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

Matthaios Olma
École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland

Farhan Tauheed
Oracle Labs
Zurich, Switzerland

Thomas Heinis
Imperial College
London, United Kingdom

Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne
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 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.

CCS CONCEPTS

• **Information systems** → **Multidimensional range search**;

ACM Reference format:
Matthaios Olma, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2017. BLOCK: Efficient Execution of Spatial Range Queries in Main-Memory. In *Proceedings of SSDBM '17, Chicago, IL,USA, June 27-29, 2017,* 12 pages.
https://doi.org/http://dx.doi.org/10.1145/3085504.3085519

1 INTRODUCTION

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

Spatial datasets in the past oftentimes 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 can store many spatial partially or entirely today. 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).

More formally, given volumetric objects $d \in$ dataset D stored in main memory, 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 $R(q) = d|q \cap d$, i.e., all objects $d \in D$ intersecting with q. Like most approaches to spatial indexing [8] we also 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 [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). Our in-memory spatial index, on the other hand, is designed for volumetric objects and thus 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. 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, to further reduce intersection tests.

While grids have been used before to index spatial data, the main 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 reduce intersection tests. Third, while indexing a dataset with several indexes typically incurs overhead, we develop a novel, space-efficient storage layout for the index structure. Our experiments on real neuroscience data show a speedup of up to 700% compared to existing spatial indexes.

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 [12, 26] (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 straightforward implementation. To alleviate the problem of replication, the loose Octree [30] 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 UB-Tree performance. A much simpler space-oriented index is the grid [6] used for static but also 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 (of replicated objects) and (c) potential deduplication overhead because of intersections detected multiple times (due to replicated objects).

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 directory entries if the cells are sparsely filled. For improved memory efficiency the twin grid file [13] improves the grid file by constant overflows but it does not offer a scalable solution.

The KDB-Tree [6] and the Bkd-Tree [24] are both based on the KD-Tree [5], a space-oriented partitioning index used in memory. Both are designed to maximize space utilization, i.e., filling disk pages as much as possible. Both approaches, however, are designed to index point datasets and cannot be used for volumetric objects.

To index imaging data, the pyramid tree [1] uses a tree to hierarchically organize uniform grids with different resolution. On the most fine-granular grid G_1 the image is indexed in detail, whereas on more coarser grained grids G_n with $n > 1$ only a summary of the finer level is stored, i.e., by combining several cells of G_n into one cell of G_{n+1}. Querying on higher levels requires less data to be retrieved and fewer comparisons need to be made.

The ST2B-Tree [7] is designed for indexing moving objects. It indexes all spatial objects in a B-Tree based on the object's ID assigned by using a space filling curve. At query time, the index is adapted by changing the resolution of the space filling curve in a subtree depending on how objects move. If, for example, many objects move into one area, the resolution of the space filling curve in this area is increased. With this the index becomes fully tunable, reacting to the workload. ST2B, however, is solely designed for point datasets and cannot easily be extended to volumetric objects.

2.2 Data-oriented Partitioning

Access methods based on data-oriented partitioning, decompose 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 partition can be chosen to fill disk pages. A directory structure needs to be used to execute range queries because the decomposition of space is irregular in all dimensions. Several data-oriented indexing approaches use a hierarchical directory structure that leads 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 R-Tree [10]. Widely used today, the R-Tree has been designed as a disk-based multidimensional generalization of the B-Tree [3]. The R-Tree packs the objects on disk pages using data-oriented partitioning and organizes the data in a hierarchical structure. Introducing a hierarchical tree structure, however, also introduces the problem of overlap that degrades R-Tree performance.

Several strategies have been devised to tackle the overlap problem. The R*-Tree [4] and the R+-Tree [27] are 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 R-Tree have been proposed to bulkload datasets which are known a priori. The Hilbert R-Tree [16], Sort-Tile-Recursive (STR) [19], Top down Greedy Split (TGS) [9] as well as the Priority R-Tree (PR-Tree) [2] all use different sort and split strategies to build an R-Tree from the dataset. While Hilbert and STR build efficient R-Trees on many real-world datasets, TGS and PR-Tree do so for extreme, mostly synthetic datasets (extreme skew and aspect ratio). Despite the numerous improvements to the basic R-Tree the fundamental problems of overlap and dead space remain. 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, data access is more efficient.

