QUASII: QUery-Aware Spatial Incremental Index

Mirjana Pavlovic
EPFL
mirjana.pavlovic@epfl.ch

Darius Sidlauskas
EPFL
darius.sidlauskas@epfl.ch

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

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

ABSTRACT

With large-scale simulations of increasingly detailed models and improvement of data acquisition technologies, massive amounts of data are easily and quickly created and collected. Traditional systems require indexes to be built before analytic queries can be executed efficiently. Such an indexing step requires substantial computing resources and introduces a considerable and growing data-to-insight gap where scientists need to wait before they can perform any analysis. Moreover, scientists often only use a small fraction of the data — the parts containing interesting phenomena — and indexing it fully does not always pay off.

In this paper we develop a novel incremental index for the exploration of spatial data. Our approach, QUASII, builds a data-oriented index as a side-effect of query execution. QUASII distributes the cost of indexing across all queries, while building the index structure only for the subset of data queried. It reduces data-to-insight time and curbs the cost of incremental indexing by gradually and partially sorting the data, while producing a data-oriented hierarchical structure at the same time. As our experiments show, QUASII reduces the data-to-insight time by up to a factor of 11.4x, while its performance converges to that of the state-of-the-art static indexes.

1 INTRODUCTION

The advances in data acquisition technologies and supercomputing for large-scale simulations rapidly increase the amounts of spatial data generated and collected. For instance, in the Human Brain Project (HBP) [27], neuroscientists build spatial models of the brain which will ultimately feature 10^{11} neurons [42], each reconstructed with thousands of $3d$ cylinders. NASA released 500 TB of earth observation data generated through remote sensing [30], while the Dutch government released point cloud data with 640 billion points [31] acquired through airborne scanning. Similarly, volunteers generate large amounts of spatial data through services such as OpenStreetMap [33]. Given these massive and growing amounts of spatial data, algorithms to query them efficiently are crucial.

Previous research has proposed many techniques [11, 26, 42] for the fast and scalable querying of spatial datasets. Existing approaches, however, have two major drawbacks. First, they require a time-consuming step to build indexes before they can be used. This pre-processing step significantly delays the analyses: indexing a model in the HBP, for example, can take several hours [42]. With increasing dataset size, the data-to-insight time grows as well. Second, scientists frequently only analyse a small fraction of the data [1, 8]. In the HBP, for example, a scientist builds a model of the brain but after a few queries may determine that it is not biorealistic (e.g., density in certain areas does not agree with measurements) and stops the analysis. Given the small number of queries executed, the overhead of indexing the entire model cannot be fully amortized.

The problems of delayed analysis (due to prior indexing) and the impossibility to amortize indexing cost (due to too few queries) are not exclusive to spatial data management. Database research has proposed incremental indexes for relational data (e.g., cracking [18] and adaptive merging [14]) and for time-series [45]. The core idea is to incrementally index only the parts of the data queried, spreading the cost of indexing over the first few queries. The major data-to-insight bottleneck is thus eliminated, i.e., queries are answered as soon as data is available (albeit the first queries run slower, as no index is initially available).

In this paper, we develop an incremental indexing approach for spatial data in main memory, with the aim of reducing data-to-insight time, as well as achieving performance comparable to traditional spatial indexes (after enough queries are executed). As no current incremental indexing approach for main memory exists, we demonstrate the limitations of applying current options to incrementally index spatial data. As we show, using the concepts for incrementally indexing one-dimensional data [18] to index three-dimensional data does not significantly reduce data-to-insight time, as the major bulk of work still has to be done for the first query. Adapting Space Odyssey [35], an incremental index for exploratory analyses of multiple spatial datasets on disk, to main memory leads to excessive reorganization of the data. As a consequence, a static index (including pre-processing cost) quickly outperforms the proposed incremental solution, in terms of total execution time.

We thus develop a QUery-Aware Spatial Incremental Index - QUASII: a novel data-oriented, query-driven incremental indexing approach. QUASII substantially reduces data-to-insight time and keeps the cost of incremental strategy low, by gradually and partially sorting the spatial objects considering all dimensions. QUASII thus distributes the cost of indexing across all queries, while preserving spatial proximity and producing a data-oriented style partitioning — which typically entails an expensive pre-processing step in the static setting. Finally, being data-oriented, it executes queries efficiently, as it adjusts to the distribution of the data, while avoiding data replication.

Our experiments show that QUASII substantially accelerates the exploratory analysis of spatial data in main memory by reducing the data-to-insight time by up 11.4×, while achieving the query performance of current algorithms for spatial indexing. Static algorithms are not able to amortize their building cost over QUASII even after 10000 queries.

To our knowledge we are the first to develop and analyze incremental indexing for spatial data. Our contributions are:

- We demonstrate the challenges of adapting and using known incremental indexing [18, 35] to spatial data in main memory. We use the resulting approaches as motivation and baseline.
- We develop *QUASII*, an incremental approach that significantly reduces the data-to-insight time, while achieving the query performance of state-of-the-art spatial indexes.
- We experimentally analyse QUASII's performance and the number of queries it needs to reach the performance of its static counterparts.

The remainder of the paper is structured as follows. We define the problem in Section 2 and motivate it in Section 3. We then describe QUASII in Sections 4 and 5 and experimentally evaluate it in Section 6. Section 7 gives an overview of related work before we conclude in Section 8.

2 PROBLEM DEFINITION

Our work is driven by the need for the exploratory analysis of spatial datasets through querying. The queries executed are ad hoc, i.e., the next query is only known after the results of the first query are analyzed, and they thus cannot be batched and executed with only one sequential read of the dataset.

Example Application. In the Human Brain Project, neuroscientists build spatial models of the brain [27]. Already now the models are so detailed that to simulate a neocortical volume of only $0.29\ mm^3$ supercomputers are needed [28].

Once the part of a model is built, neuroscientists need to validate it by choosing a subset of its regions at random and inspecting them. Each region is queried with several spatially close queries and the query results are used to verify the composition, density and other metrics agree with the real brain. The results of these analyses are crucial to determine whether or not the model can be simulated or should be abandoned (subsequently building a new one using a different configuration). Scientists currently only have two fundamentally different options: index all data a priori and execute queries with the index or scan all data each time to answer a query. Not knowing a priori how many queries will be executed (and if indexing can be amortized) makes it difficult to decide.

Data. We consider spatially extended (volumetric) objects enclosed by a minimum bounding box (MBB). In a three-dimensional ($3d$) setting, each MBB b is defined by two $3d$ points $lower(b)$ and $upper(b)$ corresponding to lower and upper coordinate at each dimension ($lower(b) = (x_l, y_l, z_l)$ and $upper(b) = (x_u, y_u, z_u)$) [11].

Queries. We focus on range (window) queries as they are broadly used in many applications and are also the building block for many other spatial queries (e.g., k-nearest neighbor queries [22]). Each query is a $3d$ box specified by two $3d$ points, e.g., (q_l, q_u). Given a query q, all objects with their bounding box b intersecting with q, i.e., where $b \cap q \neq \emptyset$, are in the result.

Setting. We assume that all data and necessary index structures fit in main memory. We consider a static setting, i.e., all raw data is available before querying.

3 MOTIVATION

No current incremental indexing approach can index spatial data in main memory. Research has developed incremental indexing for relational, one-dimensional data in main memory, i.e., cracking [18] and for spatial data on disk [35]. In the following we extend the former [18] to the spatial domain and adapt the latter [35] to use in main memory — to demonstrate the limitations of these ideas in reducing the data-to-insight time and to motivate the need for a new approach.

3.1 Cracking for Spatial Data

Relational Cracking. Database cracking [16, 18, 19] incrementally builds an index as a byproduct of query execution in the context of main memory column-stores. The proposed techniques partially sort elements based on the query execution, essentially performing an incremental quick sort. In its simplest form, cracking [18] rearranges elements in an array according to the end points of the query range (q_l, q_u): all values $< q_l$ are moved towards the beginning of the array, while values $> q_u$ are moved towards the end. With each query, the index becomes more refined until it is fully sorted and indexed.

SFCracker. Using this strategy to index spatial data is inherently challenging: spatial data has multiple dimensions and, unlike $1d$ data, no total order can be directly imposed on it. Therefore, to be able to use the strategy of cracking we transform data from the multi- to the one-dimensional domain. We perform this transformation using a space-filling curve (SFC) — a common approach to impose a total, $1d$ order on spatial objects.

A SFC maps data to $1d$ domain by visiting all the points in a d-dimensional grid exactly once; the order in which the objects are visited defines their order in $1d$ space. When mapping spatial data, it is crucial to consider SFCs that preserve proximity (such as Z-order [34] or the Hilbert curve [21]), so that data points close in multi-dimensional space remain close in $1d$ space [10, 29].

The resulting approach, *SFCracker*, incrementally sorts SFC codes based on the queried region. Both, data and queries are transformed to $1d$ space. The data transformation takes place in the first query, which makes it the most expensive one. Once the data is transformed, the queries perform cracking based on the $1d$ intervals obtained through the query transformation.

