

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 only optimized for disk access (minimizing disk page reads). Recent research has primarily focused on the optimization of known disk-based approaches for memory (through cache alignment etc.) but has not fundamentally revisited index structures for memory.

In this paper we develop BLOCK, a novel approach to execute range queries on spatial data featuring volumetric objects in main memory. Our approach is built on the key insight that in-memory approaches need to be optimized to reduce the number of intersection tests (between objects and query but also in the index structure). Our experimental results show that BLOCK outperforms known in-memory indexes as well as in-memory implementations of disk-based spatial indexes up to a factor of 7. The experiments show that it is more scalable than competing approaches as the data sets become denser.

1. INTRODUCTION

For many applications the efficient execution of spatial range queries is pivotal to extract subsets of spatial models. Executing range queries quickly is, for example, of importance in the simulation sciences where scientists analyze and visualize models through the execution of spatial range queries. Other examples are geographical systems [30], computational neuroscience [20] and applications where objects need to be extracted from a model.

Spatial datasets in the past were too big to be stored in main memory and consequently had to be stored and analyzed on disk. Main memory in desktop, server systems and in supercomputers, however, has grown considerably in recent years and today it can store many spatial datasets completely (or partially). For the purpose of analysis and visualization, scientists in numerous domains today thus load entire models into the main memory of their desktops and execute spatial range queries (and other spatial queries).

Put more formally, subsets of spatial models kept in main memory are retrieved: given volumetric objects \( d \in \text{dataset} \), 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 \( R(q) = d \cap q \), i.e., all objects \( d \in D \) intersecting with \( q \).

The numerous approaches developed for spatial indexing in the past [8] have primarily been developed for disk and, given that access to disk dominates execution time, mostly target to reduce the number of disk pages read. In memory, however, the cost of computation is fully exposed as retrieving data is substantially faster. The main motivation of this paper consequently is that in memory testing objects for intersection with the query and traversing index structures dominates overall query execution time.

Most current approaches, however, have not yet fully taken the cost shift to computations into account and optimize disk-based approaches for main memory. Current in-memory range query execution approaches consequently adopt a similar strategy like disk-based indexes and primarily optimize performance by reducing data read (e.g., through cache alignment). We, on the other hand, design our in-memory spatial index for volumetric objects so it reduces intersection tests considerably (in the index structure and testing objects for intersection with the query).

BLOCK, the approach we develop, uses indexes based on space-oriented partitioning without complex hierarchical index structure to reduce intersection tests. To further reduce intersection tests, BLOCK uses multiple indexes (several uniform grids each with a different resolution) splits the query and executes each part on the best suited index, i.e., the one needing the fewest intersection tests.

While grids have been used before to index spatial data, the major contribution of BLOCK lies in using several grids to substantially reduce the number of intersection tests - the major overhead for in-memory spatial indexes. More particularly, the contributions of BLOCK are threefold: first, BLOCK splits every query \( Q \) into several parts \( p \in Q \) and, second, chooses the grid with the best suited resolution to execute each part \( p \) on to considerably reduce intersection tests. Third, while indexing a dataset with several indexes typically comes with considerable overhead, we develop a novel, efficient indexing approach that does not incur undue overhead.

Our experiments on real neuroscience data show a speedup of up to 300% compared to existing spatial indexes.

The remainder of this paper is organized as follows. In Section 2 we review spatial indexing methods. In Section 3 we motivate our approach with a set of measurements and we explain our approach in detail in Section 4. We discuss configuration considerations in Section 5 and demonstrate the benefits of our approach in Section 6, analyze different configurations of BLOCK in more detail in Section 7 and draw conclusions in Section 8.

2. RELATED WORK

Decades of research in spatial indexing have produced numerous index structures for the execution of spatial range queries on disk but only a few in memory [8]. Many disk-based spatial indexes, however, can also be used in memory and we thus discuss both types. We classify them as space- or data-oriented indexes where the former are mostly used in memory and the latter on disk.
2.1 Space-oriented Partitioning

Indexes based on space-oriented partitioning decompose hyperspace independent of the data distribution. Doing so makes the indexing process comparatively 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 (potentially 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 unbalanced trees (if organized hierarchically). Both effects are detrimental to query execution performance on disk and consequently space-oriented indexes are mainly used in memory.

The KD-Tree [5] and the Quadtree [26, 12] (along with its variant for 3D, the Octree [15]) are today predominantly used in memory. All three recursively partition space, the Quad- and Octree split an 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. The 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 an object intersects with. All three indexes are broadly used mostly because of their rather simple implementation. To alleviate the problem of replication, the loose Octree [31] does not partition space precisely, but permits to make partitions bigger (leading to overlap), thereby curbing replication.

The UB-Tree [25] uses space-oriented partitioning but manages to build a balanced tree: it sorts points according to the z-order [23] 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 z-values inside the range. Calculating the next z-value within the range, however, is a costly operation that degrades performance of the UB-Tree. A much simpler space-oriented index is the grid [6] used for static but also for moving objects datasets [28]. 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. Doing so speeds up indexing but choosing the best resolution of the grid is difficult for datasets with volumetric objects: if the resolution is too fine grained, each object will be replicated to numerous partitions. Replicated objects lead to (a) a bigger memory footprint, (b) extra intersection tests between replicated objects and (c) potentially deduplication overhead.

To store data on disk using space-oriented partitioning, the grid file [21] defines a grid with non-overlapping cells of variable size. The objects in each cell are stored in one disk page. If a cell overflows (the disk pages overflow), it is split into two and if the cell underflows, several cells are stored on one 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 page may require several cells to be maintained in memory. All three recursively partition space, the Quad- and Octree split an 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. The 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 an object intersects with. All three indexes are broadly used mostly because of their rather simple implementation. To alleviate the problem of replication, the loose Octree [31] does not partition space precisely, but permits to make partitions bigger (leading to overlap), thereby curbing replication.