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 5×10^{-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

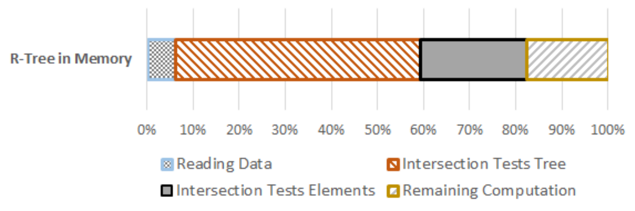

Figure 1: Query execution breakdown in memory R-Tree.

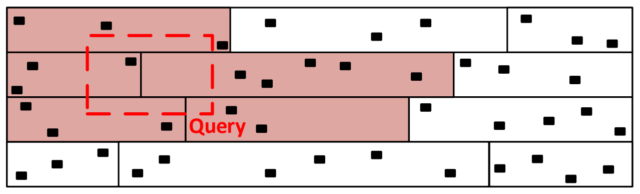

Figure 2: A range query intersecting with narrow partitions (shaded) leads to unnecessary tests of objects.

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 datatsets [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 compressioen) 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: using indexes based on data-oriented partitioning can lead to partitions of which only a small fraction indeed intersects with the query. Still, all objects in the partition need to be tested for intersection, leading to many unnecessary 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

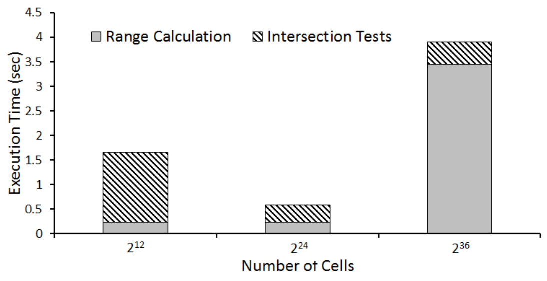

Figure 3: Performance impact of resolution.

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 result can be computed, resulting in pointer chasing and comparison tests that increase the execution time of range queries.

4 THE BLOCK APPROACH

Motivated by a scientific application — the analysis of brain models in neuroscience — we design BLOCK a novel in-memory index for the efficient execution of two types of range queries q, (i) window queries, i.e., all objects intersecting with q and (ii) containment queries, i.e., all objects fully contained in q [8].

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- and data-oriented spatial indexing. Since our motivating application only requires the efficient range queries execution on static datasets, we do not design BLOCK to support updates.

4.1 Rationale & Contributions of BLOCK

To minimize the number of intersection tests and avoid the inherent problem of overlap we refrain from using hierarchical structures or unbalanced directory trees and base BLOCK on uniform grids. In a uniform grid the partitions are quadratic and thus lead to fewer unnecessary intersection tests (e.g., no narrow partitions). The biggest challenge in using uniform grids, however, is finding the best configuration, i.e., the resolution or cell width. The best configuration primarily depends on the size of the queries which generally is 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 randomly placed. We measure the total time to execute all queries. 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 chose the best configuration.

To address the issue of unneeded intersection tests when configuring a uniform grid, 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

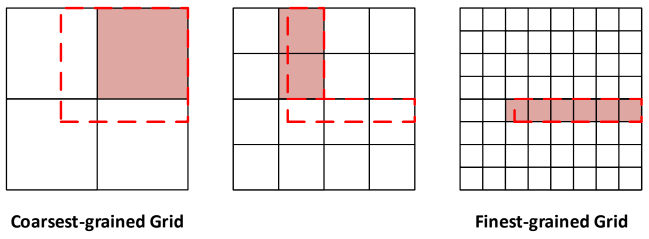

Figure 4: A query is executed on several uniform grids of different resolution.

least number of cells needs to be retrieved, therefore reducing the data retrieved 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 using 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 grids based on the benefit it may provide.

