, i.e., a deterministic or a probabilistic data structure, is chosen automatically according to query size.

these meta-data consist of min/max values of each block. Segmentation data typically require a *list of labels* contained in each block [20].

Culling for spatial queries. Culling is also required for the efficient evaluation of spatial queries on volume data, such as finding all objects in the vicinity of another object. Several different interactive query systems for volume exploration have been proposed [4, 9, 32, 36]. Braingazer [9] supports queries on brain and brain connectivity data, based on semantic and spatial relationships. ConnectomeExplorer [4] allows scientists to build domain-specific queries on segmentation volumes that are evaluated interactively. More general query techniques are dynamic queries [1], interactive visual queries [13], and DAX [38], which is a system for query-driven scientific visualization of large data sets. However, none of these are geared towards culling highly-segmented data sets comprising millions of labeled objects.

Representation of set membership. Culling a block of segmentation data requires knowing the list of labels in the block. This list can be encoded as a bit string in a straightforward fashion, by setting all bits of the integer IDs that are present in the set to 1. Bit strings are efficient to evaluate, and allow random reads and writes. However, they are not efficient in terms of storage [11]. For sparse data, *bit lists* can be efficient, where only the list of all indices with a 1-bit is stored. To make bit strings more space efficient, different lossless compression algorithms can be used [42]. Examples include LZW or run-length encoding. However, these formats typically do not support fast random access. Roaring bitmaps [11] propose to split the original bit string into smaller chunks, and to then encode each chunk individually, depending on their sparsity. Chunks can be encoded either as a dense bitmap, or as a sparse packed array. More recently, Roaring has been extended to also support chunks of run-length encoded data [31].

Probabilistic data structures. A convenient way to reduce bandwidth and memory requirements of conventional deterministic data structures is offered by probabilistic data structures. Skip lists [35] are a data structure for fast search in an ordered sequence of elements. It is based on a hierarchy of linked lists, where the elements that are skipped in a level of the hierarchy can be chosen probabilistically. Bloom filters [7] are a space-efficient data structure for quickly testing whether a given element is (could be) in a set or not. Bloom filters guarantee that no false negatives can occur, i.e., existing elements cannot be missed. However, they do have a certain false positive rate. The false positive rate of Bloom filters is influenced by the size of the bit array, the number of hash functions, and the number of entries in the Bloom filter. Extensions to Bloom filters also support deletions [16, 17], as well as better data locality [3]. However, they are less memory efficient than the original Bloom filter. Bloom filters have been very successfully applied in database indexing, search engines, and genome applications [24]. However, to our knowledge, probabilistic data structures have not yet been employed in the context of large segmented volume data.

3 Overview

Fig. 2 depicts an overview of our culling architecture. Our system consists of two major components: (1) We pre-compute a *data-adaptive* multi-resolution hierarchy for compactly storing label list data, the so-called *label list tree*. (2) At run time, we evaluate a culling query by *query-adaptive*, hierarchical traversal of the label list tree, where the best data structure is used at each node, depending on the query.

The output of our culling system is a list of *non-empty* volume blocks. Note that in our current implementation culling is performed on the CPU, but its output determines which volume blocks are transferred to GPU memory for subsequent GPU-based rendering or analysis.

3.1 Data-adaptive hierarchy of label data

The first major component of our architecture is the label list tree, which stores all label data that are required for query evaluation at run time.

Label list tree. This tree is a hierarchical representation of all label data. It is crucial to note that, in this context, label data always refers to *sets* of different label IDs in spatial regions, not to actual voxel data. While a major consideration is reducing the storage size of all label data, the performance of using the stored label information for query evaluation at run time is also crucial. In order to be able to trade-off between these two goals, we have to choose the data structures for storing label data and accessing them for queries in a way that is adapted to the actual data. However, since the data characteristics might differ significantly for different regions of a large volume, we have to adapt to the actual data in each tree node individually. That is, choosing one "best" representation for the entire label list tree is not sufficient. We adapt to the input data by using the following strategy:

- We store the label data corresponding to each tree node in one of several types of data structures. Which data structure is used for any given node depends on the characteristics of the node's data.
- We decide for each node if we actually store the same label data in *two* separate data structures, one deterministic one and one probabilistic one. This allows us to choose one at run time, depending on query characteristics that cannot be known before.

In order to determine the best data structure in each node, we consider storage size as the first major factor. However, we also take into account what the expected performance for evaluating queries at run time will be. The latter is harder to predict, because the query by itself is not known at pre-processing time when we build the label list tree. In order to compensate for this problem to some extent, we adaptively decide whether we store one data structure or two different ones in any node.

Multi-resolution label lists. In addition to the basic strategy just described, we furthermore have to accommodate an additional important consideration. To enable interactive rendering performance, large volumes have to be stored in a multi-resolution hierarchy [22]. In this way, at run time any region of the original volume can be accessed directly

at a lower (down-sampled) resolution. For label data, however, this also means that the down-sampled label data of coarser resolution levels do not anymore correspond to the label data of the original full-resolution segmentation data. However, at the same time, apart from rendering, we also have to be able to hierarchically evaluate some types of queries accurately with respect to the full-resolution data, e.g., spatial distances. We therefore conceptually store *two* label list trees. One tree corresponds to the original input data, and the other tree corresponds to the down-sampled data. Instead of actually storing two separate trees, we instead store two different kinds of label lists in each node of a single tree. One list is what we call a resolution-independent label list, and the other one is what we call a resolution-adjusted label list.

Data-adaptive label list construction. We store the raw data and label volumes in a bricked multi-resolution format on disk. From the input data, we compute the two types of label lists for each node in the multi-resolution hierarchy in a hierarchical data-adaptive preprocessing step (Fig. 2, left), constructing the label list tree. However, our system is completely independent of the actual data format used for raw volume and label data, since we only require lists of labels to be able to perform culling. We start processing with the highest-resolution label blocks, which will become the leaf nodes, and then traverse up the tree to compute the label lists for the next-lower resolution blocks, until we have computed all label lists up to the root node. For each block of label data, we first compute data distribution statistics to determine the most efficient way of storing the label lists of that block. We represent the set of labels contained in a node either as a deterministic bit string, or as a probabilistic representation, using Bloom filters [7]. Deterministic bit strings can be stored as either a full set or as a hierarchical *delta encoding*, both internally represented as a Roaring bitmap [31].

3.2 Query-adaptive run time evaluation

At run time, our system supports interactive query generation, query evaluation, as well as subsequent rendering and data analysis.

Interactive user queries. We have integrated our culling method into ConnectomeExplorer [4], an interactive volume rendering and visual query framework for large segmented neuroscience data sets. Culling is necessary for efficient volume rendering with empty space skipping, as well as for evaluating spatial proximity queries for visual analysis. Query evaluation is triggered by the user by either toggling the visibility of segments, or by using a visual query builder. See Fig. 2 (top right). The culling result (i.e., the set of non-empty volume blocks) is used as input to the volume rendering and analysis components and determines which data need to be downloaded to GPU memory.

