Efficient Bundled Spatial Range Queries

Eleni Tzirita Zacharatou
EPFL
eleni.tziritazacharatou@epfl.ch

Darius Šidlauskas
EPFL
darius.sidlauskas@epfl.ch

Farhan Tauheed*
Oracle Labs Zurich
farhan.tauheed@oracle.com

Thomas Heinis
Imperial College London
t.heinis@imperial.ac.uk

Anastasia Ailamaki
EPFL and RAW Labs SA
anastasia.ailamaki@epfl.ch

ABSTRACT

Efficiently querying multiple spatial data sets is a growing challenge for scientists. Astronomers query data sets that contain different types of stars (e.g., dwarfs, giants, stragglers) while neuroscientists query different data sets that model different aspects of the brain in the same space (e.g., neurons, synapses, blood vessels). The results of each query determine the combination of data sets to be queried next. Not knowing a priori the queried data sets makes it hard to choose an efficient indexing strategy.

In this paper, we show that indexing and querying the data sets separately incurs considerable overhead but so does using one index for all data sets. We therefore develop STITCH, a novel index structure for the scalable execution of spatial range queries on multiple data sets. Instead of indexing all data sets separately or indexing all of them together, the key insight we use in STITCH is to partition all data sets individually and to connect them to the same reference space. By doing so, STITCH only needs to query the reference space and follow the links to the data set partitions to retrieve the relevant data. With experiments we show that STITCH scales with the number of data sets and outperforms the state-of-the-art by a factor of up to 12.3.

CCS CONCEPTS

• **Information systems → Data structures**; **Data access methods**; **Multidimensional range search**.

KEYWORDS

Spatial Range Query, Multiple Spatial Data Sets, Spatial Indexing, Spatial Data Partitioning, Spatial Data Management

ACM Reference Format:
Eleni Tzirita Zacharatou, Darius Šidlauskas, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2019. Efficient Bundled Spatial Range Queries. In *27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19), November 5–8, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3347146.3359077

*This work was done while the author was at EPFL.

SIGSPATIAL '19, November 5–8, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6909-1/19/11…$15.00
https://doi.org/10.1145/3347146.3359077

1 INTRODUCTION

In several real-life applications data is naturally divided into distinct categories and users are often interested in a subset of them. Additionally, users issue queries that explore different combinations of categories as they rarely know a priori which categories need to be combined to answer a particular question or test a specific hypothesis. To build an anatomically accurate spatial atlas of the human brain, for example, the neuroscientists in the Human Brain Project (HBP) [15] study the structure and shape of neurons using bright-field microscopy, and images of the whole brain using magnetic resonance imaging (MRI). This process results in different data sets that contain data originating from different observational sources, potentially representing different types of neurons or other brain structures (e.g., synapses and blood vessels). Inspecting the same brain regions in these different data sets allows to verify that a given region contains the correct ratio and distribution of different brain structures, and is key to building a brain atlas. There is consequently the need to efficiently retrieve the same spatial region from a number of different data sets, stored on disk due to their size. This problem is challenging because the queried data sets are chosen ad-hoc depending on the results of previous queries and thus are not known a priori. As more observational sources are added, the problem becomes increasingly challenging.

Formally, let D be a set of N spatial objects. Each object has a spatial extent[1] and a tag that denotes the object's *category*. Categories have application-specific semantics; for example the different categories can correspond to neurons of different types, or to the same neurons obtained from different samples.

Definition 1 (Bundled Spatial Range Query). *Let c be the number of distinct categories; each tag thus is represented as an integer in $\{1, 2, ..., c\}$. Denote by D_i $(1 \leq i \leq c)$ the set of objects in D having category tag i. Every object has exactly one tag, and thus $D_1, D_2, ..., D_c$ are mutually disjoint and $\bigcup_{i \in c} D_i = D$. Given an axis-aligned range query r defined as a three dimensional interval $r = [l_1, u_1] \times [l_2, u_2] \times [l_3, u_3]$ and a non-empty set $Q \subset \{1, 2, ..., c\}$, a bundled spatial range query returns, for each $i \in Q$, all objects $d \in D_i$ intersecting with r. We call the query parameter Q a category selection. Note that Q can be any non-empty subset of $\{1, 2, ..., c\}$, i.e., a combination of categories. The total number of possible Q is $2^c - 1$.*

Existing spatial indexing approaches can be applied, but not knowing a priori which combination of categories will be queried together renders them inefficient. Using a single index for all categories is only efficient when a query is executed on all categories.

[1]The description in this paper focuses on 3D objects but the proposed techniques also work on 2D and higher-dimensional objects.

Otherwise the I/O overhead can be considerable as data not belonging to the categories of interest needs to be retrieved from disk and filtered out. On the other extreme, using multiple indexes, one for each category, becomes inefficient when the number of categories is large, as the same spatial region has to be repeatedly located within different index structures (that typically suffer from over-coverage, i.e., dead space, and overlap of minimum bounding boxes [9]). Overall, the dominant cost of using (i) a single index is retrieval and filtering of unnecessary data and (ii) multiple indexes is repeated traversal of index structures.

To achieve the best of both worlds, we introduce a new indexing approach that exhibits superior performance by eliminating unnecessary I/O operations. Our approach consists of two phases. The first phase is category-oblivious; it simply uniformly divides the reference space that encompasses all the categories. The second phase is category-aware and segments each individual category into partitions in a data-driven manner. To enable pinpoint access to regions in the queried categories, the category partitions are linked to the uniform partitions of the reference space. With this strategy, our approach retrieves data from precisely the categories needed without incurring the undue overhead of querying separate indexes.

Note that we target scientific use cases where all the raw data (or at least most of it) is available before querying. We thus focus on developing a bulkloading approach.

Contributions. To the best of our knowledge, we are the first to study in-depth the problem of bundled spatial range queries and to identify the lack of a solution that provides efficient access to individual categories (or data sets) that are all enclosed within the same spatial volume. Given the importance of this problem in real-world applications, we advocate that specialized efforts are required to improve the performance of existing spatial indexing approaches. The main contribution of this work is the development of a spatial indexing approach that scales with an increasing number of categories, yet without penalizing performance when the number of categories is small.

The design of our approach is based on the key observations that most overhead in baseline approaches originates from (a) traversing (hierarchical) index data structures repeatedly and unnecessarily and from (b) late pruning results from irrelevant categories.

We therefore develop an index which is based on a simple, flat, grid-based reference space. With this, query execution can quickly assess which areas are likely to be in the query result without traversing one or multiple hierarchical index structures repeatedly. Furthermore, to enable early pruning irrelevant categories in the query execution, we store information on where and how to retrieve category specific data within the grid, in a data structure that enables efficient filtering based on categories. As a result, our approach scales with an increasing number of categories, avoiding the retrieval of an excessive amount of index related data structures as well as irrelevant results. Our approach also allows to incrementally add new categories with only limited updates to the reference space (and no updates to other categories) because each category is treated, partitioned and stored independently.

