Compressed Spatial Hierarchical Bitmap (cSHB) Indexes for Efficiently Processing Spatial Range Query Workloads [*]

Parth Nagarkar	K. Selçuk Candan	Aneesha Bhat
Arizona State University	Arizona State University	Arizona State University
Tempe, AZ 85287-8809, USA	Tempe, AZ 85287-8809, USA	Tempe, AZ 85287-8809, USA
nagarkar@asu.edu	candan@asu.edu	aneesha.bhat@asu.edu

ABSTRACT

In most spatial data management applications, objects are represented in terms of their coordinates in a 2-dimensional space and search queries in this space are processed using spatial index structures. On the other hand, bitmap-based indexing, especially thanks to the compression opportunities bitmaps provide, has been shown to be highly effective for query processing workloads including selection and aggregation operations. In this paper, we show that bitmap-based indexing can also be highly effective for managing spatial data sets. More specifically, we propose a novel *compressed spatial hierarchical bitmap (cSHB)* index structure to support spatial range queries. We consider query workloads involving multiple range queries over spatial data and introduce and consider the problem of *bitmap selection* for identifying the appropriate subset of the bitmap files for processing the given spatial range query workload. We develop cost models for compressed domain range query processing and present query planning algorithms that not only select index nodes for query processing, but also associate appropriate bitwise logical operations to identify the data objects satisfying the range queries in the given workload. Experiment results confirm the efficiency and effectiveness of the proposed *compressed spatial hierarchical bitmap (cSHB)* index structure and the range query planning algorithms in supporting spatial range query workloads.

1. INTRODUCTION

Spatial and mobile applications are gaining in popularity, thanks to the wide-spread use of mobile devices, coupled with increasing availability of very detailed spatial data (such as Google Maps and OpenStreetMap [3]), and location-aware services (such as FourSquare and Yelp). For implementing range queries (Section 3.1.2), many of these applications and services rely on spatial database management systems, which represent objects in the database in terms of their coordinates in 2D space. Queries in this 2D space are then processed using multidimensional/spatial index structures that help quick access to the data [28].

1.1 Spatial Data Structures

The key principle behind most indexing mechanisms is to ensure that data objects closer to each other in the data space are also closer to each other on the storage medium. In the case of 1D data, this task is relatively easy as the total order implicit in the 1D space helps sorting the objects so that they can be stored in a way that satisfies the above principle. When the space in which the objects are embedded has more than one dimension, however, the data has multiple degrees of freedom and, as a consequence, there are many different ways in which the data can be ordered on the storage medium and this complicates the design of search data structures. One common approach to developing index structures for multi-dimensional data is to partition the space hierarchically in such a way that (a) nearby points fall into the same partition and (b) point pairs that are far from each other fall into different partitions. The resulting hierarchy of partitions then can either be organized in the form of trees (such as quadtrees, KD-trees, R-trees and their many variants [28]) or, alternatively, the root-to-leaf partition paths can be serialized in the form of strings and these strings can be stored in a string-specific search structure. Apache Lucene, a highly-popular search engine, for example, leverages such serializations of quadtree partitions to store spatial data in a *spatial prefix tree* [1].

An alternative to applying the partitioning process in the given multi-dimensional space is to map the coordinates of the data into a 1D space and perform indexing and query processing on this 1D space instead. Intuitively, in this alternative, one seeks an embedding from the 2D space to a 1D space such that (a) data objects closer to each other in the original space are also closer to each other on the 1D space, and (b) data objects further away from each other in the original space are also further away from each other on the 1D space. This embedding is often achieved through fractal-based *space-filling curves* [11,17]. In particular, the Peano-Hilbert curve [17] and Z-order curve [23] have been shown to be very effective in helping cluster nearby objects in the space. Consequently, if data are stored in an order implied by the space-filling curve, then the data elements that are nearby in the data space are also clustered, thus enabling efficient retrieval. In this paper, we leverage these properties of space-filling curves to develop a highly compressible bitmap-based index structure for spatial data.
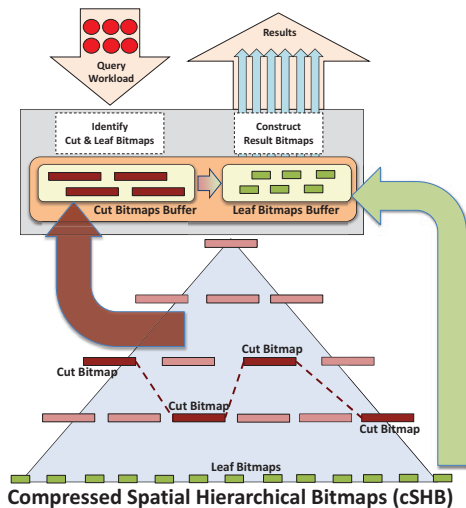
Figure 1: Processing a range query workload using *compressed spatial hierarchical bitmap (cSHB)*

1.2 Bitmap-based Indexing

Bitmap indexes [29, 32] have been shown to be highly effective in answering queries in data warehouses [34] and column-oriented data stores [5]. There are two chief reasons for this: (a) first of all, bitmap indexes provide an efficient way to evaluate logical conditions on large data sets thanks to efficient implementations of the bitwise logical "AND", "OR", and "NOT" operations; (b) secondly, especially when data satisfying a particular predicate are clustered, bitmap indexes provide significant opportunities for compression, enabling either reduced I/O or, even, complete in-memory maintenance of large index structures. In addition, (c) existence of compression algorithms [15, 33] that support compressed domain implementations of the bitwise logical operations enables query processors to operate directly on compressed bitmaps without having to decompress them until the query processing is over and the results are to be fetched from the disk to be presented to the user.

1.3 Contributions of this Paper

In this paper, we show that bitmap-based indexing is also an effective solution for managing spatial data sets. More specifically, we first propose *compressed spatial hierarchical bitmap (cSHB)* indexes to support spatial range queries. In particular, we (a) convert the given 2D space into a 1D space using Z-order traversal, (b) create a hierarchical representation of the resulting 2D space, where each node of the hierarchy corresponds to a (sub-)quadrant (i.e., effectively creating an implicit "quadtree"), and (c) associate a *bitmap* file to each node in the quadtree representing the data elements that fall in the corresponding partition. We present efficient algorithms for answering range queries using a select subset of bitmap files stored in a given cSHB index.

We then consider a service provider that has to answer multiple concurrent queries over the same spatial data and, thus, focus on query workloads involving multiple range queries. Since the same set of queries can be answered using different subsets of the bitmaps in the cSHB index structure, we consider the problem of identifying the appropriate bitmap nodes for processing the given query workload. More specifically, as we visualize in Figure 1, (a) we develop cost models for range query processing over compressed spatial

hierarchical bitmap files and (b) propose efficient *bitmap selection* algorithms that select the best bitmap nodes from the cSHB index structure to be fetched into the main-memory for processing of the query workload. In this paper, we also present an efficient disk-based organization of compressed bitmaps. To our best knowledge, this is the first work that provides an efficient index structure to execute a query workload involving multiple spatial range queries by using bitmap indexes. Experimental evaluations of the cSHB index structure and the bitmap selection algorithms show that cSHB is highly efficient in answering a given query workload.

1.4 Paper Organization

This paper is organized as follows. In the next section we provide an overview of the related work. In Section 3.1, we introduce the key concepts and notations, and in Section 3.2, we present the proposed cSHB index structure. Then, in Section 4, we describe how query workloads are processed using cSHB: in Section 4.1, we introduce the concepts of range query plans, in Section 4.2, we present cost models for alternative execution strategies, and in Section 4.3, we present algorithms for finding efficient query plans for a given range query workload. Experiment results are reported in Section 5. We finally conclude the paper in Section 6.

2. RELATED WORK

2.1 Multi-Dimensional Space Partitioning

Multi-dimensional space partitioning strategies can be categorized into two: In the first case, including quadtree, BD-tree, G-Tree, and KD-tree variants, a given bounded region is divided into two or more "*open*" partitions such that each partition borders a boundary of the input region. In the latter case, some of the partitions (often referred to as minimum bounding regions, MBRs) are "*closed*" regions of the space, not necessarily bordering any boundary of the input region. An advantage of this latter category of index structures, including the R-tree and its variants (R*-tree, R+-tree, Hilbert R-tree, and others), is that these MBRs can tightly cover the input data objects.

While most index structures have been designed to process individual queries, there are also works focusing on the execution of a workload of multiple queries on the same index structure. In [27], the Hilbert values of the centroids of the rectangles formed by the range queries are sorted, and these queries are grouped accordingly to process them over an R-tree. In [14], R-trees are used to execute multiple range queries and, in their formulation, authors propose to combine adjacent queries into one. Thus, the algorithm is not able to differentiate results of individual queries.