Culling query evaluation. We evaluate culling queries in a hierarchical top-down fashion. See Fig. 2 (middle). We start by requesting the label list of the root node and culling it against the query, before recursively traversing down the tree in depth-first order. If a node does not contain any segments of the query, traversal is stopped. For a node containing at least one segment of the query, its child nodes are visited. Whenever a leaf node containing segments of the query is reached, it is added to the culling result. The nodes in the final result can either be used for empty space skipping, or are further evaluated voxel-by-voxel for detailed quantitative queries. See Fig. 2 (bottom right).

To speed up query evaluation, we incrementally *prune* the input query while filtering it down the tree: Segments in the query that are not in the label list of a visited node are removed from the query before passing it down to the child nodes. This results in the query getting successively smaller during tree traversal, and therefore culling becomes progressively faster. To further optimize query evaluation, we adapt the type of label list representation, i.e., deterministic or probabilistic, used in each traversal step, based on the size and complexity of the query.

4 Multi-Resolution Hierarchies for Label Data

Multi-resolution representations of raw image data and integer label data pose fundamentally different challenges. For continuous image data, a multi-resolution hierarchy can be created by low-pass filtering and down-sampling, because interpolating continuous data makes sense. In contrast, integer labels identifying labeled segments cannot be meaningfully interpolated. For example, smoothly interpolating between labels 6 and 10 might give label 8, which often would not even exist in the corresponding spatial region. Instead, integer labels are usually down-sampled by simple sub-sampling or via rank filters, e.g., choosing the most-frequent label. However, then the label lists of nodes in coarser resolutions will not be the union of the label lists of their subtrees. See levels L_0, L_1, and L_2 of the multi-resolution volume depicted in Fig. 3. We create our multi-resolution segmentation volume by sub-sampling. Label list creation is described in Sec. 4.1.

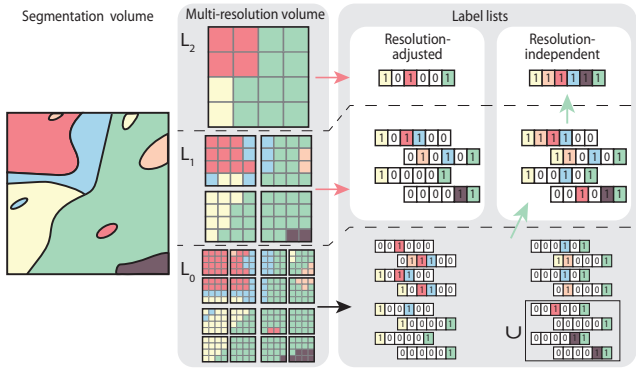


Fig. 3. ***Resolution-adjusted vs. resolution-independent label lists.*** To support different culling scenarios, we compute two types of multi-resolution label lists: (1) *Resolution-adjusted* label lists are computed directly from each resolution level L_i (red arrows). They only contain labels that are present in that resolution level. (2) *Resolution-independent* label lists are the union of *all* labels in the corresponding volume region in the original full-resolution segmentation L_0 (green arrows). The resolution-independent label list of the root node (here: L_2) will therefore contain all labels of the entire volume (here: 6), even if the down-sampled segmentation data of the root node contains much fewer labels (here: 3).

4.1 Multi-resolution label lists

Given the problems described above, we have to consider two very different scenarios of hierarchical traversal that require label lists:

- The label list of each node must be the union of all leaf nodes below, which are all labels in the corresponding spatial region.
- The label list of each node should correspond to the (down-sampled) volume block of the current resolution level.

The former scenario is required for all computations that have to be accurate with respect to the original data, such as exactly finding segments, or computing exact spatial information such as distances. The latter scenario can be used for everything that depends on the current visualization, not on the original data. For example, for empty space skipping we only need to determine which nodes do not need to be rendered and thus can be skipped. If a coarser resolution representation is currently being rendered, the label lists of the full-resolution data are irrelevant. In this case, using the label lists of full-resolution data would often lead to significantly over-estimating the number of non-empty nodes. Corresponding to these two types of requirements, we build a multi-resolution hierarchy of label lists that stores both kinds of data.

Resolution-independent label lists. In Fig. 3, the right path (green arrows) depicts how exact label lists for spatial regions are computed from the full-resolution data L_0. The initial label lists of L_0 are created directly from the label volume. Label lists of nodes in a coarser resolution level L_i ($i > 0$) are computed from the label lists of L_{i-1} by performing a union operation, indicated by $\cup$ in Fig. 3. For resolution-independent label lists, the root node will contain all labels of the whole data set. Therefore, this list is naturally very large for 24-bit or 32-bit label volumes. The farther away nodes are from the root, the sparser their label lists will be. We furthermore alleviate label data duplication by using delta encoding, see below. An important characteristic of resolution-independent label lists is that no segment will ever be present in a child node if it is not present in the parent.

Resolution-adjusted label lists. In Fig. 3, the red arrows show how the label list of each node at resolution level L_i is computed directly

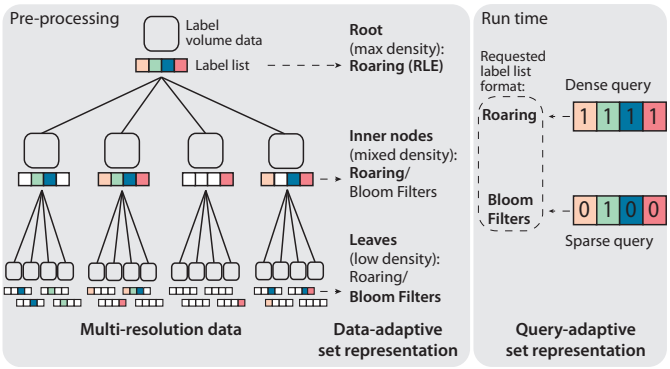


Fig. 4. ***Adaptive label list representation and usage.*** Left: We store label lists in a label list tree. Each node can store a label list in several different data structures. The optimal list (set) representation is chosen based on the data resolution and the node's label data statistics. We employ a deterministic representation based on Roaring bitmaps (i.e., bit strings, run-length encoding, sparse arrays), and a probabilistic representation based on Bloom filters. Right: At run time, we employ a query-adaptive approach. The query is always represented as a Roaring bitmap. Depending on the query cardinality and node statistics, for each node we can use the preferred (i.e., fastest) label list representation.

from the segmentation data of level L_i. This is a down-sampled representation of labels, not just a spatial subdivision as in the resolution-independent label lists, and exactly represents the down-sampled segmentation data. Resolution-adjusted label lists are more compact to store than resolution-independent lists, and the number of labels in the list is bounded by the number of (down-sampled) voxels in that node.

Adaptive label list usage during traversal. We always compute and store both types of label lists in each node to enable choosing at run time which type is required for the current task. During interaction, depending on the type of query, e.g., spatial analysis or empty space skipping, we automatically fetch and use the appropriate list type.