A naive query transformation to $1d$ space results in a substantial number of false positives (needed to be tested for intersection) because the transformed $1d$ range can be significantly larger than the original multi-dimensional range if only the *lower* and *upper* coordinates of the range query are considered. An example is shown in Figure 1: the curve segments in blue belong to the transformed range ($SFCcode_l$, $SFCcode_u$), but they are outside of the original query range (in red). To reduce the overhead of false positives, we use a technique that partitions the curve into multiple sub-intervals each of which is fully contained in the original range [43]. Consequently, a range query is transformed into a number of intervals and the data is thus cracked multiple times per query, once for every interval.

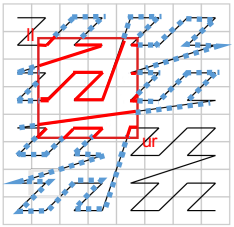


Figure 1: 1d transformation: overhead.

Limitations. Cracking in the relational domain decreases data-to-insight time, distributing the cost of sorting over all queries with fairly low overhead and initialization cost. These benefits, however, decrease for datasets with a higher number of dimensions. First, the initial query is expensive as it maps all the objects from the multi- to the one-dimensional domain. Second, as opposed to relational data, a single query has to perform multiple expensive cracks to avoid performance penalties introduced with the transformation to 1D space. Consequently, spatial cracking still has a considerable data-to-insight time, along with an

expensive incremental strategy. We demonstrate these limitations experimentally in Section 6.3.

3.2 Disk-based Incremental Indexing in Main Memory

Disk-based Incremental Indexing. Space Odyssey [35] is a recently proposed incremental index for the exploration of spatial data. However, it tackles a different problem: Space Odyssey is designed for exploratory analyses of multiple spatial datasets. Without prior information, it incrementally indexes the datasets and adapts the physical layout of the data on disk for datasets frequently queried together. Although Space Odyssey addresses a different problem, we use its ideas related to incremental indexing and adapt them for use in main memory in *Mosaic*.

Mosaic. Mosaic incrementally builds an Octree [20] by dividing the space uniformly into eight partitions. Figure 2 depicts the indexing process (in 2*d* for clarity). For every query, Mosaic identifies the partitions that overlap with the query, splits them into eight partitions and reassigns their objects to the newly created partitions. Frequently queried areas in a dataset are indexed fully, whereas less frequently queried areas are coarser grained. The top-down strategy is thus beneficial for consecutive queries, as they can reuse the previous partitioning, independent of the workload pattern. However, data in frequently queried areas is re-partitioned multiple times.

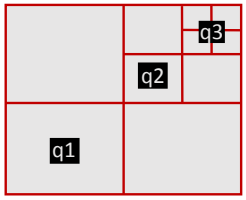


Figure 2: Mosaic: incremental strategy.

Limitations. Mosaic introduces significant overhead as the data in frequently queried areas is re-partitioned multiple times until it reaches its final configuration. Consequently, a static approach based on space-oriented partitioning, such as the uniform grid, outperforms quickly Mosaic in terms of total execution time (we provide more details in Section 6.3).

Mosaic additionally suffers from considering more objects than strictly necessary — a problem inherent in space-oriented partitioning and related to data assignment. For indexes based on space-oriented partitioning, objects can be assigned to cells with two strategies: replication and query extension. Replication assigns an object to all partitions that it overlaps with. As a consequence more objects need to be considered for intersection, the memory footprint increases and an expensive de-duplication step is needed. The alternative is to use query extension [40] which assigns an object to a cell based only on its center. This technique avoids object replication, however, it can considerably increase the number of objects necessary to be tested for intersection. More precisely, to ensure the correctness of the query result, it extends the query range by the maximum object extent. As a result, the area queried for is bigger than the initial query. Both strategies, replication and query extension, slow down query execution but, as we show in Section 6.2, replication is particularly expensive when working with volumetric spatial objects and we thus use query extension in Mosaic.

4 QUASII OVERVIEW

As discussed, an approach to incrementally index spatial data is not as straightforward as adapting known approaches. Besides the challenges, we also identify important design goals:

(i) ***minimal data-to-insight time:*** the main requirement for incremental indexing is to enable instant access to the data, i.e., the first queries must not introduce undue overhead/processing;
(ii) ***efficient query performance:*** the performance of frequently queried subsets of data should converge to that of the fully indexed approach (or better);
(iii) ***low cost incremental indexing:*** indexing should introduce as little overhead as possible, i.e., its cumulative execution time should only exceed the one of static indexes after as many queries as possible (or not at all).

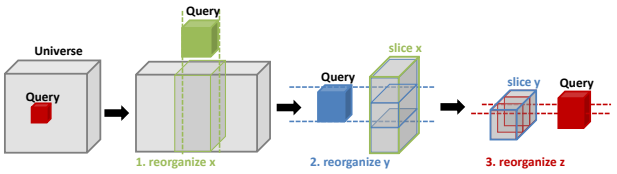
Given the design goals and our analyses, we develop QUery-Aware Spatial, Incremental Index, QUASII. QUASII is a data-oriented index, incrementally built as a side effect of query execution. It reduces data-to-insight time and curbs the cost of incremental indexing by gradually and partially sorting the data, while simultaneously producing a data-oriented hierarchical structure. It is based on a *nested reorganization strategy* which incrementally slices the space in each dimension and a *hierarchical, data-oriented structure* designed to accommodate the incremental indexing process and provide efficient query execution.

Overview. Figure 3a illustrates QUASII's incremental strategy on a high level. Given range queries of the form $q = [q_l = (x_l, y_l, z_l), q_u = (x_u, y_u, z_u)]$, QUASII reorganizes the objects based on each query's lower (q_l) and upper (q_u) coordinate by slicing each dimension and performing a nested reorganization. It first reorganizes objects on the x dimension, producing three x slices where the middle one contains the objects in the range $[x_l, x_u]$ given the query range in dimension x. Subsequently, it continues reorganizing the middle x slice on the y dimension, producing again three slices where the middle one contains objects in the range $[y_l, y_u]$. Finally, QUASII reorganizes the y slice on the z dimension producing the z slice which contains the query result. QUASII never performs a complete sort but reorganizes data locally, given the query's boundaries.

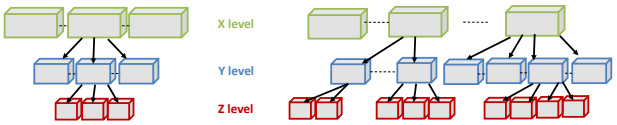
The slices produced are organized in a hierarchical structure that incrementally forms the index. Figure 3b illustrates the structure of QUASII after the very first query (left) and after an arbitrary number of queries (right) are executed. QUASII forms a hierarchical structure with one level per dimension, i.e., the first (top), second, and third (bottom) levels correspond to slices at x, y, and z dimensions, respectively. The top level has the coarsest granularity as its objects are constrained with one dimension, while the bottom level is the most fine-grained since it is constrained by all dimensions. When executing the queries, QUASII traverses the structure depth-first, performing additional refinements when necessary, as we discuss later in Algorithm 1.

Nested Reorganization Strategy. The incremental strategy of QUASII is query-driven and data-oriented. Being query-driven, it reorganizes the minimal amount of data while executing queries. At the same time, being data-oriented, it achieves query efficiency as it adjusts to the data distribution, while avoiding replication. QUASII accomplishes both through its nested reorganization.

Data-oriented partitioning typically entails an expensive pre-processing step in the static setting as it preserves spatial proximity based on a strategy for ordering multi-dimensional objects. QUASII distributes the cost of this pre-processing across all queries by performing nested and partial reorganization. It reorganizes only a subset of data driven by queries, gradually curbing the amount of data partially sorted with every dimension. This strategy is inspired by the Sort-Tile-Recursive (STR)



(a) Incremental indexing strategy



(b) Index structure after the first (left) and few more (right) queries

Figure 3: QUASII incremental indexing strategy and data structure.

R-tree bulkloading algorithm [26]. STR produces tiles that form leaf-level nodes for the R-Tree by recursively, fully sorting spatial objects in each dimension. More precisely, STR for $3d$ objects first sorts the spatial objects on the x-axis and partitions them in vertical tiles of equal size (i.e., the same number of objects). Then, within each x tile, it recursively applies the same strategy first considering y and then z dimension. This tiling strategy is particularly efficient as the resulting R-Tree has less overlap than other approaches [26]. By only performing partial reorganizations for the parts of the data that is actually queried, QUASII outputs partitions targeting these characteristics at lower cost (as opposed to complete sorts in STR).

Index Structure. QUASII's index structure is designed to support an efficient incremental strategy with as little performance penalty as possible. Its hierarchical structure is designed to accommodate the reorganization strategy: each level corresponds to one (reorganization) dimension and each parent node is represented by its children in a nested form along the dimensions QUASII reorganizes data. We discuss the data structure and how it accommodates incremental indexing in more detail in Section 5.1.