There are two problems commonly associated with multi-dimensional index structures, namely overlaps between partitions (which cause redundant I/O) and empty spaces within partitions (which cause unnecessary I/O). While there has been a significant amount of research in searching for partitioning strategies that do not face these problems, these two issues still remain [26] and are especially critical in very high-dimensional vector spaces. One way to tackle this problem has been to parallelize the work. For example, in [6], the authors describe a Hadoop-based data warehousing system with spatial support, where the main focus is to parallelize the building of the R*-tree index structure and query processing over Hadoop.
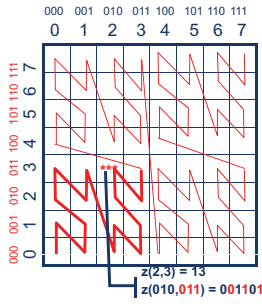
Figure 2: Z-order curve for a sample 2D Space.

2.2 Space Filling Curve based Indexing

The two most common space-filling curves are the fractal-based Z-order curve [23] and the Peano-Hilbert curve [17]. While the Hilbert curve provides a better mapping from the multidimensional space onto the 1D space, its generation is a complicated and costly process [26]. With Z-order curve, however, mapping back-and-forth between the multidimensional space and the 1D space using a process called *bit-shuffling* (visualized in Figure 2) is very simple and efficient. Consequently, the Z-order curve has been leveraged to deal with spatial challenges [12], including construction of and searches on R-trees [27] and others.

In [35], the authors present a parallel spatial query processing system called VegaGiStore that is built on top of Hadoop. This system uses a two-tiered index structure that consists of a quadtree-based global index (used for finding the necessary data blocks) and a Hilbert-ordering based local index (used for finding the spatial objects in the data block). In [26], the authors present an index called BLOCK to process spatial range queries. Their main assumption is that the data and index can fit into the main memory, and hence their aim is to reduce the number of comparisons between the data points and the query range. They create a sorted list of the Z-order values for all the data points. Given a query, they start at the coarsest level. If a block lies entirely within the given query range, they retrieve the data points in this block, otherwise, based on a branching fact, they decide whether to search the next granular level. In [7], the authors proposed a UB-tree index structure that also uses Z-ordering for storing multidimensional data in a B+ tree and in [22], the authors presented a hierarchical clustering scheme for the fact table of a data warehouse in which the data is stored using the above mentioned UB-tree. In [31], the authors present a range query algorithm specifically for the UB-tree. Unlike our approach, the above solutions are not specifically designed for multiple query workloads.

2.3 Bitmap Indexes

There have been significant amount of works on improving the performance of bitmap indexes and keeping compression rates high [20, 32, 33]. Most of the newer compression algorithms use run-length encoding for compression: this provides a good compression ratio and enables bitwise operations directly on compressed bitmaps without having to decompress them first [32]. Consequently, bitmap indexes are also shown to perform better than other database index structures, especially in data warehouses and column-oriented systems [5, 32, 34].

For attributes with a large number of distinct values, bitmaps are often created with binning, where the domain is partitioned into bins and a bitmap is created for each bin. Given a query, results are constructed by combining relevant bins using bitwise OR operations. Recognizing that many data attributes have hierarchical domains, there has also been research in the area of multi-level and hierarchical bitmap indexes [13, 24, 25, 29]. When the bitmaps are partitioned (with potential overlaps), it is necessary to select an appropriate set (or *cut* [25]) of bitmaps for query processing; results are often obtained by identifying a set of bitmaps and combining them using bitwise ORs. This work builds on some of the ideas presented in [25] from 1D data to spatial data. In [25], the cost model only considered the dominant I/O cost (reading the bitmaps from the disk), but in this work, we present an updated cost model, that appropriately includes the I/O cost as well as the cost of performing local operations on the in-memory bitmaps.

There has been some prior attempts to leverage bitmaps in spatial query processing. For example, an MBR-based spatial index structure is proposed in [30], where the leaves of the tree are encoded in the form of bitmaps. Given a query, the proposed HSB-index is traversed top-down (as in R-trees) to identify the relevant bitmaps to be combined. In this paper, we note that not only leaves, but also internal nodes of the spatial hierarchy can be encoded as bitmaps, leading to significant savings in range search time, especially for query workloads consisting of multiple spatial range queries. Thus, our work focuses on which bitmaps to read in the context of spatial range query workloads and we introduce novel algorithms to choose which bitmaps to use to answer a query workload efficiently. We generalize the problem of bitmap selection and consider alternative strategies that complement OR-based result construction. In [16], authors propose a storage and retrieval mechanism for large multi-dimensional HDF5 files by using bitmap indexes. While range queries are supported on their architecture, they neither leverage Z-order indexing, nor hierarchical bitmaps as proposed in this work. Also, their proposed mechanism is not optimized for multiple query workloads.

3. COMPRESSED SPATIAL HIERARCHICAL BITMAP (cSHB) INDEXES

In this section, we present the key concepts used in the paper and introduce the *compressed spatial hierarchical bitmap (cSHB)* index structure for answering spatial range queries.

3.1 Key Concepts and Notations

3.1.1 Spatial Database

A multidimensional database, $\mathcal{D}$, consists of points that belong to a (bounded and of finite-granularity) multidimensional space $\mathcal{S}$ with d dimensions. A spatial database is a special case where $d = 2$. We consider rectangular spaces such that the boundaries of $\mathcal{S}$ can be described using a pair of south-west and a north-east corner points, c_{sw} and c_{ne} ($c_{sw}.x \leq c_{ne}.x$ and $c_{sw}.y \leq c_{ne}.y$ and $\forall_{p \in \mathcal{S}}\ c_{sw}.x \leq p.x \leq c_{ne}.x$ and $c_{sw}.y \leq p.y \leq c_{ne}.y$).

3.1.2 Spatial Query Workload

In this paper, we consider query workloads, Q, consisting of a set of *rectangular* spatial range queries.

- **Spatial Range Query:** A range query, $q \in Q$, is defined by a corresponding range specification $q.rs =$

$\langle q_{sw}, q_{ne} \rangle$, consisting of a south-west point and a north-east point, such that $q_{sw}.x \leq q_{ne}.x$ and $q_{sw}.y \leq q_{ne}.y$.

Given a range query, q, with a range specification, $q.rs = \langle q_{sw}, q_{ne} \rangle$, a data point $p \in \mathcal{D}$ is said to be contained within the query range (or is a *range point*) if and only if $q_{sw}.x \leq p.x \leq q_{ne}.x$ and $q_{sw}.y \leq p.y \leq q_{ne}.y$.

3.1.3   Spatial Hierarchy

In cSHB, we associate to the space $\mathcal{S}$ a hierarchy H, which consists of the node set $\mathcal{N}(H) = \{n_1, \ldots, n_{maxn}\}$:

- **Nodes of the hierarchy:** Intuitively, each node, $n_i \in \mathcal{N}(H)$ corresponds to a (bounded) subspace, $S_i \subseteq S$, described by a pair of corner points, $c_{i,sw}$ and $c_{i,nw}$.

- **Leaves of the hierarchy:** L_H denotes the set of leaf nodes of the hierarchy H and correspond to all potential point positions of the finite space S. *Assuming that the database, $\mathcal{D}$, contains only points, only the leaves of the spatial hierarchy occur in the database.*

- **Parent of a node:** For all n_i, $parent(n_i)$ denotes the parent of n_i in the corresponding hierarchy; if n_i is the root, then $parent(n_i) = \bot$.

- **Children of a node:** For all n_i, $children(n_i)$ denotes the children of n_i in the corresponding hierarchy; if $n_i \in L_H$, then $children(n_i) = \emptyset$. In this paper, we assume that the children induce a partition of the region corresponding to the parent node:

$$\left(\underset{n_h \neq n_j \in children(n_i)}{\forall} S_h \cap S_j = \emptyset \right) \text{and} \left(S_i = \bigcup_{n_h \in children(n_i)} S_h \right).$$

- **Descendants of a Node:** The set of descendants of node n_i in the corresponding hierarchy is denoted as $desc(n_i)$. Naturally, if $n_i \in L_H$, then $desc(n_i) = \emptyset$.

- **Internal Nodes:** Any node in H that is not a leaf node is called an internal node. The set of internal nodes of H is denoted by I_H. Each internal node in the hierarchy corresponds to a (non-point) sub-region of the given space. If $\mathcal{N}(H, l)$ denotes the subset of the nodes at level l of the hierarchy H, then we have

$$\left(\forall_{n_i \neq n_j \in \mathcal{N}(H,l)} S_i \cap S_j = \emptyset \right) \text{ and } \left(\mathcal{S} = \bigcup_{n_i \in \mathcal{N}(H,l)} S_i \right).$$

The root node corresponds to the entire space, $\mathcal{S}$.