To showcase these techniques, we develop STITCH, a bundled spatial index that achieves efficient query execution while scaling with an increasing number of categories. Albeit we base STITCH on simple ideas, these ideas prove to be effective. As our experiments on real-world data show, STITCH achieves a speedup of $\sim$ 4.5× compared to indexing each category separately with a state-of-the-art index, and 1.3× - 12.3× compared to indexing all categories with a single index.

2 MOTIVATION

Spatial data is at the core of many scientific processes as scientists study entities using their morphological or topological properties. Observational data is acquired using a variety of different instruments and techniques and originates from a variety of input samples, resulting in multiple data sets describing the same spatial volume. Given these data sets, scientists need to efficiently perform ad-hoc queries collecting data from only a subset of them. Querying a certain combination of data sets that are arbitrarily chosen from a pool of tens or hundreds of data sets is the general problem that drives the design of STITCH.

Use Cases. The main motivation behind our work stems from our collaboration with the Human Brain Project (HBP) [15]. Neuroscientists in the Human Brain Project aim to build an atlas of the human brain which will serve as a unifying "spatial scaffold" for studying different aspects of the brain. The input to create this atlas is observational data collected using a variety of light microscopy modalities (such as laser-scanning, wide-field epifluorescence, and bright-field microscopy) and magnetic resonance imaging (MRI). Scientists then need to ensure that their spatial model is biorealistic. To do so, they compute statistical properties for different regions in the model and compare them with the observational data. Overall, in both the building and the validation phase, scientists need to retrieve complementary information about a given spatial region from a subset of the data sources that are available to them in an exploratory fashion. The bulk of the data used in this analysis is static, as scientists incorporate new information by adding new data sources rather than updating existing ones. With the increase in the number and size of the analyzed data sets, the exploration process is significantly hampered.

In other disciplines (e.g., cosmology [12] and seismology [1]) scientists simulate phenomena on a large scale. The outcome of the simulations are several data sets, each containing a different representation of the simulation result and, in the analysis of the result, parts of different data sets need to be combined. In cosmology, for example, N-body simulations of different particle types (dark matter, gas, stars, etc.) are used to study the evolution of the universe. Each resulting data set stores the locations of one particle type and for the final analysis, astronomers need to query different data sets together without knowing a priori the exact combination.

Data Management Challenge. There are three straightforward strategies for evaluating bundled spatial range queries using existing spatial indexes. The first strategy, `1-for-each`, builds a dedicated spatial index (e.g., R-Tree [9]) for each category. A bundled spatial range query is evaluated by searching the $|Q|$ spatial indexes on the categories in Q. Adding a new category incurs negligible overhead as it simply entails building an index for the new category. The second strategy, `all-in-1`, builds a single index structure containing all categories. Given a bundled spatial range query, it traverses the index to get a set of spatial objects potentially qualifying the query predicates, it filters out irrelevant items that do not

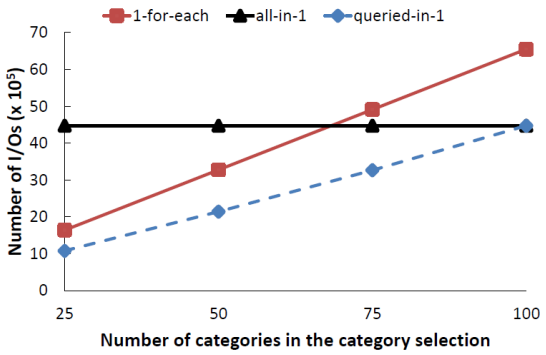

Figure 1: Scaling with an increasing number of categories in the category selection. `1-for-each` does not scale well as the number of categories increases, `all-in-1` introduces an overhead when only a small subset is queried, while `queried-in-1` provides the best performance, but is a practically infeasible solution.

belong to any of the queried categories in Q, and finally evaluates the spatial predicate on the remaining items. New categories are added by updating the index, which can incur substantial overhead. The third strategy, `queried-in-1`, takes the `all-in-1` strategy to the extreme and builds indexes for all possible category combinations. That way, we can answer a bundled spatial range query by searching the specific index that contains the exact combination of queried categories (and nothing but those).

We implement all strategies using the R-Tree [9] spatial index, which is arguably the most widely used index structure for spatial data. In a motivation experiment we index 100 neuroscience data sets (categories) representing the same brain volume, $\sim 1GB$ each, and measure the total number of I/Os (i.e., disk pages read) for 200 range queries corresponding to different brain regions of size $10^{-3}\%$ of the total volume. To evaluate the performance of the `queried-in-1` strategy, we create indexes that contain exactly the combination of the 25, 50, 75, and 100 categories that are queried together. The precise experimental setup is described in Section 7.

The results of the motivation experiment in Figure 1 show the trade-offs of each strategy as the number of queried categories increases. The `all-in-1` strategy achieves the same consistent performance regardless of the number of categories, while the performance of `1-for-each` is linearly decreasing with the growing number of categories. This is expected, as the former always has to inspect the same (big) index, while the latter has to probe an increasing number of (smaller) indexes. Interestingly, none of these two strategies achieves the best performance in all cases and the crossing point clearly indicates what strategy is preferred for what scenario. As expected, `queried-in-1` provides the best performance, minimizing the query cost irrespective of how many categories are queried. Note, however, that this strategy is unrealistic because the cost of constructing and storing indexes for all possible category combinations ($2^{100} - 1$ in our example) is prohibitively high.

The goal of this work is to develop an indexing approach that has the same querying behavior as if there was a dedicated index for the queried combination, without building indexes for all possible combinations a priori. As we will show next, this is achieved by physically bundling the indexes for different categories.

3 RELATED WORK

Data-oriented partitioning. Arguably the seminal spatial index structure is the R-tree [9]. The R-tree is a disk-based index consisting of a hierarchy of minimum bounding boxes (MBBs) which recursively enclose data objects. By doing so, the R-Tree is resilient to data skew, but faces the problems of over-coverage and overlap of MBBs, which results in multiple (partial) paths being explored during querying. Many extensions to the basic approach have been proposed to address these issues and optimize the node MBBs during dynamic index maintenance. To increase robustness against different data distributions, the R*-tree [4] employs multiple optimization criteria to choose the node into which a new object should be inserted. In addition, it removes and reinserts the spatial objects of an overflowing node in an attempt to minimize the dead space and the margin in each node. The improved query performance comes at the expense of higher update costs. The RR*-tree [5] introduces more adaptive optimization strategies to further reduce I/O costs and enhance search performance. The R+-Tree [16] creates non-overlapping nodes by inserting objects into multiple leaves, which makes the index larger. The cR-tree [7] considers the R-tree node splitting procedure as a typical clustering problem. To find a good split upon a node overflow, it partitions the data in multiple nodes using k-means. Instead of modifying the index structure or the splitting procedure, the approach presented in [17] proposes to solve the problem of over-coverage by improving MBBs. It converts node MBBs to CBBs (Clipped Bounding Boxes) by clipping away dead space that is concentrated around the MBB corners.