To store data on disk using space-oriented partitioning, the grid file [21] defines a grid with non-overlapping cells of variable size. The objects in each cell are stored in one disk page. If a cell overflows (the disk pages overflow), it is split into two and if the cell underflows, several cells are stored on one 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 page may require several cells to be maintained in memory. All three recursively partition space, the Quad- and Octree split an 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. The 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 an object intersects with. All three indexes are broadly used mostly because of their rather simple implementation. To alleviate the problem of replication, the loose Octree [31] does not partition space precisely, but permits to make partitions bigger (leading to overlap), thereby curbing replication.

To store data on disk using space-oriented partitioning, the grid file [21] defines a grid with non-overlapping cells of variable size. The objects in each cell are stored in one disk page. If a cell overflows (the disk pages overflow), it is split into two and if the cell underflows, several cells are stored on one 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 page may require several cells to be maintained in memory. All three recursively partition space, the Quad- and Octree split an 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. The 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 an object intersects with. All three indexes are broadly used mostly because of their rather simple implementation. To alleviate the problem of replication, the loose Octree [31] does not partition space precisely, but permits to make partitions bigger (leading to overlap), thereby curbing replication.

To store data on disk using space-oriented partitioning, the grid file [21] defines a grid with non-overlapping cells of variable size. The objects in each cell are stored in one disk page. If a cell overflows (the disk pages overflow), it is split into two and if the cell underflows, several cells are stored on one 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 page may require several cells to be maintained in memory. All three recursively partition space, the Quad- and Octree split an 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. The 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 an object intersects with. All three indexes are broadly used mostly because of their rather simple implementation. To alleviate the problem of replication, the loose Octree [31] does not partition space precisely, but permits to make partitions bigger (leading to overlap), thereby curbing replication.

To store data on disk using space-oriented partitioning, the grid file [21] defines a grid with non-overlapping cells of variable size. The objects in each cell are stored in one disk page. If a cell overflows (the disk pages overflow), it is split into two and if the cell underflows, several cells are stored on one 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 page may require several cells to be maintained in memory. All three recursively partition space, the Quad- and Octree split an 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. The 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 an object intersects with. All three indexes are broadly used mostly because of their rather simple implementation. To alleviate the problem of replication, the loose Octree [31] does not partition space precisely, but permits to make partitions bigger (leading to overlap), thereby curbing replication.

The CR-Tree [17] essentially is an optimized R-Tree 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 reduce size by 60% and thus improve performance by a constant factor over the regular R-Tree but do little to address the problem of overlap (in fact, quantization of the MBRs leads to more overlap and more computations).

An extensive evaluation [14] compares variants of the R-Tree (R*-Tree, Hilbert R-Tree and others) in memory to the CR-Tree. The CR-Tree generally outperforms other variants on range queries.

3. POTENTIAL FOR IMPROVEMENT

The major motivation for BLOCK is that most of today’s spatial indexes are not adequately designed for memory. Approaches developed in the past are primarily optimized to retrieve as little data as possible. For disk-based indexes this is crucial as the vast
majority of time is spent on retrieving data from disk. In memory, however, the time needed to read data is very small.

The biggest potential for improving today’s indexes for use in memory lies in further reducing computation. We demonstrate this with an experiment where we index a dataset of 200 million spatial objects with an R-Tree and execute 200 queries with a selectivity of $5 \times 10^{-4}$ at random locations (the experimental setup is described in Section 6). As the result in Figure 1 shows, only 3.3% of the time is spent on reading data when using the R-Tree in memory. Computations, on the other hand, take the overwhelming majority or 95.3% of the total time.

As the breakdown of the query execution time in the same experiment shows, most of the time spent for computations is used to perform intersection tests. Analyzing the time spent on intersection tests further reveals that the majority of intersection tests (and 55% of the total time) is performed in the tree structure of the R-Tree. The considerable time spent on intersection tests within the tree structure degrades performance and indicates overlap of the bounding boxes, a well known problem of the R-Tree [10] and data-oriented tree structures. With increasing density of the spatial datasets, the overlap will also increase in future datasets [29], thereby further increasing the intersection tests in the tree structure.

Recent work like the CR-Tree [17] optimizes the R-Tree for memory (by aligning the data structures for the cache line, quantization and compression) manages to reduce the execution time by a constant factor, but does little address the fundamental problem of overlap. Optimizing indexes based on data-orientation for use in memory leads to indexes which still require excessive reads of the index structure and an excessive number of intersection tests.

Even without overlap the number of intersection tests accounts for a considerable share of the overall time: 25% of the time is spent on testing individual objects for intersection with the query. Many of these tests are unnecessary as well. Using indexes based on data-oriented partitioning can lead to partitions of which only a small fraction intersects with the query (see Figure 2). Still, all objects in the partition need to be tested for intersection, leading to many unnecessary intersection tests. Figure 2 illustrates this problem as objects in the narrow partitions need to be tested for intersection although they are far from the query.

The problem of excessive comparisons, however, does not only affect data-oriented indexes. Space-oriented approaches like the KD-Tree or the Octree that avoid overlap 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 tree is unbalanced and has a depth of up to 90 levels, meaning that in the worst case 90 nodes need to be retrieved before the query result can be computed, resulting in pointer chasing and additional comparison tests that ultimately increase the execution time of range queries.

4. THE BLOCK APPROACH

With BLOCK we design a novel in-memory index for the efficient execution of range queries $q$, either window queries, i.e., all objects intersecting with $q$ or containment queries, i.e., all objects completely contained in $q$ [8]. Motivated by our example from science — the analysis of brain models in neuroscience — BLOCK is designed for static workloads as it does not need to support updates.