- **Leaf Descendants of a Node:** Leaf descendants, $leafDesc(n_i)$, of a node are the set of nodes such that

$$leafDesc(n_i) = desc(n_i) \cap L_H.$$

3.2   Compressed Spatial Hierarchical Bitmap (cSHB) Index Structure

In this section, we introduce the proposed *compressed spatial hierarchical bitmap (cSHB)* index structure:

DEFINITION 3.1   (cSHB INDEX STUCTURE). *Given a spatial database $\mathcal{D}$ consisting of a space, $\mathcal{S}$, and a spatial hierarchy, H, a cSHB index is a set, $\mathcal{B}$ of bitmaps, such that for each $n_i \in \mathcal{N}(H)$, there is a corresponding bitmap, $B_i \in \mathcal{B}$, where the following holds:*
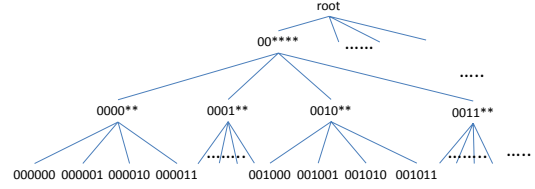
Figure 3: A sample 4-level hierarchy defined on the Z-order space defined in Figure 2 (the string associated to each node corresponds to its unique label)

- *if n_i is an internal node (i.e., $n_i \in I_H$), then $\left(\exists_{o \in \mathcal{D}} \exists_{n_h \in leafDesc(n_i)} located\_at(o, n_h) \right) \leftrightarrow (B_i[o] = 1)$, whereas*
- *if n_i is a leaf node (i.e., $n_i \in L_H$), then $\left(\exists_{o \in \mathcal{D}} located\_at(o, n_i) \right) \leftrightarrow (B_i[o] = 1)$*   ○

3.2.1   Our Implementation of cSHB

A cSHB index structure can be created based on any hierarchy satisfying the requirements[1] specified in Section 3.1.3.

In this paper, *without loss of generality*, we discuss a Z-curve based construction scheme for cSHB. The resulting hierarchy is analogous to the MX-quadtree data structure, where all the leaves are at the same level and a given region is always partitioned to its quadrants at the center [28]. As introduced in Sections 1.1 and 2.2, a space-filling curve is a fractal that maps a given finite multidimensional data space onto a 1D curve, while preserving the locality of the multi-dimensional data points (Figure 2): in other words nearby points in the data space tend to be mapped to nearby points on the 1D curve. As we also discussed earlier, Z-curve is a fractal commonly used as a space-filling curve (thanks to its effectiveness in clustering the points in the data space and the efficiency with which the mapping can be computed).

A key advantage of the Z-order curve (for our work) is that, due to the iterative (and self-similar) nature of the underlying fractal, the Z-curve can also be used to impose a hierarchy on the space. As visualized in Figure 3, each internal node, n_i, in the resulting hierarchy has four children corresponding to the four quadrants of the space, S_i. Consequently, given a 2^h-by-2^h space, this leads to an $(h+1)$-level hierarchy, (analogous to an MX-quadtree [28]) which can be used to construct a cSHB index structure[2]. As we show in Section 5, this leads to highly compressible bitmaps and efficient execution plans.

3.2.2   Blocked Organization of Compressed Bitmaps

Given a spatial database, $\mathcal{D}$, with a corresponding hierarchy, H, we create and store a *compressed bitmap* for each node in the hierarchy, except for those that correspond to regions that are *empty*. These bitmaps are created in a bottom-up manner, starting from the leaves (which encode for each point in space, $\mathcal{S}$, which data objects in $\mathcal{D}$ are located at that point) and merging bitmaps of children nodes into the bitmaps of their parents. Each resulting bitmap is stored as a *compressed* file on disk.

It is important to note that, while compression provides significant savings in storage and execution time, a naive storage of compressed bitmaps can still be detrimental for

---

[1]In fact, cSHB can be created even when some of the requirements are relaxed – for example children do not need to cover the parent range entirely (as in R-trees).

[2]Without loss of generality, we assume that the width and height are 2^h units for some integer $h \geq 1$.

Algorithm 1 Writing blocks of compressed bitmaps to disk
1: **Input:**
- A spatial database, $\mathcal{D}$, defined over 2^h-by-2^h size space, $\mathcal{S}$ and a corresponding $(h+1)$-level (Z-curve based) hierarchy, H, with set of internal nodes, I_H
- Minimum block size, K

2: **procedure** WRITEBITMAPS
3:     Block $T = \emptyset$
4:     $availableSize = K$
5:     **for** level $l = (h+1)$ (i.e., leaves) to 0 (i.e., root) **do**
6:         **for** each node n_i in l in increasing Z-order **do**
7:             **if** $l == (h+1)$ **then**
8:                 Initialize a compressed bitmap B_i
9:             **else**
10:                 $B_i = \underset{n_j \in children(n_i)}{\text{OR}} B_j$
11:             **end if**
12:             **if** $size(B_i) \geq K$ **then**
13:                 write B_i to disk;
14:             **else**
15:                 $T = append(T, B_i)$
16:                 $availableSize = availableSize - size(B_i)$
17:                 **if** $(availableSize \leq 0)$ or (n_i is the last node at this level) **then**
18:                     write T to disk;
19:                     Block $T = \emptyset$
20:                     $availableSize = K$
21:                 **end if**
22:             **end if**
23:         **end for**
24:     **end for**
25: **end procedure**

performance: in particular, in a data set with large number of objects located at unique points, there is a possibility that a very large number of leaf bitmaps need to be created on the secondary storage. Thus, creating a separate bitmap file for each node may lead to inefficiencies in indexing as well as during query processing (as directory and file management overhead of these bitmaps may be non-negligible).

To overcome this problem, cSHB takes a *target block size*, K, as input and ensures that all index-files written to the disk (with the possible exception of the last bitmap file in each level) are at least K bytes. This is achieved by concatenating, if needed, compressed bitmap files (corresponding to nodes at the same level of hierarchy). In Algorithm 1, we provide an overview of this block-based bottom-up cSHB index creation process. In Line 10, we see that the bitmap of an internal node is created by performing a bitwise OR operation between the bitmaps of the children of the node. These OR operations are implemented in the compressed bitmap domain enabling fast creation of the bitmap hierarchy. As it creates compressed bitmaps, the algorithm packs them into a block (Line 15). When the size of the block exceeds K, the compressed bitmaps in the block are written to the disk (Line 18) as a single file and the block is re-initialized.

EXAMPLE 3.1. *Let us assume that $K = 10$ and also that we are considering the following sequence of nodes with the associated (compressed) bitmap sizes:*

$$\langle n_1, 3 \rangle; \langle n_2, 4 \rangle; \langle n_3, 2 \rangle; \langle n_4, 15 \rangle; \langle n_5, 3 \rangle; \ldots$$

This sequence of nodes will lead to following sequence of bitmap files materialized on disk:

$$\underbrace{[B_4]}_{size=15} \; ; \; \underbrace{[B_1 \| B_2 \| B_3 \| B_5]}_{size=3+4+2+3=12} \; ; \; \ldots$$

Note that, since the bitmap for node n_4 is larger than the target block size, B_4 is written to disk as a separate bitmap

(a) 2D query range   (b) Corresponding 1D ranges
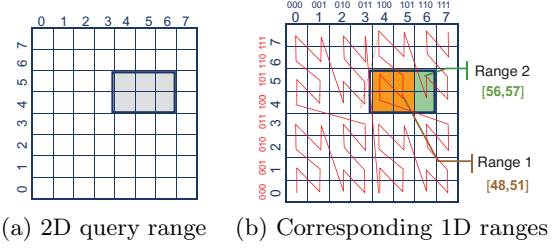
Figure 4: Mapping of a single spatial range query to two 1D ranges on the Z-order space: (a) A contiguous 2D query range, $[sw = (4,4); ne = (6,5)]$ and (b) the corresponding contiguous 1D ranges, [48,51] and [56,57], on the Z-curve

file; on the other hand, bitmaps for nodes n_1, n_2, n_3, and n_5 need to be concatenated into a single file to obtain a block larger than $K = 10$ units. ◇

Note that this block-based structure implies that the size of the files and the number of bitmap files on the disk will be upper bounded, but it also means that the cost of the bitmap reads will be lower bounded by K. Therefore, to obtain the best performance, repeated access to a block to fetch different bitmaps must be avoided through bitmap buffering and/or bitmap request clustering. In the next section, we discuss the use of cSHB index for range query processing. In Section 5, we experimentally analyze the impact of block-size on the performance of the proposed cSHB index structure.

4. QUERY PROCESSING WITH THE cSHB INDEX STRUCTURE

In this section, we describe how query workloads are processed using the cSHB index structure. In particular, we consider query workloads involving multiple range queries and propose *spatial bitmap selection* algorithms that select a subset of the bitmap nodes from the cSHB index structure for efficient processing of the query workload.