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.

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.

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_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. 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 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. Figure 5 illustrates the correlation of BLOCK's components.

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_l on level l has an entry that stores a *containment list* and *intersection list*.

Containment 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.

Intersection list. 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 typically not stored consecutively in the object store and thus we store pointers to individual objects in a linked list. Nevertheless, if at least two objects in the list are stored adjacent in the object store 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.

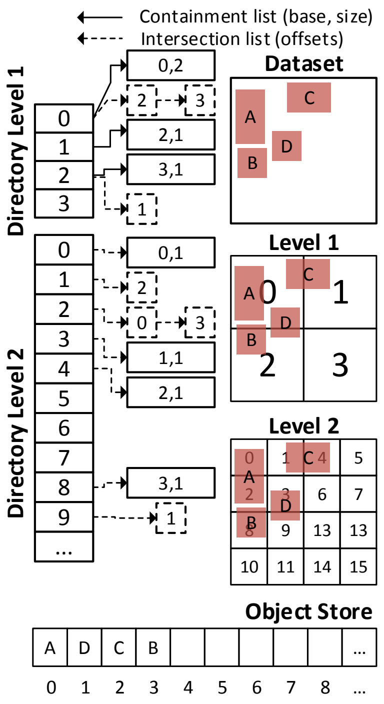

Figure 5: Data structures of BLOCK: the object store and the level directories.

4.4 Building the Index

To build the index structures for dataset D, BLOCK (i) builds the object store and (ii) builds the grid levels.

When building the object store, BLOCK sorts all objects $o_i \in D$ based on the z-value of their center and stores them in an array. Subsequently, BLOCK builds the directory of each level $l_i \in L$ by iterating over the object store and testing for each object o_i in which cell c_i o_i's center falls in and with what cells C o_i intersects.

For a cell c_i that contains the center of the object o_i, BLOCK appends to c_i's containment list a pointer to object o_i. 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 centers in the same cell are adjacent to each other in the object store as they are ordered by their z-value and we thus only to store the offset and the number of objects of this group, thereby effectively compressing the pointers.

Accelerated building. 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_{coarse} exactly contains several cells on level l_{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_{coarse} from a finer-grained level l_{fine}, the directory entries of cells C_{fine} of l_{fine} contained in a cell c_{coarse} of l_{coarse} are combined.

More precisely, all the containment lists of C_{fine} are merged to obtain the containment list of c_{coarse}. To obtain the intersection list of c_{coarse} the intersection lists of C_{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 $c_j \in C_{fine}$ and in the intersection list of a different cell $c_k \in C_{fine}$. o_i is thus contained in both lists of c_{coarse} and has to be removed from the intersection list (because it already is in the containment list).

Pointer compression. 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. The pointers to the object store can consequently 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. The novel storage approach of BLOCK enables to store multiple grids space- and time-efficient.

4.5 Query Execution

To execute a range query rq on a single level BLOCK computes all cells C_{rq} intersecting rq. The cells are divided into two categories (i) fully contained and (ii) intersecting.

Fully contained. Initially, BLOCK computes all cells contained entirely in the query ($c_{contained} \in C_{rq}$) and retrieves their intersection and containment lists. As the cells $c_{contained}$ 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 thus reduced cache misses).

The list resulting from merging the intersection lists of all cells $c_{contained}$ 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 $c_{contained}$ 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).

Partial intersection. BLOCK uses a similar approach to retrieve the cells $c_{intersect}$ which are only intersecting (but are not contained) in rq. The difference is 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. Specifically, the bit array stores a bit for every object in the dataset, the bit is set when an object is returned. To reduce the required space and provide high access and update efficiency, the bit array uses run length encoding.

Multi-level execution. 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 $c_{contained}$ and retrieves the objects in these cells as described before. For the cells $c_{intersect}$, BLOCK decides dynamically whether to retrieve the cells $c_i \in C_{intersect}$ 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 level proceeds as before and may recursively execute remainders of the query on all levels.