In designing BLOCK, we are driven by the insights of Section 3 and draw inspiration from previous work, building on the benefits and drawbacks of past work in space- as well as data-oriented spatial indexing.

4.1 Rationale & Contributions of BLOCK

In order to minimize the number of intersection tests we avoid hierarchical structures (and the associated problem of overlap) as well as unbalanced directory trees and consequently base BLOCK on uniform grids. 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 do not need to be tested anyway. Finally, because the partitions are quadratic in a uniform grid, no unnecessary intersection tests with objects in narrow partitions are performed.

The biggest challenge with uniform grids, however, is finding the best configuration, i.e., the resolution or cell width, because it depends on the size of the queries which is typically not known a priori. The results of an experiment shown in Figure 3 with a grid illustrate this problem. We use the same dataset as in the motivation and a workload with five queries of size 500 space units per dimension uniform random placed. We measure the total time to execute all queries. As Figure 3 shows, the best configuration is difficult to choose. If there are too many cells, i.e., the cells are too small, then...
too much time is spent on calculations. If we chose the granularity too coarse (bigger cells), a considerable share of the time is used on unnecessarily testing objects for intersection with the query. Not knowing the size of the queries a priori makes it difficult to choose the best configuration, particularly if the query size varies.

To address the issue of unneeded intersection tests when configuring a uniform grid wrong. BLOCK features the novel idea of using several uniform grids (each with a different resolution) to reduce intersection tests. Queries are executed on several grids with different resolutions: a query $Q$ is split into several parts $p \in Q$ and each part $p$ is executed on the grid where the least number of cells needs to be retrieved, therefore reducing the data retrieved unnecessarily and hence also reducing intersection tests.

While both, the idea of using grids of different resolution [18] and splitting a query [1] have been used in the past for spatial joins and for imaging data respectively, their application to execute range query on arbitrary spatial data (featuring volumetric objects) in general and to reduce the intersection tests in memory in particular, is novel and is BLOCK’s main contribution. An additional key contribution of BLOCK is the novel space- and time-efficient indexing of a dataset with several grids.

### 4.2 BLOCK Overview

BLOCK indexes the same dataset with a hierarchy of grids where each level is a uniform grid with different resolution. To execute range queries, BLOCK starts with the grid with the coarsest resolution and finds all cells completely contained in the range query. Objects in these cells do not need to be tested for intersection because they are by definition contained in the query. The remaining parts of the query, i.e., parts not covered by cells completely contained in the query, are subsequently executed as queries either on the same or the next finer grained grid based on the benefit it may provide. The process continues to increasingly finer grained grids.

Figure 4 illustrates the query execution process with BLOCK on three different grids. Query execution starts on the coarsest level and tests to what degree cells overlap with the range query. Fully contained cells are retrieved from the current level. For partially overlapping cells, BLOCK computes whether it is beneficial to execute the overlapping parts on the same or on a finer level. If splitting is beneficial, the query is split into several smaller queries which are then recursively executed on a more fine grained level based on the same idea.

![Figure 4: A query is executed on several uniform grids of different resolution.](image)

The approach reduces the number of intersection tests in two ways. First, by primarily retrieving cells that are completely contained in the query, none of the cells’ objects needs to be tested for intersection. Second, BLOCK ensures that the cells only intersected by the query but not fully contained in it are small. Consequently, they contain substantially fewer irrelevant (and therefore unnecessarily retrieved and tested) objects. As we will demonstrate in the evaluation, doing so considerably reduces the number of intersection tests.

By optimizing the data structures for the cache line, i.e., aligning them for the cache line, and by reordering access to memory in order to reduce cache misses, we further optimize BLOCK for use in main memory.

### 4.3 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 $l_i$ 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 $l_k$. BLOCK chooses the configuration based on a cost model described in Section 5.

Each object is indexed using the grid on every level. To avoid replication of objects 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.

#### 4.3.1 Object Store

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 [22], i.e., based on the z-value of their center. Storing objects according to their z-order in the object store ensures spatial locality 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. Other orders (e.g., Hilbert [11] or Peano [23]) could be used but we found the z-order to be the most efficient for coordinate to z-value transformations.

#### 4.3.2 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 stores all pointers (location in the object store) to objects that have their center in the cell. Because the objects are stored in the object store consecutively, i.e., ordered on the z-value of their center, we can compress the containment list and only store the offset and the number of objects which have their center in this cell.

Only storing pointers to the objects which have their center in the cell, however, will not guarantee that BLOCK retrieves all objects intersecting with the cell. A volumetric object $o$ may overlap with a cell $c$ but $o$’s center may be in a cell neighboring $c$. We thus maintain an intersection list for every cell $c$ and store in it pointers (location in the object store) to all volumetric objects intersecting with $c$ (but not containing the center of the object). These objects are not stored consecutively in the object store (they may be stored elsewhere in the object store) and we thus need to store pointers to individual objects in a linked list. If at least two objects in the list are stored adjacent we compress them into pairs of offset and size.

Figure 5 shows the data structures of BLOCK. Based on the dataset and the two levels each with a uniform grid, BLOCK builds the objects store sorting the objects based on the z-order value of their center. Subsequently it builds the directories based on the 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 on the z-value of the cells.

![Diagram of BLOCK data structures](image)

Figure 5: Data structures of BLOCK: the object store and the level directories.