4.2 Data-adaptive label list construction

A major goal of our hybrid approach is to combine different data structures for storing label lists, to combine the best characteristics of each. We will consider two specific goals: (1) achieving the smallest memory consumption; and (2) achieving the fastest culling (query evaluation) performance at run time. To facilitate this, we first analyze the input data in a data-adaptive pre-processing step. This step performs a hierarchical data analysis, starting at the leaf nodes corresponding to the highest resolution, and going up to the root (lowest resolution).

For the label list of each node, we consider the following: (a) Is it smaller to encode the whole label list, or the difference between the list of the node and the list of its parent node (Sec. 4.3)? And, (b) In what data structure should we store the label list (Sec. 5)?

In order to be able to determine the best representation, we first hierarchically compute data statistics, based on the cardinality of a set, the size of the data structure in memory, and predicted data access times. We compute these for both, the *resolution-independent* as well as the *resolution-adjusted* label lists. Fig. 4 illustrates our hybrid data representation. Typically, the root node of the *resolution-independent* label list has a very high cardinality (i.e., many labels) and is stored with run-length encoding (using Roaring bitmaps). Nodes with medium cardinality (i.e., inner nodes) are stored either as Roaring bitmaps or using Bloom filters. We also support the option of storing both Roaring bitmaps and Bloom filters, and decide on the actual data structure usage only at run time, based on the actual culling query (see Sec. 6.3).

4.3 Top-down delta encoding of label list hierarchies

Instead of always fully encoding the label list of each node, we employ a delta encoding scheme that only stores the difference between a node and its parent node. This concept is somewhat similar to standard video compression, where each video frame can be stored either as a complete *keyframe*, or it is only encoded via the *difference* to the previous frame. In our case, the "frames" are different label lists, and the encoding order is from the tree's root node toward the leaf nodes.

Starting from the root node, which is always a keyframe, i.e., it always stores a full label list, we perform top-down tree traversal, and for every node check whether storing the difference to its parent node consumes less storage than storing the full label list. We quantify the difference between two label lists by the *Hamming distance* $\mathrm{Ham}(S_i, S_{i+1})$ of the corresponding bit strings, where S_i is the bit string of a node in level i, and S_{i+1} is the bit string of its parent node (we assign level 0 to the leaf nodes, not the root). The Hamming distance can be computed as the number of 1-bits after a bitwise XOR of the two bit strings. This is correct for *resolution-independent* and *resolution-adjusted* label lists.

In principle, when encoding the delta between S_i and S_{i+1}, we can check the Hamming distance $\mathrm{Ham}(S_i, S_{i+1})$ against the cardinality of the child set S_i. Then, if $|S_i| < \mathrm{Ham}(S_i, S_{i+1})$, we would disable delta encoding, because it is larger to store. Instead, the bit string of S_i would then be stored completely. However, in practice, before deciding at any node whether delta encoding should be used or not, we test actually encoding both the full label list and the delta label list. Then, instead of comparing the Hamming distance, we compare the actual memory consumption of both variants and choose the smaller one. The reason for this approach is that the size of encoding a Roaring bitmap does not only depend on the cardinality of the encoded bit string, but does in fact depend on the actual *pattern* of 1-bits. For example, if many consecutive bits are set, they will often be stored more compactly.

After the choice of either full or delta encoding is made, we use an additional flag to indicate the meaning of the stored bit string as either

1. The bit string encodes the original set S_i (no delta encoding).
2. The bit string encodes the difference set $D_i := \mathrm{XOR}(S_i, S_{i+1})$.

The bit string of the root node is always encoded as the whole set S_{L-1}, where L is the number of resolution levels, and level $L-1$ is the level of the root node. Below the root node, the choice is data-adaptive.

5 Data Structures for Label Set Membership

We now describe the major two data structures that our culling architecture uses to represent label lists in each label list tree node. See Fig. 5. Each label list is in fact a *set* of integer label IDs, which we conceptually treat as a very long bit string. A 1-bit means that the corresponding ID is present in the set, a 0-bit means that it is not. For storing these bit strings, we employ two main data structures in order to be able to adaptively choose between the two, and thus be able to combine the advantages of both. A label list can be stored as either a

1. Deterministic data structure (a Roaring bitmap), providing exact set membership queries with *logarithmic-time random access*.
2. Probabilistic data structure (a Bloom filter), providing approximate set membership queries with *constant-time random access*.

We note that we have designed our system in a modular way to allow switching out individual components easily in the future, e.g., for using different underlying data structures. However, we have chosen Roaring bitmaps and Bloom filters particularly for their efficient storage sizes, flexibility, and fast access times, as described in more detail below.

5.1 Deterministic set membership: Roaring bitmaps

We use Roaring bitmaps [31] as a lossless compression method for label bit strings, because they provide deterministic set membership queries with logarithmic-time random access. Roaring has been developed for compressing large bit string indices, as they typically occur in large databases or search engines. Roaring bitmaps are by themselves a hybrid data structure that splits the input range (2^{24} or 2^{32} different label IDs) into *chunks*, where each chunk can be stored using a different container data structure. Each chunk in Roaring corresponds to 16 bits, i.e., 2^{16} labels. For 24-bit labels, there can be at most $2^{(24-16)} = 256$ chunks; for 32-bit labels at most 64K chunks. All non-empty chunks are held in a sorted list (array). When evaluating a query, the required chunks are found using binary search. The labels in a chunk are stored in one of three types of internal storage containers (see Fig. 5, left):

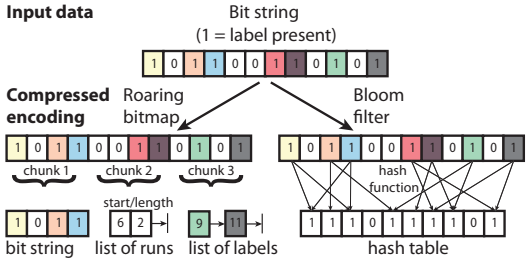


Fig. 5. ***Data structures for label set membership.*** Given an input set of labels (a conceptual bit string), we support two different encodings: (1) Our deterministic data structure is built on Roaring bitmaps, which divide 32-bit data into chunks of 16-bit size each. The list in each chunk is stored as either an uncompressed bit string, a run length-encoded bit string, or a list of IDs. (2) Our probabilistic data structure is built on Bloom filters. This enables the compact representation of a sparse bit string with a fixed amount of storage, while guaranteeing no false negatives.

- *Uncompressed bit string:* Stores $2^{16} = 64$K label IDs. It is therefore always 8 KB in size, for storing 64K bits. It is intended for dense chunks (i.e., many label IDs set to 1), and allows direct access to each bit. However, if there are less than 4K elements (1-bits), a sorted array will be used instead, because it is smaller.
- *Sorted array:* This container explicitly stores a list of all label IDs (indices of 1-bits). Due to the previous chunking, only the lowest 16 bits of the label IDs need to be stored. The higher 16 bits are already determined by the chunk ID. This container is stored as a sorted array for fast binary search. This representation is only used for 4,096 elements or less ($4,096 \cdot 2$ bytes $= 8$ KB).
- *Run length encoding:* Here, *runs* of 1-bits are encoded as pairs of $(startID, count)$ records, stored in sorted order for access with binary search. A run length container is only used if its size is smaller than the size of a bit string or list container for the same data. This container's size is always $2 + 4r$ bytes, given r runs.