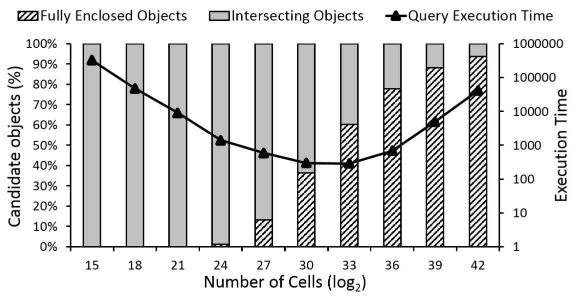

Figure 6: Cost vs Benefit balance.

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.

BLOCK bases the decision when to split and where to execute the query on the cost and benefits of using a given 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. The benefit of executing parts of the query on finer-grained level, on the other hand, is avoiding intersection tests due to more fully contained cells by the query.

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.

BLOCK therefore uses a cost model ensuring that finer-grained levels are only used if the benefit (avoiding unnecessary intersection tests) of their use is bigger than the cost (random memory accesses). Equation 1 captures the cost model on a high level (Table 1 provides a description of the variables used). Query execution has three major cost factors: (1) calculation of the execution plan, (2) scanning a number of cells on each level that are fully contained within the query and (3) scanning a number of cells on each level that the objects have to be tested for intersection with the query.

$$Q_{cost} = Calc + \mathop{}_{i=1}^{L} (D_i \cdot Scan_i + ND_i \cdot (Scan_i + Test_i)) \quad (1)$$

Equation 2 further breaks down the cost of scanning the objects contained in a cell. The cost of object retrieval from a

Variable	*Description*
L	The set of level granularities
X	Size of the universe dimensions
N	Number of objects in the dataset
ow	Average object width
Q_{cost}	Query cost
$Calc$	Calculation of execution strategy
$Scan_i$	Cost of scanning a cell on level i
$Test_i$	Cost of testing a cell on level i
cw_i	Cell width on level *i*
D_i	Number of fully covered cells on level i
ND_i	Number of not fully covered cells on level i
I_i	Number of objects only intersecting a cell
C_i	Number of objects having center in a cell
O_i	Average number of objects in cell on level i
Intersection TestCost	Average cost of intersection test
SeqMem AccessCost	Average sequential memory access cost in seconds
RandomMem AccessCost	Average random memory access cost in seconds

Table 1: Variables used in the cost model.

cell has two major components: (1) $SeqAccessScan_i$, the cost for the sequential memory access to read all objects whose centers are in the cell, and (2) $RandAccess_i$, the cost random memory access to retrieve the objects that intersect but center is not in the cell.

$$Scan_i = SeqAccessScan_i + RandAccess_i \tag{2}$$

Sequential Access. Assuming C_i the number of objects in a cell on level *i* and *SeqMem AccessCost* the average sequential memory access cost in seconds is as follows:

$$SeqAccessScan_i = C_i * SeqMemAccessCost \tag{3}$$

Random Access. Assuming I_i the number of objects intersecting a cell but their center being in a different cell on level *i* and *RandomMemAccessCost* the average random memory access cost in seconds.

$$RandomAccessScan_i = I_i * RandomMemAccessCost \tag{4}$$

The benefit, on the other hand, is avoiding unnecessary intersection test through fully contained cells. Equation 5 gives the idea of 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.

$$Test_i = NumberOfObjects_i * IntersectionTestCost \tag{5}$$

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 granularity. The number of levels corresponds to the number of different granularities BLOCK will use and the granularity is the number of cells a level uses.

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.

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 $cw_0 = \frac{width_{query}}{2}$ with $width_{query}$ 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 least the center cell can be enclosed by a query that is not aligned.