Since our data sets are massive and known a priori, we focus on bulkloaded R-Trees. Bulkloading approaches group spatially close objects and store them on the same disk page to improve locality and reduce overlap between nodes. Then, an R-Tree is built on top of those disk pages, typically bottom-up. The Hilbert R-Tree [11] uses the Hilbert space-filling curve to order the objects according to their spatial proximity. Sort-Tile-Recursive (STR) [13] recursively tiles the space, sorts the objects in a tile along each dimension and thereby also guarantees spatial proximity as well as small MBBs, outperforming the Hilbert R-Tree [11]. In contrast, the Top-down Greedy Split (TGS) [8] works top down: it splits the data set into partitions so that on each level the area of each partition is minimized. This process continues recursively until each partition fits on a disk page. While bulkloading with TGS takes much longer than with other approaches, the resulting R-Tree outperforms the Hilbert R-Tree and STR on extreme data sets (with respect to skew and aspect ratio). The Priority R-Tree (PR-Tree) [3] groups all objects with extreme coordinates in the same dimension in the same node, thereby reducing the area and overlap of the remaining nodes. This improves performance on extreme data sets, making the PR-Tree outperform TGS. As recently shown [19], R-Tree-based approaches still suffer considerably from unnecessary I/Os caused by overlap. FLAT [19] consequently adds connectivity (neighborhood) information so that the R-Tree is only used to locate any single data object inside a query volume and the remaining objects are found by crawling through neighbors.

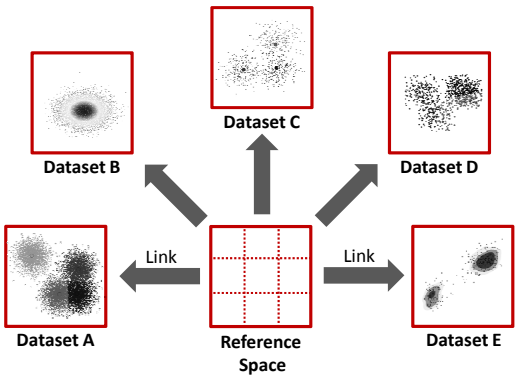

Figure 2: STITCH links multiple data sets (categories) to the same index/reference space (bottom center) and directs queries to the destination data sets via corresponding links.

Space-oriented partitioning. Instead of grouping objects hierarchically based on their proximity and allowing groups to overlap, another family of spatial indexing methods splits the space using hyperplanes into a set of disjoint partitions that are stored flatly [2] or in a hierarchical structure [6, 10, 20]. The simplest space-oriented indexing technique is the uniform grid [2], where a predefined area is divided into rectangular cells. Each cell stores together all the objects that overlap with it. In contrast, the KD-Tree [6] divides the space hierarchically in a data-driven manner. At each level, it splits the objects along one dimension in two partitions such that each partition contains approximately the same number of objects.

Finally, we note that the term *category* in this work simply refers to a *group* of objects (e.g., a data set) rather than a textual attribute. Thus, research on *keyword* search (i.e., [14]) is not related to us.

4 STITCH OVERVIEW

To overcome the aforementioned challenges, the proposed method STITCH avoids the repeated traversal of multiple index structures on disk and the retrieval of unnecessary data. This is achieved by combining data-oriented partitioning with space-oriented indexing.

First, similar to the `1-for-each` and unlike the `all-in-1` strategy, spatial objects belonging to different categories are stored in separate data files to enable retrieval of data from precisely the categories needed. To retrieve data from each file efficiently, spatially close objects are stored on the same disk page. The assignment of spatial objects to disk pages is achieved by applying a data-oriented partitioning method which adapts to the distribution of objects in each individual category.

Second, unlike the `1-for-each` but similar to the `all-in-1` strategy, instead of using one index per category to index the minimum bounding boxes (MBBs) of the pages, STITCH builds a single index on the reference space (the common universe that encloses all the underlying objects from all categories). To make access to this *reference index* efficient, STITCH avoids a hierarchical structure and organizes it in a uniform grid. The grid cells store links to the categories, i.e., each cell stores links to the category pages it overlaps with. To have more pruning power in the reference index, along each page link, the cell also stores the page MBB. Within each cell, the links (and corresponding page MBBs) are arranged such that links to a specific category can be efficiently accessed. Figure 2 shows the overview structure of STITCH.

With the reference index and the category pages, the result of range queries is computed in two phases. STITCH first probes the reference index to find all grid cells overlapping with the query range and retrieves all the page MBBs from those cells for the queried categories. This phase is based on the key insight that unlike data-oriented hierarchical indexes, our reference index does not suffer from overlap. Furthermore, STITCH avoids traversing multiple index structures because all the categories are mapped to the same reference index. In the second phase, STITCH discards the page MBBs that do not fall in the queried area and only visits the qualifying disk pages by following the corresponding links. That way, STITCH retrieves spatial objects from exactly those categories needed and for the queried area, thereby significantly reducing the amount of unnecessary data retrieved. Comparing to existing grid-based indexing approaches, STITCH suffers less from the problem of data replication and can thus accommodate categories having objects of varying sizes without the need for expensive fine-tuning of the grid configuration. This is because the uniform grid in STITCH indexes page MBBs rather than individual spatial objects - the actual spatial objects are organized using a data-oriented space partitioning strategy which is resilient to data skew. As we discuss in more details in the following, we adapt the data-oriented partitioning strategy to reduce the amount of replicated data even further.

5 STITCH INDEXING

At the core of STITCH indexing is the algorithm that partitions and links each individual category to the common reference space. In the following section, we first discuss this partitioning and linking algorithm and then present the data structures that store the partitions and the linking information for all categories.

5.1 Partitioning & Linking

We segment the entire space of each category into partitions, each partition corresponding to one disk page. We then create a link between a partition P and a grid cell C of the reference index if and only if at least one element contained in P overlaps C. The created link is essentially the pointer to the partition P (i.e., the location on disk where P is stored) and is stored in grid cell C.

In theory, any partitioning method can be used to partition the categories. Nevertheless, to avoid the problem of replication associated with space-oriented partitioning, STITCH follows a data-oriented partitioning strategy. In particular, we base our partitioning strategy on an existing algorithm, Sort-Tile-Recursive (STR [13]). STR first sorts the spatial objects in the x-dimension and partitions them along this dimension into fixed sized partitions. Each such partition is subsequently sorted and partitioned in the other dimensions (y, and then z). The partition sizes in each dimension are chosen so that the final partitions contain at most as many spatial objects as can be stored on a single disk page.

