

GeoBlocks: A Query-Cache Accelerated Data Structure for Spatial Aggregation over Polygons

Christian Winter Andreas Kipf^{*} Christoph Anneser

Eleni Tzirita Zacharatou[◊] Thomas Neumann Alfons Kemper

Technische Universität München MIT CSAIL^{*} Technische Universität Berlin[◊]

{winterch, anneser, neumann, kemper}@in.tum.de kipf@mit.edu eleni.tziritazacharatou@tu-berlin.de

ABSTRACT

As individual traffic and public transport in cities are changing, city authorities need to analyze urban geospatial data to improve transportation and infrastructure. To that end, they highly rely on spatial aggregation queries that extract summarized information from point data (e.g., Uber rides) contained in a given polygonal region (e.g., a city neighborhood). To support such queries, current analysis tools either allow only predefined aggregates on predefined regions and are thus unsuitable for exploratory analyses, or access the raw data to compute aggregate results on-the-fly, which severely limits the interactivity. At the same time, existing pre-aggregation techniques are inadequate since they maintain aggregates over rectangular regions. As a result, when applied over arbitrary polygonal regions, they induce an approximation error that cannot be bounded.

In this paper, we introduce GeoBlocks, a novel pre-aggregating data structure that supports spatial aggregation over arbitrary polygons. GeoBlocks closely approximate polygons using a set of fine-grained grid cells and, in contrast to prior work, allow to bound the approximation error by adjusting the cell size. Furthermore, GeoBlocks employ a trie-like cache that caches aggregate results of frequently queried regions, thereby dynamically adapting to the skew inherently present in query workloads and improving performance over time. In summary, GeoBlocks outperform on-the-fly aggregation by up to three orders of magnitude, achieving the sub-second query latencies required for interactive exploratory analytics.

1 INTRODUCTION

Nowadays, the amount of geospatial data collected in cities is increasing rapidly, thanks to the widespread use of mobility applications such as Uber [53]. To analyze this data and make data-driven decisions, city officials and planners often rely on visualization frameworks that allow users to visualize data of interest at different spatial and temporal resolutions [4, 8, 41, 50, 53]. To generate common visualizations, such as heatmaps, visual tools perform *spatial aggregation queries* that partition the data over different polygonal-shaped regions and then compute summarized aggregate information for each region. To support *exploratory analyses*, visual tools must provide interactive response times as high latency reduces the rate at which users make observations, draw generalizations, and generate hypotheses [22]. However, the sheer size of the data combined with the complexity of spatial queries prohibit interactivity, which severely limits analyses. As shown in [28], current tools operating over raw geospatial data cannot produce results fast enough for interactive analysis.

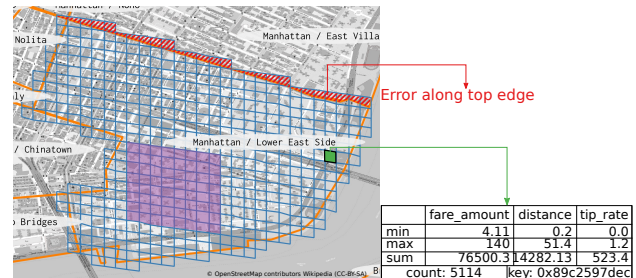


Figure 1: Cell covering (blue) of the Lower East Side (border in orange) with bounded error (red), a cell aggregate (green), and a cached commonly queried region (purple).

On the bright side, interactive analyses are often repetitive in nature. Analysts, for example, typically run multiple aggregate queries for the same area (e.g., the city center) in a sequence, changing only the aggregate function (e.g., count, sum) or the data attribute over which the aggregation is performed. Furthermore, they often focus on certain geospatial regions during their analysis. They might, for example, iteratively resize the boundary of the spatial region of interest, extracting an aggregate every time, or calculate aggregates for neighboring, potentially overlapping, regions. Such analyses can greatly benefit from query-driven materialization approaches that store and reuse intermediate or even full query results.

Naturally, in classical OLAP settings, query-driven materialization and result recycling are widely used and well understood [24, 35, 42, 45]. However, these methods do not address multi-dimensional spatial data. While methods have also been proposed for spatio-temporal OLAP queries, such as nanocubes [21] and the aR-tree [30, 31], these do not provide *precision guarantees* for spatial aggregation queries over *arbitrary polygons*. Both nanocubes and the aR-tree store aggregate information in a hierarchy of rectangles, maintained using a quadtree and an R-tree, respectively. Therefore, they are designed for aggregate queries over rectangular regions while their precision depends on the granularity of the underlying index structure. Using them to compute aggregates over *polygonal* regions introduces an approximation error, which *cannot be bounded*. There are also some analysis tools, such as Uber Movement [53], that rely on pre-computation to provide exact results for spatial aggregations over polygons. However, they require the polygonal regions to be pre-defined at aggregation time. This assumes a priori knowledge of the workload and is thus not applicable in *exploratory* analyses, where the query polygons are chosen *ad-hoc*.

