BLOCK: Efficient Execution of Spatial Range Queries in Main-Memory

Matthaios Olma†, Farhan Tauheed††, Thomas Heinis†, Anastasia Ailamaki†
†Data-Intensive Applications and Systems Lab, École Polytechnique Fédérale de Lausanne, Switzerland
‡Brain Mind Institute, École Polytechnique Fédérale de Lausanne, Switzerland

Abstract—The execution of spatial range queries is at the core of many applications, particularly in the simulation sciences but also in many other domains. Although main memory in desktop and supercomputers alike has grown considerably in recent years, most spatial indexes supporting the efficient execution of range queries are still mostly optimized for disk access (minimizing disk pages read). Recent research has primarily focused on the optimization of known disk-based approaches for memory (through cache alignment, quantization, compression etc.) but has not fundamentally redesigned the index structures for memory.

In this paper we develop BLOCK, a novel approach to execute range queries on spatial data in main memory. Our approach is built on the key insight, that in-memory approaches need to be designed to reduce the number of intersection tests (between objects and query but also in the index structure) instead of reducing the data read. Our experimental results show that BLOCK outperforms known in-memory as well as in-memory implementations of disk-based spatial indexes between a factor of 2 and 4. The experiments show that it is more scalable than competing approaches as the data sets become more dense.

I. INTRODUCTION

For many applications in a broad range of domains the efficient execution of spatial range queries is pivotal to extract subsets of the spatial model. Executing range queries quickly is of particular importance in the simulation sciences where it is used to build, analyze and visualize models through the repeated execution of range queries or to analyze the spatial component of the simulation output (to run statistical tests etc.). Examples include geographical information systems, where range queries are used to extract subsets of geographical features [1], i.e., landmarks, houses, roads, etc., medical imaging where brain regions are retrieved from MRIs using range queries, computational neuroscience where parts of the spatial brain models [2] are retrieved and many other applications.

Main memory in desktop, server systems and also in supercomputers has grown considerably in recent years and many spatial datasets fit comfortably into it. For scientists in many domains this means that they can efficiently analyze spatial models loaded into the main memory of their desktop or server systems. More importantly, however, it means that scientists can simulate bigger spatial models stored in the main memory of their simulation infrastructure (clusters or supercomputers). In either case, for the purpose of analysis and visualization, spatial queries (primarily range queries) need to be executed to retrieve parts of the model from main memory.

To define the problem, for the purpose of analysis and visualization, subsets of spatial models residing in main memory need to be retrieved. More formally, given a dataset $D$ containing volumetric objects $d \in D$, the result $R$ of executing an axis-aligned range query $q$ defined as a three dimensional interval $q = [l_1, u_1] \times [l_2, u_2] \times [l_3, u_3]$ will be $R(q) = d \mid q \cap d \neq \emptyset$. As most approaches to spatial indexing [3], also we assume that all objects $d$ can be approximated with an axis-aligned minimum bounding box.

The numerous approaches developed for spatial indexing in the past [3] have primarily been developed for disk and are mainly optimized to minimize the pages read from disk. For approaches on disk, the disk latency effectively hid the cost of computation. In memory, however, the cost of computation is now fully exposed because retrieving data from memory is substantially faster. Using disk based approaches in memory will still reduce the amount of data read and will still speed up the execution of range queries. In this paper, however, we argue that in memory the cost of query execution has shifted and that testing objects for intersection with the query range as well as traversing hierarchical index structures now dominates overall cost.

We consequently argue that an in-memory approach has to avoid complex hierarchical index structures and needs to minimize the number of intersection tests and design BLOCK, our new approach, accordingly. By using space-oriented partitioning, BLOCK reduces intersection tests by maximizing the number of objects that can be read from the index without testing for intersection. Using a grid, BLOCK does not need to test objects in cells that are entirely contained in the range query.

BLOCK is implemented as multiple levels of uniform grids, each with a different resolution and each indexing the dataset. The grid is used to identify the largest grid cell enclosed inside the query region and therefore enables retrieval of the maximum number of objects within the query without intersection tests. To reduce the memory overhead of using a grid (a space-oriented approach prone to object duplication), we develop an object store where objects are stored only once
and couple it with a z-order offset based pointer compression scheme.

Testing BLOCK on synthetic data shows that it outperforms existing spatial indexes in terms of number of intersection tests and execution time when executing range queries. Experiments on synthetic datasets show an improvement over $2 \times$ to $4 \times$ over the fastest existing approach.

The remainder of this paper is organized as follows. In section Section II we review related work for memory- as well as disk-based spatial indexing methods. In Section III we motivate our approach with an initial set of measurements confirming our assumptions and explain our approach in detail in Section IV. We demonstrate the benefits of our approach with an extensive set of measurements in Section V, analyze different configurations of BLOCK in more detail in Section VI and draw conclusions in Section VII.

II. RELATED WORK

Decades of research in spatial indexing have produced numerous index structures for the efficient execution of spatial range queries on disk but only few in memory [3]. Many disk-based spatial join algorithms, however, can also be used in memory and in the following we therefore discuss both types. We classify them as data- or space-oriented indexes where the former are mostly used on disk and the latter in memory.

In the following we first discuss access methods based on data-oriented partitioning and then those based on space-oriented partitioning.