To minimize partial overlap with the grid cells of the reference index, STITCH extends STR to align the data partitions with the (conceptual) cell boundaries. This is achieved by adding spatial objects in the sorted order to partitions in each dimension until: (i) a grid boundary is crossed, or (ii) the current partition is full. The

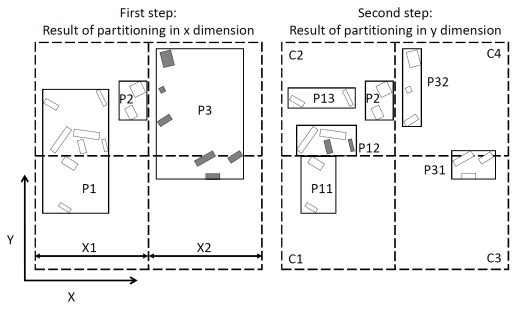

Figure 3: The partitioning procedure packs spatially close elements on the same disk page (rectangle) and aligns the page boundaries as much as possible with the grid boundaries (dashed lines) of the reference index.

first condition makes the partitioning grid-aware and minimizes the number of cells that a final category partition overlaps with (thus minimizing the number of replicated links in the reference index). While doing so may underfill disk pages, i.e., less than the maximum elements are stored in a partition, our strategy proves effective in reducing replicated links and providing fine-grained data access. The strategy ultimately trades disk space for performance.

We call our grid-aligned data-oriented partitioning strategy Sliced Data-Oriented Partitioning (`Sliced-DOP`). `Sliced-DOP` has the same complexity as STR, since it does not introduce any additional passes over the data. The pseudocode of `Sliced-DOP` is given in Algorithm 1. Note that the only information needed to detect a grid boundary crossing is the grid resolution, defined by the input parameter g. Also note that the linking is performed alongside the partitioning and it is based on object overlap and not on page MBB overlap: if the intersection between a page and a grid cell does not contain any objects (dead space), the cell is not linked to the page.

Figure 3 illustrates the intuition behind our partitioning technique using a 2D example. This example assumes that each partition contains at most 4 spatial elements — the same number a disk page can store. Since there are 16 objects in total, they are first divided in x-partitions of maximum 8 objects each along the x dimension, and then they are further divided in the y dimension so that the final partitions contain at most 4 objects. The conceptual grid boundaries are shown with dashed lines. When dividing in the x dimension, partition $P1$ contains the maximum number of objects, as it is fully enclosed in a single grid x-tile ($X1$). On the other hand, even though $P2$ is not full, we do not add the next elements in the sorted order (shown with solid color) in it, so that it remains fully enclosed in the first grid x-tile ($X1$). The remaining 6 elements are then inserted in $P3$. Similarly, when dividing in the y dimension, although $P11$ is not full, we do not extend it with the 2 elements shown with solid grey color so that it remains fully enclosed in $C1$, and a new page, $P12$, starts in $C2$. The same logic applies in the division of $P3$.

The following links are introduced in the reference index: $C1$ is linked to page $P11$, $C2$ is linked to pages $P12$, $P13$ and $P2$, $C3$ to $P31$, and lastly $C4$ to $P31$ and $P32$. This example also highlights that in some cases, avoiding link replication is not possible: $P31$ contains two objects that overlap with both $C3$ and $C4$. As a result, both cells need to contain a link pointing to $P31$. In general, as the number of objects falling on grid boundaries increases, more links need to be replicated. The impact of replication, however, is significantly smaller compared to existing grid-based indexing approaches which replicate individual objects because replication happens at the disk page level and typically objects overlapping the same neighboring cells are stored in the same page. With a larger

Algorithm 1: `Sliced-DOP`

Input: D: array of spatial objects
g: # of uniform grid cells in the reference space
ps: partition size (e.g., # objects per disk page)
Output: P: array of all partitions (stored on disk)
L: reference index (stored on disk)

// Init. running partitions for each dimension:
$P_x \leftarrow \emptyset$; $P_y \leftarrow \emptyset$; $P_z \leftarrow \emptyset$;
$C \leftarrow \emptyset$ // set of grid cells overlapping a running partition
// partition sizes:
$s = \sqrt[3]{|D|/ps}$; $s_x = |D|/s$; $s_y = s_x/s$; $s_z = s_y/s$;
`sortByX(`D`)`; // sorts by x-coord.of object centers
$tile_x \leftarrow$`cellNrAtX(`$g, D[0]$`)`; // current tile at x-dim.
foreach $i \in D$ **do**
  $nextTile_x \leftarrow$ `cellNrAtX(`g, i`)`;
  if $tile_x == nextTile_x$ **then**
    $P_x \leftarrow P_x \cup \{i\}$;
  if $|P_x| == s_x$ **or** $tile_x \neq nextTile_x$ **then**
    `sortByY(`P_x`)`;
    $tile_y \leftarrow$ `cellNrAtY(`$g, P_x[0]$`)`;
    foreach $j \in P_x$ **do**
      $nextTile_y \leftarrow$ `cellNrAtY(`g, j`)`;
      if $tile_y == nextTile_y$ **then**
        $P_y \leftarrow P_y \cup \{j\}$;
      if $|P_y| == s_y$ **or** $tile_y \neq nextTile_y$ **then**
        `sortByZ(`P_y`)`;
        $tile_z \leftarrow$ `cellNrAtZ(`$g, P_y[0]$`)`;
        foreach $k \in P_y$ **do**
          $nextTile_z \leftarrow$ `cellNrAtZ(`g, k`)`;
          if $tile_z == nextTile_z$ **then**
            $P_z \leftarrow P_z \cup \{k\}$;
            // Keep track of overlapping cells:
            $C \leftarrow C \cup$ `cellNr(`g, k`)`;
          if $|P_z| == s_z$ **or** $tile_z \neq nextTile_z$ **then**
            $P \leftarrow P \cup \{P_z\}$; // ready partition
            foreach $c \in C$ **do**
              store in cell c of L the pointer to P_z and the MBB of P_z ; // linking
            $tile_z \leftarrow nextTile_z$; $P_z \leftarrow \emptyset$; $C \leftarrow \emptyset$;
        $tile_y \leftarrow nextTile_y$; $P_y \leftarrow \emptyset$;
  $tile_x \leftarrow nextTile_x$; $P_x \leftarrow \emptyset$;
return P, L

Algorithm 2: STITCH Indexing Algorithm

Input: *datasets*: set of spatial data sets, each corresponding to a distinct category
g: # of uniform grid cells in the reference space
ps: partition size (e.g., # objects per disk page)

ht: in-memory hash table storing the disk offsets of non-empty grid cells;

foreach $D \in datasets$ **do**
  $P, L \leftarrow$ `Sliced-DOP`(D, g, ps);
  foreach grid cell c *of* L **do**
    store in the header page of c the number of pointers in c for dataset D;
    store in *ht* the disk offset of the header page of c for dataset D;

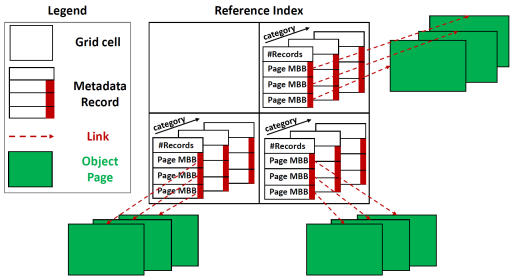

Figure 4: STITCH's data structures and their interaction: The disk-based reference index stores the metadata records in its grid cells which point to the object pages. STITCH also maintains an in-memory hash table indicating the non-empty grid cells which is not shown in the Figure.

page size, more boundary-crossing objects are grouped together and the number of replicated links decreases.