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 avoiding intersection tests is smaller than the cost of randomly accessing memory for intersection lists retrieval. 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 1 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 resulting 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 $cw_i/2$ 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. The finest-grained level for BLOCK is chosen so that (6) applies.

$$Scan_{i+1} - Scan_i \leq O_i * IntersectionTestCost \tag{6}$$

To decide on the resolution of the finest-grained without information about the data we use an anlytical model to determine $Scan_i$ cost for each level. In the following analysis we assume that the distribution of objects is uniform. Based on the equations used to calculate $Scan_i$ (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 or cells in a level).

$$C_i = \frac{N}{G_i} \tag{7}$$

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, (7) becomes:

$$C_i = \frac{N * {cw_i}^3}{X^3} \tag{8}$$

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.

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:

$$Iarea_i = ((cw_i + ow)^3 - {cw_i}^3) \tag{9}$$

Multiplying $Iarea_i$ with the average area per object, we obtain:

$$I_i = \frac{((cw_i + ow)^3 - {cw_i}^3) * N}{X^3} \tag{10}$$

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

$$\begin{aligned} Scan_i = \frac{N}{X^3} * ({cw_i}^3 * SeqMemAccessCost \\ + ((cw_i + ow)^3 - {cw_i}^3) * (RandomMemAccessCost)) \end{aligned} \tag{11}$$

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 random memory access. If the input dataset contains large objects it is important to avoid adding multiple fine-grained levels as this increases indexing time unduly. Instead, levels with coarser resolutions (closer to the coarsest resolution) should be used for the levels added. Similarly, for very small objects finer-grained levels should be added to reduce intersection tests as random access is unlikely.

6 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) as well as real spatial datasets from neuroscience.

6.1 Setup

Hardware: The experiments are conducted in a Sandy Bridge server with a dual socket Intel(R) Xeon(R) CPU E5-2660 (8 cores per socket @ 2.20 Ghz), equipped with 64 KB L1 cache and 256 KB L2 cache per core, 20 MB L3 cache shared, and 128 GB RAM running Red Hat Enterprise Linux 6.5 (Santiago - 64 bit) with kernel version 2.6.32. The server is equipped with a RAID-0 of 7 250GB 7500 RPM SATA disks.
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: We use a parameter sweep to find the best possible configuration for every index. In the experiments we compare BLOCK against the Octree with node size of 50'000 objects, a loose Octree with overlap factor of 0.5, a MX-CIF tree, an in-memory implementation of STR-bulkloaded R-Tree (superior performance compared to the R*-Tree [19]) using a page size of 4KB and a fanout of 111 and a CR-Tree with 30 bit quantization and fanout of 20 for both inner and leaf nodes. The resolution/granularity BLOCK uses on a level is denoted by the number of cells on the level. The maximum resolution is 2^{42}. In all experiments, BLOCK uses the query execution cost model developed in Section 4.5 (Query Execution) to decide what part of the query to execute on what level. We use the configuration cost model developed in Section 5 (Block Configuration) to set the configuration of BLOCK, i.e., to determine the configuration of the levels.

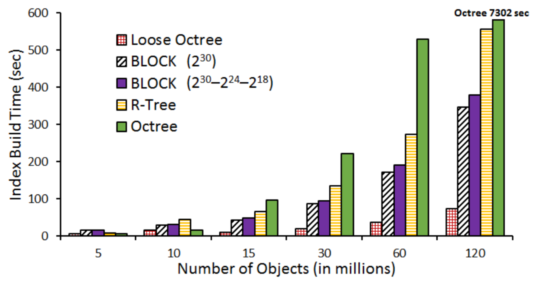

Figure 7: Index build time for 30M uniformly distributed objects of size [0,1].

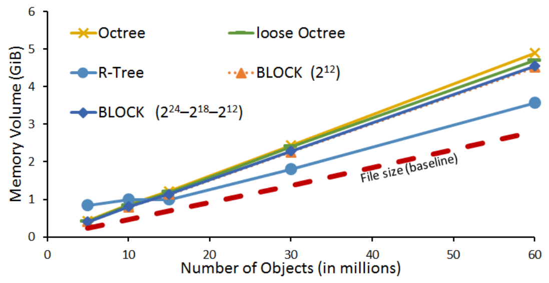

Figure 8: Memory footprint of different index approaches.