### 4.4 Building the Index

To build the index structures for dataset $D$, BLOCK first organizes the object store: all objects $o_i \in D$ are sorted based on the z-value of their center and are stored in an array. The directory of each level $l_k \in L$ is subsequently built by iterating over the object store and testing for each object $o_i$ in which its center resides. For cells $c \in C$ that object $o_i$ intersects with, a pointer to $o_i$ is inserted into the containment list of the directory entry of the cell $c_j$ in which its center resides. For cells $c \in C$ that object $o_i$ intersects with, a pointer to $o_i$ is inserted into the intersection list of each $c$.

All objects having their centers in a cell are adjacent to each other in the object store because they are ordered by their z-value and we thus only need to store the offset and the 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. When using uniform space partitioning, i.e., a cell on $l_{\text{coarse}}$ exactly contains several cells on level $l_{\text{fine}}$ (with coarse $< $ fine) we can optimize the indexing process by building the directory of a coarse-grained level based on the directory of a fine-grained level. To do so BLOCK builds the directory of the finest-grained granularity level first. Then, to compute the directory of a coarser-grained level $l_{\text{coarse}}$ from a finer-grained level $l_{\text{fine}}$, the directory entries of cells $C_{\text{fine}}$ of $l_{\text{fine}}$ contained in a cell $c_{\text{coarse}}$ of $l_{\text{coarse}}$ are combined.

More precisely, all the containment lists of $C_{\text{fine}}$ are merged to obtain the containment list of $c_{\text{coarse}}$. To obtain the intersection list of $c_{\text{coarse}}$ the intersection lists of $C_{\text{fine}}$ are merged equally. The intersection list, however, needs to be updated: an object $o_i$ 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 $c_k \in C_{\text{fine}}$. $o_i$ is thus contained in both lists of $c_{\text{coarse}}$ and has to be removed from the intersection list (because it already is in the containment list). Algorithm 1 describes the indexing procedure in pseudocode.

```pseudocode
Algorithm 1: BLOCK Index Building Algorithm

Input: $D$: spatial dataset containing all objects

1. foreach object $o_i \in D$ do
2.     insert object $o_i$ into object store and get pointer $p$;
3.     sort object store based on object's center z-value
4. foreach object $o_j \in object store$ do
5.     calculate cells $C$ covered by $o_j$ in most granular level;
6.     foreach cell $c_j \in C$ do
7.         foreach level $l_k$ do
8.             calculate cell $c_j$'s id for level $l_k$ ($c_{jl_k}$); if object $o_j$
9.                 not inserted to cell $c_{jl_k}$ before then
10.                if cell $c_{jl_k}$ contains object's $o_j$ center then
11.                   insert $p$ into $c_{jl_k}$ containment list;
12.                 else
13.                     insert $p$ into $c_{jl_k}$ intersection list;
```

As the objects in the object store are sorted based on the z-value of their center, the objects in a cell on a coarser level will still be stored consecutively. Consequently, the pointers to the object store can be stored compressed and the level directory for each coarser level will shrink in size as on the coarser level more objects are contained in each cell (thus increasing compression). This novel storage approach of BLOCK enables to store multiple grids space- and time-efficient.

### 4.5 Querying

To execute a range query $rq$ on a single level BLOCK computes all cells $c_{\text{cont}} \in C_{rq}$ intersecting $rq$. For all cells $c_{\text{cont}} \in C_{rq}$ entirely contained in the query, the intersection list as well as the containment list are retrieved. Because the cells $c_{\text{cont}}$ 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 thus reduced cache misses).

The list resulting from merging the intersection lists of all cells $c_{\text{cont}}$ may contain duplicates (the same object may intersect with several cells) and BLOCK therefore removes duplicates based on the hash value of objects. Duplicates are thus already removed at this stage of query execution and further deduplication is not needed. Access to the objects intersecting with the cells $c_{\text{cont}}$ is more random, i.e., no contiguous blocks of objects can be read. To reduce cache misses, BLOCK sorts the objects according to their z-order value (their linear order in memory).

The cells $c_{\text{mem}}$ 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 from memory (and to avoid filtering of duplicate objects), BLOCK keeps a compressed bit array storing per query what objects have already been returned. Using the bit array ensures that duplicate results found on different levels can be filtered so that only unique results are reported.

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 \(c_{\text{contained}}\) and retrieves the objects in these cells as described before. For the cells \(c_{\text{intersect}}\), BLOCK decides dynamically whether to retrieve the cells \(c_i \in c_{\text{intersect}}\) on the current level or to execute the part of \(rq\) not yet retrieved on a finer-grained level. The execution of the remainder of \(rq\) on a finer-grained level proceeds as before and may recursively execute remainders of the query on all levels.

Executing a query on multiple levels can lead to an object being considered multiple times. As discussed before, BLOCK maintains a compressed bit array containing already returned results to avoid the overhead of retrieving and testing an object multiple times. BLOCK thus only returns unique results early in the query execution and no further deduplication is needed.

Algorithm 2: BLOCK Range Query Algorithm

```
Input: rq: three dimensional range query
Data: 
\(c_{\text{contained}}\): array of lists containing covered cells
\(c_{\text{intersect}}\): array of lists containing intersecting cells
\(i\) ← coarsest level number;
\(c_{\text{contained}}\) ← cells contained in \(rq\) on level \(i\);
\(c_{\text{intersect}}\) ← cells intersecting with \(rq\) on level \(i\);

while \(\text{intersect, not empty} \) do
  foreach cell \(c_i \in c_{\text{intersect}}\), do
    cost\((c_i)\) cost of executing on level \(i\); cost\((\sum c_{i+1})\) cost of executing on level \(i + 1\);
    if cost\((\sum c_{i+1})\) < cost\((c_i)\) then
      append to \(c_{\text{contained},i+1}\) containing cells of \(c_i\) in level \(i + 1\);
      append to \(c_{\text{intersect},i+1}\) intersecting cells of \(c_i\) in level \(i + 1\);
    end
  i ← i + 1
  foreach level \(i\) do
    append to \(c_{\text{contained}}\) object pointers stored in cells
    append to \(c_{\text{intersect}}\) object pointers stored in cells
  foreach object \(c_i \in c_{\text{contained}}\) do
    return object;
  foreach object \(c_i \in c_{\text{intersect}}\) do
    if object \(\cap rq\) then
      return object;
    else
      continue;
end
```

BLOCK bases the decision when to split and where to execute the query on the cost and benefits of using a finer-grained level. The most significant cost factor is the random memory accesses used to retrieve objects from the intersection list. Objects in this list are not stored consecutively in memory. Clearly, the more fine-grained a level is, the longer the intersection lists become due to smaller cells and consequently the cost of accessing the objects is higher. Several levels and may recursively execute remainders of the query on all levels.