We use a publicly available C++ implementation of the Roaring bitmap data structure [30] that supports fast addition and deletion of elements from a set, and also offers fast intersection, union, and difference operators. In correspondence with the description in Sec. 4.2, we use the same data structure for either encoding a full label list or a delta label list. The latter encodes only the difference between a parent and a child node in the label list tree. We chose Roaring because it allows deterministic logarithmic-time access to label lists, is an open-source, optimized library, and conceptually still represents a simple bit string.

5.2 Probabilistic set membership: Bloom filters

We use Bloom filters [7] as a fully scalable probabilistic encoding of label bit strings. Bloom filters offer constant-time random access, independent of the input range of label IDs as well as of the number of labels stored in one data structure. This makes them a good candidate for scalability to extremely large data. Bloom filters employ a hashing strategy that enables checking directly whether an element is present in a set or not, without the binary search steps required by Roaring bitmaps. A Bloom filter uses k different hash functions, indexing a single hash table that is a bit vector of m bits (see Fig. 5, right). We add each label ID by hashing it, and setting the k bits at the indices given by the k hashing results in the bit vector to one. To check whether a label is in a Bloom filter, the label ID is hashed, and the hash table bit vector is checked whether *all k bits* at the k corresponding indices are set. If any of the bits is zero, the label is definitely not in the set. If all k bits are set, the label *may be* in the set, but it could also be a false positive.

Bloom filters are very space efficient when the universe U (all possible label IDs) is large, and the cardinality of the label set S is small:

$$|U| \gg |S|. \tag{1}$$

The universe refers to the number of potential elements that can occur (e.g., 2^{32} in 32-bit label data), and the cardinality is the actual number of elements in a set, i.e., the number of different labels in a label list. The false positive rate of a Bloom filter can be approximated by

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

where m is the length of the hash table bit vector in bits, k is the number of hash functions, and n is the number of inserted elements, i.e., $n = |S|$. It is important to notice that Eq. 2 is *completely independent of the universe size* $|U|$. For this reason, Bloom filters are very well suited to very large ranges of label IDs (e.g., 2^{32} labels), when, in contrast, the actual label set cardinality $|S|$ in a label list tree node is small.

We use the characteristics of Bloom filters for probabilistic, but conservative, culling. This will never result in incorrectly culling nodes that should not be culled (i.e., no false negatives). On the other hand, some nodes might be reported as non-empty that could have been culled (i.e., false positives), leading to unnecessary processing or rendering.

By changing the hash table size m, we can control the trade-off between false positives and memory size. In a Bloom filter, set membership has to be queried for each label individually. Therefore, in a serialized implementation of Bloom filters, the size of the query, i.e., the number of elements the query contains, should be relatively small for quickly evaluating if the set contains any element of the query.

To summarize, Bloom filters are very memory efficient for label lists of nodes containing a relatively small number of labels (i.e., typically the nodes at higher resolution levels). Furthermore, Bloom filters are fast for queries with a low cardinality, such as detailed distance queries or renderings of a few specific structures.

6 Adaptive Hierarchical Culling

We employ a fully hierarchical approach for evaluating culling queries, which allows us to quickly cull large spatial regions in one step. Our culling algorithm is modular, and can easily be integrated into existing volume rendering or query systems, including *ray-guided* volume renderers [12, 20]. Generally speaking, the goal of a culling operation is to locate those *leaf nodes* of the label list tree that contain at least one of the labels specified in the query. These can be, for example, the segments enabled for rendering, or segments that are otherwise of interest, e.g., for computing distances between segments. Given an arbitrary input query, it is hierarchically "filtered" through the label list tree, comparing it against each visited node. Empty nodes and sub-trees are skipped. Only *non-empty* leaf nodes are reported in the final result.

6.1 Query representation

Conceptually, the input query is a set of labels, like the label list of each segmented volume block. This set is created either by a computational algorithm, or based on direct user input, e.g., by clicking on a volume-rendered segment on screen. All nodes that do not contain any of the labels in the query should not be reported in the culling result. To facilitate fast, optimized set intersection computations during query evaluation, we represent the label list of a query as a Roaring bitmap, and employ the Roaring library for individual bit-level computations.

6.2 Hierarchical query evaluation

Given an arbitrary input query Q, i.e., a list (set) of labels of interest, we can formally define the output of culling as the list (set) of nodes

$$S_{\text{out}} = \{n_j\}, \quad \text{such that} \quad S(n_j) \cap Q \neq \emptyset \text{ for all } j, \tag{3}$$

where $S(n_j)$ denotes the set of labels of a *leaf node* n_j. Fig. 6 illustrates our hierarchical query evaluation and incremental pruning scheme for computing S_{out}. We then perform depth-first label list tree traversal starting from the root, and evaluate the query for each visited node.

If a node is classified as *empty*, i.e., when $S(n_j) \cap Q = \emptyset$, its entire sub-tree is culled, i.e., not put into the set S_{out}, and traversal of that sub-tree stops. Otherwise, traversal continues toward the leaves. To determine if a node is empty, we compare the labels in the query with the node's labels. However, for efficiency we employ a hierarchical query pruning scheme (see below). If a label occurs both in the query and in the node, i.e., when $S(n_j) \cap Q \neq \emptyset$, the node is *non-empty*.

We add all visited, non-empty leaf nodes to S_{out}.

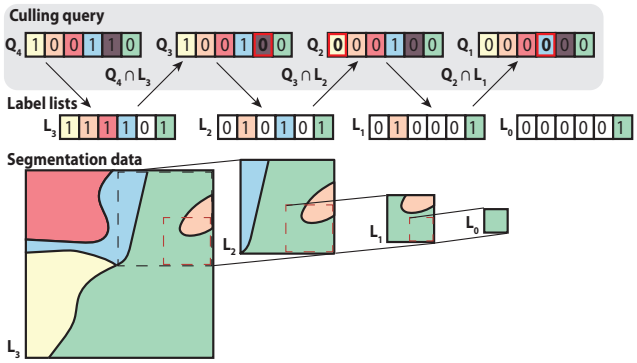


Fig. 6. ***Hierarchical query evaluation.*** We cull label lists (middle) against the input query Q, updating temporary queries Q_i (top). Starting with the temporary query $Q_4 := Q$, we update Q_i at every step to at most a subset of the labels in the current node. Thus, the cardinality of a query decreases as tree traversal proceeds from the root node toward the leaf nodes. We depict the multi-resolution label data of levels $L_3 - L_0$, and the corresponding label lists. $Q_4 - Q_1$ show how we prune query information while traversing the tree. In this example, traversal stops recursion one level above the shown leaf node at level L_0, because Q_1 is already empty.