Note that `Sliced-DOP` is *not equivalent* to first applying uniform grid partitioning and then performing STR within each grid cell. The difference is that `Sliced-DOP` sorts the objects *globally* in each dimension, not *locally* within each grid cell. As a result, `Sliced-DOP` preserves the relative one-dimensional distances between objects across grid cells.

The pseudocode for indexing in STITCH, taking into account multiple categories, is given in Algorithm 2.

5.2 Data Structures

This section discusses how we store the partitioning and linking information to support efficient query execution with STITCH. In particular, we discuss STITCH's core data structures, the *metadata* records containing information about each object page, the *reference index* used to retrieve the metadata records in the queried range, and finally *object pages* storing the actual spatial elements.

1) *Metadata:* A metadata record refers to a particular object page and contains a link (pointer) to the object page and the page MBB. By recording the MBB with each link, the data partitions that do not overlap query volumes can be filtered immediately without the need of reading the disk page(s) storing the actual objects.

2) *Reference Index:* To start query execution, the *reference index* must return all the metadata records that fall inside the query range. In STITCH we use a disk-based uniform grid to organize all the metadata records for all the categories in the reference index. A metadata record is stored in a grid cell if the corresponding object page contains at least one object that overlaps with the grid cell. Real simulation data sets have a skewed data distribution and as a result the number of metadata records stored with each grid cell can vary significantly, while a majority of the grid cells are empty. STITCH therefore also maintains an in-memory hash table to store the disk offsets of all the non-empty grid cells. Additionally, the header disk page of each grid cell records the number of links that fall in the grid cell for each category. The links are ordered per category, so that all links for category i precede any link for category j for $i < j$. The metadata records of each grid cell are flushed to disk sequentially. Spatially close records are very likely to be stored on the same grid cell and thereby on the same disk page, resulting in good disk locality for fast retrieval of the metadata.

3) *Object Pages:* On each disk page, `Sliced-DOP` packs the maximum possible number of elements while at the same time ensuring that in most cases an object page is linked with only one grid cell of the reference index. The exact number of elements that a page can hold depends on their size, e.g. for an axis aligned box with an id the size is 6 floats plus 1 integer. Spatial locality is preserved by storing spatially close objects on the same page.

All data structures and their relations are illustrated in Figure 4: several metadata records are stored on each grid cell of the reference index and each metadata record contains a link to an object page.

6 STITCH QUERY EXECUTION

STITCH answers a query on a subset Q of all categories in two phases: it first retrieves the set of links that overlap with the query range and point to the queried categories and then, by following those links, it retrieves the actual objects.

More precisely, STITCH first probes the reference index and finds the cells overlapping the query volume. It reduces the search space instantly and only fetches the links to partitions as well as the partition's MBB from the grid cells for each of the queried categories $q \in Q$. STITCH initially purges any duplicate links retrieved from different grid cells that intersect with the same data partition. In a next step STITCH further reduces the number of category data pages needed to be retrieved by discarding the links associated with an MBB that does not overlap with the query volume. With the set of remaining links, STITCH retrieves the disk pages that contain the actual objects. In a last filtering step, STITCH discards the objects whose MBB does not overlap with the query volume. The pseudo code of the complete STITCH querying algorithm is described in Algorithm 3.

Clearly, STITCH avoids the repeated traversal of the index structure because we query the reference index only once. In addition, using a uniform grid as the reference index enables efficient query execution: STITCH can readily calculate the intersection between the query and the grid cells and obtain the offsets of those intersecting grid cells on disk. Finally, STITCH reduces the amount of

Algorithm 3: STITCH Querying Algorithm

Input: L: reference index
g: # of uniform grid cells in the reference space
query: spatial range query
datasets: set of queried spatial data sets, each corresponding to a distinct category
Output: *objects*: result set of objects

```
// set of grid cells overlapping the query:
C ← cellNr(g, query);
foreach dataset D ∈ datasets do
    P ← ∅ ;                    // set of qualifying links
    foreach c ∈ C do
        fetch from cell c of L the metadata_records of D;
    foreach m ∈ metadata_records do
        if m.MBB does not overlap query then
            discard m;
        else
            P = P ∪ m.link;
    foreach p ∈ P do
        retrieve the disk page pointed by p;
        foreach object o on the disk pages do
            if o overlaps query then
                objects = objects ∪ o;
return objects
```

retrieved data effectively by first only retrieving links pointing to categories that are queried for and ultimately discarding the links associated with MBBs that do not overlap with the query range.

7 EXPERIMENTAL EVALUATION

In the following section we describe the experimental setup & methodology and demonstrate the benefits of STITCH using real neuroscience workloads. We compare STITCH against the two previously introduced strategies, `1-for-each` (one index per category) and `all-in-1` (a single index for all categories), and present a detailed breakdown of the performance. Lastly, we conduct a sensitivity analysis where we vary specific data set, workload and configuration parameters to better understand STITCH's behavior.

7.1 Experimental Setup & Methodology

Hardware Configuration. The experiments were performed on a Linux Ubuntu 12.04 machine equipped with 2× Intel Xeon Processors each with 6 cores running at 2.8GHz, and 48GB RAM. The storage consists of 2 SAS disks of 300GB capacity each.

Competing Approaches. We experimentally compare STITCH against the following spatial indexes: FLAT `1-for-each`, FLAT `all-in-1` and GRID (`1-for-each`). We omit comparisons against the R-Tree because it is outperformed by FLAT [19]. For our workload, FLAT `1-for-each` answers queries by up to 2× faster compared to R-tree `1-for-each`, and FLAT `all-in-1` is ∼ 6× faster than R-tree `all-in-1`. We use the original implementation of FLAT [19] that the authors made available to us and our own implementation of a uniform disk-based grid index. Given that it is unrealistic to index an entire data set in-memory before flushing the data to disk, GRID inherently writes data for each grid cell to disk individually (and thus distributed) when the memory buffer becomes full, which can result in random reads during the querying phase. Similarly to STITCH, GRID maintains an in-memory hash table storing the disk offsets of all the non-empty grid cells. To avoid replicating objects that overlap with multiple grid cells, in our GRID implementation we adopted the following strategy proposed in [18]: each object is assigned only to the grid cell that encloses its center. During querying, to ensure that all the objects intersecting a grid cell are retrieved, the query range is enlarged by the width of the biggest object in each dimension.