We propose GeoBlocks, a novel pre-aggregating data structure for geospatial point data that guarantees error-bounded results for spatial aggregation queries over arbitrarily shaped polygons. Essentially, GeoBlocks are materialized views on geospatial point data that pre-compute filters and aggregations on pre-defined

© 2021 Copyright held by the owner/author(s). Published in Proceedings of the 24th International Conference on Extending Database Technology (EDBT), March 23-26, 2021, ISBN 978-3-89318-084-4 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

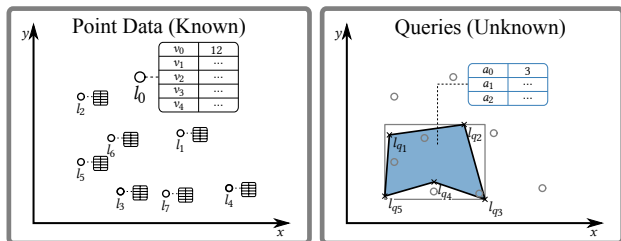


Table 1: Terminology

columns. Instead of pre-computing aggregates over a hierarchy of rectangles as in prior work, GeoBlocks pre-compute aggregates over *fine-grained grid cells*. As depicted in Figure 1, GeoBlocks subdivide the spatial domain into grid cells, keeping aggregates for each individual cell. We allow the user to specify the geospatial granularity, and thereby *bound* the spatial approximation error. In addition, we propose a trie-like data structure that caches aggregates for commonly queried regions in a compact manner, enabling even faster response times. GeoBlocks are designed for historical point data and are thus write-once/read-only. However, while GeoBlocks currently do not support updates, they can be adapted to do so, as we briefly discuss in Section 5. Our contributions are summarized as follows:

- We propose GeoBlocks, the first, to the best of our knowledge, data structure that supports spatial aggregation over arbitrary polygons, while guaranteeing a bounded error.
- We develop a query-driven caching mechanism that further accelerates aggregate queries by leveraging the skew commonly found in exploratory query workloads.

The advantages of our approach are amply clear from our extensive experimental evaluation on real-world data. The results show that GeoBlocks achieve up to three orders of magnitude speedup compared to on-the-fly aggregation approaches and support sub-second response times.

In the remainder of this paper, we first formalize the problem in Section 2. Section 3 describes our approach, which we then experimentally evaluate in Section 4. Section 5 summarizes the key points discovered in the evaluation and discusses updates for GeoBlocks. Finally, we present an overview of related work in Section 6 before concluding in Section 7.

2 PROBLEM STATEMENT

In this paper, we propose a new data structure to speed up the execution of spatial aggregation queries. Formally, the query can be defined in SQL-like notation as follows:

```
SELECT AGG(P.v0), . . . , AGG(P.vk) FROM P
WHERE P.l INSIDE R(lq1, lq2, . . . , lqm) [AND filterCondition]*
```

Given a set of annotated points of the form $P(l, v_0, v_1, \dots, v_n)$, where $l = (x_l, y_l)$ is the location of the point and v_i are numerical or temporal attributes, this query extracts multiple aggregates $a_i = AGG(v_i)$ over all the points contained in a query region R . The query region can be any *arbitrary polygon*, and its geometry is defined by the locations of the polygon's vertices $l_{q1}, l_{q2}, \dots, l_{qm}$. The aggregates are non-holistic functions such as count, sum, min, max, or average. Finally, the query can have zero or more filter conditions on the attributes.

cell	Rectangular area, hierarchically subdividable into four children
cell level	Number of subdivisions performed on the spatial domain to obtain the cell
cell id/spatial key	Unique one-dimensional identifier of a cell
block level	Level of grid cells in a GeoBlock
cell aggregate	Aggregates of all tuples of a grid cell
cell covering	Error-bounded approximation of a polygon using cells

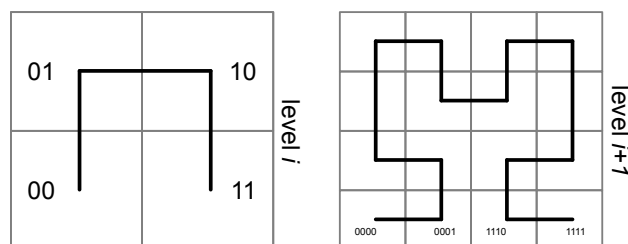


Figure 3: Hierarchical cell decomposition [16].

In exploratory interactive analyses, users can dynamically and unpredictably change not only the filtering conditions and the requested aggregates but also the polygonal query region. The data points, on the other hand, are known a priori. Figure 2 presents an example of this scenario: The left-hand side shows the input points that are located at (l_0, \dots, l_7) and have five attributes each. The right-hand side shows the query; the polygonal region is marked in blue, while three different aggregates are extracted. As can be seen in the figure, this query applies the aggregation over the three points that are contained in the query region, located at l_5, l_6 , and l_7 .