Depth-first traversal and incremental query pruning. During label list tree traversal, we successively prune the input query to the smallest possible cardinality, which corresponds to the set intersection of the input query Q with the label list of the currently visited node.

We can do this very efficiently in an incremental manner, by maintaining a temporary copy Q_i of the original query Q in every node along the path from the root node to the current node (see Fig. 6). That is, we employ a set of temporary queries $Q_i \in \{Q_0, Q_1, \cdots, Q_{L-1}\}$. The associated overhead is negligible, since this path always has only one node per tree level. Therefore, the total number of temporary queries is always bounded by L, the total number of tree levels. During traversal, we keep track of the temporary queries, and compute the current Q_i at every traversal step to a child node. We compute each Q_i as either:

- $Q_i := \text{AND}(Q_{i+1}, S_i)$, if S_i is a whole set (no delta encoding),
- $Q_i := \text{AND}(Q_{i+1}, \text{NOT}(D_i))$, if D_i is a difference set,

where S_i and D_i are as defined in Sec. 4.3. We start recursive traversal at the root node with $i = L-1$, and the initial query passed into the root node (tree level $L-1$) is defined as $Q_L := Q$.

Whenever the current temporary query Q_i is empty, the corresponding sub-tree (including the node itself) is culled, and the recursion of depth-first traversal is stopped. Traversal then continues on level $i+1$, after going back to the parent of the node whose query Q_i was empty.

Roaring bitmap evaluation. If the label set S_i, or alternatively the difference set D_i, is represented as a Roaring bitmap, the two operations given above are evaluated directly using the Roaring library, which either performs a bit-wise AND operation in a bit string container, or otherwise updates the sorted list or run-length containers accordingly.

Bloom filter evaluation. If the set S_i is represented probabilistically using a Bloom filter, we evaluate the updated temporary query $Q_i := \text{AND}(Q_{i+1}, S_i)$ by creating it as a pruned copy of Q_{i+1} as follows. For every label in the temporary query Q_{i+1}, we check whether that label is contained in the Bloom filter of S_i (a hashed label list of the current node). If it is not, we delete that label from the query Q_i. If it *may be* in the set S_i (including possible false positives of the Bloom filter), the label is kept in the temporary query Q_i. We note that this never prunes the query too much, but it can conservatively prune it too little. The latter cannot lead to incorrect results, but can reduce efficiency.

6.3 Query-adaptive choice of label list representation

During traversal, out of the label list representations available at some visited node n_i, we dynamically choose the best one for culling against the temporary query Q_i, depending on the cardinality of Q_i.

Choosing the best representation. If the label list of a node n_i is stored in both a Roaring bitmap as well as in a Bloom filter, we choose one of the two representations according to the following criteria:

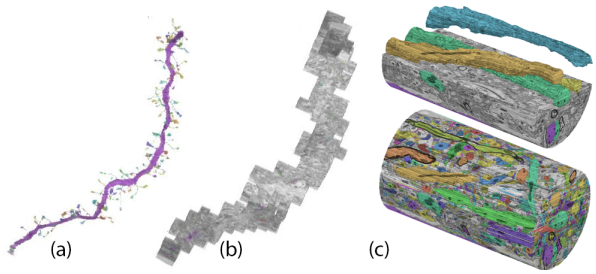


Fig. 7. ***Empty space skipping and spatial queries.*** (a) The empty space around the segmented dendrite is skipped during volume rendering. (b) We highlight the remaining volume blocks that need to be traversed after empty space skipping. (c) Finding all labels inside a cylindrical region of interest, and only showing the largest structures.

- If $|Q_i| > c_B$, i.e., the cardinality of the temporary query at the node n_i is higher than a certain threshold c_B, we always cull against the Roaring bitmap of n_i. The threshold is chosen to specify what cardinality is considered too slow for culling using a Bloom filter.
- If $|Q_i| \leq c_B$, we cull against the Bloom filter of the node n_i.

Culling against a Bloom filter is conservative and less accurate than using a Roaring bitmap. However, if a node n_i contains a Bloom filter representation of its label list, the pre-computation step has determined that, given the known cardinality $|S(n_i)|$ of the node n_i, a Bloom filter is the most space-efficient representation, and that it should be used for run time queries with a cardinality less than c_B. We have heuristically chosen $c_B \approx 10$. Optionally, we can enforce deterministic culling for leaf nodes, to guarantee final culling results without false positives.

7 Applications

To demonstrate the usefulness of our general culling method, we have integrated our architecture into two different applications. The first one is a ray-guided volume renderer for large segmented neuroscience data sets [20], and the second application is ConnectomeExplorer [4], a system for interactive visual queries and visual analysis. See Fig. 7.

7.1 Empty space skipping for volume rendering

We have integrated our culling framework into SparseLeap [20], a multi-resolution volume renderer aimed at high-resolution neuroscience data. The system supports rendering of segmented data, and uses a hybrid image- and object-order approach for empty space skipping. However, the original implementation uses uncompressed bit strings for representing label lists, and therefore does not scale to highly segmented volumes with millions of objects. Therefore, we have replaced the previous label representation and uncompressed bit string culling of SparseLeap with our novel method (i.e., the label list tree and hierarchical query traversal) to support empty space skipping with millions of objects. See Fig 1 (a,b) and Fig. 7. Volume rendering is performed in a ray-guided manner, meaning that data blocks are loaded and culled in a deferred way, only after the ray-caster has reported a block as missing. Once a missing block has been reported, the system first requests its label list and executes the culling operation on that block. Only if that block is *non-empty* is it actually loaded and subsequently rendered.

Culling for empty space skipping works in the following way: We first start hierarchical traversal using resolution-independent label lists. During traversal, this leads to conservative culling. Once we hit a node of the resolution that is currently used in rendering, we stop traversal, because we are not interested in any higher resolution than what is being rendered on screen. At that resolution, we perform culling with resolution-adjusted labels, to get a completely accurate cull state for that resolution. Standard empty space skipping would stop here, but the SparseLeap architecture uses an additional step to propagate the cull state of the leaf nodes back up to the root, to further optimize the rendering step. However, this can be done in a completely transparent manner, simply using the culling information our architecture provides. Similarly, our internal culling data structures and culling optimizations are hidden from the volume rendering architecture, as we only provide

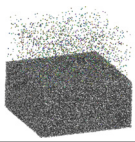
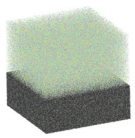
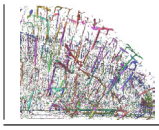
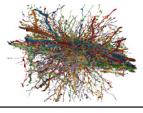
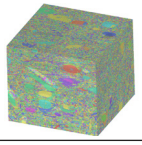
a high-level abstract interface. Results and timings for our culling architecture for empty space skipping are provided in Sec. 8.3. Using Sparseleap, all our data sets can be rendered at interactive framerates. A more detailed evaluation of rendering performance is given in [20].