4.1 Range Query Plans and Operating Nodes

In order to utilize the cSHB index for answering a spatial range query, we first need to map the range specification associated with the given query from the 2D space to the 1D space (defined by the Z-curve). As we see in Figure 4, due to the way the Z-curve spans the 2D-space, it is possible that a single contiguous query range in the 2D space may be mapped to multiple contiguous ranges on the 1D space. Therefore, given a 2D range query, q, we denote the resulting set of (disjoint) 1D range specifications, as RS_q.

Let us be given a query, q, with the set of 1D range specifications, RS_q. Naturally, there may be many different ways to process the query, each using a different set of bitmaps in the cSHB index structure, including simply fetching and combining only the relevant leaf bitmaps:

EXAMPLE 4.1 (ALTERNATIVE RANGE QUERY PLANS). *Consider a query q with $q.rs = \langle (1,0), (3,1) \rangle$ on the space shown in Figure 2. The corresponding 1D range, $[2, 11]$, would cover the following leaf nodes of the hierarchy shown in Figure 3: $RS_q = (000010, 000011, 001000, 001001, 001010, 001011)$. The following are some of the alternative query plans for q using the proposed cSHB index structure:*

- Inclusive query plans: *The most straightforward way to execute the query would be to combine (bitwise OR operation) the bitmaps of the leaf nodes covered in 1D range,* [2, 11]. *We refer to such plans, which construct the result by combining bitmaps of selected nodes using the OR operator, as* inclusive plans.

  An alternative inclusive plan for this query would be to combine the bitmaps of nodes $000010, 000011, 0010**$:

  $$B_{000010} \texttt{ OR } B_{000011} \texttt{ OR } B_{0010**}.$$

- Exclusive query plans: *In general, an exclusive query plan includes removal of some of the children or descendant bitmaps from the bitmaps of a parent or ancestor through the ANDNOT operation. One such exclusive plan would be to combine the bitmaps of all leafs nodes, except for* B_{000010}, B_{000011}, B_{001000}, B_{001001}, B_{001010}, B_{001011}, *into a bitmap* B_{non_result} *and return*

  $$B_{root} \texttt{ANDNOT} B_{non\_result}.$$

- Hybrid query plans: *Both inclusive and exclusive only query plans may miss efficient query processing alternatives. Hybrid plans combine inclusive and exclusive strategies at different nodes of the hierarchy. A sample hybrid query plan for the above query would be*

  $$\big(B_{0000**} \texttt{ ANDNOT } (B_{000000} \texttt{ OR } B_{000001})\big) \texttt{ OR } B_{0010**}. \diamond$$

As illustrated in the above example, a range query, q, on hierarchy H, can be answered using different query plans, involving bitmaps of the leaves and certain internal nodes of the hierarchy, collectively referred to as the *operating nodes of a query plan*. In Section 4.3, we present algorithms for selecting the operating nodes for a given workload, Q; but first we discuss the cost model that drives the selection process.

4.2 Cost Models and Execution Strategies

In cSHB, the bitwise operations needed to construct the result are performed on compressed bitmaps directly, without having to decompress them.

4.2.1 Cost Model for Individual Operations

We consider two cases: (a) logical operations on disk-resident compressed bitmaps and (b) logical operations on in-buffer compressed bitmaps.

Operations on Disk-Resident Compressed Bitmaps.

In general, when the logical operations are implemented on compressed bitmaps that reside on the disk, the time taken to read a bitmap from the secondary storage to the main memory dominates the overall bitwise manipulation time [15]. The overall cost is hence proportional to the size of the (compressed) bitmap file on the secondary storage.

Let us consider a logical operation on bitmaps B_i and B_j. Let us assume that $T(B_i)$ and $T(B_j)$ denotes the blocks in which B_i and B_j are stored, respectively. Since multiple bitmaps can be stored in a single block, it is possible that B_i and B_j are in the same block. Hence, let us further assume that $\mathcal{T}_{(B_i, B_j)}$ is the set of unique blocks that contain the bitmaps, B_i and B_j. Then the overall I/O cost is:

$$cost_{io}(B_i \texttt{ op } B_j) = \alpha_{IO}\big(\sum_{T \in \mathcal{T}_{(B_i,B_j)}} size(T)\big),$$

where α_{IO} is an I/O cost multiplier and op is a binary bitwise logical operator. A similar result also holds for the unary operation NOT.

Operations on In-Buffer Compressed Bitmaps.

When the compressed bitmaps on which the logical operations are implemented are already in-memory, the disk access cost is not a factor. However, also in this case, the cost is proportional to the sizes of the compressed bitmap files in the memory, independent of the specific logical operator that is involved [33], leading to

$$cost_{cpu}(B_i \texttt{ op } B_j) = \alpha_{cpu}\big(size(B_i) + size(B_j)\big),$$

where α_{cpu} is the CPU cost multiplier. A similar result also holds for the unary operation NOT.

4.2.2 Cost Models for Multiple Operations

In this section, we consider a cost model which assumes that blocks are disk-resident. Therefore, we consider a storage hierarchy consisting of disk (storing all bitmaps), RAM (as buffer storing relevant bitmaps), and L3/L2 caches (storing currently needed bitmaps).

Buffered Strategy.

In the *buffered* strategy, visualized in Figure 1, the bitmaps that correspond to any leaf or non-leaf operating nodes for the query plan of a given query workload, Q, are brought into the buffer once and cached for later use. Then, for each query $q \in Q$, the corresponding result bitmap is extracted using these buffered operating node bitmaps. Consequently, if a node is an operating one for more than one $q \in Q$, it is read from the disk only once (and once for each query from the memory). Let us assume that $\mathcal{T}_{ON_Q}$ denotes the set of unique blocks that contains all the necessary operating nodes given a query workload $Q(ON_Q)$. This leads to the overall processing cost, $time\_cost_{buf}(Q, ON_Q)$, of

$$\underbrace{\alpha_{IO}\left(\sum_{T \in \mathcal{T}_{ON_Q}} size(T)\right)}_{read\ cost} + \underbrace{\alpha_{cpu}\left(\sum_{q \in Q} \sum_{n_i \in ON_q} size(B_i)\right)}_{operating\ cost}.$$

Since all operating nodes need to be buffered, this execution strategy requires a total of $storage\_cost_{buf}(Q, ON_Q) = \sum_{n_i \in ON_Q} size(B_i)$ buffer space. Note that, in general, $\alpha_{IO} > \alpha_{cpu}$. However, in Section 5, we see that the number of queries in the query workload and query ranges determine the relative costs of in-buffer operations vs. disk I/O.

The buffered strategy has the advantage that each query can be processed individually on the buffered bitmaps and the results for each completed query can be pipelined to the next operator without waiting for the results of the other queries in the workload. This reduces the memory needed to temporarily store the result bitmaps. However, in the buffered strategy, the buffer needed to store the operating node bitmaps can be large.

Incremental Strategy.

The *incremental* strategy avoids buffering of all operating node bitmaps simultaneously. Instead, all leaf and non-leaf operating nodes are fetched from the disk one at a time *on demand* and results for each query are constructed incrementally. This is achieved by considering one internal operating node at a time and, for each query, focusing only on the leaf operating nodes under that internal node. For this purpose, a *result accumulator bitmap*, Res_j, is maintained for each query in $q_j \in Q$ and each operating node read from the disk is applied directly on this result accumulator bitmap.
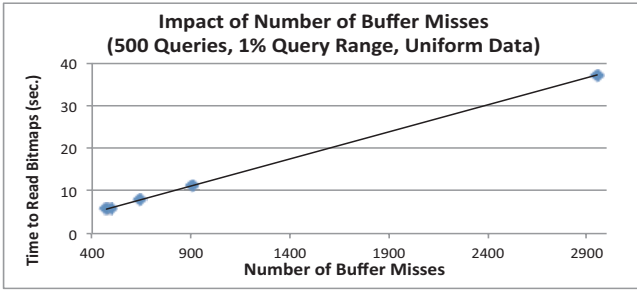
Figure 5: Buffer misses and the overall read time (data and other details are presented in Section 5)

While it does not need buffer to store all operating node bitmaps, the incremental strategy may also benefit from partial caching of the relevant blocks. This is because, while each internal node needs to be accessed only once, each leaf node under this internal node may need to be brought to the memory for multiple queries. Moreover, since the data is organized in terms of blocks, rather than individual nodes (Section 3.2.2), a single block may serve multiple nodes to different queries. When sufficient buffer is available to store the *working set* of blocks (containing the operating leaf nodes under the current internal node), the execution cost, $time\_cost_{inc}(Q, ON_Q)$, of the incremental strategy is identical to that of the buffered strategy. Otherwise, as illustrated in Figure 5, the read cost component is a function of buffer misses, $\alpha_{IO} \times \#\_buffer\_misses$, which itself depends on the size of the buffer and the clustering of the data.

The storage complexity[3] is $storage\_cost_{inc}(Q, ON_Q) = \sum_{q_j \in Q} size(Res_j)$ plus the space needed to maintain the most recently read blocks in the current working set. Experiments reported in Section 5 show that, for the considered data sets, the sizes of the working sets are small enough to fit into the L3-caches of many modern hardware.