We demonstrate the cost/benefit trade-off with an experiment where we execute the same queries on grids with increasingly fine resolution and measure fully enclosed objects as well as the query execution time. As the results in Figure 6 show, there is indeed a trade-off in using finer-grained levels: more fully contained objects decrease the execution time but there is a tipping point where the cost of retrieving smaller cells starts to slow down query execution.

We use Equation 2 further breaks down the cost of scanning the objects contained in a cell. The cost of object retrieval from a cell has two major components: (1) \(\text{SeqAccessScan}\), the cost for the sequential memory access to read all objects whose centers are in the cell, and (2) \(\text{RandAccess}\), the cost random memory access to retrieve the objects that intersect but center is not in the cell.

\[
Q_{\text{cost}} = \text{Calc} + \sum_{i=1}^{L} (D_i \cdot \text{Scan}_i + \text{ND}_i \cdot (\text{Scan}_i + \text{Test}_i)) \tag{1}
\]

```

Figure 6: Cost vs Benefit balance.
```

Equation 2 further breaks down the cost of scanning the objects contained in a cell. The cost of object retrieval from a cell has two major components: (1) \(\text{SeqAccessScan}\), the cost for the sequential memory access to read all objects whose centers are in the cell, and (2) \(\text{RandAccess}\), the cost random memory access to retrieve the objects that intersect but center is not in the cell.

\[
\text{Scan}_i = \text{SeqAccessScan}_i + \text{RandAccess}_i \tag{2}
\]

### Sequential Access.
Assuming \(C_i\) the number of objects in a cell on level \(i\) and \(\text{SeqMemAccessCost}\) the average sequential memory access cost in seconds is as follows:

\[
\text{SeqAccessScan}_i = C_i \ast \text{SeqMemAccessCost} \tag{3}
\]

### Random Access.
Assuming \(I_i\) the number of objects intersecting a cell but their center being in a different cell on level \(i\) and \(\text{RandomMemAccessCost}\) the average random memory access cost in seconds.

\[
\text{RandomAccessScan}_i = I_i \ast \text{RandomMemAccessCost} \tag{4}
\]

The benefit, on the other hand, is avoiding unnecessary intersection test through fully contained cells. Equation 5 gives the idea of...
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>The set of level granularities</td>
</tr>
<tr>
<td>$X$</td>
<td>Size of the universe dimensions</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of objects in the dataset</td>
</tr>
<tr>
<td>$Q_{\text{cost}}$</td>
<td>Query cost</td>
</tr>
<tr>
<td>$\text{Calc}$</td>
<td>Calculation of execution strategy</td>
</tr>
<tr>
<td>$\text{Scan}_i$</td>
<td>Cost of scanning a cell on level $i$</td>
</tr>
<tr>
<td>$\text{Test}_i$</td>
<td>Cost of testing a cell on level $i$</td>
</tr>
<tr>
<td>$D_i$</td>
<td>Number of fully covered cells on level $i$</td>
</tr>
<tr>
<td>$ND_i$</td>
<td>Number of not fully covered cells on level $i$</td>
</tr>
<tr>
<td>$I_i$</td>
<td>Number of objects only intersecting a cell</td>
</tr>
<tr>
<td>$C_i$</td>
<td>Number of objects having center in a cell</td>
</tr>
<tr>
<td>$O_i$</td>
<td>Average number of objects in cell on level $i$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intersection TestCost</th>
<th>Average cost of intersection test</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeqMem AccessCost</td>
<td>Average sequential memory access cost in seconds</td>
</tr>
<tr>
<td>RandomMem AccessCost</td>
<td>Average random memory access cost in seconds</td>
</tr>
<tr>
<td>$cw_i$</td>
<td>Cell width on level $i$</td>
</tr>
<tr>
<td>$ow$</td>
<td>Average object width</td>
</tr>
</tbody>
</table>

Table 1: Variables used in the cost model.

calculating the benefit by using the average cost of a single intersection test and the total number of objects in a cell as all objects in the cell will have to be tested.

$$\text{Test}_i = \text{NumberOfObjects}_i \times \text{IntersectionTestCost} \quad (5)$$

When executing a query, BLOCK has access to all parameters (size of intersection or containment list of all cells involved, cost of intersection test, cost of random/sequential memory access) and computes cost and benefit of executing part of a query (or a whole query) on two different layers. Through calculating the total cost for executing the query on a cell on level $i$ and over the cells in the same area in level $i+1$, BLOCK decides whether the query should be split and executed on the next level or not.

5. BLOCK CONFIGURATION

The configuration of BLOCK centers around two factors, the number of levels and their respective granularity. The number of levels corresponds to the number of different granularities BLOCK will use and the granularity is the number of cells used on a level.