Benefits. Ultimately, the design choices behind our approach enable us to achieve the goals we outlined. To reduce data-to-insight time (i), QUASII keeps data in the multi-dimensional, spatial domain. This avoids transforming all data at the very beginning which significantly hurts performance of the first query. Next, to achieve query efficiency (ii), QUASII uses data-oriented partitioning that preserves spatial proximity, adjusts to the distribution of data, and avoids object replication. Finally, to keep the cost of the incremental indexing low (iii), QUASII gradually and partially sorts the data using a nested reorganization strategy.

5 DATA STRUCTURE & QUERY PROCESSING

In the following, we explain the QUASII index structure and data organization before we proceed with discussing querying and incremental indexing algorithms.

Throughout this section, we refer to a $2d$ example given in Figure 4. It depicts a dataset $D = \{o_0, \ldots, o_9\}$ of ten (gray) rectangular spatial objects. All subfigures have three main parts: the top part shows a $2d$ view of the dataset D and how the space is conceptually "sliced" by QUASII, the middle ("Data array") depicts how (raw) data objects are re-organized in main memory, and the bottom shows QUASII's hierarchical data structure that is incrementally built. All x- and y-axis related slicing is marked in green and blue, respectively. Figure 4a) shows the initial state: the "slice-less" view of the data space with D objects and the very first query q_1, the data array of spatial objects in an arbitrary initial order, and the data structure containing the initial slice, s_0 (capturing the entire dataset).

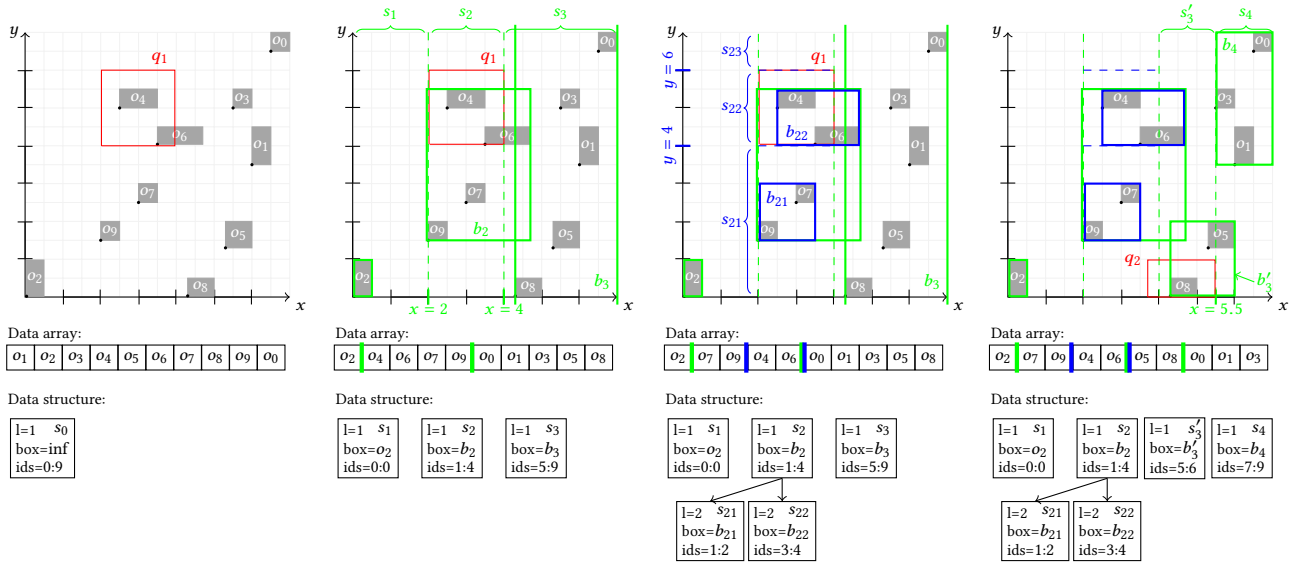
5.1 Data Structure

QUASII forms a d-level hierarchical structure, organized according to the number d of dimensions. Each level l has a one-to-one mapping to the corresponding dimension. That is, the first level ($l = 1$) represents slicing of data at x, the second level ($l = 2$) slices at y, and the third level ($l = 3$) slices at the z dimension. The top level always slices data objects at the coarsest granularity, while the bottom level is the most fine-grained. Each slice is described with four attributes: (i) its level, (ii) a minimum bounding box capturing all its objects, (iii) indices to the data array corresponding to the first and last entry of the objects that belong to the slice, and (iv) pointers to sub-slices refining the slice further on the subsequent dimension. In Figure 4, this corresponds to the four fields present in each node of the data structure (next to slice label, e.g., s_0): l, box, ids, and arrow pointers (when not null). In our two-dimensional view of the dataset, we mark boxes with a solid line (in the corresponding color), while the slice cuts are marked as dashed lines.

Data-oriented Slicing. One of the main advantages of data-oriented partitioning is that each spatial object is always assigned to just one partition (slice). However, QUASII determines the slices in each dimension based on query ranges. Given volumetric spatial objects, objects can be sliced through and thus overlap with multiple slices. To overcome this problem, QUASII represents each object using only one of its coordinates and uses this coordinate to identify a slice where an object will be assigned to. In particular, during indexing, it uses each object's lower coordinate (x_l, y_l, z_l). Being part of object's MBB, this does not require any additional computation or storage[1]. In Figure 4, this coordinate is marked as a black dot for all objects. Figure 4b illustrates slicing based on the very first query q_1 and its range $[2, 4]$ on the x-axis. Slicing at $x = 2$ and $x = 4$ results in three x-slices (s_1, s_2, and s_3). While object o_6 overlaps two slices (s_2 and s_3), it is assigned to s_2 based on its lower coordinate (x_l). Note how the objects are re-organized in the data array and correspond to three partitions (slices) with coordinates $x < 2$, $2 \leq x \leq 4$, and $4 < x$. Accordingly, the data structure is updated with three new (more refined) slices replacing the initial (coarser) slice s_0 (capturing the whole dataset).

While QUASII assigns objects to slices based on their single (lower) coordinate, it records a minimum bounding box for each slice taking into account the actual spatial extent of the objects and thus ensures the correctness of the query result. This also results in slice representations (their MBBs) that are often much smaller but not necessarily within the originally sliced bounds. For example, s_1 contains only one object and thus has a very small MBB (i.e., its $box = o_2$), while the MBB of s_2 is b_2 and exceeds the original cut at $x = 4$ (Figure 4b). As we show later, this enables QUASII to discard many unnecessary slices during

[1] The upper coordinate (x_u, y_u, z_u) or the object's center (requires to be computed, though) can equally be used.



(a) Spatial data and query q_1 **(b) After slicing based on x of q_1** **(c) After slicing based on y of q_1** **(d) After processing q_2**

Figure 4: An example of query processing and incremental indexing in QUASII (configured with $\tau_x = 4$ and $\tau_y = 2$), given ten spatial objects (o_0–o_9) and two range queries (q_1 and q_2).

query execution. To limit unnecessary computation (as a slice can be reorganized multiple times until it is fully refined), QUASII computes a full MBB only when a slice is completely refined. Otherwise, a slice is represented with an open-ended MBB, i.e., the MBB has bounds only on the dimension it has been sliced on.

Configuration. QUASII has only one configuration parameter, a size threshold τ, that determines the maximum number of objects in a slice at the finest level. That is, at the bottom level, whenever a slice s contains less or τ number of objects (i.e, $|s| \leq \tau$), it is considered to be fully refined. Intuitively, this is similar to setting a (leaf) node size in the R-Tree.

The sizes of the remaining $d-1$ levels are calculated as follows. Since QUASII performs data-oriented slicing, the total number of partitions required to satisfy threshold τ is $\lceil n/\tau \rceil$, where n is the total number of objects (i.e., $n = |D|$). Consequently, the number of times QUASII has to slice the data space across each dimension to produce $\lceil n/\tau \rceil$ partitions is equal to:

$$r = \left\lceil \sqrt[d]{n/\tau} \right\rceil \tag{1}$$

If we use τ_d to denote the slice threshold at the bottom level $l = d$ (i.e., $\tau_d = \tau$), then the maximum number of objects per slice for the remaining levels (up to the top) can be expressed recursively as $\tau_{d-1} = r \times \tau_d$. Note that r corresponds to the number of sub-slices (within a slice) at each index level.

Turning to our 2d example[2], after x-based slicing in Figure 4b, s_1 contains one object and thus is considered fully refined (i.e., $|s_1| = 1 \leq \tau_x$), while s_3 has five objects and may be refined in the future. Also note that s_3 stores an open-ended MBB ($s_3.box = b_3$).