A. Space-oriented Partitioning

Indexes based on space-oriented partitioning typically decompose hyperspace independent of the distribution of the data. Doing so makes the indexing process comparatively fast (as the computation assigning the object to a partition is very fast) but has two major disadvantages: (a) if volumetric objects are indexed, objects (or their reference) intersecting several partitions need to be replicated and (b) skew in the dataset may lead to an uneven (and sometimes extreme) distribution of objects to partitions. When used on disk, the object replication leads to a bigger index and to random access on disk. Similarly, extreme distributions can lead to unevenly filled disk pages as well as potentially unbalanced trees (if organized hierarchically). Both effects, replication and uneven distribution, are detrimental to query execution performance on disk and consequently space-oriented indexes are mainly used in memory.

The KD-Tree [4] and the Quadtree [5], [6] (along with its variant for 3D space, the Octree [7]) are today predominantly used in memory. All three recursively partition space, the Quad- and Octree split and overflowing cell in the (spatial) middle whereas the KD-Tree splits the cell so that the resulting two cells contain the same number of objects. Either way to split, however, leads to an unbalanced tree and to unevenly filled cells. Furthermore, to index volumetric objects, the object or a reference to it has to be replicated to all partitions the object intersects with. All three indexes are broadly used (particularly for computer graphics and simulation applications) mostly because of their rather simple implementation.

The UB-Tree [8] uses space-oriented partitioning but manages to build a balanced tree: it sorts points according to the Z-order [9] and builds a B+-Tree on the points. To execute a range query, it finds the intersection of the Z-order curve with the range query and scans through all a z-values inside the range. Calculating the next Z-value within the range, however, is a very costly operation that considerably degrades performance of the UB-Tree.

A much simpler space-oriented index is the grid [10] used for static but also moving objects datasets [11]. Instead of building a hierarchy of decomposed space, the grid defines a uniform grid in space and assigns each object to all partitions it intersects with. Doing so speeds up the indexing process but choosing the proper resolution of the grid is difficult, for example, if the resolution is chosen too fine grained, each object will be replicated to numerous partitions.

To store data on disk using space-oriented partitioning, the grid file [12] defines a grid with cells of different size (but not overlapping). The objects in each cell are stored in one disk page. If a cell overflows (i.e., its disk pages overflows), it is split into two and if the cell underflows, one or several cells are stored on the same disk page. Because the cells are not uniform, a directory mapping cell to disk pages needs to be maintained in memory. The disadvantage of the grid file is the superlinear growth of the directory: one disk pages may require several directory entries if the cells are sparsely filled. For improved memory efficiency improvements over the initial design of the grid file were introduced by the twin grid file [13] which solves partly the constant overflows but does not offer a completely scalable solution.

The KDB-Tree [14] and the Bkd-Tree [15] are both based on the KD-Tree [4], a space-oriented partitioning index typically used in memory. Both are designed to maximize space utilization, i.e., filling disk pages as much as possible. Either approach, however, is only designed to index point datasets and cannot be used for volumetric objects.

B. Data-oriented Partitioning

Access methods based on data-oriented partitioning, decompose the hyperspace into partitions such that each partition contains approximately the same number of objects. This lends itself perfectly for use on disk as the number of objects in a partitions can be chosen to fill disk pages. Because the decomposition of space, however, is irregular in all dimensions, a directory structure needs to be used to execute range queries. Several data-oriented indexing approaches use a hierarchical organization of the directory structure leading to the problem of overlap.
Based on data-oriented partitioning, arguably the seminal data structure developed for the execution of spatial range queries is the Rtree [16]. Widely used today, the Rtree has been designed as a disk-based multidimensional generalization of the B-Tree [17]. The Rtree packs the objects on disk pages using data-oriented partitioning and organizes the data in a hierarchical structure to enable range query execution. Introducing a hierarchical tree structure, however, also introduces the problem of overlap that considerably degrades Rtree performance.

Several strategies have been devised to tackle the overlap problem. The R*-Tree [18] and the R+Tree [19] have been designed for the case where the tree is built by inserting objects consecutively. The former addresses overlap through an improved node split algorithm and the latter through object duplication.

To reduce overlap, several packing methods for the Rtree have been proposed to bulkload data sets which are known a priori. The Hilbert Rtree [20], Sort-Tile-Recursive (STR) [21], Top down Greedy Split (TGS) [22] as well as the Priority Rtree (PRtree) [23] all use different sort and split strategies to build an Rtree from the dataset. While Hilbert and STR build efficient Rtrees on many real-world datasets, TGS and PRtree do so for extreme, mostly synthetic datasets (extreme skew and aspect ratio).

Despite the numerous improvements and approaches enhancing the Rtree, however, the basic problems of overlap and dead space remain the same.

The CRtree [24] essentially is an optimized Rtree for the memory hierarchy. The optimizations target at reducing cache misses (through alignment of the nodes for the cache line) and at reducing the index size (through quantizing the MBRs as well as compressing the nodes). The optimizations improve performance by a factor of 2.5 over the regular Rtree (and require 60% less space), but do little to address the problem of overlap (in fact, quantization of the MBRs leads to more overlap).