6.2 Experimental Methodology

Datasets: To demonstrate the general applicability of our approach and to stress that we do not exploit any particularity of the 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 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. Futhermore, 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 are identical to 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

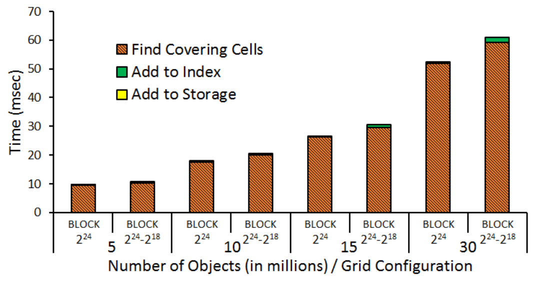

Figure 9: Breakdown of Index build time for Normally distributed objects of size [0,5]

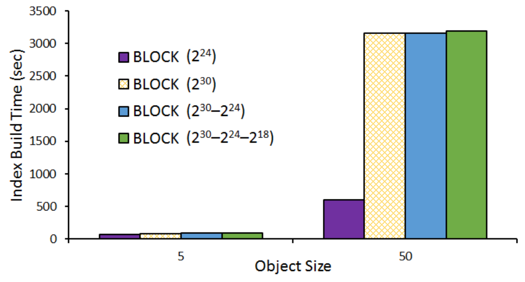

Figure 10: Index build time for file size of 30M uniformly distributed objects.

thus requires less memory than the previous one. As 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. On the other hand, adding the cell to the index, i.e., to the level directories, as well as to the object store takes insignificant time.

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 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.

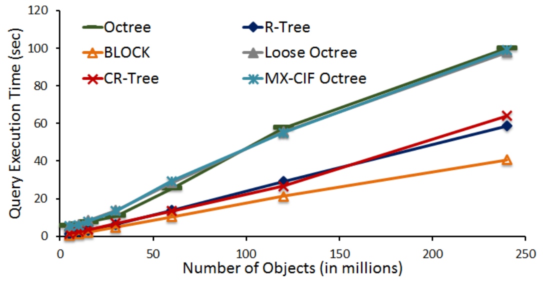

Figure 11: Comparing BLOCK, STR R-Tree, CR-Tree, loose and MX-CIF Octree. Uniform data, size [0,1], bench. B.

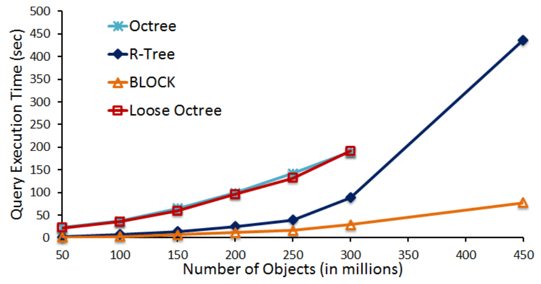

Figure 12: Comparison of BLOCK, STR R-Tree, loose Octree, Octree on neuroscience data for queries of benchmark B.

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 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 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× faster than the STR-bulkloaded R-Tree. BLOCK also clearly outperforms the loose Octree and the Octree. We do not include the results for either Octrees for 450 million object dataset 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

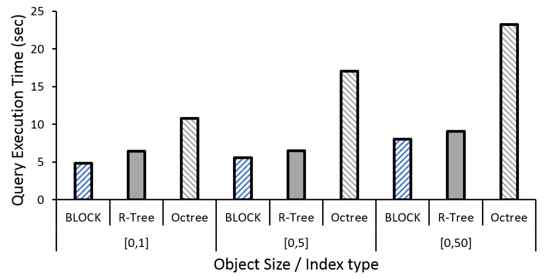

Figure 13: Comparison of BLOCK, STR-bulkloaded R-Tree and Octree on different object sizes.