The number of levels in QUASII is fixed and always equals to the dimensionality of the queried dataset. That is, it does not depend on the size of the dataset. Therefore, to accommodate the index growth (the index grows in breadth) and enable efficient query execution, QUASII keeps the children (within a slice) organized/sorted according to the level's dimension. QUASII uses this order and the minimum bounding boxes (box) of each node to prune the amount of objects necessary to be tested during the query execution.

[2]To minimize the required number of objects in Figure 4, we fix $\tau_x = 4$ and $\tau_y = 2$.

5.2 Query Processing and Index Refinement

Having defined QUASII's data structure, we discuss how it is incrementally built and maintained as a side effect of each query.

Query Processing. Algorithm 1 shows the pseudo-code for query processing. Each query traverses the d-level structure depth-first, starting from the first level (having x-slices). Because the slices are sorted, QUASII performs a binary search (Line 3) to find the starting slice. It then scans all the slices $S[i]$ within the query range on the current dimension (i.e., while the loop conditions in Line 4 hold). The loop conditions guarantee that each slice $S[i]$ intersects q only in the current dimension. To discard potential false positive slices early, Line 5 checks if its actual boundaries ($S[i].box$) also intersect with the query range.

Next, QUASII potentially refines $S[i]$ (Line 6), which may be further sliced into multiple more fine-grained slices S'' if it is larger than the maximum size threshold τ (discussed in the next algorithm). In Lines 7—16, QUASII traverses (potentially refined) slices S''. For each $s \in S''$, it either checks all s objects for intersection in case of the bottom level or recursively proceeds querying its children based on the next level/dimension (a default child is assigned to a not fully refined slice, Line 15). Finally, all the newly created slices are accumulated in S' (Line 17), appended to S (Line 19), and re-sorted (Line 20). The slices are sorted based on their ids, i.e., the position (index) of the first slice's object in the data array.

Index Refinement. With each query, QUASII attempts to refine all query intersecting slices (i.e., Line 6 in Algorithm 1). Algorithm 2 provides the simplified pseudo-code for this refinement process. Note that the processing within Algorithm 2 is always based only on the current dimension/level of slice s ($s.l$).

The input slice s is considered for slicing only if it exceeds the threshold τ. Given s is coarse enough, QUASII proceeds with determining the type of slicing based on the intersection between

Algorithm 1: query(query q, data D, slices S, result R)

```
1:  S' ← ∅                          // to store newly created (refined) slices
2:  dim ← S[0].l                    // current level/dimension of slices in S
3:  i ← binarySearch( S, lower(q[dim]) )
4:  while i < |S| and lower(S[i].box[dim]) ≤ upper(q[dim])
    do
5:      if q ∩ S[i].box = ∅ then continue
6:      S'' ← refine( S[i], D, q )      // as per Algorithm 2
7:      for each slice s ∈ S'' do
8:          if q ∩ s.box ≠ ∅ then
9:              if s.l is the bottom level then
10:                 for each j ∈ {s.ids} do
11:                     if D[j] ∩ q ≠ ∅ then
12:                         R ← R ∪ D[j]
13:             else
14:                 if |s.children| = 0 then
15:                     createDefaultChild(s)
16:                 query(q, D, s.children, R)
17:     S' ← S' ∪ S''
18:     i ← i + 1
19: S ← S ∪ S'
20: sort(S)
```

query q and slice s. It considers three types of slicing. If both q's lower and upper coordinates are within s, a three-way slicing is performed splitting s into three sub-slices (Line 5). If only one of q's coordinates is within s, a two-way slicing is performed splitting s into two sub-slices (Line 6). Finally, if q contains s (i.e., both q's coordinates are outside of s's bounds), QUASII performs a two-way slicing based on an artificially introduced coordinate.

QUASII iterates through the generated slices and for the ones that still exceed τ (and overlap with the query) it applies additional refinement according to artificially introduced boundaries in Line 10 (it repeats the process recursively until a slice is fully refined in the corresponding dimension). The three- and two-way slicing algorithms (Line 5 and Line 6) reorganize the data (D) following the incremental quick sort strategy introduced in database cracking [18]. In the reorganization process, QUASII also records the information about the boundaries (box) of newly created or modified slices.

Example. Continuing with our example in Figure 4, after refining s_0 into three x sub-slices in Line 6 of Algorithm 1 (and resulting in Figure 4b), QUASII recursively continues with the intersecting (and just refined) slice s_2 based on the y dimension (Figure 4c). As such, s_2 is further refined based on the queried y range and results in three new slices (s_{21}, s_{22}, and s_{23}). In this step, only the objects within the s_2 range ($ids = [1..4]$) are three-way sliced and re-organized in the data array. The two new slices (s_{23} is empty) are appended to the data structure as children of s_2. They are fully refined (as $|s_{21}| \leq 2$ and $|s_{22}| \leq 2$) and have much smaller MBBs (b_{21} and b_{22}, respectively) than the initial slice cuts. Finally, because it is the bottom level, the objects within s_{22} are checked against the query range and the two qualifying objects $\{o_4, o_6\}$ are added to the result set (R).

The subsequent query q_2 benefits greatly from previous slicing, as illustrated in Figure 4d. For example, x-slices s_1 and s_2 are skipped completely because query q_2 does not intersect with their MBBs (i.e, test on Line 5 in Algorithm 1). Therefore, QUASII proceeds with the only intersecting slice s_3, which is not fully refined and requires further slicing. As per Algorithm 2, this time a two-way slicing is performed (at $x = 5.5$) resulting in two finer

Algorithm 2: refine(slice s, data D, query q) $\longrightarrow$ slices S

```
1:  if |s| ≤ τ[s.l] then
      return {s}
2:  S ← ∅                                   // to store refined slices
3:  t ← determineSliceType(s, q)
4:  switch ( t )
5:      case both: S' ← sliceThreeWay(s, q, D)
6:      case one: S' ← sliceTwoWay(s, q, D)
7:      default: S' ← sliceArtificial(s, q, D)
8:  for each slice s ∈ S' do
9:      if |s| > τ[s.l] and q[s.l] ∩ s.box[s.l] ≠ ∅ then
10:         S'' ← sliceArtificial(s, q, D)
11:         S ← S ∪ S''
12:     else
13:         S ← S ∪ s
14: return S
```

slices (s'_3 and s_4) replacing the previous slice s_3. Next, QUASII continues with y-based slicing of the fully q_2-contained slice s'_3. Since s'_3 reaches the size threshold τ_y, it is not refined further. Finally, the actual data array objects within s'_3 range ($ids = [5..6]$) are checked for intersection with q_2 and the qualifying o_8 is added to the result set.

Artificial Refinement. To produce a balanced hierarchical structure QUASII has to conform with the defined thresholds when forming the slices and using only query boundaries does not meet these requirements. One query is usually not sufficient and we cannot use the subsequent queries for this purpose, as they may interfere with the existing order of the slices. For instance, reorganizing a slice again (that has been organized according to all dimensions) based on the x dimension, may disrupt the previously established partitioning for y and z dimensions.

To address this problem, QUASII reorganizes a slice s (Lines 7 and 10 in Algorithm 2) until it meets a size threshold τ in the corresponding dimension. It achieves this by forcing a two-way slicing based on artificially introduced coordinate and thus splitting the slice into two sub-slices. Given the range (x_l, x_u), the new coordinate is $c = \lfloor (x_l + x_u)/2 \rfloor$. The two new slices are recursively sliced further until the threshold τ is satisfied.

While more advanced approaches, e.g., based on the concepts from R*-Tree node splitting algorithms [6], would minimize overlap in data structure, they would also significantly increase the cost of incremental strategy. Therefore, QUASII employs the above uniform and low-cost artificial slicing strategy to meet τ thresholds at each of d levels.

Query & Refine. The outcome of QUASII's reorganization strategy are the slices that are within the query range and consequently only the objects in these slices are checked for intersection. However, performing the reorganization following strictly the query's boundaries would produce an incomplete result, as illustrated in Figure 5. For instance, the object o overlaps with the query range q, however, its lower coordinate is outside the query's boundaries and consequently o would not be identified as a part of the result.

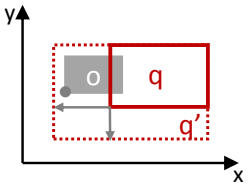


Figure 5: Refinement step: query extension.

To ensure correct query execution while preforming refinement, QUASII employs the query extension technique [40]. More

precisely, it extends the query for maximum object extent in each dimension, considering lower coordinate. This extension is done only when performing refinement and only within not fully refined slice. Consequently, the query that performs refinement potentially considers more objects for intersection as its range is enlarged. However, this introduces a minimal overhead as the only alternative is the expensive scan of the entire unrefined slice. We apply the same logic for the binary search where, to avoid missing any slices due to the overlap within them, we extend the query range (while performing binary search) for the maximum slice extent in the corresponding dimension.