A more extensive evaluation [25] compared several variants of the unoptimized Rtree (R*-Tree, Hilbert Rtree and others) in memory to the CRtree. The CRtree generally outperformed other variants on range queries.

III. MOTIVATION

The development of BLOCK is driven by the requirements of the computational neuroscientists [2] we collaborate with. In order to better understand how the brain works, the neuroscientists simulate biophysically realistic models of a neocortical column on the molecular level on their desktops, but more importantly on clusters and the BlueGene/Q with 16K CPUs. To analyze the simulations during the simulation or after, they need to execute numerous spatial range queries on the spatial model in memory. Efficient of the range queries is key to speed up the analysis.

As outlined before, both, space- as well as data-oriented have issues that considerably deteriorate query execution performance. Disk-oriented approaches based on the the Rtree suffer from the problem of overlap and dead space. A brief experiment with the an in-memory Rtree (nodes size 100K objects, fanout 111) indexing a dataset of 60 million objects (location uniformly distributed and object volume 5) demonstrates this impressively: the height of the resulting tree is 4, yet a point query at a random location reads 48 nodes. The point query is an excellent indication of overlap in a Rtree: the number of disk pages read to execute this query in an Rtree without overlap is equal to the height of the tree. The more nodes need to be read, the more overlap the Rtree has and the slower query execution is. The exact experimental setup is described in Section V (and is used throughout the paper). Avoiding the issue of overlap, space-oriented approaches like the KD-Tree or the Octree, however, do not perform substantially better as a further experiment with an Octree (50K objects per node, replicating objects to all intersecting partitions) indexing the same dataset shows. The height of the tree is at most 90, meaning that in the worst case 90 nodes need to be retrieved before the query result can be retrieved, considerably adding to the execution time of range queries.

Uniform grids avoid the hierarchical organization of space-oriented partitioning and are thus potentially very efficient. The difficulty of using a uniform grid, however, is finding the best possible cell width. If the grid is set too coarse (the queries are substantially smaller than the grid cells) too many unnecessary intersection tests (between query and objects) are needed to compute the result. If, on the other hand, the grid is set too fine-grained, then too much time is spent on calculations and retrieving many small cells.

The results of an experiment shown in Figure 1 with a grid shows this impressively. In this experiment we use the same datasets and a workload with 5 queries of size 500 space units per dimension. The grid granularity is the number of bits per coordinate used to represent the cell position in the grid. The bigger the number the smaller but more in number are the cell. If the cells are chosen too small, then too much time is spent on calculations. If on the other hand we chose the granularity too coarse (bigger cells), much time is wasted on unnecessarily testing objects for intersection with the query range. Finding the best granularity to minimize comparisons and computations is difficult particularly if the query size is not known a priori.

IV. APPROACH

To develop BLOCK, a novel index for the efficient execution of spatial range queries in memory, we draw inspiration from previous work, building on the benefits and issues of past work in space- as well as data-oriented spatial indexing.
A. Key Insights Inspiring BLOCK

Spatial indexes developed in the past for use on disk have primarily been optimized to reduce moving data from disk to memory because disk I/O is slow (compared to memory) and takes a considerable share of query execution. Today where entire datasets and indexes fit into memory, we can take advantage of the memory speed and proximity to CPU. We consequently argue that in-memory indexes no longer spend the majority of the query execution time moving data to the CPU, but do so traversing indexing structures as well as testing objects for intersection with the query.

The overhead of traversing index structures can be illustrated with examples from space- as well as data-oriented indexes. The unbalanced directory structure of space-oriented approaches have in face of skewed data slows down the execution of range queries considerably: too many nodes need to be read and tested, resulting on the one hand in excessive data read from memory, but more importantly, in many more intersection tests.

As the example of the CRtree shows, aligning the data structure for the cache line, quantization and compression only manages to reduce the execution time by a constant factor, but it does not fundamentally address the problem of overlap or imbalance (long access paths) some data- as well as space-oriented indexes suffer from. While a uniform grid (without any index structure) can suffer from data skew (some cells overly full while others are empty) this does not need to result in a problem: if the size of the cells is chosen appropriately most of the objects in the cell need to be tested for intersection anyway. Finally, because the partitions are quadratic in a uniform grid, no unnecessary intersection tests with objects in narrow partitions are necessary.

B. BLOCK Overview

The challenge of using a uniform grid to enable the efficient execution of range queries lies in finding the best configuration, i.e., the best resolution of the grid. The optimal resolution of a uniform grid depends on the size of the objects indexed, but primarily on the size of the queries executed. While the size of the objects can be known a priori and the grid can be configured accordingly, the size of the queries is typically not known beforehand.

Not choosing the right granularity for queries either means too many intersection tests or too much computational overhead. In case the grid is too coarse, i.e., the partitions are too big, too many objects are needlessly tested for intersection with the query. If, on the other hand, the granularity is too fine, computing what partitions are within the query range (particularly if using an order based on a space-filling curve) takes a considerable share of the query execution time.

To address the problem of a priori configuring the uniform grid, we use several uniform grids, each with a different resolution. Each object is stored in a central data structure keeping all objects and a reference to each object is indexed with every grid. Storing each object only once and indexing only references effectively reduces the memory footprint. At the same time, however, it makes data organization challenging: when retrieving a cell on a particular level, the objects should be stored together (no matter on what level) to ensure cache coherence. As we discuss later, we use the z-order to organize objects in memory, reducing the cache misses substantially.