All approaches are implemented single-threaded in C++ and compiled with g++ with the maximum optimization level.

Configuration Parameters. Given the absence of heuristics, we identify the best performing configuration for STITCH and GRID with a parameter sweep. The resolution of GRID is set to 60^3 cells, which balances the number of objects retrieved and disk spatial locality. STITCH performs best with 100^3 cells for the reference index. We set the disk page to 4KB for all approaches and for both metadata and data files. The memory footprint of all indexes is limited to 1GB during index building, and we use only one disk (i.e., no RAID configuration). We assume cold system caches in all experiments: the OS caches and disk buffers are cleared (overwritten with an empty file) before executing each query.

Data Sets. We use data sets that model a small part of the brain with a surface mesh consisting of 3D triangles in a volume of 285 μm^3. Each neuron type forms one category. Different categories are stored in different data sets (similarly to the cosmology use case described in Section 2). In other words, each data set we use in our experiments (10 data sets in total) corresponds to a subset of the neurons that are contained in the same brain volume. We approximate the 3D triangles with axis aligned MBBs and store only these MBBs along with an object identifier in the object partitions. All approaches therefore only test for overlap between the stored MBBs and the query volume. The MBB coordinates are represented with double precision floating point numbers and the object identifier is an integer. Each data set occupies ≤ 5 GB on disk, and in total the indexed data is ∼ 45 GB.

Benchmark. We define a benchmark which consecutively executes 200 spatial range queries of varying sizes. The size, aspect ratio and the location of each query is randomly chosen. The average query volume in the benchmark is 10^{-6}% of the entire universe. This benchmark is derived from our neuroscience use-case where specific subvolumes are retrieved with range queries for analysis purposes. As different subregions in the brain vary significantly in volume, the size of the issued queries varies accordingly.

7.2 Comparative Analysis

We first perform a comparative analysis among all competing approaches. For each experiment, we execute the same 200 queries taken from our real neuroscience benchmark to compare the total query execution time and the total amount of data retrieved. Additionally, we compare the time to build the indexes and the storage space required for each approach. Lastly, we compare the cost of adding a new data set in each approach. In each experiment, we increase the number of data sets that are queried to show how each

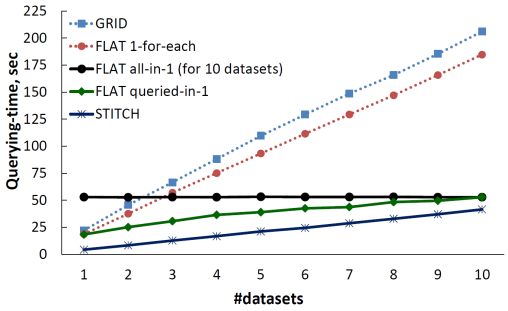

Figure 5: Scaling-up with the number of queried data sets.

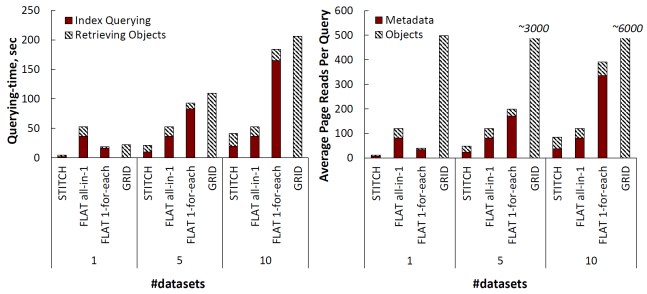

Figure 6: Breakdown of: Query execution time (left) and Pages read per query (the page size is 4KB) (right).

approach scales as scientists increase the number of data sets (thus the total amount of data that is queried increases as well).

Query Execution Time. Figure 5 shows the query execution time as we increase the number of data sets queried, when all 10 data sets have been indexed a priori. In the same figure we also plot the line which shows the optimal strategy based on FLAT, i.e., indexing precisely the data sets needed for each query combination (`queried-in-1`, see Section 2). STITCH executes the queries fastest because the majority of the time (~ 53%) is spent on useful work, i.e., retrieving objects, not on traversing index structures or retrieving metadata information as other index structures do.

FLAT on the other hand incurs a higher overhead for retrieving metadata information. As Figure 6 shows, querying the index can take up to ~ 90% of the total time for FLAT `1-for-each`. The FLAT `all-in-1` strategy requires roughly the same time as the same number of objects are always retrieved irrespective of how many data sets are queried. In addition to the cost of navigating the FLAT index, FLAT `all-in-1` retrieves all the objects within the query volumes from all data sets, as Figure 6 shows.

As a consequence, STITCH is 12.3× faster than FLAT `all-in-1` when only a single data set needs to be queried. When all data sets are queried, STITCH performs comparably to the `all-in-1` strategy, but is still 1.3× faster. Crucially, the trends in Figure 5 *do not imply a crossover point* where STITCH is outperformed by FLAT `all-in-1` if we add more data sets: the only reason why the query execution for FLAT `all-in-1` remains constant is because it always indexes 10 data sets throughout the experiment. If we add more data sets, the query execution time for FLAT `all-in-1` will increase (i.e., the flat black line corresponding to `all-in-1` will move higher up in the graph).

The `1-for-each` approach scales poorly compared to the other approaches. In particular, compared to STITCH, FLAT `1-for-each` performs ~ 4.5× slower on average. The key reason is that the cost of navigating each FLAT index grows due to having a separate index for each data set as shown in Figure 6. GRID `1-for-each` performs ~ 5× slower on average compared to STITCH. The primary reason is that the skew in the data sets results in grid cells that contain many objects and a lot of unnecessary data is retrieved during querying as Figure 6 indicates.

Data Retrieved. Comparing the measurements in the left side of Figure 6 with the respective approaches in the right side of the same figure, we can see that the data retrieved from disk correlates with

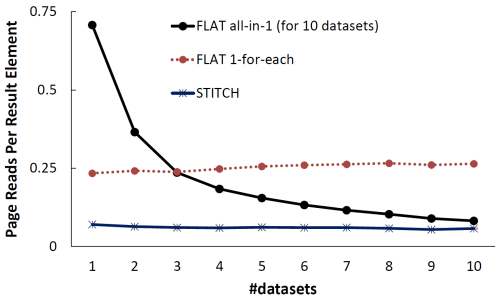

Figure 7: Total number of page reads per result element.

the total query execution time and therefore is the most significant factor that defines the trend of performance for each approach (i.e., the execution time for all approaches is I/O bound). Operations such as testing overlap between partitions and querying MBBs - as well as filtering objects in case of the `all-in-1` approach - are performed while the data resides in memory and therefore do not significantly affect query execution. Although GRID retrieves by far the largest amount of data compared to all the other approaches, the query execution time is not affected severely because the access pattern is mainly sequential.