Both factors primarily depend on the distribution of query location, query size as well as the distribution of the location and size of objects in the dataset. Even if the distribution of size and location of the queries is not known a priori BLOCK can still be configured for efficient performance as we discuss in the following. Crucially, however, even if not configured optimally, BLOCK adjusts itself at runtime and by collecting information of object distribution and size it chooses the most efficient set of granularity levels. In the following we first discuss the granularities (finest and coarsest) and then further discuss how many levels should be used.

5.1 Grid Granularity

For BLOCK to execute queries as efficient as possible, it needs to be configured with the appropriate number of levels and the optimal grid resolutions. The grid resolution depends on the distribution of query size and location and the distribution of object size and location. In this discussion we focus on the query size and assume the distribution of their positions as well as the distribution of the objects to be uniform. As we will see in the experiments, even when using extreme distributions, configurations based on reasoning with uniform distributions are very efficient.

5.1.1 Coarsest Granularity Level

As discussed in Section 4.5, key to make BLOCK efficient is to reduce intersection tests while balancing the cost of retrieving objects. Reducing intersection tests directly translates into ensuring that queries will completely enclose several cells, thereby avoiding any intersection tests for objects in the enclosed cells.

If the query size is known a priori, we consequently argue to choose the granularity of the coarsest grained grid such that a query encloses cells as big as possible. Using as resolution the query size is not optimal. In the best possible case this will exactly retrieve one cell if the query position is perfectly aligned with the uniform grid. In all other cases, however, 4 cells are retrieved in 2D and 8 in 3D, leading to many unnecessary intersection tests. We therefore argue that the best resolution for the coarsest grid (on level 0), or more precisely its cell width is $cw_0 = \frac{\text{width}_{\text{query}}}{2}$ with $\text{width}_{\text{query}}$ being the width of the query in each dimension. With this cell width we can ensure that at least one cell will be enclosed in the query and in the optimal case 8 cells in 3D.

In case the query size is not known a priori, we argue that the best resolution is such that a query can enclose at least one grid cell. In case we use three cells in each dimension, at lest the center cell can be enclosed by a query that is not perfectly aligned.

5.1.2 Finest Granularity Level

Finding the finest granularity follows similar reasoning as the trade off BLOCK strikes when deciding what levels to use when querying (Section 4.5): if the granularity is too fine, the benefit of the avoided intersection tests is smaller than the cost of randomly accessing memory for retrieving the intersection lists. Crucially, the more fine-grained a level is, the smaller the cells are and consequently the number of cells an object intersects with grows. This leads to increasing random memory accesses thus increasing the cost of query execution on a level. We thus argue that the finest level is chosen so that the benefit of using one of its cells is bigger than the cost to access it. In the following we develop the model to determine the finest granularity (cf. Table 20 for the variables).

As cost we take into account the additional cost introduced by retrieving objects from smaller cells (with longer intersection lists). The benefit is the saved cost from avoiding intersection tests. For this discussion we only consider immediately neighboring levels $i$ and $i+1$ of BLOCK (with cell width $cw_i$ and $cw_{i+1}$ respectively) and focus on the case where the finer-grained level avoids all intersection tests. Cost and benefit then lead to the inequality in 6.

$$\text{Scan}_{i+1} - \text{Scan}_i \leq O_i \times \text{IntersectionTestCost} \quad (6)$$

The finest-grained level for BLOCK should be chosen so that inequality 6 still applies.

To decide on the resolution of the finest-grained without information about the data we use an analytical model to determine $\text{Scan}_i$ cost for each level. In the following analysis we assume that the distribution of objects is uniform. Based on the equations used to calculate $\text{Scan}_i (2, 3, 4)$ and the assumption of uniform distribution, we can assume that the number of objects that have their center in each cell is as described in Equation 7 (with $G_i$ being the number of objects in a cell).

$$\text{Scan}_{i+1} = \text{Scan}_i - \frac{G_{i+1}}{G_i}$$
\[ C_i = \frac{N}{G_i} \]  \hspace{1cm} (7)

If we further assume that \(cw_i\) is the width of a cell on a level \(i\) and \(X\) the size of one dimension of the universe Equation 7 becomes:

\[ C_i = \frac{N \times cw_i^3}{X^3} \]  \hspace{1cm} (8)

which instead of the number of cells takes into account the object width and thus shows clearer that on finer-grained levels the number of object per cell becomes smaller. Analytically, every cell is surrounded by an area to be positioned close to the neighboring cell or it has to be of substantial size. To determine analytically the length of the intersection list we calculate the number of objects that only intersect a cell (while their center is in another cell). To only intersect, the object either has to be positioned close to the neighboring cell or it has to be of substantial size. Analytically, every cell is surrounded by an area filled with potentially intersecting objects. This area is calculated as follows:

\[ I_{area} = ((cw_i + ow)^3 - cw_i^3) \]  \hspace{1cm} (9)

Multiplying \(I_{area}\) with the average area per object, we obtain:

\[ I_i = \frac{((cw_i + ow)^3 - cw_i^3) \times N}{X^3} \]  \hspace{1cm} (10)

Thus by connecting these equations we obtain the object scan cost or one cell on level \(i\):

\[ Scan_i = \frac{N}{X} \times (cw_i^3 \times SeqMemAccessCost + ((cw_i + ow)^3 - cw_i^3) \times (RandomMemAccessCost)) \]  \hspace{1cm} (11)

Based on this cost model and on the measured cost for memory access and for intersection tests we can precisely set the finest-granular level of BLOCK.

5.2 Number of Levels

The number of levels has direct impact on performance of BLOCK. As shown in the experiments additional levels do not require substantial additional memory and we thus argue to use as many levels as the memory allows between the finest and coarsest level. The granularities of the levels added between the coarsest and the finest, on the other hand, depend on the size of the objects in the dataset. As we argued, the biggest cost factor for BLOCK is the memory to buffer disk pages. For a fair comparison the implementations of all approaches are single threaded. All approaches are implemented in C++.