To execute range queries, the grid with the coarsest resolution is first used to find all cells contained in the range query. After, the remaining grids are used in order of coarseness.
to find all the cells intersecting with the query while still limiting the amount of data read and, more importantly, while still ensuring that not too many unnecessary comparisons are performed.

Figure 3 illustrates how a spatial range query is executed with BLOCK on three different grids: a range query is executed on several uniform grids of different resolution. Query execution starts on the coarsest level and identifies through simple calculations to what degree cells overlap with the range query. Cells fully contained (or covered to some predefined degree) are retrieved from the current level. The remainder of the query is then executed (as several smaller queries) on more fine grained levels according to the same principles.

![Coarsest-grained Grid](image1.png) ![Finest-grained Grid](image2.png)

Fig. 3. A range query is executed on several uniform grids of different resolution.

### C. Index Structure

The index is built with $L$ levels where each level $l_i \in L$ has a resolution $r_i$, defined as the number of cells. A level $l_k$ is more fine-grained than level $l_j$ if it has a higher resolution, i.e., more cells and we say $j < k$ (consequently $l_0$ has the fewest cells). The granularity of each grid on every level can be chosen independently of each other as long as the grids use uniform space-partitioning ensuring that on two subsequent levels $j \& k$ (with $j < k$ and $k - j = 1$) each cell in $l_j$ will entirely cover several cells on $l_k$.

1) **Object Store:** To avoid replication of objects and to curb the memory overhead, each object is only stored once and on each level only a compressed pointer to the object is stored. BLOCK maintains two basic data structures to accomplish this: first the object store that holds all objects and second, for each level, one directory storing the pointer to the object.

The object store stores all objects sequentially in an array-like data structure. To preserve spatial locality and to improve cache coherence, the objects are stored in z-order [26], i.e., based on the z-value of their center. Storing objects according to their z-order in the object store also ensures spatial locality (and therefore cache coherence) across different levels: all objects with their center contained in any cell on any level are stored next to each other in the object store. While other orders like Hilbert [27] or Peano [9] could also be used, but we found the z-order to be the most efficient for coordinate to z-value transformations.

2) **Level Directory:** Each level has a directory that stores pointers to objects in the object store. The directory is organized according to the cells of each level, i.e., each cell $C_i$ on level $l$ has an entry that stores a containment list and an intersection list. The containment list contains pointers to all objects in the object store that have their center in the cell. Because the objects are stored next to each other in the object store (ordered on the z-value of their center), we can compress the containment list and only need to store the offset as well as the size, i.e., the number of objects which have their center in this cell.

By only storing pointers to the objects which have their center in the cell we have no guarantee to retrieve all objects intersecting with the cell. After all, we design our index for volumetric objects which may extend into neighboring cells without having the center in the neighboring cell. Consequently, we store in the entry of each cell also the intersection list which contains pointers to all objects intersecting with the cell (but not containing the center of the object). These objects are not stored consecutively in the object store (they may be stored in an entirely different place in the object store) and consequently we need to store pointers to individual objects in a linked list. If two (or more) objects in the list are stored adjacent, then we can still compress this into pairs of offset and size.

Figure 4 shows the data structures of BLOCK. Based on the dataset and the two levels each with a uniform grid, BLOCK builds the object store sorting the objects based on the z-order value of their center and then builds the directories based on the uniform grids, one directory per grid. The intersection list is shown with dashed lines whereas the containment list is illustrated with solid lines. The directory entries of the cells in this example are named/addressed by based on the z-value of the cells.

### D. Building the Index

To build the index structures for dataset $D$, BLOCK first organizes the object store: all objects $d \in D$ are sorted based on the z-value of their center and are stored in an array.

Following this, the directory of each level $l_i \in L$ is built. For this BLOCK iterates over the object store and tests for each object $d$ in which cell $c_i$ $d$’s center is and with what cells $C$ $d$ intersects. A pointer to the object is inserted into the intersection list of the directory entry of each cell $c \in C$. Furthermore, a pointer is inserted into the containment list of the directory entry of $c_i$. Because of their z-order in the object store, all objects in a cell are adjacent to each other in the store and we only have to store the offset and the size (number of objects) of this group, thereby effectively compressing the pointers.

Depending on the number of levels (as well as the size and number of the objects), indexing can take considerable time. Because the grids use uniform space-partitioning, i.e., a cell...
on \( L_{\text{coarse}} \) exactly contains several cells on level \( L_{\text{fine}} \) (with \( \text{coarse} < \text{fine} \)), we can optimize the indexing process by building the directory of a coarse grained level on the directory of a fine-grained level.

To do so, BLOCK builds the directory for the most fine-grained granularity (smallest cells) level first. Then, to compute the directory of a coarser-grained level \( L_{\text{coarse}} \) from \( L_{\text{fine}} \), the directory entries of cells \( C_{\text{fine}} \) of \( L_{\text{fine}} \) that are contained in a cell \( C_{\text{coarse}} \) of \( L_{\text{coarse}} \) are combined. Because the objects in the object store are sorted based on the z-value of their center, they will still all objects in a cell on a coarser-grained level will still be consecutively stored. Consequently, the pointer to the object store can still be stored compressed.