4.3 Selecting the Operating Bitmaps for a Given Query Workload

To process a range query workload, Q, on a data set, $\mathcal{D}$, with the underlying cSHB hierarchy H, we need to select a set of operating bitmap nodes, ON_Q, of H from which we can construct the results for all $q_j \in Q$, such that $time\_cost(Q, ON_Q)$ is the minimum among all possible sets of operating bitmaps for Q. It is easy to see that the number of alternative sets of operating bitmaps for a given query workload Q is exponential in the size of the hierarchy H. Therefore, instead of seeking the set of operating bitmaps among all subsets of the nodes in H, we focus our attention on the *cuts* of the hierarchy, defined as follows:

DEFINITION 4.1 (CUTS OF H RELATIVE TO Q). *A complete cut, C, of a hierarchy, H, relative to a query load, Q, is a subset of the* internal nodes (including the root) *of the hierarchy, satisfying the following two conditions:*

- validity: *there is exactly one node on any root-to-leaf branch in a given cut; and*
- completeness: *the nodes in C collectively cover every possible root-to-leaf branch for all leaf nodes in the result sets for queries in Q.*

---
[3]The space complexity of the incremental strategy can be upper-bounded if the results for the queries in Q can be pipelined to the next set of operators *progressively* as partial results constructed incrementally.

If a set of internal nodes of H only satisfies the first condition, then we refer to the cut as an incomplete cut. ∘

As visualized in Figure 1, given a cut C, cSHB queries are processed by using **only the bitmaps of the nodes in this cut, along with some of the leaf bitmaps necessary to construct results of the queries in Q.** In the rest of this subsection, we first describe how queries are processed given a cut, C, of H and then present algorithms that search for a cut, C, given a workload, Q.

4.3.1 Range Query Processing with Cuts

It is easy to see that any workload, Q, of queries can be processed by any (even incomplete) cut, C, of the hierarchy and a suitable set of leaf nodes: Let R_q denote the set of leaf nodes that appear in the result set of query $q \in Q$ and $\bar{R}_q$ be the set of leaf nodes that do not appear in the result set. Let also R_q^C be the set of the result leaves covered by a node in C. Then, one possible way to construct the result bitmap, B_q, is as follows:

$$B_q = \left(\left(\underset{n_i \in C}{\text{OR}} B_i \right) \text{OR} \underbrace{\left(\underset{n_i \in R_q \setminus R_q^C}{\text{OR}} B_i \right)}_{inclusions} \right) \underset{n_j \in R_q^C \cap \bar{R}_q}{\text{ANDNOT}} \underbrace{B_j}_{exclusions} .$$

Intuitively any result nodes that are not covered by the cut need to be *included* in the result using a bitwise OR operation, whereas any leaf node that is not in any result needs to be *excluded* using an ANDNOT operation. Consequently,

- if $C \cap R_q = \emptyset$, an *inclusion-only* plan is necessary,
- an *exclusion-only* plan is possible only if C covers R_q completely.

Naturally, given a range query workload, Q, different query plans with different cuts will have different execution costs. The challenge is, then,

- to select an appropriate cut, C, of the hierarchy, H, for query workload, Q, and
- to pick, for each query $q_j \in Q$, a subset $C_j \in C$ for processing q_j,

in such a way that these will minimize the overall processing cost for the set of range queries in Q. Intuitively, we want to include in the cut, those nodes that will not lead to a large number of exclusions and cannot be cheaply constructed by combining bitmaps of the leaf nodes using OR operations.

4.3.2 Cut Bitmap Selection Process

Given the above cut-based query processing model, in this section we propose a cut selection algorithm consisting of two steps: (a) a per-node cost estimation step and (b) a bottom-up cut-node selection step. We next describe each of these two steps.

Node Cost Estimation.

First, the process assigns an estimated cost to those hierarchy nodes that are relevant to the given query workload, Q. For this, the algorithm traverses through the hierarchy, H, in a top-down manner and identifies part, R, of the hierarchy relevant for the execution of at least one query, $q \in Q$ (i.e., for at least one query, q, the range associated with the node and the query range intersect). Note that this process

Algorithm 2 Cost and Leaf Access Plan Assignment Algorithm

1: **Input:** Hierarchy H, Query Workload Q
2: **Outputs:** Query workload, $Q(n_i)$, and cost estimate, $cost_i$, for each node, $n_i \in H$; leaf access plan, $E_{i,j}$, for all node/query pairs $n_i \in H$ and $q_j \in Q(n_i)$; a set, $R \subseteq I_H$, or relevant internal nodes
3: **Initialize:** $R = \emptyset$
4: **procedure** COST_AND_LEAFACCESSPLANASSIGNMENT
5:     **for** each internal node $n_i \in I_H$ in top-down fashion **do**
6:         **if** $n_i = $ "$root''$ **then**
7:             $Q(n_i) = Q$
8:         **else**
9:             $Q(n_i) = \{q \in Q(parent(n_i)) \ s.t. \ (q.rs \cap S_i) \neq \emptyset\}$
10:         **end if**
11:         **if** $Q(n_i) \neq \emptyset$ **then**
12:             add n_i into R
13:         **end if**
14:     **end for**
15:     **for** each node $n_i \in R$ in a bottom-up fashion **do**
16:         **for** $q_j \in Q(n_i)$ **do**
17:             Compute $icost(n_i, q)$
18:             Compute $ecost(n_i, q)$
19:             Compute the leaf access plan, $E_{i,j}$, as
                $E_{i,j} = [\texttt{ecost}(\texttt{n}_\texttt{i}, \texttt{q}_\texttt{j}) < \texttt{icost}(\texttt{n}_\texttt{i}, \texttt{q}_\texttt{j})]$
20:         **end for**
21:         Compute the leaf access cost, $leaf\_cost_i$, as
        $\left(\sum_{q_j \in Q(n_i)} E_{i,j} \times ecost(n_i, q_j) + (1 - E_{i,j}) \times icost(n_i, q_j) \right)$
22:     **end for**
23: **end procedure**

---

also converts the range in 2-D space into 1-D space by identifying the relevant nodes in the hierarchy. Next, for each internal node, $n_i \in R$, a cost, $cost_i$, is estimated assuming that this node and its leaf descendants are used for identifying the matches in the range S_i. The outline of this process is presented in Algorithm 2 and is detailed below:

• *Top-Down Traversal and Pruning.* Line 5 indicates that the process starts at the root and moves towards the leaves. For each internal node, n_i, being visited, first, the set, $Q(n_i) \subseteq Q$, of queries for which n_i is relevant is identified by intersecting the ranges of the queries relevant to the parent (i.e., $Q(parent(n_i))$) with the range of n_i. More specifically,

$$Q(n_i) = \{q \in Q(parent(n_i)) \ s.t. \ (q.rs \cap S_i) \neq \emptyset\}.$$

If $Q(n_i) = \emptyset$, then n_i and all its descendants are ignored, otherwise n_i is included in the set R.
• *Inclusive and Exclusive Cost Computation.* Once the portion, R, of the hierarchy relevant to the query workload is identified, next, the algorithm re-visits all internal nodes in R in a bottom-up manner and computes a cost estimate for executing queries in $Q(n_i)$: for each query, $q \in Q(n_i)$, the algorithm computes inclusive and exclusive leaf access costs:

• *Inclusive leaf access plan (Line 17):* If query, q, is executed using an inclusive plan at node, n_i, this means that the result for the range $(q.rs \cap S_i)$ will be obtained by identifying and combining (using bitwise ORs) all relevant leaf bitmaps under node n_i. Therefore, the cost of this leaf access plan is

$$icost(n_i, q) = \sum_{(n_j \in leafDesc(n_i)) \wedge ((q.rs \cap S_j) \neq \emptyset)} size(B_j).$$

This value can be computed incrementally, simply by summing up the inclusive costs of the children of n_i.

• *Exclusive leaf access plan (Line 18):* If query, q, is executed using an exclusive leaf access plan at node, n_i, this means that the result for the range $(q.rs \cap S_i)$ will be obtained by using B_i and then identifying and excluding (using bitwise ANDNOT operations) all irrelevant leaf bitmaps under node n_i. Thus, we compute the exclusive leaf access plan cost, $ecost(n_i, q)$, of this query at node n_i as

$$
\begin{aligned}
ecost(n_i, q) = \ & size(B_i) \\
& + \sum_{(n_j \in leafDesc(n_i)) \wedge ((q.rs \cap S_j) = \emptyset)} size(B_j)
\end{aligned}
$$

or equivalently as

$$
\begin{aligned}
ecost(n_i, q) = \ & size(B_i) + \left(\sum_{n_j \in leafDesc(n_i)} size(B_j) \right) \\
& - icost(n_i, q)
\end{aligned}
$$

Since the initial two terms above are recorded in the index creation time, the computation of exclusive cost is a constant time operation.