To further study the different index structures and quantify their overheads, we measure the number of page reads per result element as we increase the number of data sets, focusing on FLAT and STITCH. Figure 7 shows that STITCH has a fixed overhead irrespective of the number of data sets. FLAT `all-in-1` has a big overhead when only 1 data set is queried out of the 10 indexed data sets, but it converges to STITCH's performance when all 10 data sets are queried. Finally, FLAT `1-for-each` has a higher overhead than STITCH which is slightly increasing with more data sets.

Index Time. Considering the time to build the indexes, the `all-in-1` approach is the most time-consuming. FLAT index building requires (externally) sorting the data on each dimension to create partitions, building a tree on top of the partitions and then using the tree to find neighboring partitions. Building many smaller indexes is more efficient, mostly because each index operates on a smaller data set and thus requires fewer (or even zero) external passes for sorting. In our experiments, `1-for-each` outperforms `all-in-1` by ~ 40% as shown in Figure 8. STITCH indexes faster than FLAT, because virtually no time is required for creating a grid index on

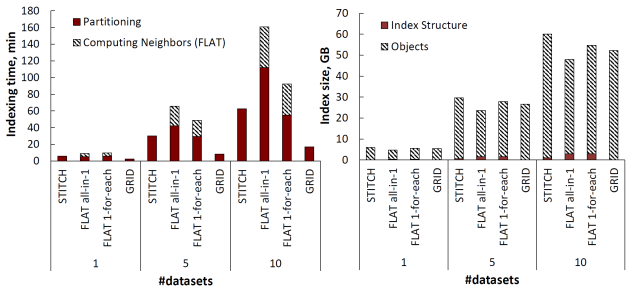

Figure 8: Overall time to index (left) and Index size (right).

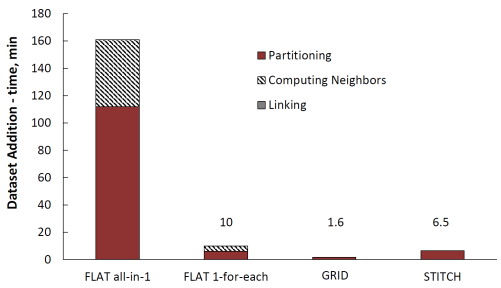

Figure 9: Extending an existing index with a new data set.

the reference space and constant time is required for computing the overlap between the grid cells and the data partitions. Therefore STITCH spends only 1% of the time for linking the reference index and the data sets while the remaining 99% is spent on partitioning the data sets. GRID indexes the fastest because it simply partitions the data uniformly and does not require an external sort.

Index Size. If we compare the storage space needed by all indexes in each approach, we see that the majority of the space is taken by the objects themselves (partitioned data sets), roughly 45 GB for all 10 data sets. STITCH requires the least amount of space (~1 GB) to store the metadata information. Both FLAT `all-in-1` and `1-for-each` have virtually the same index structure and therefore require a similar amount of space - around 3GB for 10 data sets while GRID does not need to store any index structure at all. In terms of the space needed to store the objects, both GRID and mainly STITCH introduce some empty space in the object pages, requiring 16% and 31% more space compared to FLAT respectively. In the case of STITCH, this extra space enables more fine-grained filtering and thus results in faster overall query execution.

Index Update. Scientists often acquire more data of the phenomena being studied. Providing indexing support for newly added data sets is therefore important. We thus analyze the cost of adding a new data set in each approach. We initially index 9 out of our 10 data sets, and measure the time for adding the 10th data set (of size 4.9 GB). FLAT is designed for bulkloading and therefore it is more efficient to re-build `all-in-1` from scratch. `1-for-each` only requires to build the index for the new data set and is therefore much cheaper as shown in Figure 9. STITCH needs to partition the new data set and link the partitions to the reference index which is a faster process than building the whole index in `1-for-each`. Overall STITCH finishes the update about 35% faster than `1-for-each`. GRID is the fastest approach as it simply partitions the new data set uniformly without the need to sort it first.

7.3 Sensitivity Analysis

In the following, we perform an analysis of STITCH to understand the impact of different workload and configuration characteristics on the performance. The following experiments are performed using the same query workload described in Section 7.1 and indexing 4 neuroscience data sets while querying all 4 of them.

Scaling with Data Set Size. In this experiment we execute the same queries on data sets with increasing size and we study the impact on the number of links between the reference space and the

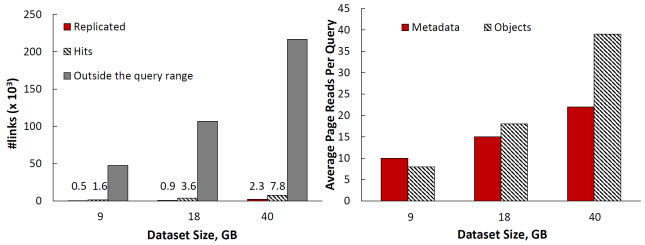

Figure 10: Number of links (left) and Pages read per query (right) for increasing data set sizes.

category partitions. We define three cases: (1) we index and query successively four small sized data sets (a total size of 9 GB), each one containing a small set of neurons, (2) we use four data sets of medium size (18 GB in total) and (3) we use four big data sets (40 GB in total) containing a large set of different types of neurons. For a fair comparison, the granularity of the reference index is not adjusted to the size of the data sets (the default configuration of 100^3 cells is used). As all the data sets are contained within the same reference space, adding more neurons results in increasingly denser data sets. The overlap between the grid cells and the category partitions thus increases. As shown in Figure 10 (left), for a given grid cell, there are more links to category partitions outside the query range. The number of replicated links (links retrieved from multiple grid cells) and the number of links that point to the actually overlapping category partitions (hits) increase as well (because the query size is constant). The right side of Figure 10 shows that as the data set size increases, there are more objects in the query result, and retrieving them becomes the dominant factor.

Grid Resolution. The number of grid cells has an impact on the number of links that need to be retrieved from disk to evaluate a query. When the grid resolution is too low, the grid cannot effectively prune the links to the category partitions that fall outside the query volume. Figure 11 shows that the amount of data retrieved from the reference index (left-hand side) decreases as the resolution increases from 100 to 160 cells per dimension. However, increasing the resolution further does not help in reducing the amount of retrieved data. The right-hand side of Figure 11 shows the impact of the grid resolution on the category partitions. As the resolution increases, `Sliced-DOP` leaves more empty space in the object pages (i.e., creates category partitions with a smaller number of objects), resulting in an overhead of up to 40% for 200 cells per dimension.