7.2 Spatial queries for analysis

The second use case of our culling architecture is ConnectomeExplorer [4], an interactive exploration and query-guided visual analysis system for large segmented electron microscopy (EM) data sets. ConnectomeExplorer provides a visual query builder, based on a query algebra, that allows users to build custom queries by either manually selecting segments in a volume or list view, or by using the provided query algebra of predicates and operators. Queries are evaluated interactively and result either in a set of segments, or use a set of segments to compute quantitative results. For example, topological (connectivity) operators can be used to find connected biological structures, and spatial predicates and operators can be used to compute distances between two segments. See Fig 1 (c,d). Previously, accurate spatial queries of labeled segments were slow, and limited to a few thousand segments. We have integrated our culling architecture into ConnectomeExplorer to speed up spatial queries and support highly segmented data.

The supported spatial queries include finding the locations of labels, subsequently computing the distance between labels, and finding all labels within a region of interest. The algorithm for computing distances between two segments is based on previous work on distance algorithms for octree-encoded objects [14]. We have integrated our culling architecture into the spatial query system and use resolution-independent (i.e., region-based) label data, through all levels of the hierarchy. This allows us to quickly and accurately identify candidate volume bocks for the distance calculations (i.e., blocks containing the labels used in the distance calculation). The distance computation algorithm then maintains a list of pairs of tree nodes that potentially minimize the distance between the two segments. The algorithm processes this list by hierarchically subdividing one of the nodes in a pair, until only pairs with leaf nodes are left. To get a voxel-exact distance (i.e., smaller than the block size used for the node), the final leaf nodes need to be manually inspected to compute the exact result.

Table 1. ***Data set statistics*** of the volumes used for evaluation (Sec. 8). We list data resolution, storage size, and number of labels, as well as the number of resolutions in the label list tree, and the used block size.

Data set	Description	Label lists info
	Phantom Spheres (PS1K) 1,024 x 1,024 x 1,024 Images: 1 GB (8 bit) Labels: 3 GB (24 bit)	Block size: 32^3 Resolution levels: 6 # Labels: 614 K
	Phantom Spheres 2 (PS2K) 2,048 x 2,048 x 2,048 Images: 16 GB (8 bit) Labels: 48 GB (24 bit)	Block size: 32^3 Resolution levels: 7 # Labels: 4.91 M
	KESM Mouse Brain (KESM) 2,380 x 9,216 x 2,039 Images: 42.7 GB (8 bit) Labels: 128.1 GB (24 bit)	Block size: 32^3 Resolution levels: 9 # Labels: 224 K
	SEM Mouse Cortex (MC1) 21,494 x 25,790 x 1,850 Images: 955 GB (8 bit) Labels: 489 GB (16 bit)	Block size: 32^3 Resolution levels: 10 (11 levels for images) # Labels: 4,107
	Mouse Cortex 2 (MC2) 4,096 x 4,096 x 4,096 Images: 64 GB (8 bit) Labels: 192 GB (24 bit)	Block size: 32^3 Resolution levels: 8 # Labels: 13.25 M

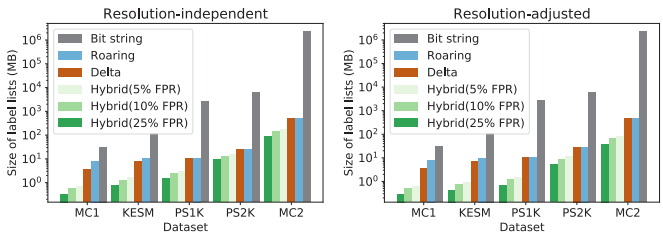


Fig. 8. ***Memory consumption of different label list representations.*** We report the total size of the label list tree and compare uncompressed bit strings, Roaring, combined Roaring/delta encoding, and our hybrid method (using different false positive rates (FPR) for Bloom filters). Left: Resolution-independent label lists. Right: Resolution-adjusted label lists.

8 Evaluation and Results

We evaluate our culling approach using the data sets listed in Table 1. We report memory sizes and culling performance with respect to reference implementations using standard bit strings. Our framework is implemented in C++, OpenGL and GLSL, and our data structures are based on the CRoaring library [30] and Boost Bloom filters [10]. The culling system is implemented as a CPU component, and can be easily integrated into larger GPU volume rendering or visual query systems. We report more evaluation results in our supplemental material.

8.1 Data sets

Table 1 lists the data sets we use for evaluation and their basic properties. Our data sets have between 4,107 and 13 million labels, with memory sizes of from several GB to more than one TB. We include two densely labeled phantom sphere data sets and three different neuroscience data sets: SEM Mouse Cortex was manually segmented, KESM Mouse Brain was labeled with an automatic, but sparse segmentation algorithm, and Mouse Cortex 2 was densely labeled. We analyze block-based distribution statistics of labels in the supplemental material.

8.2 Memory footprint

Fig. 8 shows the overall memory consumption of the label list tree for pre-defined data representations for the label lists. We compare standard bit strings to Roaring, Roaring with delta encoding, and our hybrid method. Uncompressed bit strings are by far the most memory-intensive representation, especially for volumes with many labels. Our hybrid method is consistently the most compact, by a factor of more than 50×-100×. The memory consumption of resolution-independent label lists (Fig. 8, left) is higher than for resolution-adjusted label lists (Fig. 8, right), especially at lower resolution levels. However, since the highest resolution makes up roughly 90% of the overall memory consumption, and both types of label lists share their highest-resolution representation, the difference in memory size is almost negligible.

Fig. 9 (top) shows the memory consumption of the label list tree for different resolution levels of the KESM data set. Especially for level 0 (i.e., the highest resolution), our hybrid method is very compact. The stacked bar chart in Fig. 9 (bottom) depicts the distribution of the different data encodings used in our hybrid approach (i.e., Roaring, deltas, and Bloom filters). We depict this distribution for three different false positive rates (light to dark green), and per resolution level. It can be seen that Bloom filters are primarily used in the highest-resolution levels, while Roaring is used mainly for the lowest-resolution levels.

8.3 Culling performance

Table 2 gives detailed performance numbers of our culling method, as applied to empty space skipping. We list the memory consumption, as well as the query evaluation time for different data sets, and compare our hybrid culling approach to a standard brute-force culling approach. The standard approach iterates over all visible blocks of the current view (non-hierarchically), and checks whether queried labels are contained or not. Our method consistently performs better in both, memory consumption and query evaluation time. The only outlier is query Q2 for SEM Mouse Cortex, where our method is more memory efficient, but the bit string-based approach is slightly faster. The most likely