The intersection lists of all cells \( C_{\text{fine}} \) are also combined to compute the intersection list of \( C_{\text{coarse}} \). The intersection list, however, needs to be updated: a object may have been in the containment list of any of the cells \( c_j \in C_{\text{fine}} \) and in the intersection list of a different cell \( c_k \in C_{\text{fine}} \). It is thus in the containment list and the intersection list of \( c_{\text{coarse}} \) and has to be removed from the latter.

### Algorithm 1: BLOCK Index Building Algorithm

```
for each object \( o_i \) do
    insert object \( o_i \) into object store and get pointer \( p_i \);
    calculate cells \( C \) covered by \( o_i \) in most granular level;
    for each cell \( c_j \) from \( C \) do
        for each level \( l_k \) do
            calculate cell \( c_j \) id for level \( l_k \) \( (c_{jl_k}) \);
            if object \( o_i \) not inserted to cell \( c_{jl_k} \) before then
                if cell \( c_{jl_k} \) contains object’s \( o_i \) center
                    then
                        insert \( p_i \) into \( c_{jl_k} \) block store;
                    else
                        insert \( p_i \) into \( c_{jl_k} \) list store;
                end
            end
        end
    end
end
```

**Fig. 4.** The data structure BLOCK is using, the objects store and directories built based on the dataset.

**E. Querying**

To execute a range query \( rq \) on a single level BLOCK computes all cells \( C_{rq} \) intersecting \( rq \). For all cells \( c_{\text{contained}} \in C_{rq} \) entirely contained in the query, the intersection list as well as as the containment list are retrieved. Because the cells \( c_{\text{contained}} \) are fully contained in \( rq \), the objects on either list do not need to be tested for intersection with the query. All objects referenced by the containment lists are thus returned immediately and because the containment list is contiguous in the object store, reading it will be very quick due to good spatial locality (and therefore fewer cache misses).

Furthermore, BLOCK merges the intersection lists of all cells \( c_{\text{contained}} \) and removes duplicates (the same object may intersect with several cells). Access to the objects intersecting with the cells \( c_{\text{contained}} \) is more random, i.e., no contiguous blocks of objects can be read (as is the case for objects on the containment list). To reduce cache misses, BLOCK sorts the objects according to their z-order (their linear order in memory) to reduce cache misses.

The cells \( c_{\text{intersect}} \) only intersecting (but not contained) with \( rq \) are retrieved similarly, except that all objects need to be tested for intersection before they are reported. To minimize reading objects multiple times form memory (and to avoid filtering of duplicate objects), BLOCK keeps a compressed bit array storing per query what objects have already been returned.

Key to our approach, however, is to execute range queries on several levels. To do so, BLOCK first uses the coarsest level to
find all cells \( c_{\text{contained}} \) contained in \( rq \) and proceeds as before. For the cells \( c_{\text{intersect}} \), BLOCK uses the branching factor (\( BR \)) to decide whether it retrieves the cells \( c_i \in C_{\text{intersect}} \) on the current level or if it executes the part of \( rq \) not yet retrieved on a finer-grained level. Executing the remainder (potentially several queries) of \( rq \) on a finer-grained level proceeds as before and may recursively execute remainders of the query on all levels, thereby effectively reducing the number of intersection tests.

\[
\text{Algorithm 2: BLOCK Range Query Algorithm}
\]

<table>
<thead>
<tr>
<th>Data: query ((rq)), branching factor ((BR))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i \leftarrow \text{most coarsest level number} )</td>
</tr>
<tr>
<td>( D_i \leftarrow \text{cells definitely covered by query} ((rq)) on level} ( i );</td>
</tr>
<tr>
<td>( ND_i \leftarrow \text{cells non-definitely covered by query} ((rq)) on level} ( i );</td>
</tr>
<tr>
<td>while ( ND_i ) not empty do</td>
</tr>
<tr>
<td>\hspace{1em} for each cell ( c_{\text{intersect}} ) in ( ND_i ) do</td>
</tr>
<tr>
<td>\hspace{1em} \hspace{1em} if cell ( c_{\text{intersect}} ) volume covered by ( Q ) less than ( BR ) then</td>
</tr>
<tr>
<td>\hspace{1em} \hspace{1em} \hspace{1em} append into ( D_{i+1} ) definite cells of ( c_{\text{intersect}} ) in level} ( i + 1 );</td>
</tr>
<tr>
<td>\hspace{1em} \hspace{1em} \hspace{1em} append into ( ND_{i+1} ) non definite cells of ( c_{\text{intersect}} ) in level} ( i + 1 );</td>
</tr>
<tr>
<td>\hspace{1em} end</td>
</tr>
<tr>
<td>\hspace{1em} ( i \leftarrow i + 1 )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>for each level} ( i ) do</td>
</tr>
<tr>
<td>\hspace{1em} append into \text{ObjectDefinite} \ object pointers stored in cells} ( D_i );</td>
</tr>
<tr>
<td>\hspace{1em} append into \text{ObjectNonDefinite} \ object pointers stored in cells} ( ND_i );</td>
</tr>
<tr>
<td>while \text{ObjectDefinite} not empty do</td>
</tr>
<tr>
<td>\hspace{1em} return next object from} \text{ObjectDefinite};</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>while \text{ObjectNonDefinite} not empty do</td>
</tr>
<tr>
<td>\hspace{1em} object \leftarrow \text{get next object from} \text{ObjectNonDefinite};</td>
</tr>
<tr>
<td>\hspace{1em} if object in range} ( rq ) then</td>
</tr>
<tr>
<td>\hspace{1em} \hspace{1em} return object;</td>
</tr>
<tr>
<td>\hspace{1em} else</td>
</tr>
<tr>
<td>\hspace{1em} \hspace{1em} continue;</td>
</tr>
<tr>
<td>\hspace{1em} end</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

\[ F. \text{ Updating} \]