6 EXPERIMENTAL EVALUATION

In this section, we first describe the experimental setup & methodology and then present a thorough experimental analyses that illustrates the benefits of our incremental approach, both on a real-world neuroscience and synthetic datasets. We start the analyses by outlining the shortcomings of the approaches based on space-oriented partitioning in Section 6.2. We then study the incremental approaches by comparing them with their static counterparts in Section 6.3 and cross-evaluating their performance in Section 6.4. Finally, Section 6.5 describes the sensitivity analyses of QUASII.

6.1 Experimental Setup & Methodology

Hardware. We run our experiments on a Red Hat Enterprise Linux Server release 7.3 machine equipped with 2 Intel Xeon CPU E5-2650L processors at 1.80GHz and 768GB of RAM. Each processor has 12 cores (24 hardware threads) with private L1 (32KB) and L2 (256KB) caches and 30MB of shared L3 cache.

Implementations. All indexing techniques are implemented in C++ and compiled with `g++ 4.9.3` with the maximum optimization level. The list below summarizes the implementations that we study:

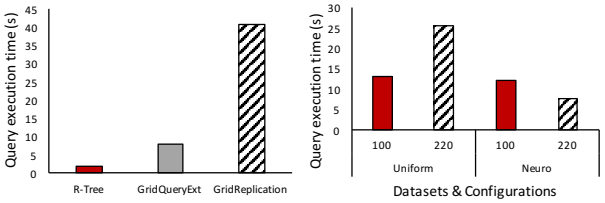
Scan: performs a full data scan to answer each query.

SFCracker: is our incremental variant of database cracking [18] for spatial data, described in Section 3.1. We use the Z-order as a SFC order. The average farthest distance of neighbours in the Z-order is (slightly) higher than in the Hilbert order [10] (i.e., it has better locality), however, we opt to use the Z-order due to its simplicity and the huge body of work on its efficient range query algorithms [5, 39, 43, 44]. We use 32-bit to represent *zcodes* (i.e., 10 bits per dimension) as a trade-off between memory resources and precision (the number of false positives to be filtered).

SFC: is a static counterpart of SFCracker. In the pre-processing phase, SFC transforms data from multi- to one-dimensional domain and sorts it according to the produced SFCcodes. During querying, a ($3d$) query range is also converted to a $1d$ range and a binary search is used to locate the objects in the $1d$ interval. We employ the same representation of *zcodes* and query optimization as in SFCracker (described in Section 3.1).

QUASII: is our incremental approach discussed in Section 4. We use 60 objects as a node capacity τ_z.

R-Tree: According to our setting, all data is available before querying. Therefore, we use a bulk-loading approach to build the R-Tree index as it reduces overlap and decreases pre-processing time compared to the R-Tree built by inserting one object at a time [26]. For this purpose, we use an efficient STR [26] bulk-loading strategy that balances well the overhead of partitioning the data and query performance. It outperforms Hilbert R-Tree [23] in terms of query performance [26], while its pre-processing cost is not significantly higher [42]. Similarly, TGS [12] and PR-Tree [4] can outperform STR on datasets with extreme skew and aspect ratio, however, they incur considerable overhead for data partitioning. We use the same configuration for node capacity (60) as in QUASII.



(a) Query execution **(b) Configuration**

Figure 6: The impact of space-oriented partitioning.

Mosaic: corresponds to the space-oriented incremental approach described in Section 3.2.

Grid: is a uniform grid-based index used as a static counterpart of Mosaic. We use query extension [40] technique (as discussed in Section 3.2) to assign an object to a grid cell. We use two configurations with 100 and 220 partitions per dimension for synthetic and neuroscience datasets, respectively. Both configurations are obtained through a parameter sweep.

Dataset and Queries. We use real-world neuroscience and synthetic datasets.

Neuroscience: we use a small part of the rat brain model represented with 450 million cylinders as elements in a volume of 285 μm^3. We approximate the cylinders with MBBs, resulting in the total number of 450 million MBBs with a size of 21GB on disk. Based on the previously described use cases, we synthetically generate queries, each having a fixed volume *qvol* of $10^{-2}\%$ of the queried brain volume and a clustered distribution. We generate 5 query clusters each with 100 queries, where query centers are distributed around the cluster centers following a Gaussian distribution ($\mu = 0$, $\sigma = qvol$).

Synthetic: we create synthetic datasets by distributing spatial boxes in a space of 10 000 units in each dimension of the $3d$ space. The length of each side of each box is determined uniform randomly between 1 and 10 for 99% of the objects, while 1% of the objects has a side ranging from 10 - 1000 units. The spatial elements are distributed according to a uniform distribution. The datasets have 500 million and 1 billion elements (size on disk 22.5GB and 45GB). For completeness and to test non-skewed cases, we generate *uniform* workload. The uniform workload contains up to 10 000 uniformly distributed queries. To have range queries of different selectivity, we vary *qvol*: $10^{-3}\%$, $10^{-1}\%$, 1%, and 10% of the universe.

6.2 Space-oriented Partitioning Challenges

Both, Mosaic and SFCracker (introduced in Section 3), use space-oriented partitioning at their core — Mosaic partitions space, while SFCracker assigns the SFCcodes using a uniform grid. Before we start the analysis of incremental approaches we experimentally demonstrate the shortcoming of space-oriented partitioning — the overhead introduced with data assignment strategy — since it also affects incremental solutions. Further on, we illustrate why a static approach based on space-oriented partitioning, such as a uniform grid, is not a suitable replacement for an incremental index despite having a comparatively cheap pre-processing step (once properly configured).

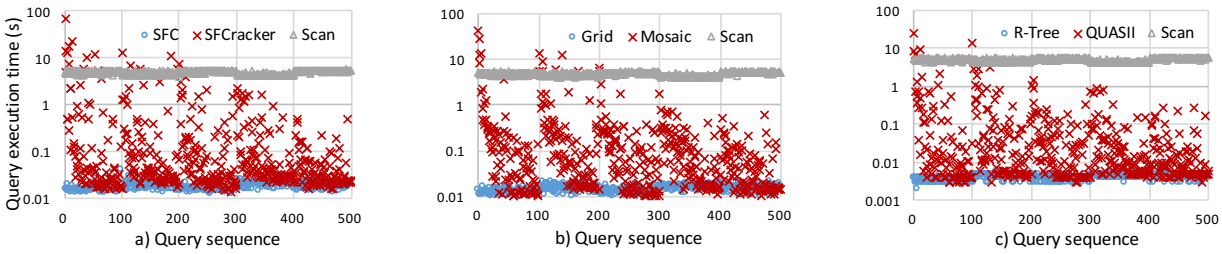


Figure 7: Convergence of a) one-dimensional, b) space-oriented c) data-oriented based approaches.

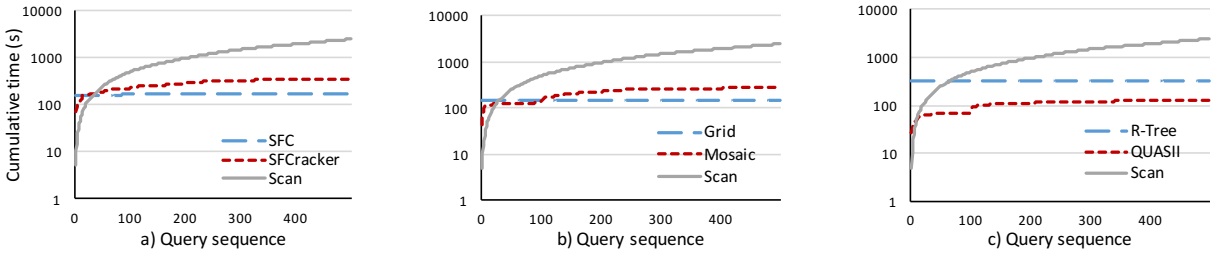


Figure 8: Cumulative time of a) one-dimensional, b) space-oriented c) data-oriented based approaches.

Data Assignment. In the first experiment we illustrate the impact of data assignment strategies by comparing the performance of Grid and R-Tree. We use two variants of the Grid approach: GridQueryExt avoids the objects replication by using the query extension technique — it assigns an object to the grid partition based on its center, while GridReplication replicates the objects — it assigns an object to all the overlapping partitions.

Figure 6a) shows the results of the experiments where we execute 500 clustered queries of selectivity 0.01% on the neuroscience dataset. GridReplication is heavily affected by object replication which increases the number of objects necessary to be checked for intersection and introduces an expensive de-duplication step (needed due to objects replication). GridQueryExt achieves better performance, however, it still considers 3.1× more objects for intersection than the R-Tree as it extends the initial query for the maximum object extent. The R-Tree clearly outperforms both GridReplication and GridQueryExt with a speedup of 19.4× and 3.7× respectively.

Configuration. In the second experiment we demonstrate the difficulty to configure the grid-based approaches. We use two datasets with identical extent and number of elements but different data distributions: Uniform (uniform distribution, synthetic dataset) and Neuro (skewed distribution, the neuroscience dataset). We use the same experimental setup as for the previous experiment. The best configuration (number of partitions per dimension) is 100 for Uniform and 220 for the Neuro dataset and is determined in a parameter sweep. We measure the execution time when using both configurations for each dataset and illustrate the results in Figure 6b).