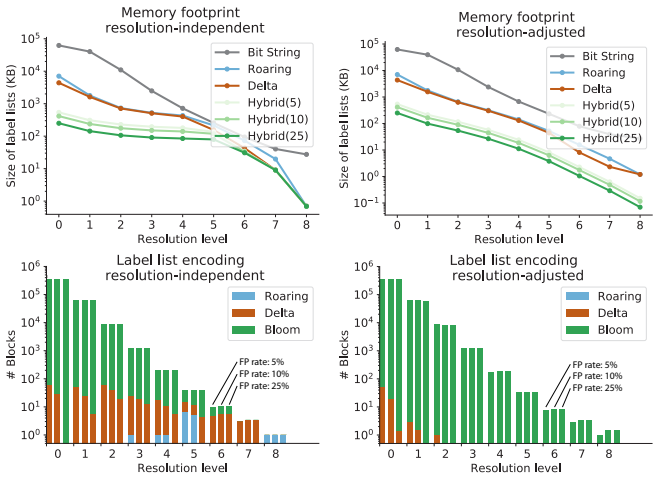


Fig. 9. ***Memory consumption of label list and internal encoding.*** Top: We show the size of different label list representations at different resolution levels (full resolution is level 0). We use false positives rates of 5, 10, and 25 in our hybrid method. Bottom: Distribution of the different data encodings (Roaring, delta, bloom) in our hybrid approach (FP rate: 5, 10, 25), per resolution level. Left: Resolution-independent label lists. Right: Resolution-adjusted label lists. Data set: KESM Mouse Brain.

reason for this is that the SEM Mouse Cortex data set is very sparsely labeled (i.e., 4,107 labels), which is small enough that bit strings are a feasible alternative for storing label lists. A more detailed evaluation of culling for spatial queries is included in the supplemental material.

8.4 Bloom filter evaluation

In Fig. 10, we evaluate the relationship between the memory consumption of Bloom filters and their false positive rate, based on the KESM Mouse Brain data set. The smaller the false positive rate, the more memory a Bloom filter needs. In our framework we have tested different settings, but heuristically decided on a false positive rate of 5-10 %, as this seems to be a good trade-off between size and performance. Fig. 8 show the influence of the false positive rate that is used for Bloom filters on the overall size of the label list tree. Fig. 9 (bottom) indicates that Bloom filters are primarily used for the highest-resolution levels, where the nodes contain the smallest number of distinct labels. This plays into the strength of Bloom filters: A large number of possible label IDs, but a small number of actual label IDs in the node. A more detailed analysis is given in the supplemental material.

Table 2. ***Culling performance for empty space skipping.*** We compare our approach using hybrid label lists to standard, non-hierarchical culling with bit strings. We list the number of nodes touched for culling, the size of the touched label lists, and the evaluation time for two different queries (Q1: 2 labels, Q2: 1,000 labels). FP rate 10%, $c_B = 10$.

data set	culling method		# nodes touched	label data touched	time (ms)
KESM Mouse Brain	hybrid culling	Q1	261 (1.6 %)	203 KB	7.6
		Q2	3,109 (19.5 %)	647 KB	25
	non-hierar.	Q1	13,895 (86.9 %)	381 MB	153.2
		Q2	13,895 (86.9 %)	381 MB	142.6
SEM Mouse Cortex	hybrid culling	Q1	141 (1.5 %)	48 KB	7
		Q2	3,269 (34.3 %)	188 KB	27.2
	non-hierar.	Q1	8,190 (85.9 %)	4.1 MB	7.4
		Q2	8,190 (85.9 %)	4.1 MB	6.6
Mouse Cort. 2	hybrid culling	Q1	417 (1.9 %))	13.4 MB	12.8
		Q2	1,713 (7.83 %)	27.7 MB	25.2
	non-hierar.	Q1	12,986 (87.5 %)	25.4 GB	7,019
		Q2	12,986 (87.5 %)	25.4 GB	6,549
Phantom Spheres	hybrid culling	Q1	273 (1.74%)	3.8 MB	8
		Q2	12,329 (78.7%)	25.7 MB	92.4
	non-hierar.	Q1	16,332 (87.5%)	9.8 GB	3,357
		Q2	16,332 (87.5%)	9.8 GB	2,976

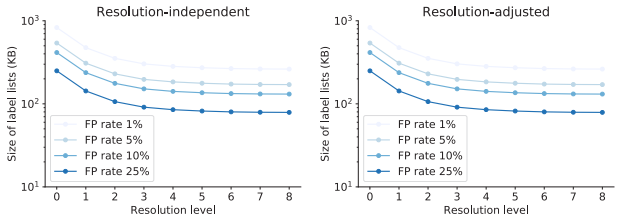


Fig. 10. ***Bloom filter memory consumption vs. false positive rate.*** We evaluate Bloom filter sizes at different resolution levels (full resolution is level 0), for different false positive rates. Left: Resolution-independent label lists. Right: Resolution-adjusted label lists. Data set: KESM Brain.

8.5 Discussion

One of the main advantages of our culling approach is its flexibility: Our label list tree can handle sparse as well as dense segmentation volumes, and chooses the underlying data representation adaptively, based on local data set characteristics. This makes our approach suitable for large, segmented volumes with areas of different label densities.

Roaring and Bloom filters. Roaring bitmaps offer a compact representation of label lists and are especially suited for dense (in terms of labels) nodes with a high cardinality of label IDs. Roaring is the most compact data structure if there are long runs in the label list that can be stored with run-length encoding. The strength of Bloom filters lies in encoding sparse labels in a large universe of potential labels, which is typically the case for nodes in the highest resolution levels of the label list tree. Furthermore, Bloom filters are agnostic to the actual distribution or pattern of label IDs in a node. Random labels are stored as efficiently as clustered label IDs. Currently, we set the false positive rates of Bloom filters heuristically. Ideally, the most efficient false positive rate would be adjusted not only based on the data set, but also for each resolution level. While we currently use Roaring bitmaps and Bloom filters, in the future the underlying data structures of our culling method could be easily extended or exchanged. Any data structure just needs to support (a) fast checking of whether an element is part of a set, and (b) fast intersection operations for updating the query.

Label list tree node size. A major consideration for all block-based culling techniques is how to choose the best size for a block. Smaller blocks allow for more accurate culling, but also result in larger label list trees. Larger blocks are more efficient to store, but culling will be more conservative. A more detailed analysis was given in [20]. In principle, our hybrid method is agnostic to the actual block size used.

Data-adaptive, hierarchical queries. Our hierarchical query pruning approach significantly speeds up query evaluation. Smaller (pruned) queries result in Bloom filters being requested more often during the query-adaptive culling step. This, in turn, results in a smaller memory footprint and faster query evaluation. Furthermore, our query-adaptive approach allows us to trade disk storage space for run-time performance. By storing two data representations on disk, we can select the ideal (i.e., most compact and fastest) data representation at run time.

9 Conclusions

Our culling method for large, labeled volumes combines two main concepts: First, a novel hierarchical data structure for compact storage of integer label lists, and second, a hierarchical culling algorithm which is optimized for efficient handling of large label lists and complex culling queries. The combination of deterministic and probabilistic data structures allows us to find the best trade-off between exact culling with logarithmic-time data access, and conservative culling with constant-time data access. We have combined these data structures with a novel hierarchical culling algorithm. Complex culling queries get pruned at each step of the hierarchy traversal, leading to faster culling times, and label lists get requested based on the current (pruned) query. This combination has enabled significant improvements for fast culling of large, labeled volumes and is scalable to millions of labels.