6. 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 create cubes and vary the length of the edges (either uniformly distributed between 0 and 1, 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 240M resulting in a size on disk between 229MB and 12GB. All synthetic datasets cover 1500 space units in each dimension of three-dimensional space.

In addition to synthetic datasets we use real spatial datasets from the motivating neuroscience application. The dataset models a part, i.e., the neurons, of the neocortex. The structure of each neuron is represented by thousands of small cylinders. The size of the data spans from 50 up to 450 million cylinders, or more precisely their bounding box. The size of the neuroscience models ranges from 2.3GB to 22GB on disk.

Queries: Driven by the neuroscience motivation, we have designed BLOCK to perform efficiently in face of queries of varying sizes. Inspired by a visualization use case in neuroscience we consequently use six microbenchmarks with queries of different sizes: (A) covering 50 space units per dimension, (B) covering 300 space units per coordinate and (C) covering 750 space units.

To test the performance of the BLOCK multi-level grid structure as well as the query execution cost model (Section 4.5) based on which BLOCK decides what levels to execute the query on, we also experiment with single level grids.

6.3 Building the Index

Although indexing is a one off operation, the time to index as well as the memory overhead are crucial. In the following section we compare the index building time and memory footprint of BLOCK, the STR-bulkloaded R-Tree and Octree. Furthermore, we study the efficiency of BLOCK’s multi-layer grid building algorithm and finally, we measure the effect of object size on BLOCK’s index building time.
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 normally distributed and each object has a uniformly distributed size between 0 and 1. We first compare the STR R-Tree, the Octree, BLOCK with a single level with $2^{30}$ cells and, to show the impact of additional levels, with three levels with $2^{30}$ cells, $2^{24}$ cells and $2^{18}$ cells. For clarity, we neither include the MX-CIF and Loose Octree as they have the same indexing process as the Octree nor the CR-Tree as it uses the same indexing like the R-Tree.

As the result in Figure 7 shows, BLOCK outperforms the other approaches for either configuration. Indexing with a grid is 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 as the corresponding structures have to be computed. Building any levels with BLOCK beyond the first one, however, increases the build time only minimally as Figure 7 shows. BLOCK takes advantage of the relationship between grids on different levels: it builds only the most fine-grained grid and then summarizes this recursively into coarser levels as discussed in Section 4.4. This is possible because bigger cells on a coarser level always contain several smaller cells from a finer level and consequently no calculations of the element intersections with the grid are necessary.

The indexing process of the R-Tree is more complex as the dataset needs to be sorted in each dimension, resulting in a higher indexing time. Surprisingly, however, is the almost exponential growth of the Octree’s indexing time. This can be explained by frequent node splits due to the dense regions in the dataset. Indexing 120 million objects takes 10× longer for the Octree than for the R-Tree as can be seen in Figure 7.

Memory Footprint. All indexing approaches need additional data structures requiring space beyond the dataset. Figure 8 shows the memory usage of the indexes with increasing dataset size. We use two different configurations of BLOCK to show the impact of multiple levels.

The level directories of BLOCK are 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 intersecting with multiple cells, on the other hand, cannot be efficiently compressed, thus increasing the memory footprint. Adding additional levels to BLOCK does not considerably increase its memory footprint as most of the memory is used for the object store. When adding coarser levels, more objects are entirely contained in cells, and therefore the easily compressible containment lists become longer and the harder to compress intersection lists become shorter. Each additional coarser level thus requires less memory than the previous one. Because the Octree needs to replicate data, i.e., objects intersecting several cells, it requires considerably more memory. With its data-oriented organization, the R-Tree avoids replication (of objects or pointers) completely and, although it organizes the index with a hierarchical structure, it requires the least memory.

Build Time Breakdown. To analyze where time is spent in BLOCK’s indexing process, we index a dataset with normally distributed objects with different static configurations (both single level and multiple levels). The major parts of the index build time are: (1) insertion into the object store (2) find covering cells, i.e., calculation of the cells that the object has to be inserted into and (3) insertion into the level directories. Figure 9 shows the build time breakdown of these operations. The finer the grid granularity, i.e., the more cells, the longer indexing takes because an object will intersect with more cells. Calculating the overlap of objects with cells is particularly expensive because it requires to calculate the z-order value of all the overlapping cells. Adding the cell to the index, i.e., to the level directories, on the other hand, only takes insignificant time. Similarly, the time to add the elements to the object store is insignificant.

Object Size Effect. Figure 10 shows the effect of object size on the efficiency of BLOCK’s index building time. We use four static configurations two single layer, one two layer and one three-layer.
The build time of BLOCK depends both on the object size and the granularity of the most fine-grained grid layer. The three layer BLOCK configuration requires nearly the same index building time as the single layer BLOCK with $2^{30}$ cells.

### 6.4 Query Execution

In this section we evaluate BLOCK and compare it to the STR-bulkloaded R-Tree, the CR-Tree, the Octree, the loose Octree and the MX-CIF Octree on synthetic and neuroscience data. BLOCK uses the query execution cost model (see Section 4.5) to decide on what levels to execute which parts of the query. Thanks to the use of the query execution cost model, no parameters need to be set.

![Figure 11: Comparison of BLOCK, STR-bulkloaded R-Tree, CR-Tree, Octree, loose Octree and MX-CIF Octree. Uniform data, size [0,1], benchmark B.](image1)

**Synthetic Dataset.** We first compare the approaches on uniformly distributed data using benchmark B (300 units). As Figure 11 shows, although the Octree is very efficient on point data, it does not scale well in case of objects with spatial extent. The loose Octree improves performance but because a considerable share of the tree has to be traversed for a range query, the improvement is barely noticeable. The R-Tree as well as the CR-Tree perform better despite of overlap but BLOCK proves to be the most efficient approach. The quantization of the MBR’s in the CR-tree incurs more overlap thus reducing the benefits of its smaller size. Furthermore testing for intersection in the CR-Tree, i.e., testing the quantized queries against the quantized MBR’s, is computationally more complex than in the R-Tree.