• *Overall Cost Estimation and the Leaf Access Plan.* Given the above, we can find the best strategy for processing the query set $Q(n_i)$ at node n_i by considering the overall estimated cost term, $cost(n_i, Q(n_i))$, defined as

$$\underbrace{\left(\sum_{q_j \in Q(n_i)} E_{i,j} \times ecost(n_i, q_j) + (1 - E_{i,j}) \times icost(n_i, q_j) \right)}_{leaf\ access\ cost\ for\ all\ relevant\ queries}$$

where $E_{i,j} = 1$ means an exclusive leaf access plan is chosen for query, q_j, at this node and $E_{i,j} = 0$ otherwise.

Cut Bitmap Selection.

Once the nodes in the hierarchy are assigned estimated costs as described above, the cut that will be used for query processing is found by traversing the hierarchy in a bottom-up fashion and picking nodes based on their estimated costs[4]. The process is outlined in Algorithm 3. Intuitively, for each internal node, $n_i \in I_H$, the algorithm computes a revised cost estimate, $rcost_i$, by comparing the cost, $cost_i$, estimated in the earlier phase of the process, with the total revised costs of n_i's children:

• In Line 13, the function $findBlockIO(n_i)$ returns the cost of reading the block $T(B_i)$. If this block has already been marked "to-read", then the reading cost has already been accounted for, so the cost is zero. Otherwise, the cost is equal to the size of the block $T(B_i)$, as explained in Section 4.2.1.

• As we see in Line 21, it is possible that a block T is first marked "to-read" and then, later in the process, marked "not-to-read", because for the corresponding nodes in the cut, more suitable ancestors are found and the block is no longer needed.

• If $cost_i$ is smaller (Line 17), then n_i and its leaf descendants can be used for identifying the matches to the queries in the range S_i. In this case, no revision is

---

[4]Note that this bottom-up traversal can be combined with the bottom-up traversal of the prior phase. We are describing them as separate processes for clarity.

Algorithm 3 Cut Selection Algorithm

1: **Input:** Hierarchy H; per-node query workload $Q(n_i)$; per-node cost estimates $cost_i$; and the corresponding leaf access plans, $E_{i,j}$, for node/query pairs $n_i \in H$ and $q_j \in Q(n_i)$; the set, $R \subseteq I_H$, or relevant internal nodes
2: **Output:** All-inclusive, C_I, and Exclusive, C_E, cut nodes
3: **Initialize:** $Cand = \emptyset$
4: **procedure** FINDCUT
5:     **for** each relevant internal node n_i in R in a bottom-
6:     up fashion **do**
7:         Set $internal\_children = children(n_i) \cap I_H$;
8:         **if** $internal\_children = \emptyset$ **then**
9:             add n_i to $Cand$;
10:             $rcost_i = cost_i$
11:         **else**
12:             $costChildren = \sum_{n_j \in internal\_children} rcost_j$
13:             $rcostIO_i = findBlockIO(n_i)$
14:             **for** each child n_j in $internal\_children$ **do**
15:                 $costChildrenIO = costChildrenIO + findBlockIO(n_j)$
16:             **end for**
17:             **if** $(rcost_i + rcostIO_i) \leq (costChildren + costChildrenIO)$ **then**
18:                 **for** each descendant n_k of n_i in $Cand$ **do**
19:                     remove n_k from $Cand$;
20:                     **if** n_k is the only node to read from $T(B_k)$ **then**
21:                         mark $T(B_k)$ as "not-to-read";
22:                     **end if**
23:                 **end for**
24:                 add n_i to $Cand$;
25:                 $rcost_i = cost_i$
26:                 mark $T(B_i)$ as "to-read";
27:             **else**
28:                $rcost_i = costChildren$
29:             **end if**
30:         **end if**
31:     **end for**
32:     $C_E = \{n_i \in Cand \text{ s.t. } \exists_{q_j \in Q(n_i)} E_{i,j} == 1\}$
33:     $C_I = Cand/C_E$
34: **end procedure**

---

necessary and the revised cost, $rcost_i$ is equal to $cost_i$. Any descendants of n_i are removed from the set, $Cand$, of cut candidates and n_i is inserted instead.

- If, on the other hand, the total revised cost of n_i's children is smaller than $cost_i$, then matches to the queries in the range S_i can be more cheaply identified by considering the descendants of n_i, rather than n_i itself (Line 27). Consequently, in this case, the revised cost, $rcost_i$, is set to

$$rcost_i = \sum_{n_j \in children(n_i)} rcost_j.$$

As we experimentally show in Section 5, the above process has a small cost. This is primarily because, during bottom-up traversal, only those nodes that have not been pruned in the previous top-down phase are considered. Once the traversal is over, the nodes in the set, $Cand$, of cut candidates are reconsidered and those that include exclusive leaf access plans are included in the exclusive cut set, C_E, and the rest are included in the all-inclusive cut set, C_I.

Caching of Cut and Leaf Bitmaps.

During query execution, the bitmaps of the nodes in C_E are read into a cut bitmaps buffer, whereas the bitmaps for the nodes in C_I do not need to be read as the queries will be answered only by accessing relevant leaves under the nodes in C_I. We store the blocks containing the bitmaps of these relevant leaves in an LRU-based cache so that leaf bitmaps can be reused by multiple queries.

4.3.3 Complexity

The bitmap selection process consists of two steps: (a) a per-node cost estimation step and (b) a cut bitmap selection step. Each of these steps visit only the *relevant* nodes of the hierarchy. Therefore, if we denote the set of nodes of the hierarchy, H, that intersect with any query in Q, as $H(Q)$, then the overall work is linear in the size of $H(Q)$.

During the cost estimation phase, for each visited node, n_i, an inclusive and exclusive cost is estimated for any query that intersects with this node. Therefore, the worst case time cost of the overall process (assuming that all queries in Q intersect with all nodes in $H(Q)$) is $O(|Q| \times |H(Q)|)$.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness of the proposed *compressed spatial hierarchical bitmap (cSHB)* index structure using spatial data sets with different characteristics, under different system parameters. To assess the effectiveness of cSHB, we also compare it against alternatives.

We ran the experiments on a quad-core Intel Core i5-2400 CPU @ 3.10GHz machine with 8.00GB RAM, and a 3TB SATA Hard Drive with 7200 RPM and 64MB Buffer Size, and in the same Windows 7 environment. All codes were implemented and run using Java v1.7.

5.1 Alternative Spatial Index Structures and the Details of the cSHB Implementation

As alternatives to cSHB, we considered different systems operating based on different spatial indexing paradigms. In particular, we considered spatial extensions of PostgreSQL called PostGIS [2], of a widely used commercial DBMS (which we refer to as DBMS-X), and of Lucene [1]:

- PostGIS [2] creates spatial index structures using an R-tree index implemented on top of GiST.

- DBMS-X maps 2D space into a 1D space using a variation of Hilbert space filling curve and then indexes the data using B-trees.

- Apache Lucene [1,18], a leading system for text indexing and search, provides a spatial module that supports geo-spatial range queries in 2D space using quadtrees and prefix-based indexing. Intuitively, the space is partitioned using a MX-quadtree structure (where all the leaves are at the same level and a given region is always partitioned to its quadrants at the center [28]) and each root-to-leaf path is given a unique path-string. These path-strings are then indexed (using efficient prefix-indexing algorithms) for spatial query processing.

Since database systems potentially have overheads beyond pure query processing needs, we also considered disk-based implementations of R*-tree [8] and the Hilbert R-tree [19]. For this purpose, we used the popular XXL Java library [10]:

- A packed R*-tree, with average leaf node utilization $\sim 95\%$ (page size 4MB).

- A packed Hilbert R-tree, with average leaf node utilization $\sim 99\%$ (page size 4MB).

Table 1: Data sets and clustering

Data set	#points	#points per (non-empty) cell ($h = 10$)		
		Min.	Avg.	Max.
Synthetic (Uniform)	100M	54	95	143
Gowalla (Clustered)	6.4M	1	352	312944
OSM (Clustered)	688M	1	3422	1.2M
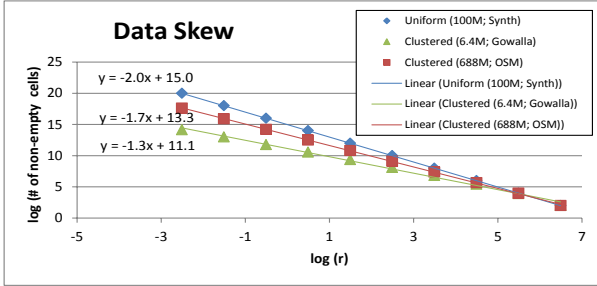
Figure 6: Data Skew

Table 2: Parameters and default values (in bold)

Parameter	Value range
Block Size (MB)	0.5; **1**; 2.5; 5; 10
Query range size	0.5% **1%**; 5%
$|Q|$	100; **500**; 1000
h	9; **10**; 11
Buffer size (MB)	2; 3; 5; 10; **20**; 100

We also implemented the proposed cSHB index structure on top of Lucene. In particular, we used the MX-quadtree hierarchy created by Lucene as the spatial hierarchy for building cSHB. We also leveraged Lucene's (Java-based) region comparison libraries to implement range searches. The compressed bitmaps and compressed domain logical operations were implemented using the JavaEWAH library [21]. Due to space limitations, we only present results with the incremental strategy for query evaluation.