Acknowledgments

We thank John Keyser for the 'KESM Mouse Brain' data set [34]. This work is partially supported by King Abdullah University of Science and Technology (KAUST) and the KAUST Office of Sponsored Research (OSR) award OSR-2015-CCF-2533-01.

REFERENCES

[1] C. Ahlberg, C. Williamson, and B. Shneiderman. Dynamic Queries for Information Exploration: an Implementation and Evaluation. In *SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 619–626, 1992.

[2] R. S. Avila, L. M. Sobierajski, and A. E. Kaufman. Towards a comprehensive volume visualization system. In *IEEE Visualization*, pages 13–20, 1992.

[3] 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. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

[4] J. Beyer, A. Al-Awami, N. Kasthuri, J. W. Lichtman, H. Pfister, and M. Hadwiger. ConnectomeExplorer: Query-Guided Visual Analysis of Large Volumetric Neuroscience Data. *IEEE Trans. on Visualization and Computer Graphics (Proc. IEEE SciVis '13)*, 19(12):2868–2877, 2013.

[5] J. Beyer, M. Hadwiger, A. Al-Awami, W.-K. Jeong, N. Kasthuri, J. Lichtman, and H. Pfister. Exploring the Connectome - Petascale Volume Visualization of Microscopy Data Streams. *IEEE Computer Graphics and Applications*, 33(4):50–61, 2013.

[6] J. Beyer, M. Hadwiger, and H. Pfister. State-of-the-art in GPU-based large-scale volume visualization. *Computer Graphics Forum*, 8(34):13–37, 2015.

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

[8] I. Boada, I. Navazo, and R. Scopigno. Multiresolution Volume Visualization with a Texture-based Octree. *The Visual Computer*, 17(3):185–197, 2001.

[9] S. Bruckner, V. Šoltészová, M. E. Gröller, J. Hladuvka, K. Bühler, J. Yu, and B. Dickson. BrainGazer - Visual Queries for Neurobiology Research. *IEEE Trans. on Visualization and Computer Graphics (Proc. IEEE Visualization '09)*, 15(6):1497–1504, 2009.

[10] A. Cabrera. Boost Bloom Filters. `https://github.com/queertypes/boost-bloom-filters`, 2018. Last accessed 2018-7-31.

[11] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Softw. Pract. Exper.*, 46(5):709–719, 2016.

[12] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Interactive 3D Graphics and Games*, pages 15–22, 2009.

[13] M. Derthick, J. Kolojejchick, and S. F. Roth. An Interactive Visual Query Environment for Exploring Data. In *Tenth Annual ACM Symposium on User Interface Software and Technology (UIST '97)*, pages 189–198, 1997.

[14] E. Dyllong and C. Grimm. A modified reliable distance algorithm for octree-encoded objects. *Applied Mathematics and Mechanics (PAMM)*, 7(1):4010015–4010016, 2007.

[15] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. A. K. Peters, Ltd., 2006.

[16] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 75–88, 2014.

[17] 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.

[18] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[19] E. Gobbetti, F. Marton, and J. Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):797–806, 2008.

[20] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister. SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE SciVis '17)*, 24(1):974–983, 2018.

[21] M. Hadwiger, C. Berger, and H. Hauser. High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware. In *IEEE Visualization*, pages 301–308, 2003.

[22] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach. *IEEE Trans. on Visualization and Computer Graphics (Proc. IEEE SciVis '12)*, 18(12):2285–2294, 2012.

[23] M. Hadwiger, C. Sigg, H. Scharsach, and K. Bühler. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum (Proc. Eurographics '05)*, 24(3):303–312, 2005.

[24] S. D. Jackman, B. P. Vandervalk, H. Mohamadi, J. Chu, S. Yeo, S. A. Hammond, G. Jahesh, H. Khan, L. Coombe, R. L. Warren, and I. Birol. Abyss 2.0: Resource-efficient assembly of large genomes using a bloom filter. *bioRxiv*, page 068338, 2016.

[25] N. Kasthuri, K. J. Hayworth, D. R. Berger, R. L. Schalek, J. A. Conchello, S. Knowles-Barley, D. Lee, A. Vázquez-Reina, V. Kaynig, T. R. Jones, et al. Saturated reconstruction of a volume of neocortex. *Cell*, 162(3):648–661, 2015.

[26] V. Kaynig, A. Vazquez-Reina, S. Knowles-Barley, M. Roberts, T. R. Jones, N. Kasthuri, E. Miller, J. Lichtman, and H. Pfister. Large-scale automatic reconstruction of neuronal processes from electron microscopy images. *Medical Image Analysis*, 22(1):77 – 88, 2015.

[27] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature sensitive surface extraction from volume data. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 57–66, 2001.

[28] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *IEEE Visualization*, pages 287–292, 2003.

[29] E. Lamar, B. Hamann, and K. I. Joy. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *IEEE Visualization*, pages 355–362, 1999.

[30] 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. spe.2560.

[31] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Softw. Pract. Exper.*, 46(11):1547–1569, Nov. 2016.

[32] C.-Y. Lin, K.-L. Tsai, S.-C. Wang, C.-H. Hsieh, H.-M. Chang, and A.-S. Chiang. The neuron navigator: Exploring the information pathway through the neural maze. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium*, pages 35–42, 2011.

[33] B. Matejek, D. Haehn, F. Lekschas, M. Mitzenmacher, and H. Pfister. Compresso: Efficient compression of segmentation data for connectomics. In *International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, pages 781–788, 2017.

[34] D. Mayerich, J. Kwon, C. Sung, L. C. Abbott, J. Keyser, and Y. Choe. Fast macro-scale transmission imaging of microvascular networks using KESM. *Biomedical Optics Express*, 2:2888–2896, 2011.

[35] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[36] A. Sherbondy, D. Akers, R. Mackenzie, R. Dougherty, and B. Wandell. Exploring connectivity of the brain's white matter with dynamic queries. *IEEE Trans. on Vis. and Computer Graphics*, 11(4):419–430, 2005.

[37] L. M. Sobierajski and R. S. Avila. A hardware acceleration method for volumetric ray tracing. In *IEEE Visualization*, pages 27–34, 1995.

[38] K. Stockinger, J. Shalf, K. Wu, and E. W. Bethel. Query-Driven Visualization of Large Data Sets. In *IEEE Visualization*, pages 167–174, 2005.

[39] K. R. Subramanian and D. S. Fussell. Applying space subdivision techniques to volume rendering. In *IEEE Visualization*, pages 150–159, 1990.

[40] V. Vidal, X. Mei, and P. Decaudin. Simple empty-space removal for interactive volume rendering. *Journal of Graphics Tools*, 13(2):21–36, 2008.

[41] R. Westermann and B. Sevenich. Accelerated volume ray-casting using texture mapping. In *IEEE Visualization*, pages 271–278, 2001.

[42] K. Wu, K. Stockinger, and A. Shoshani. Breaking the curse of cardinality on bitmap indexes. In *Scientific and Statistical Database Management*, pages 348–365, 2008.