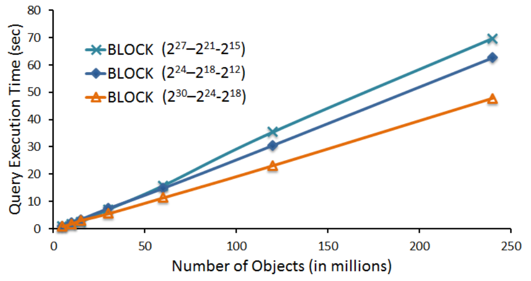

Figure 14: Compare multi-level configurations on Uniformly distributed objects with size [0,5] for benchmark B.

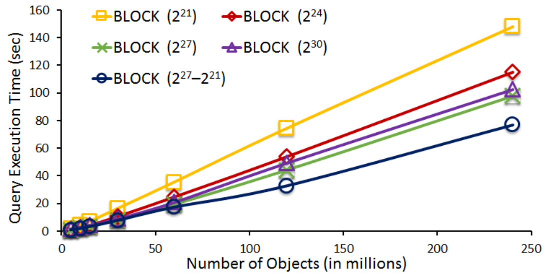

Figure 15: BLOCK compared single to multi-level grids on uniform distributed objects of size [0,5] for benchmark B.

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.

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, 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 are not known a priori, making optimal configuration challenging. Considerations discussed in Section 5 can be used to choose 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 15 show, the single level grid with granularity 2^{27} performs best. Using any other configuration results in degraded performance. For coarser granularities (fewe cells) performance degrades due to unnecessary intersection tests because of bigger cells and for finer granularities (more cells), 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 memory and build time overhead of additional levels is minimal.

Figure 14 shows a comparison between multi-level BLOCK configurations over uniform data using benchmark B (300 units). Using the configuration cost model for BLOCK developed in Section 5 (and given measured parameters for cost of intersection test and for memory access) suggests a three level configuration with levels 2^{27},2^{24},2^{21}. As this experiment confirms, the configuration suggested by the cost model indeed proves to be the most efficient.

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^{33},2^{27},2^{21}) finally crosses over the trade-off point shown in the cost model and thus executes inefficiently.

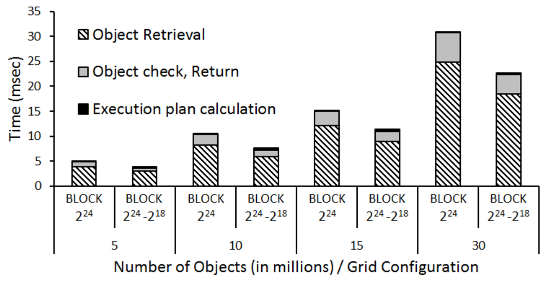

Figure 16: Query execution breakdown.

7.1 BLOCK Query Execution Breakdown

We break the query execution operation into three major parts: (1) *execution plan calculation*, i.e., the calculation of the cells covering the query, (2) *object retrieval*, the iteration through the index and gathering the objects within the query and finally, (3) *object check and return*, the validation that objects read indeed intersect with the query range and then return them to the user. As the result of the experiment with the benchmark C (750 units) on normally distributed files and an object size between 0 and 5 in Figure 16 shows, the object retrieval part dominates the overall time. In the object retrieval part, the objects both from the containment and intersection lists are retrieved.

In our experiments, multiple levels with properly chosen granularities 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, BLOCK executes queries up to 7 times faster than competing approaches.

As we demonstrated with the experiments in Figure 15 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 time scales on object size, number of objects and number of levels. Similarly the memory overhead is lower than the Octree and it does not grow substantially with more levels.

Acknowledgments. We would like to thank the reviewers for their valuable comments. This work is partially funded by the EU FP7 programme (ERC-2013-CoG), Grant No 617508 (ViDa), the EU FP7 Collaborative project Grant No 317858 (BigFoot) and EU Horizon 2020 research and innovation programme Grant No 650003 (Human Brain project).