Deleting object \( o \) means removing it from the object store. All cells with which \( o \) intersects are identified and their intersection list is updated in the directory of all levels. Similarly, the cell which contains the center of \( o \) is identified on all levels and it is removed from the containment list of the directory of all levels.

Like for other spatial indexes with optimally packed data structures (e.g., the Rtree with filled nodes), inserting object \( o \) is a more complicated process. Also here, the position of \( o \) in the object store is calculated based on the \( z \)-value of \( o \)’s center. If no empty space is available in the object store, objects need to be moved/copied to make space. After modifying the object store, the containment lists as well as the intersection lists in all directories need to be updated on all levels: first, the changes in the object store need to be reflected in the directories and second, due to addition of the object, the containment lists of all cells containing the center of \( o \) are updated and the intersection list of all cells with which \( o \) intersects need to be updated as well.

If the object store becomes too fragmented, it is compacted by copying all objects into a new array in memory. Both lists, intersection and containment list need to be updated as the object position in the object store/array may have changed. Because this is a costly operation, it is only performed infrequently and the object store is split into several arrays to limit the cost of copying data. As with the indexing process, optimization potential lies in deleting the object from the finest-grained level and by computing all coarser grained levels by summarizing the finer-grained levels. Other indexes, however, face similar challenges.

\[ G. \text{ Configuration} \]

BLOCK approach is based on three major parameters: (1) branching factor, (2) granularity and (3) number of levels. The reasoning over the choice of the parameters will be discussed further on the experimental section Section V. The branching factor as was mentioned earlier is the volume of coverage of a cell by the given query that is set to be enough for a cell to be scanned through on a specific level. If the coverage volume is less than the branching factor then the cell is analyzed in the next available level. The granularity of the cells is the number of bits per coordinate that is used to describe the position of a cell in the grid. As in a more granular level more cells exists, consequently more bits are needed to describe their position. The number of levels represents the number of different granularities BLOCK will use. As will be discussed later, the overhead introduced by additional levels in build time is minimal whereas with the optimal selection of branching factors they can minimize the query response time.

\[ V. \text{ EXPERIMENTAL EVALUATION} \]

In this section we describe the experimental setup & methodology and compare BLOCK against state-of-the-art indexing approaches in terms of time to build the index, memory overhead and, most importantly, query execution time. To this end we use synthetic datasets with different configurations (distribution, number of objects, object size) in order to simulate real use cases.
A. Setup

Hardware: The experiments are run on a Linux Ubuntu v2.6 machine equipped with 2 quad CPUs AMD Opteron (each with 2 MB L2, i.e., 512KB × 4 cores and 6 MB L3 cache), 64-bit @ 2.7GHz and 64GB RAM. The storage consists of 4 SAS disks of 300GB capacity each, striped to 1TB.

Software: For all the experiments the OS can use the remaining memory to buffer disk pages. For a fair comparison the implementations of all approaches are single threaded. All approaches are implemented in C++.

Settings: In experiments we compare BLOCK with different grid level configurations against Octree using a node size of 50 thousand objects and an in-memory implementation of Rtree using a page size of 4KB and a fanout of 111. The configuration for each approach is determined using a parameter sweep that yield the best performance.

The resolution/granularity BLOCK uses on a particular level is denoted by the number of bits used for the z-value, e.g., a resolution of 4 means that $2^4 \times 2^4 \times 2^4$ cells of uniform width are used. The maximum resolution is 14.

B. Experimental Methodology

Datasets: To demonstrate the general applicability of our approach and to stress that we do not exploit any particularity of datasets, we primarily use 3D synthetic spatial datasets to evaluate BLOCK. In the synthetic datasets we use we vary the size of objects (either uniformly distributed between 0 and 5 or between 0 and 50) and the distribution of the object’s location (normal distribution with $\mu = 500$, $\sigma = 220$ and uniform). To emulate increasingly large spatial model datasets, we also vary the spatial objects in the datasets between 5M and 120M resulting in a size on disk between 115MB and 2.7GB. All datasets cover 1500 space units in each dimension of three-dimensional space.

Queries: We have designed BLOCK to perform efficiently in face of queries of varying sizes. Consequently we use three microbenchmarks with queries of different sizes: (A) small, covering 300 space units per dimension, (B) big, covering 750 space units per coordinate and (C) mixed, a heterogeneous microbenchmark with queries of type A and B.

C. Building the Index

Although indexing is only a one off operation, the time to index as well as the memory overhead are crucial. In the following we compare BLOCK against both, the Rtree and the Octree, and measure index building time as well as memory overhead.