Although both datasets have the same number of elements and extent, the best configuration significantly depends on the data distribution — the neuroscience dataset requires more partitions compared to the Uniform dataset since it has the very dense regions that require fine grained partitioning. Furthermore, the grid configuration significantly affects performance — the grid performance on the Uniform dataset deteriorates notably when using the Neuro dataset configuration and vice versa.

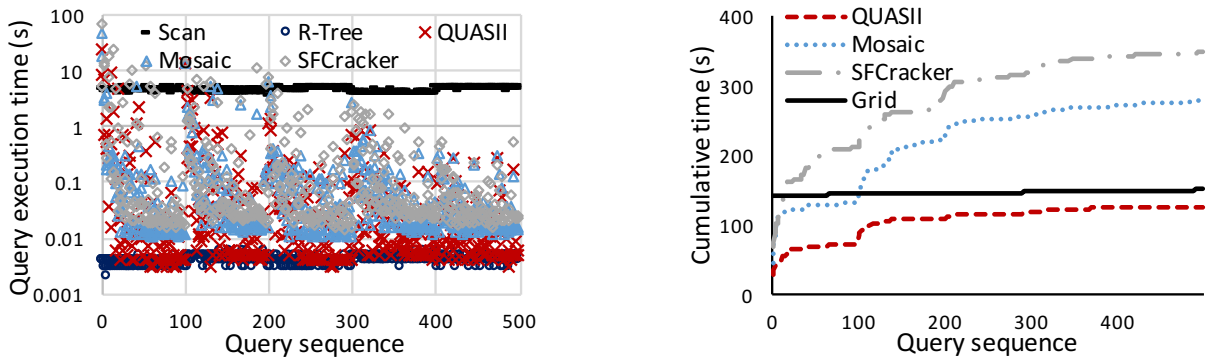
Summary. Space-oriented partitioning introduces performance penalties. Depending of data assignment strategy, we either consider more elements or suffer from replication. Additionally, the grid configuration is non-trivial and using the wrong one has a detrimental impact on the execution time. In practice we have to use a parameter sweep to find the configuration for a given workload. Consequently, grid configuration turns into a time-consuming process, increasing data-to-insight time.

6.3 Incremental versus Static

We first analyze the incremental approaches by comparing their performance with the performance of their static counterparts (introduced in Section 6.1). Each static approach has similar properties as its incremental counterpart, however, it involves necessary pre-processing. We categorize the approaches according to these properties as a) one-dimensional, b) space-oriented and c) data-oriented approaches. For each category we present the performance of the incremental approach, its static counterpart and Scan. We first evaluate if and when the approaches converge to the performance of their static counterparts and then analyze the overhead of the incremental strategy. For this purpose we execute the clustered query workload with 500 queries of selectivity 0.01% on the *neuroscience* dataset.

Convergence. In the first experiment we evaluate the convergence of the incremental approaches — how fast an approach converges to the execution time of a fully indexed dataset. Figure 7 measures the execution time of each query for all approaches.

The results show five peaks in execution time, one for each query cluster. The execution of the first cluster of queries (and the associated processing of the data) takes the longest as no index structure exists at the beginning. The first queries therefore exceed the cost of Scan, because at this point, the entire dataset has to be scanned along with building partial index structures. Subsequent queries within a cluster use a partial index and thus execute in less time than a full scan, but take longer than queries on the static approach. This process continues as queries in the



(a) Convergence **(b) Cumulative time**

Figure 9: Comparative analysis of incremental approaches.

same cluster further refine the index. Queries in one cluster not only refine the index locally but also carry out limited, global refinement. The queries in a subsequent cluster thus benefit from previous clusters and execute faster. As the index converges to its full structure, the query execution time approaches that of the to static approach.

Cumulative Response Time. While in the previous set of experiments we measure the individual query performance, in this analysis we measure the cumulative execution time (including index building step for the static approaches). Figure 8 illustrates the experimental results.

Similar to the convergence experiment, the query clusters are visible: the cumulative response time jumps each time the experiment moves to a new cluster. The most expensive is the transition from the first to the second cluster while subsequent transitions become less evident as the index becomes more refined.

The cumulative cost of SFCracker is comparatively high and, crucially, with a very expensive first query. One reason is that the first query takes 12.9% of the total pre-processing by assigning the objects to the grid cells and calculating the *zcode* values for the entire dataset. Adding to this the cost of cracking, the total execution time of the first query grows to 43% of the total pre-processing time. More precisely, in order to minimize the overhead introduced by the transformation to $1d$ space, we partition the $1d$ query range into sub-intervals that tightly cover its original $3d$ range. This optimization [43] results in a high number of small intervals per query — on average 197. As a consequence, the first queries crack the previously uncracked area into a number of small adjacent intervals and therefore reorganize significant amounts of data.

The static (SFC) index, on the other hand, is not substantially slower for the first queries or, put differently, the building cost of SFC is not much higher than the first query of SFCracker. In fact, the cumulative execution time of SFCracker exceeds the one of SFC after 23 queries already. The incremental approach SFCracker thus does not offer a considerable benefit over SFC.

The incremental strategy of Mosaic is less expensive compared to SFCracker — the objects within the partition queried are potentially reassigned to the eight newly created partitions based on their location. Therefore, it takes Mosaic longer, i.e., 100 queries, before it exceeds the cumulative time of the static Grid. However, its cumulative execution time is still considerable with the biggest overhead being its top-down incremental strategy. The top-down strategy ensures fast convergence but it also introduces overhead as the data in frequently queried areas is re-partitioned multiple times until Mosaic reaches its final level of refinement.

QUASII, at the same time, does not exceed the cumulative execution of the R-Tree in our experiments. Even after 500 executed queries, the cumulative execution time for QUASII is 39.4% of that of the R-Tree. The main benefit comes from its partial reorganization strategy where the objects are gradually reorganized within the query boundaries, as opposed to the complete sort.

Summary. While all the incremental approaches reach the performance of their static counterparts, the incremental strategies of SFCracker and Mosaic are comparatively expensive. As we show for SFCracker, the major bulk of work has to be done when executing the first query — as the data needs to be transformed to $1d$ space and a single query has to perform multiple cracking operations to avoid performance penalties due to the transformation to $1d$ space. Mosaic increases its cumulative time considerably due to its top-down partitioning strategy — it reorganizes data in frequently queried areas multiple times until it reaches its final level of refinement. Only the cumulative execution time of QUASII does not exceed the one of its static counter part, the R-Tree, in our experiments.

6.4 Comparative Analysis

We now compare the performance of incremental approaches. We use the same setup as previously and measure the convergence of execution time as well as the cumulative execution time.

Convergence. Figure 9a) depicts the single query execution time for all the incremental approaches compared with the R-Tree and Scan. We use the R-Tree approach as a reference because it is the fastest approach among the static indexes for the workloads tested. We analyze the execution time of the first query and then focus on the performance of the converged data structure.

The execution time of the first query determines data-to-insight time and thus has to be as small as possible. Among the incremental approaches, SFCracker has the most expensive first query due to the transformation of data to the $1d$ space. Mosaic's first query is faster, but still expensive as it has to reassign all the objects to new partitions, examining all three coordinates. Finally, QUASII has the least expensive first query due to the nested data reorganization — the number of objects necessary to be examined and reorganized becomes smaller as more dimensions are taken into account: all objects are scanned on the x-dimension, but on the y-dimension only the objects with a x-value satisfying the

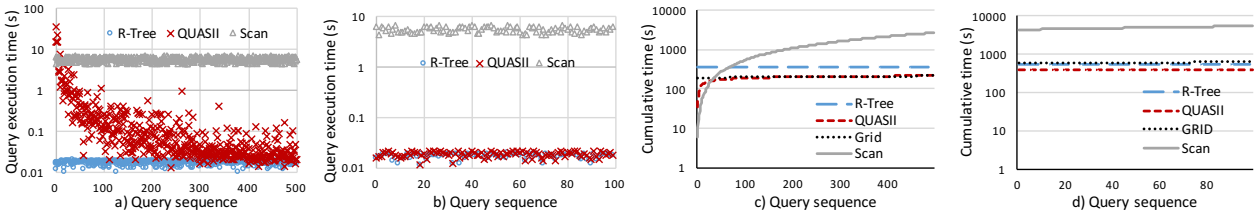


Figure 10: Convergence and cumulative time: the first 500 (a & c) and last 100 (b & d) queries.

query will be scanned (accordingly for the z-dimension). Overall, Scan is 13.7, 9.2 and 4.6 times faster compared to SFCracker, Mosaic and QUASII respectively, when executing the first query.