5.2 Data Sets

For our experiments, we used three data sets: *(a) a uniformly distributed* data set that consists of 100 million synthetically generated data points. These points are mapped to the range $\langle -180, -90 \rangle$ to $\langle 180, 90 \rangle$, *(b) a clustered* data set from Gowalla, which contains the locations of check-ins made by users. This data set is downloaded from the Standford Large Network Dataset Collection [4], and *(c) a clustered* data set from OpenStreetMap (OSM) [3] which contains locations of different entities distributed across North America. The OSM data set consists of approximately 688 million data points in North America. We also normalized both the real data sets to the range $\langle -180, -90 \rangle$ to $\langle 180, 90 \rangle$. In order to obtain a fair comparison across all index structures and the data sets, all three data sets are mapped onto a $2^h \times 2^h$ space and the positions of the points in this space are used for indexing. Table 1 provides an overview of the characteristics of these three very different data sets. Figure 6 re-confirms the data skew in the three data sets using the box-counting method proposed in [9]: in the figure, the lower the negative slope, the more skewed the data. The figure shows that the clustered Gowalla data set has the largest skew.

5.3 Evaluation Criteria and Parameters

We evaluate the effectiveness of the proposed *compressed spatial hierarchical bitmap (cSHB)* index structure by com-

Table 3: Index Creation Time (sec.)

Data set	cSHB	Luc.	DBMS-X	Post GIS	R*-tree	Hilb. R-tree
Synthetic	1601	2396	3865	4606	2160	2139
Gowalla	24	114	232	112	22	20
OSM	2869	12027	30002	76238	18466	17511

Table 4: Index Size on Disk (MB)

Data set	cSHB	Luc.	DBMS-X	Post GIS	R*-tree	Hilb. R-tree
Synthetic	10900	5190	1882	8076	3210	1510
Gowalla	44	220	121	600	211	100
OSM	2440	22200	12959	61440	22100	10400

paring its *(a)* index creation time, *(b)* index size, and *(c)* query processing time to those of the alternative index structures described above under different parameter settings. Table 2 describes the parameters considered in these experiments and the default parameter settings.

Since our goal is to assess the contribution of the index in the cost of the query plans, all index structures in our comparison used index-only query plans. More specifically, we executed a count(∗) query and configured the index structures such that only the index is used to identify the relevant entries and count them to return the results. Consequently, only the index files are used and data files are not accessed.

Note that all considered index structures accept square-shaped query ranges. The range sizes indicated in Table 2 are the lengths of the boundaries relative to the size of the considered 2D space. These query ranges in the query workloads are generated uniformly.

5.4 Discussion of the Indexing Results

Indexing Time. Table 3 shows the index creation times for different systems and index structures, for different data sets (with different sizes and uniformity): cSHB index creation is fastest for the larger Synthetic and OSM data sets, and competitive for the smaller Gowalla data set. As the data size gets larger, the alternative index structures become significantly slower, whereas cSHB is minimally affected by the increase in data size. The index creation time also includes the time spent on creating the hierarchy for cSHB.

Index Size. Table 4 shows the sizes of the resulting index files for different systems and index structures and for different data sets. As we see here, cSHB provides a competitive index size for uniform data (where compression is not very effective). On the other hand, on clustered data, cSHB provides very significant gains in index size – in fact, even though the clustered data set, OSM, contains more points, cSHB requires less space for indexing this data set than it does for indexing the uniform data set.

Impact of Block Size. As we discussed in Section 3.2.2, cSHB writes data on the disk in a blocked manner. In Figure 7, we see the impact of the block sizes on the time needed to create the bitmaps. As we see here, one advantage of using blocked storage is that the larger the blocks used, the faster the index creation becomes.

5.5 Discussion of the Search Results

Impact of the Search Range. Table 5 shows the impact of the query range on search times for 500 queries under the default parameter settings, for different systems. As we expected, as the search range increases, the execution time
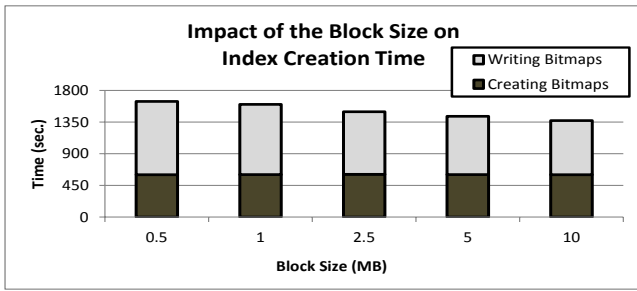
Figure 7: Impact of the block size on index creation time of cSHB (uniform data set)

Table 5: Comparison of search times for alternative schemes and impact of the search range on the time to execute 500 range queries (sec.)

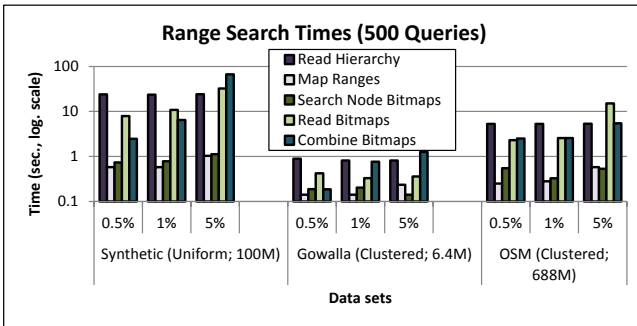| Range	cSHB	Luc.	DBMS-X	Post GIS	R*-tree	Hilb. R-tree	cSHB-LO
Synthetic (Uniform; 100M)							
0.5%	35	123	414	12887	2211	4391	52
1%	42	131	345	28736	2329	4480	59
5%	137	187	368	72005	2535	4881	1700
Gowalla (Clustered; 6.4M)							
0.5%	2	2	24	19	8	24	2
1%	3	3	29	34	11	26	3
5%	3	48	37	194	20	45	5
OSM (Clustered; 688M)							
0.5%	13	23	303	1129	3486	4368	13
1%	15	30	645	4117	3889	5599	14
5%	28	66	15567	18172	4626	6402	78

Figure 8: cSHB execution breakdown

becomes larger for all alternatives. However, cSHB provides the best performance for all ranges considered, especially for the clustered data sets. Here, we also compare cSHB with its leaf-only version (called cSHB-LO), where instead of a cut consisting of potentially internal nodes, we only choose the leaf nodes for query processing. As you can see from the figure, while cSHB-LO is a good option for very small query ranges (0.5% and 1%), it becomes very slow as the query range increases (since the number of bitwise operations increases, and it is not able to benefit from clustering).

Execution Time Breakdown. Figure 8 provides a breakdown of the various components of cSHB index search (for 500 queries under the default parameter settings): The bitmap selection algorithm presented in Section 4.3 is extremely fast. In fact, the most significant components of the execution are the times needed for reading the hierarchy into memory[5], and for fetching the selected bitmaps from the disk into the buffer, and performing bitwise operations on them. As expected, this component sees a major increase

---

[5]Once a hierarchy is read into the memory, the hierarchy does not need to be re-read for the following queries.
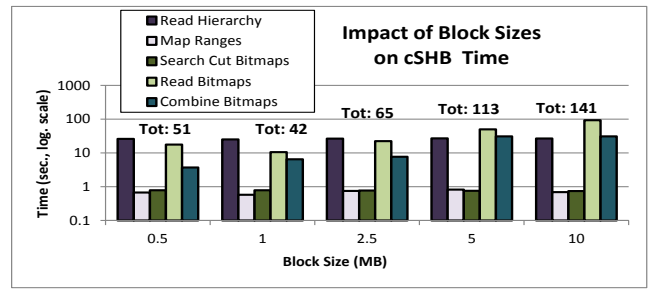
(a) Impact of block size on overall cSHB execution time
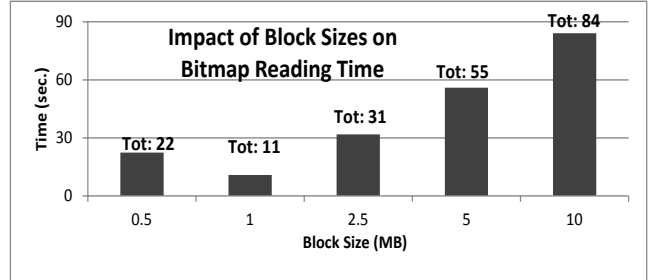
(b) Impact of block size on bitmap reading time

Figure 9: Impact of the block size (500 queries, 1% q. range, uniform data)
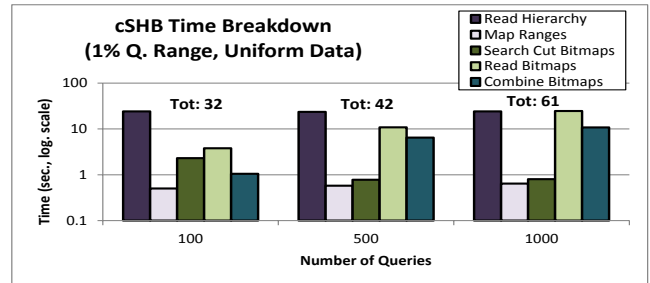

Figure 10: Impact of the number of queries on the execution time of cSHB (1% q. range, uniform data)

as the search range grows, whereas the other costs are more or less independent of the sizes of the query ranges.