1) Building Time: In a first experiment we measure the time to build the index for the different approaches with files of increasing size where the object locations are normal distributed and each object has a uniformly distributed size between 0 and 5. The approaches compared are the STR Rtree, the Octree, the BLOCK with a single level with granularity 10 and BLOCK with two levels of granularity 10 and 8.

As can be seen in Figure 5, the BLOCK outperforms the other approaches for either configuration. Indexing with a grid is, after all, rather straightforward: only the overlap of each object with the grid has to be calculated. The indexing process therefore is linear in the number of objects. Clearly, indexing a dataset with any more than one grid level takes longer than with just one level. Intuitively, the indexing time in case for multiple levels is expected to be linear in the number of objects to be indexed as well as the number of grids, e.g., indexing for BLOCK with two levels takes twice as long as indexing BLOCK with one level. Building any levels beyond the first as discussed in Section IV-D, however, optimizes the build process and each additional level takes only little more time as Figure 5 shows.

The indexing process of the Rtree is more complex as the dataset needs to be sorted in each dimension, resulting in a higher indexing time. Most surprisingly, however, is the almost exponential growth of the Octree’s index time. This can be explained by the node splits that occur frequently due to the dense regions in the dataset.

Figure 6 shows a breakdown of the index build time of BLOCK. The major parts of the index build time are: (1) insertion into the object store (2) find the covering cells, i.e., calculation of the cells that the object has to be inserted into and (3) insertion into the level directories (hash table).

As Figure 6 shows, the vast majority of index build time is spent on the calculation with what cells the objects overlap as this operation scans through the grid. With a growing object size the time for this operation grows as well as Figure 7 shows. The same effect can also be observed when comparing different granularities in Figure 6: the finer the granularity, i.e., the more cells, the longer indexing does take (increasing the time spent on calculating the covering cells).
2) Memory Overhead: All approaches need additional data structures that require space beyond the bare dataset. Figure 8 shows the memory consumption of the different approaches with increasing dataset size.

The grid design is very efficient in minimizing the overhead of the pointers, by compressing the containment list into two integers: offset and size of the block of consecutive objects in the object store. Objects only intersecting with cells, on the other hand, have to be stored in all entries of the level directories, thereby increasing the memory footprint.

Also the Octree needs to replicate data, i.e., objects intersecting several cells, and therefore requires more memory. With its data-oriented organization, the Rtree avoids replication (of objects or pointers). Although, Rtree needs to organize the index with a hierarchical structure, it still requires the least memory.

D. Query Execution

We evaluate BLOCK in different configurations on synthetic datasets and compare its execution time with the STR Rtree and the Octree. We first compare the approaches on uniformly distributed data with the small query (A) microbenchmark.

As Figure 9 shows, the Octree is not scaling well at all. Although, the Octree is very efficient for point data, it does not perform or scale well when storing objects with spatial extent. The three-level configuration of BLOCK proves to be most efficient, as the query is big enough, all three levels are used, and the cells enclosed within the range query are found in all three levels. Finding cells enclosed within the range query is essential to the efficiency of query execution, as objects in these cells do not need to be tested for intersection with the query.

We also use datasets with a normal distribution for the same comparison. As Figure 10 shows, the trends are similar except that the Rtree performs worse. The normal distribution leads to
hotspots in the dataset, i.e., locations where the density is high, and where the R-tree consequently suffers from considerable overlap, deteriorating the query execution performance.

VI. BLOCK ANALYSIS

In the following we analyze BLOCK in more detail. We particularly break down its query execution performance and study its parameters. BLOCK has three different configuration parameters: the number of levels, the branching factor, i.e., the threshold determining when to use the next higher level and the granularity.

A. BLOCK Query Execution Breakdown

We first break the query execution operation into three major parts: (1) hypercube calculation, i.e., the calculation of the cells covering the query, (2) pointer calculation, the iteration through the index and gathering the objects within the hypercube and finally, (3) object check and return, the validation that the objects read indeed intersect with the query range and then return them to the user.

As the result of the experiment with the AC microbenchmark on files with normal distribution and an object size between 0 and 5 in Figure 11 shows, the "pointer calculation" operation dominates the overall time. In the pointer calculation, lists containing objects only intersecting with the cell (not containing the center) are iterated and combined together.

As the result of the experiment with the AC microbenchmark on files with normal distribution and an object size between 0 and 5 in Figure 11 shows, the "pointer calculation" operation dominates the overall time. In the pointer calculation, lists containing objects only intersecting with the cell (not containing the center) are iterated and combined together.

Multiple levels always outperform single level configurations of BLOCK. The breakdown explicitly shows that the configurations with multiple levels save time on the hypercube calculation as well as on the object intersection tests. Time in the intersection tests is saved because on multiple levels, more cells are contained completely in the range query. The same effect also reduces the pointer calculation: with multiple levels fewer objects are read from the intersection list and hence fewer lists have to be combined together.

B. Number of Levels

Figure 12 shows the comparison of BLOCK with different configurations featuring one and two levels. More precisely, BLOCK with two levels of granularity 10 and 8 is compared against single level configurations with granularity 6, 8 and 10.