Among the incremental approaches, only QUASII attains the query execution time of R-Tree on a fully converged index. Mosaic and SFCracker have at their core space-oriented partitioning and therefore, their performance is affected by the data assignment strategy as well as the skew in distribution, as Section 6.2 shows. SFCracker additionally transforms data to $1d$ domain and thus cannot preserve spatial proximity to the same extent as the other approaches. Consequently, QUASII outperforms Mosaic and SFCracker with a speedup of $3.68x$ and $4.9x$ respectively for the average execution time of a query in a fully refined area.

Cumulative Execution Time. We use the cumulative execution time as metric to evaluate the decrease in the data-to-insight time as well as the "break-even" point — the point when the cumulative cost of incremental exceeds that of static indexing — to assess the quality of an incremental index. Figure 9b) shows the experimental results. We use Grid as a reference since it has the smallest cumulative execution time among the static approaches — its pre-processing step is comparatively cheap (once its optimal configuration is determined).

As discussed in Section 6.3, SFCracker and Mosaic have comparatively expensive strategies and thus reach the performance of Grid after 13 and 100 queries respectively. Grid, on the other hand, compared to QUASI, has not amortized its building cost after 500 queries. More precisely, QUASII reaches 84% of the Grid cumulative execution time and, more importantly, it achieves $3.66x$ faster query performance for completely refined areas. QUASII executes the first query the fastest and consequently achieves the highest decrease in data-to-insight time — $5.1x$ and $11.4x$ compared to Grid and R-Tree.

For single query execution, the major benefit of QUASII comes from its data-oriented partitioning. Similar the to R-Tree, it adjusts to the distribution of the data and, as opposed to Grid and SFC, it does not replicate the objects or extend the query. It additionally keeps the data in multidimensional space and does consequently not suffer from decrease in dimensionality. Its low cumulative cost is mostly attributed to its incremental strategy. QUASII does not sort all objects, but rather reorganizes them within the specific bounds, gradually curbing the amount of data necessary to be reorganized.

Summary. QUASII outperforms other incremental approaches with respect to the convergence of execution time and cumulative time. It achieves performance comparable to the R-Tree (the fastest static approach) in the areas of the dataset where enough queries have been executed, while not exceeding the cumulative time of Grid (the static approach with the least expensive pre-processing) or the R-Tree. Its major benefits come from the data-oriented partitioning and the nested reorganization strategy which reorganizes precisely the data touched and used.

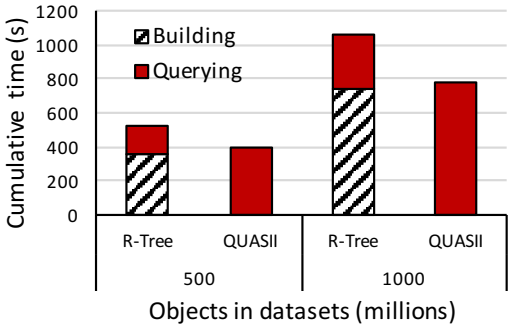


Figure 11: Analysis of QUASII: scalability.

6.5 Analysis of QUASII

In this section we focus on QUASII. We evaluate its performance on the workloads other than neuroscience, analyze its scalability and the impact of query selectivity.

6.6 Uniform Workload

In the previous analyses we used workloads with query clusters that show the benefit of incremental approaches: the index quickly converges to the final performance as the queries are targeting the same areas. In this experiment we evaluate the performance of QUASII for a uniform workload. We execute 10000 uniformly distributed queries of selectivity 0.1% on the dataset with uniform distribution and 500M elements. We compare the performance of QUASII with Scan and R-Tree and additionally consider Grid for the cumulative execution time. Figure 10 illustrates both convergence and cumulative time for the first 500 and last 100 queries of the workload.

None of the first 500 queries is executed on a completely refined index. Starting with the 300th query, however, the single query execution is close to the final performance. Among the last 100 queries, 64 are executed on a completely refined index. The performance of queries on the refined structure is equal or very close to the performance of the R-Tree, i.e., on average 7.5% slower than the R-Tree.

After 10000 executed queries QUASII reaches 75% and 63.8% of the cumulative time of the R-Tree and Grid approaches respectively (y axis is in log scale). Likewise, it decreases data-to-insight time by $10.3x$ and $5.6x$ compared to R-Tree and Grid. Although the pre-processing step of Grid is significantly cheaper compared to the R-Tree, its cumulative time deteriorates with more queries executed due to the expensive single query performance.

6.7 Performance Trends

In the following experiment we evaluate the scalability of QUASII by executing 10000 queries of selectivity 0.1% on datasets with

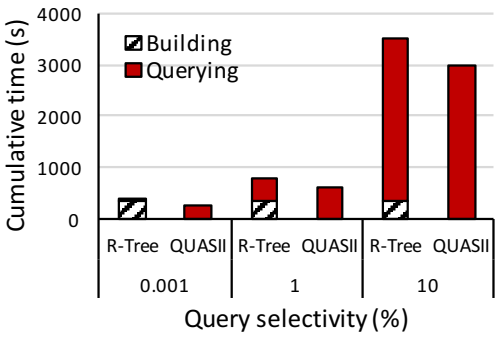


Figure 12: Analysis of QUASII: impact of selectivity.

500 million and 1 billion elements. In Figure 11 we compare the cumulative time of QUASII with R-Tree, where we additionally split the execution time of R-Tree into Building and Querying.

After 10000 executed queries QUASII reaches 75% and 73.7% of the cumulative time of the R-Tree with datasets of 500M and 1B elements respectively. By the time the R-Tree finishes its building process QUASII has executed around 8000 queries in both cases. QUASII also decreases data-to-insight time by 10.3x (on the 500M dataset) and 10.6x (on the 1B dataset) compared to the R-Tree. As illustrated in this experiment, QUASII maintains the performance trends as the dataset size increases.

6.8 Impact of Selectivity

In this set of experiments we evaluate the impact of query selectivity on the performance of QUASII. We measure the cumulative time for a uniform workload: 500M dataset and 5000 queries of 0.001%, 1%, and 10% selectivity. Figure 12 illustrates the results where we consider both the R-Tree and QUASII.

Intuitively, a static index (R-Tree) takes more time to amortize its building cost when executing 0.001% selectivity queries. On the other hand, the lower selectivity queries (10%) touch and reorganize a significant amount of data and QUASII thus reaches the break-even point with the R-Tree faster. Overall, after 5000 executed queries, QUASII reaches 68.8%, 79.8% and 85.6% of the cumulative time of the R-Tree for queries with 0.001%, 1%, and 10% selectivity.

7 RELATED WORK

To the best of our knowledge, no incremental strategy has been proposed to spatial indexing in main memory. While recently an incremental indexing technique has been proposed for exploration of multiple spatial datasets [35], the addressed problem is different (spatial search within multiple datasets) and the proposed solution focuses solely on disk-based efficiency, i.e., reducing the number of expensive disk I/O operations. Nevertheless, there has been considerable interest in incremental data processing within relational databases. Therefore, before giving an overview of related (but not incremental) spatial indexing techniques, we briefly describe incremental approaches used by relational database systems.

7.1 Relational Incremental Indexing

Incremental (one-dimensional) indexing techniques are extensively studied in database cracking [16, 18, 19] and adaptive merging [13, 14]. The former partially sorts keys in an in-memory relation, essentially performing quicksort. The latter, adaptive merging, takes the idea further and devises an incremental, external sort to make use of external memory as well.

Driven by the same goal (minimize data-to-insight time), novel systems have been proposed that bypass data pre-processing step and execute queries on raw data files. Instead, auxiliary data structures are built incrementally so that the most popular data subsets are serviced at the speeds of fully loaded/indexed data. For example, NoDB [2], RAW [25], and ViDa [24] incrementally build positional maps to track the position of frequently accessed data fields. This enables the systems to "jump" to previously queried data regions and potentially reduce the costs of tokenizing and parsing raw data sources.

7.2 Spatial Indexing

Research in indexing spatial data has produced numerous approaches in past decades [11]. In the following we briefly review spatial indexing approaches that we group into three classes depending on how amenable they are to incremental indexing.

One-dimensional Indexing. One way to address the curse of dimensionality in the context of spatial data is to transform it from multi- to one-dimensional domain. Typical methods to perform this transformation are space-filling curves like the Z-order [34], the Hilbert curve [21], and the Gray-code curve [9]. They impose a total order and preserve spatial proximity between objects - if the objects are close in higher-dimensional space, they are also close on the space-filling curve - reasonably well. Given such a mapping of spatial data, the existing 1d access methods, such as B-Tree [5], can be used for querying.