Impact of the Block Sizes. As we see above, reading bitmaps from the disk and operating on them is a major part of cSHB query execution cost; therefore these need to be performed as efficiently as possible. As we discussed in Section 3.2.2, cSHB reads data from the disk in a blocked manner. In Figure 9, we see the impact of the block sizes on the execution time of cSHB, including the time needed to read bitmaps from the disk. As we see here, small blocks are disadvantageous (due to the directory management overhead they cause). Very large blocks are also disadvantageous as, the larger the block gets, the larger becomes the amount of redundant data read for each block access. As we see in the figure, for the configuration considered in the experiments, 1MB blocks provided the best execution time.

Impact of the Number of Queries in the Workload. Figure 10 shows the total execution times as well as the breakdown of the execution times for cSHB for different number of simultaneously executing queries. While the total execution time increases with the number of simultaneous queries, the increase is sub-linear, indicating that there are savings due to the shared processing across these queries.
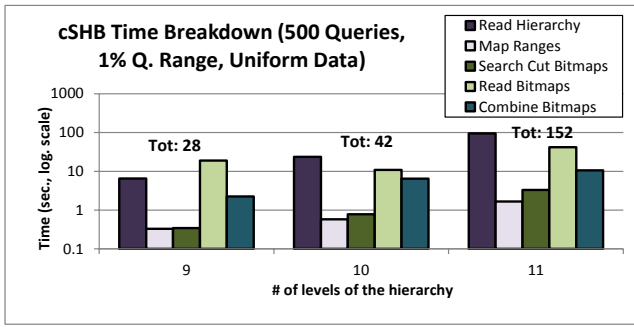
Figure 11: Impact of the depth of the hierarchy (500 queries, 1% query range, uniform data)

Table 6: Working set size in terms of 1MB blocks

Q.Range (on 100M data)	Min	Avg.	Max.
0.5%	1	2.82	36
1%	1	2.51	178
5%	1	1.02	95

Table 7: Impact of the buffer size on exec. time (in seconds, for 500 queries, 100M data)

Query Range	Buffer Size					
	2MB	3MB	5MB	10MB	20MB	100MB
0.5%	11.8	11.3	10.9	10.6	10.5	10.2
1%	24.2	19.1	18.1	17.5	17.3	16.3
5%	823.8	399.9	155.9	105.8	101.6	94.9

Also, in Section 4.2.2, we had observed that the number of queries in the query workload and query ranges determine the relative costs of in-buffer operations vs. disk I/O. In Figures 8 and 10, we see that this is indeed the case.

Impact of the Depth of the Hierarchy. Figure 11 shows the impact of the hierarchy depth on the execution time of cSHB: a $4\times$ increase in the number of cells in the space (due to a 1-level increase in the number of levels of the hierarchy) results in $< 4\times$ increase in the execution time. Most significant contributors to this increase are the time needed to read the hierarchy and the time for bitmap operations.

Impact of the Cache Buffer. As we discussed in Section 4.2.2, the incremental scheduling algorithm keeps a buffer of blocks containing the working set of leaf bitmaps. As Table 6 shows, the average size of the working set is fairly small and can easily fit into the L3 caches of modern hardware. Table 7 confirms that a small buffer, moderately larger than the average working set size, is sufficient and larger buffers do not provide significant gains.

6. CONCLUSIONS

In this paper, we argued that bitmap-based indexing can be highly effective for running range query workloads on spatial data sets. We introduced a novel *compressed spatial hierarchical bitmap (cSHB)* index structure that takes a spatial hierarchy and uses that to create a hierarchy of compressed bitmaps to support spatial range queries. Queries are processed on cSHB index structure by selecting a relevant subset of the bitmaps and performing *compressed-domain* bitwise logical operations. We also developed bitmap selection algorithms that identify the subset of the bitmap files in this hierarchy for processing a given spatial range query workload. Experiments showed that the proposed cSHB index structure is highly efficient in supporting spatial range query workloads. Our future work will include implementing and evaluating cSHB for data with more than two dimensions.

7. REFERENCES

[1] Apache Lucene. http://lucene.apache.org/core/4_6_0/ spatial/org/apache/lucene/spatial/prefix/tree/ SpatialPrefixTree.html
[2] Using PostGIS: Data Management and Queries. http://postgis.net/docs/using_postgis_dbmanagement.html
[3] OpenStreetMap. http://www.openstreetmap.org/
[4] J. Leskovec and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data
[5] D. Abadi *et al.* Compression and Execution in Column-Oriented Database systems. SIGMOD 2006.
[6] A. Aji *et al.* Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. PVLDB 2013.
[7] Rudolf Bayer. The Universal B-tree for Multidimensional Indexing: General concepts. WWCA 1997.
[8] N. Beckmann *et al.* The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD 1990.
[9] A. Belussi and C. Faloutsos. Self-Spatial Join Selectivity Estimation Using Fractal Concepts. TOIS 1998.
[10] J.Bercken *et al.* XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. VLDB 2001.
[11] A.R. Butz. Alternative Algorithm for Hilbert's Space-Filling Curve. TOC 1971.
[12] A. Cary *et al.* Experiences on Processing Spatial Data with MapReduce. SSDBM 2009.
[13] J. Chmiel *et al.* Dimension Hierarchies by means of Hierarchically Organized Bitmaps. DOLAP 2010.
[14] P. Chovanec and M. Krátký. On the Efficiency of Multiple Range Query Processing in Multidimensional Data Structures. IDEAS 2013.
[15] F. Deliège and T. Pedersen. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. EDBT 2010.
[16] L.Gosink *et al.* HDF5-FastQuery: Accelerating Complex Queries on HDF Datasets using Fast Bitmap Indices. SSDM 2006.
[17] David Hilbert. Ueber stetige abbildung einer linie auf ein flachenstuck. Mathematische Annalen 1891.
[18] Y. Jing *et al.* An Empirical Study on Performance Comparison of Lucene and Relational Database. ICCSN 2009.
[19] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. VLDB 1994
[20] O. Kaser *et al.* Histogram-Aware Sorting for Enhanced Word-Aligned Compression in Bitmap Indexes. DOLAP 2008.
[21] D. Lemire *et al.* Sorting improves Word-Aligned Bitmap Indexes. DKE 2010.
[22] V. Markl *et al.* Improving OLAP Performance by Multidimensional Hierarchical Clustering. IDEAS 1999.
[23] G. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. IBM 1966.
[24] M. Morzy *et al.* Scalable Indexing Technique for Set-Valued Attributes. ADBIS 2003.
[25] P. Nagarkar and K. S. Candan. HCS:Hierarchical Cut Selection for Efficiently Processing Queries on Data Columns using Hierarchical Bitmap Indices. EDBT 2014.
[26] M. A. Olma *et al.* BLOCK: Efficient Execution of Spatial Range Queries in Main-Memory. Technical report EPFL 2013.
[27] A. N. Papadopoulos and Y. Manolopoulos. Multiple Range Query Optimization in Spatial Databases. ADBIS 1998.
[28] H. Samet. Foundations of Multidimensional and Metric Data Structures, 2005.
[29] R. Sinha and M. Winslett. Multi-Resolution Bitmap Indexes for Scientific Data. TODS 2007.
[30] T. Siqueira *et al.* The SB-index and the HSB-index: Efficient Indices for Spatial Data Warehouses. Geoinformatica 2012.
[31] T. Skopal *et al.* Algorithm for Universal B-trees. Inf. Syst. 2006.
[32] K. Wu *et al.* On the Performance of Bitmap Indices for High Cardinality Attributes. VLDB 2004.
[33] K. Wu *et al.* An Efficient Compression Scheme for Bitmap Indices. TODS 2004.
[34] M. Zaker *et al.* An Adequate Design for Large Data Warehouse Systems: Bitmap Index versus B-tree Index. IJCC 2008.
[35] Y. Zhong *et al.* Towards Parallel Spatial Query Processing for Big Spatial Data. IPDPSW 2012.