REFERENCES

[1] Walid G. Aref and Hanan Samet. Efficient Processing of Window Queries in the Pyramid Data Structure. In *PODS '90*.
[2] Lars Arge, Mark Berg, Herman J. Haverkort, and Ke Yi. The Priority R-tree: a practically efficient and worst-case optimal R-tree. In *SIGMOD '04*.
[3] Rudolf Bayer. 1971. Binary B-Trees for Virtual Memory. In *Proceedings of the Workshop on Data Description, Access and Control*.
[4] Norbert Beckmann, Hans Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD '90*.
[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975).
[6] Jon Louis Bentley and Jerome H. Friedman. 1979. Data Structures for Range Searching. *Comput. Surveys* 11, 4 (1979).
[7] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A. Nascimento. ST2B-tree: A Self-tunable Spatio-temporal B+-tree Index for Moving Objects. In *SIGMOD '08*.
[8] Volker Gaede and Oliver Guenther. 1998. Multidimensional Access Methods. *Comput. Surveys* 30, 2 (1998), 170–231.
[9] Yván J. García, Mario A. López, and Scott T. Leutenegger. A Greedy Algorithm for Bulk Loading R-trees. In *GIS '96*.
[10] Antonin Guttman. R-trees: a Dynamic Index Structure for Spatial Searching. In *SIGMOD '84*.
[11] David Hilbert. 1891. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.* 38 (1891), 459–460.
[12] Gisli R. Hjaltason and Hanan Samet. Improved Bulk-loading Algorithms for Quadtrees. In *GIS '99*.
[13] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. The Twin Grid File: A Nearly Space Optimal Index Structure. In *EDBT '88*.
[14] Sangyong Hwang, Keunjoo Kwon, SangK Cha, and ByungS Lee. Performance Evaluation of Main-Memory R-tree Variants. In *SSTD '03*.
[15] Chris L. Jackins and Steven L. Tanimoto. 1980. Oct-trees and Their Use in Representing Three-dimensional Objects. *Computer Graphics and Image Processing* 14, 3 (1980).
[16] Ibrahim Kamel and Christos Faloutsos. Hilbert R-Tree: An Improved R-Tree using Fractals. In *VLDB '94*.
[17] Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. In *SIGMOD '01*.
[18] Nick Koudas and Kenneth Sevcik. Size Separation Join. In *SIGMOD '97*.
[19] S. T. Leutenegger, M. A. Lopez, and J. Edgington. STR: a Simple and Efficient Algorithm for R-tree Packing. In *ICDE '97*.
[20] Henry Markram. 2006. The Blue Brain Project. *Nature Reviews Neuroscience* 7, 2 (2006), 153–160.
[21] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems* 9, 1 (1984), 38–71.
[22] Jack A. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In *SIGMOD '86*.
[23] Giuseppe Peano. 1890. Sur une courbe, qui remplit toute une aire plane. *Math. Ann.* 36, 1 (1890), 157–160.
[24] Octavian Procopiuc, Pankaj K. Agarwal, Lars Arge, and Jeffrey Scott Vitter. Bkd-Tree: A Dynamic Scalable kd-Tree. In *SSTD '03*.
[25] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, and Rudolf Bayer. Integrating the UB-Tree into a Database System Kernel. In *VLDB '00*.
[26] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *Comput. Surveys* 16, 2 (1984), 187–260.
[27] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB '87*.
[28] Darius Sidlauskas, Simonas Saltenis, Christian W. Christiansen, Jan M. Johansen, and Donatas Saulys. Trees or grids?: Indexing Moving Objects in Main Memory. In *GIS '09*.
[29] Farhan Tauheed, Laurynas Biveinis, Thomas Heinis, Felix Schürmann, Henry Markram, and Anastasia Ailamaki. Accelerating Range Queries For Brain Simulations. In *International Conference on Data Engineering (ICDE '12)*.
[30] Thatcher Ulrich. 2000. Loose Octrees. In *Game Programming Gems*.