Data-oriented Indexing. The data-oriented partitioning approaches create an index structure taking into consideration the data distribution, where the prominent representatives are the KD-Tree [7], the R-Tree [15], and its variants. The R-Tree, arguably the most important data-oriented spatial index, is multi-dimensional generalization of the B-Tree which recursively encloses objects in minimum bounding rectangles (MBRs). The basic R-Tree definition faces the problems of overlap and dead space, both detrimental to query execution performance [15, 42]. Multiple approaches have been devised to address the issue. The R*-Tree [6], for example, uses an improved version of the node split algorithm and reinsertion of objects while the R+-Tree [37] tries to avoid overlap through the duplication of MBRs (leading to a bigger index). A priori knowledge of the entire dataset may help to reduce the above problems of the R-Tree by packing spatially close objects on the same disk page. The Hilbert R-Tree [23] achieves this using the Hilbert curve, Sort-Tile-Recursive (STR) [26] recursively sorts objects in all dimensions to do so, while Top-down Greedy Split (TGS) [12] recursively splits the data set into partitions minimizing the area on each level.

Adaptive index structures [41] rearrange the nodes of data-oriented hierarchical indexes (including the R-Tree index) in response to queries so that they can be accessed sequentially on disk. However, this reorganization is performed to improve query performance by optimizing disk-access without taking into consideration data-to-insight time.

A recently proposed partitioning mechanism for large-scale spatial data also adapts to an incoming query workload [3]. In contrast to QUASII, however, its primary goal is not to minimize data-to-insight time (as all necessary data structures are still built during pr-eprocessing), but to efficiently accommodate changes in data and workload. Also, it considers a distributed setting.

Space-oriented Indexing. The space-oriented partitioning approaches assign the data to the partitions based on space, regardless of data distribution. A typical representative is the uniform grid that partitions the space uniformly into partitions of equal size [38]. Similarly, the Quadtree [36] and its variant for $3d$ space, the Octree [20], recursively divide space into four/eight partitions of equal size to build a hierarchy of partitions. Further approaches, for example the grid file [32], use a non-uniform grid to better accommodate skew in the data (and to also optimize for disk access). The downside, is a more complex query execution due to the cells of different size and location. The two level grid file [17] addresses the issue by introducing an additional level with a coarser grid. Still, the overhead of testing the query against multiple cells can be substantial.

8 CONCLUSIONS

The advances in data acquisition technologies and supercomputing for large-scale simulations rapidly increase the amounts of spatial data generated and collected. This data helps the scientist tremendously to gain insights and understand natural phenomena, however, at the same time, it leaves them with the challenge of analyzing it. Known approaches to spatial indexing have two major drawbacks with respect to exploratory analyses. First, they require a time-consuming pre-processing step that delays analyses. Second, given the massive amounts of data, a scientist frequently only analyzes a small fraction of it and consequently indexing the entirety of the data does not always pay off.

In this paper we propose a novel incremental index for the exploration of spatial data, where the ultimate goal is to let the scientists perform exploratory analyses as soon as the data is available, while using their queries to incrementally index the data. Our approach, QUASII, reduces data-to-insight time and curbs the cost of incremental indexing, by gradually and partially sorting the data, while producing a data-oriented hierarchical structure. As our experiments show, QUASII reduces the data-to-insight time by up to a factor of 11.4x, while its performance converges to that of the fastest state-of-the-art static indexes.

ACKNOWLEDGEMENTS

We would like to thank the reviewers, the DIAS laboratory members, and Georgios Chatzopoulos for their comments and suggestions on how to improve the paper. This work is partially funded by the EU FP7 programme (ERC-2013-CoG), Grant No 617508 (ViDa) and EU Horizon 2020, GA No 720270 (HBP SGA1).

REFERENCES

[1] C.L. Abad, N. Roberts, Yi Lu, and R.H. Campbell. 2012. A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns. In *IISWC*. 100–109.
[2] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *SIGMOD*. 241–252.
[3] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. 2015. AQWA: Adaptive Query-Workload-Aware Partitioning of Big Spatial Data. *PVLDB* 8, 13 (2015), 2062–2073.
[4] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The Priority R-tree: a practically efficient and worst-case optimal R-tree. In *SIGMOD '04*.
[5] Rudolf Bayer. 1997. The Universal B-Tree for Multidimensional Indexing: General Concepts. In *WWCA*. 198–209.
[6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*.
[7] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *CACM* 18, 9 (1975), 509–517.
[8] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads. *PVLDB* 5, 12 (2012), 1802–1813.
[9] Christos Faloutsos. 1988. Gray codes for partial match and range queries. *TSE* 14, 10 (1988), 1381–1393.
[10] Christos Faloutsos and Shari Roseman. 1989. Fractals for secondary key retrieval. In *PODS*. 247–252.
[11] Volker Gaede and Oliver Guenther. 1998. Multidimensional Access Methods. *CSUR* 30, 2 (1998).
[12] Yván J. García, Mario A. López, and Scott T. Leutenegger. 1996. A Greedy Algorithm for Bulk Loading R-trees. In *GIS*.
[13] G. Graefe and H. Kuno. 2010. Adaptive Indexing for Relational Keys. In *ICDEW*.
[14] Goetz Graefe and Harumi Kuno. 2010. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *EDBT*.
[15] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*. 47–57.
[16] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores.. In *VLDB*.
[17] Klaus Hinrichs. 1985. Implementation of the Grid File: Design Concepts and Experience. *BIT* 25, 4 (1985).
[18] Stratos Idreos, Martin L Kersten, and Stefan Manegold. Database Cracking.. In *CIDR*.
[19] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Graefe Goetz. 2011. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores.. In *VLDB*.
[20] Chris L Jackins and Steven L Tanimoto. 1980. Oct-trees and their use in representing three-dimensional objects. *Comp. Graphics and Image Proc.* 14, 3 (1980), 249–270.
[21] Hosagrahar V Jagadish. 1990. Linear clustering of objects with multiple attributes. *SIGMOD Rec.* 19, 2 (1990), 332–342.
[22] Christian S Jensen, Dan Lin, and Beng Chin Ooi. 2004. Query and update efficient B^+-tree based indexing of moving objects. In *VLDB*. 768–779.
[23] Ibrahim Kamel and Christos Faloutsos. 1993. Hilbert R-tree: An improved R-tree using fractals. In *VLDB*. 500–509.
[24] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR'15*.
[25] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive query processing on RAW data. *PVLDB* 7, 12 (2014), 1119–1130.
[26] Scott T Leutenegger, Mario Lopez, Jeffrey Edgington, et al. 1997. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*. 497–506.
[27] Henry Markram et al. 2011. Introducing the Human Brain Project. *Procedia Computer Science* 7, 1. FET '11.
[28] Henry Markram et al. 2015. Reconstruction and simulation of neocortical microcircuitry. *Cell* 163, 2 (2015), 456–492.
[29] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H Saltz. 2001. Analysis of the clustering properties of the Hilbert space-filling curve. *TKDE* 13, 1 (2001), 124–141.
[30] EarthData NASA. *https://earthdata.nasa.gov/*.
[31] Actueel Hoogte Bestand Nederland. 2017. *AHN datasets*. http://www.ahn.nl.
[32] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *TODS* 9, 1 (1984).
[33] OpenStreetMap. *https://www.openstreetmap.org*.
[34] Jack A Orenstein and Tim H Merrett. 1984. A class of data structures for associative searching. In *PODS*. 181–190.
[35] Mirjana Pavlovic, Eleni Tzirita Zacharatou, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. 2016. Space Odyssey: Efficient Exploration of Scientific Data. In *ExploreDB*. 12–18.
[36] Hanan Samet. 1984. The quadtree and related hierarchical data structures. *CSUR* 16, 2 (1984), 187–260.
[37] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-tree: A dynamic index for multi-dimensional objects. In *VLDB*.
[38] Darius Šidlauskas, Simonas Šaltenis, et al. 2009. Trees or grids? Indexing moving objects in main memory. In *SIGSPATIAL*. 236–245.
[39] Tomáš Skopal, Michal Krátkỳ, Jaroslav Pokornỳ, and Václav Snášel. 2006. A new range query algorithm for Universal B-trees. *Information Systems* 31, 6 (2006), 489–511.
[40] Emmanuel Stefanakis, Yannis Theodoridis, Timos Sellis, and Yuk-Cheung Lee. 1997. Point representation of spatial objects and query window extension: A new technique for spatial access methods. *GIScience* 11, 6 (1997), 529–554.
[41] Yufei Tao and Dimitris Papadias. 2002. Adaptive Index Structures. In *VLDB*.
[42] Farhan Tauheed, Laurynas Biveinis, Thomas Heinis, Felix Schürmann, Henry Markram, and Anastasia Ailamaki. 2012. Accelerating Range Queries For Brain Simulations. In *ICDE*. 941–952.
[43] Hermann Tropf and H. Herzog. 1981. Multimensional Range Search in Dynamically Balanced Trees. *Angewandte Informatik* 23, 2 (1981), 71–77.
[44] Peter van Oosterom, Oscar Martinez-Rubi, Milena Ivanova, Mike Hörhammer, Daniel Geringer, Siva Ravada, Theo Tijssen, Martin Kodde, and Romulo Goncalves. 2015. Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Computers & Graphics* 49 (2015), 92–125.
[45] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *SIGMOD*. 1555–1566.