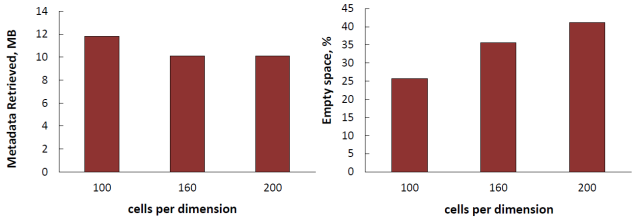

Figure 11: Amount of retrieved metadata (left) and Percentage of empty space in object pages (right) for increasing grid resolution.

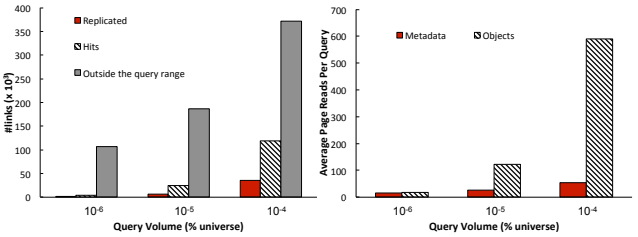

Figure 12: Number of links (left) and Pages read per query (right) for increasing query volume.

Query Volume. In this experiment, we increase the volume of the queries from 10^{-6}% to 10^{-4}% of the entire universe volume. Since the grid resolution is fixed, when bigger queries are executed, an increasing number of grid cells overlap the query. However, not all the links inside those cells are pointing to a category partition that overlaps the query. As Figure 12 shows, the increase in the number of overlapping cells results in an increasing number of links that do not overlap the query volume. The same figure (right) shows that as the query volume (and the number of cells that overlap with the query) increases, more pages are read for both metadata and objects, but retrieving objects is the dominant factor.

8 DISCUSSION AND CONCLUSIONS

In this paper we identify the challenge of efficiently exploring multiple spatial data sets with the same range query—a common task across scientific applications. Not knowing a priori which data sets will be queried makes it particularly challenging to accelerate access: indexing all possible combinations of data sets is not feasible as it takes too long and requires too much space leaving us to choose between two extremes – indexing each data set individually and using one index for all data sets. As we show, neither of the two extremes is efficient: the first does not scale well with an increasing number of queried data sets, and the second is inefficient when only a small subset of the indexed data sets is queried.

Based on these key insights we develop STITCH, a novel disk-based index that combines data-oriented partitioning with space-oriented indexing. Using data-oriented partitioning for the data sets we can effectively address skew in the distribution of spatial objects. At the same time we refrain from using a hierarchical structure to access the partitioned data sets (and thus avoid the associated overhead) and instead link the data set partitions to a central space-oriented index. Using space-oriented partitioning in the reference space, we cover the entire universe of all categories without having a priori knowledge of the data distribution, and thus ensure that all category partitions (even those of future categories) are "stitched" to at least one partition in the reference space. In particular, STITCH employs a uniform grid for its efficiency in building, querying and updating with new categories. Links are stored for all intersecting pairs of grid cells with data set partitions. The only drawback is that the grid resolution has to be defined statically. Alternatively, other space-partitioning indexes could be used as the reference index, such as octrees [10] or kd-trees [6].

Key to the approach is the use of Sliced Data-Oriented Partitioning (`Sliced-DOP`): to avoid storing and ultimately following an excessive number of links, the uniform grid guides the partitioning of the data sets. Our extensive experimental analysis shows that with these measures our approach outperforms the state-of-the-art by up to a factor of 12.3 for a real neuroscience workload.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive feedback. This work was supported by the EU Horizon 2020 research and innovation programme (Human Brain Project).

REFERENCES

[1] V. Akcelik, J. Bielak, et al. 2003. High resolution forward and inverse earthquake modeling on terascale computers. In *Supercomputing*. 52–52.
[2] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. 1989. Geometric computing and uniform grid technique. *CAD* 21, 7 (1989), 410–420.
[3] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. 2004. The Priority R-tree: A Practically Efficient and Worst-case Optimal R-tree. In *Proc. SIGMOD*. 347–358.
[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. SIGMOD*. 322–331.
[5] N. Beckmann and B. Seeger. 2009. A Revised R*-tree in Comparison with Related Index Structures. In *Proc. SIGMOD*. 799–812.
[6] J. L. Bentley. 1975. Multidimensional binary search trees used for associative searching. *CACM* 18, 9 (1975), 509–517.
[7] S. Brakatsoulas, D. Pfoser, and Y. Theodoridis. 2002. *Revisiting R-tree Construction Principles.* Technical Report. Revision.
[8] Yván J. García R., Mario Lopez, and Scott T. Leutenegger. 1998. A greedy algorithm for bulk loading R-trees. In *Proc. GIS*. 163–164.
[9] A. Guttman. 1984. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.* 14, 2 (1984), 47–57.
[10] C. L. Jackins and S. L. Tanimoto. 1980. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing* 14, 3 (1980), 249 – 270.
[11] I. Kamel and C. Faloutsos. 1994. Hilbert R-tree: An improved R-tree using fractals. In *Proc. VLDB*. 500–509.
[12] YC. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman. 2010. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Proc. SSDBM*. 132–150.
[13] Scott T. Leutenegger, Mario Lopez, and J. Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proc. ICDE*. 497–506.
[14] Zhisheng Li, Ken C. K. Lee, Baihua Zheng, Wang-Chien Lee, Dik Lee, and Xufa Wang. 2011. IR-Tree: An Efficient Index for Geographic Document Search. *IEEE TKDE* 23, 4 (2011), 585–599.
[15] Henry Markram et al. 2011. Introducing the Human Brain Project. *Procedia Computer Science* 7 (2011), 39–42.
[16] T. Sellis, N. Roussopoulos, and C. Faloutsos. 1987. The R+-tree: A dynamic index for multi-dimensional objects. In *Proc. VLDB*. 507–518.
[17] D. Šidlauskas, S. Chester, E. Tzirita Zacharatou, and A. Ailamaki. 2018. Improving Spatial Data Processing by Clipping Minimum Bounding Boxes. In *Proc. ICDE*. 425–436.
[18] E. Stefanakis, Y. Theodoridis, T. Sellis, and Y.-C. Lee. 1997. Point Representation of Spatial Objects and Query Window Extension: A new Technique for Spatial Access Methods. *IJGIS* 11, 6 (1997), 529–554.
[19] F. Tauheed, L. Biveinis, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. 2012. Accelerating Range Queries For Brain Simulations. In *Proc. ICDE*. 941–952.
[20] T. Ulrich. 2000. Loose Octrees. In *Game Programming Gems*, Mark DeLoura (Ed.). Charles River Media, 444–453.