The coarse granularity (containing the largest cells) grid 6, is the slowest because the "refinement" step, i.e., testing objects for intersection with the range query is not efficient, leading to the need of a massive scan through almost all the objects indexed. As the granularity becomes higher, the efficiency of the index improves but peaks as granularity 8 is faster than granularity 10. The higher granularity does not necessarily provides higher efficiency as a considerable amount of time is spent on calculating the grid cells. Throughout the experiment the multi-level configuration of BLOCK is considerably faster than the rest and scales better. This is because it takes advantage of all 10 and 8 granularities in order to respond to the query.

Clearly, the more level are used, the faster query execution becomes. The overhead of using several levels is at the same time minor: the memory overhead as well as the build time overhead are not substantial. At the same time, adding numerous levels will not improve performance indefinitely as adding each additional level only leads to smaller improvements.

C. Granularity

The choice of granularity as is clear from both motivation and comparisons of single-level grid with BLOCK has immense impact on the efficiency of the index. The choice of granularity depends mostly on query size and object size. Query size cannot be known a priori, which makes the choice of a granularity based on the query size difficult as has already been discussed problems of single-level grid structures.

The object size also influences the efficiency of the grid through its access methods. Bigger objects in small granularity
cover more cells, thus pointers to the objects need to be stored in multiple cells. The pointers on the covering list which leads to sequential access to the object is stored only in the cell where the object center resides. As a result bigger objects are more probable to be checked in more cells and bigger objects on small cells also results in more random read access to memory.

D. Impact of the Branching Factor

The branching factor has a considerable impact on the performance of BLOCK. The decision whether a cell should be scanned in lower level or recursively analyzed on more granular grids has a huge impact on the efficiency of the multi-level index. What branching factor to use for a particular configuration is not trivial and for the work presented here we determined the best branching factor with a parameter sweep.

In Figure 13 the number of comparisons, accepted objects and definite tuples is presented in columns and as a trend line the query execution time all for executions of a three level BLOCK configuration (levels of 10, 8 and 6) with different configurations of the branching factor. The AC microbenchmark was used and the dataset contains 15M normally distributed objects. From this graph it is clear that the less comparisons and more definite cells one has, the better the query execution time is. In addition, sets of branching factors are observed within which the query execution time remains stable. To reason over the best branching factor, however, it is equally important to measure each of the branches on BLOCK configurations of granularities 10-8 and 8-6 separately.

In Figure 14 and in Figure 15 the amount of comparisons, accepted tuples and definite tuples is presented in columns and as line the query execution time all for executions of 2 two-level BLOCK configurations (10, 8 and 8, 6) with different configurations of the branching factor. Both follow the same principles as Figure 13, but since 10-8-6 contains both other configurations, the expectation is that it would be at least as fast as the fastest of the two. Despite this hypothesis the 10-8-6 has minimum execution time, 1136.6 msec whereas 10-8 has 1081 msec. We conclude that a more complex, flexible branching factor calculation needs to be introduced in order to take full advantage of the multiple grid dynamics.

VII. CONCLUSIONS

Spatial indexing bottlenecks changed when the indexes became fully memory-based in recent years. Disk-based approaches are optimized to reduce access to the slow disk in general and random access to it in particular while in-memory approaches are CPU-bound and need to reduce the number of objects tested for intersection with the query range. As the hot-spots change it is only logical that the approaches solving the problem should adapt as well. Through the experiments done it has been shown that BLOCK can adapt well in different input conditions, while being faster than the competitive approaches. By being dynamic and adaptive BLOCK, takes advantage of
offering better response time. Range query over a grid structure can be easily parallelized common, the operations over the grid can be parallelized. are single threaded, and as multi-core processors are very all possible environment. Currently all the implementations be used always offering scalable and efficient solutions for overhead that multiple levels add, a big number of levels could Having such a mechanism, in combination with the minimal number of levels and their granularity and the data density BLOCK has shown that can be more memory efficient than any intermediate structure -as trees have- they require less memory if replicates are stored and compressed efficiently. Because grid structures do not have further improvement of query efficiency and, (2) minimization of memory consumption, leading to higher efficiency and scalability.

As proven by Figure 12 the multilevel grid scales better than the single level and from Figure 9 and Figure 10 follows that it performs better than the STR Rtree and Octree. Despite the multiple levels and the added complexity, the build time is scalable both on object size, file size and number of levels. Similarly the memory overhead is lower than Rtree and Octree and does not grow substantially with an increasing number of levels.

VIII. FUTURE WORK

Future work on BLOCK, is divided into two parts: (1) further improvement of query efficiency and, (2) minimization of memory consumption. Because grid structures do not have any intermediate structure -as trees have- they require less memory if replicates are stored and compressed efficiently. BLOCK has shown that can be more memory efficient than the Rtree and Octree but it still has optimization potential.

Considering query efficiency, by introducing a dynamic branching approach which depending on the size of query, number of levels and their granularity and the data density will decide on each cell which level is the best to access. Having such a mechanism, in combination with the minimal overhead that multiple levels add, a big number of levels could be used always offering scalable and efficient solutions for all possible environment. Currently all the implementations are single threaded, and as multi-core processors are very common, the operations over the grid can be parallelized. Range query over a grid structure can be easily parallelized offering better response time.

REFERENCES


![Branching factor analysis](image1.png)

Fig. 15. branching sweep on BLOCK 8-6.