**Neuroscience Dataset.** As a test to demonstrate BLOCK's usefulness for real world applications, we experiment with a dataset from neuroscience featuring up to 450 million cylinders which model the spatial structure of one neuron. In this experiment we use benchmark B (300 units), modeled after a visualization use case from neuroscience with comparatively big queries. The results in Figure 12 show that BLOCK is $7 \times$ faster than the STR-bulkloaded R-Tree. BLOCK also clearly outperforms the loose Octree and the Octree. We do not include the measurement for either Octrees for a dataset of 450 million objects as the query execution takes longer than 500 seconds.

The super-linear increase in R-Tree execution time is due to the dataset density which creates large overlapping MBR’s in the R-Tree. As we will also see in more detail in the next experiment, due to the comparatively large objects, the Octree needs to traverse multiple paths to answer a query and is thus less efficient. The loose Octree is similarly inefficient when indexing large spatial objects despite the reduction in object replication based on the loose branch borders.

![Figure 13: Comparison of BLOCK, STR-bulkloaded R-Tree and Octree on different object sizes.](image2)

**Object Size Effect.** Spatial data can scale both in density (more objects) as well as in size. By indexing large objects on the coarser grids BLOCK avoids scanning through many cells in order to deduplicate results in contrast to the R-Tree structures where bigger objects lead to increasing overlap.

In a final experiment we thus test different approaches (we only take the fastest representative from the R-Tree as well as Octree family of approaches) for 30 million uniformly distributed objects.
with different sizes (from 0 to 1, from 0 to 5 and from 0 to 50). As the results in Figure 13 show, BLOCK indeed outperforms the R-Tree which suffers from overlap.

7. BLOCK ANALYSIS

In the following we analyze BLOCK in more detail. We study the impact of different configurations for indexing on query execution performance. BLOCK has two different configuration parameters for indexing: the number of levels and their respective granularity. Configuring BLOCK optimally depends on the distribution of size and location of the objects in the dataset as well as of the queries. While the former can be determined before configuring the index, the queries cannot be known a priori, making the optimal configuration challenging. The considerations discussed in Section 5 can be used to determine an efficient configuration as we demonstrate with experiments.

In the following we consequently test different configurations for indexing (number of levels as well as their resolution). During query execution BLOCK uses the configuration cost model (see Section 5) to decide how to split the queries and on what levels to execute the parts of the query.

In a first experiment we test different BLOCK configurations on a uniform dataset using microbenchmark C (300 units). We set the grid resolution of the level with the finest granularity according to the cost model (discussed in Section 5) to $2^{27}$ cells because for an average object size of 5, the finest granularity that does not degrade performance uses $2^{27}$ cells. We compare the performance of this configuration with the immediately finer and with coarser grained configurations as well as with a configuration with two levels.

As the results in Figure 14 show, the single level grid with granularity $2^{27}$ performs best. Using any other configuration results in degraded performance. For coarser granularities performance degrades due to unnecessary intersection tests because of bigger cells and for finer granularities, e.g., $2^{30}$, due to the longer intersection lists and random memory access. The BLOCK configuration using two levels outperforms all single level configurations, including the one with $2^{27}$ cells. The addition of the coarser level with $2^{21}$ cells to the configuration allows some queries (or at least parts of them) to be executed on it as they may enclose entire cells. Clearly, adding one level improves performance and as we have shown previously, the overhead of additional levels is minimal: the memory and build time overhead are not substantial.

The performance of configuration $2^{27}, 2^{21}, 2^{15}$ is slower because with this query size, the level with granularity $2^{15}$ cannot be used as its cells are bigger than the query. Configurations with larger steps than the one chosen (such as $2^{27}, 2^{21}$) have poorer performance as well because of the small size of objects. A configuration with a more fine-grained top level of granularity ($2^{13}, 2^{27}, 2^{21}$) finally crosses over the trade-off point shown in the cost model and thus executes inefficiently.

In our experiments, multiple levels with properly chosen granu-
larities always outperform single level configurations of BLOCK. The breakdown shows that the configurations with multiple levels save time on the execution plan calculation and on the object intersection tests. Time in the intersection tests is saved because on multiple levels, more cells are fully enclosed in the range query. Depending on the query position the intersection tests can be reduced by approximately 50% when using multiple levels as this experiment demonstrates. As discussed, enclosing more cells fully when using multiple levels also reduces the overhead of object retrieval: on coarser levels the intersection lists are smaller and therefore less random main memory access is required.

8. CONCLUSIONS

The bottlenecks of spatial indexing have shifted in recent years when an increasing number of datasets started to fit into memory. While disk-based approaches were disk-bound and hence optimized to reduce (particularly random) access to the slow disk, in-memory approaches are CPU-bound and need to reduce computations in general and the number of objects tested for intersection with the query range in particular.

Clearly, as the bottlenecks have shifted, indexes should be adapted for the new medium as well. With BLOCK we have developed an efficient approach for the execution of spatial range queries in memory. By effectively reducing the number of intersection tests needed in the index structure and to test objects for intersection with the query, BLOCK executes queries up to 7 times faster than competing approaches and scales better.

As we demonstrated with the experiments in Figure 14 the multilevel grid scales better than the single level and from Figure 11 follows that it performs better than the state of the art. Despite the multiple levels and the added complexity, the build timescales both with the query range in particular.

9. REFERENCES