Existing approaches for spatial aggregation queries, such as the aR-tree [30, 31], are designed for rectangular regions, and thus *do not support arbitrary polygons*. Applying them to the example of Figure 2 requires to approximate the query polygon with a minimum bounding rectangle, displayed in grey, over which the aggregation is performed. This introduces an extra point in the results, l_3 , which is outside the actual query region.

3 GEOBLOCKS

In this section, we first present the geospatial decomposition that forms the basis of our approach. We then discuss how we can quantify and *bound* the error that this decomposition introduces. Next, we explain the core concepts of GeoBlocks, their storage layout, and the efficient evaluation of spatial aggregation queries using GeoBlocks. Finally, Section 3.6 outlines our query-driven caching mechanism that further improves performance by leveraging the characteristics of the query workload. Table 1 provides an overview of the concepts introduced in this section.

3.1 Geospatial Decomposition

GeoBlocks rely on a hierarchical, quadtree-based spatial decomposition. In this decomposition, a given area (cf. the outer rectangle in Figure 3) is recursively subdivided into equally-sized smaller areas that we call cells. Each cell has four children, which

leads to an exponentially growing number of 4^n cells after recursively subdividing a cell n times. We encode each subdivision using two bits, which allows us to uniquely identify a cell at level n by concatenating the encoding of levels 0 to n . Equivalently, all cells at a given level can be enumerated using an *order-preserving space-filling curve*. Since children cells share a common prefix with their parent cell, containment tests are reduced to efficient bitwise operations. This encoding further allows storing cell ids in prefix-encoded index structures such as radix trees [16, 17] or in learned indices [52] to speed up containment queries. Figure 3 shows the decomposition of a cell in four (level i) and 16 (level $i + 1$) sub-cells, and the corresponding enumeration with a Hilbert curve. Applying our decomposition strategy to the Earth’s surface, we only need 64 bits to address every single square centimeter. That way, we map two-dimensional geospatial locations (lat/long coordinates) to one-dimensional 64-bit keys. In our implementation, we use the Google S2 library [38] to perform the spatial decomposition and cell enumeration. Note, however, that our approach is not restricted to S2 or the Hilbert curve. Any other framework that supports recursive geospatial subdivisions and order-preserving cell enumerations can be used instead.

Point Approximation. We map locations (i.e., *points*) to the smallest cell that contains them. The imprecision introduced by this approximation (e.g., at most 6.1 mm for any point in the US) is negligible, as the imprecision of GPS data is often orders of magnitude worse [54].

Polygon Approximation. Similarly, we approximate the query polygons on-the-fly by mapping them to a set of cells, possibly at different levels, as shown in Figure 4 (center and right). We call this geometric approximation a *cell covering*. In our implementation, we calculate cell coverings using the S2 library.

3.2 Bounded Error

Similarly to all geometric approximations, our cell covering introduces a spatial error. This is because all the cells that intersect the polygon outline, even minimally, are considered to be part of the polygon. However, in contrast to other coverings like the widely used minimum bounding rectangle (MBR), our cell covering is much more fine-grained. As can be seen in Figure 4, the cell covering approximates the polygon outline much more closely compared to the MBR. More importantly, the introduced approximation error can be bounded. In fact, any point on the cell covering is within a distance $\sqrt{\epsilon_1^2 + \epsilon_2^2}$ from the polygon outline, where ϵ_1, ϵ_2 are the side lengths of the cell. Clearly, the smaller the cell size, the smaller the approximation error. Consequently, our cell covering can *guarantee* a user-defined error bound, i.e., a bound on the spatial distance between the approximate and the original polygon, by using an appropriately small cell size. The MBR cannot guarantee such a bound, because its spatial extent, and thus its distance from the polygon outline, depends on the polygon’s minimum and maximum coordinates in each dimension and cannot be controlled [52]. The user can specify the error bound by choosing an appropriate cell level¹ so that the cell’s diagonal is not greater than her desired error. This user-controlled and bounded spatial error is the only error in GeoBlocks. All further operations are exact and do not introduce any additional error. While the error bound should be the driving factor when selecting a cell level, there are other points to consider: (1) The cell diagonal is the maximum error, and the average error can

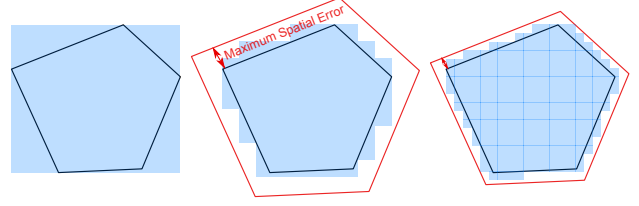


Figure 4: MBR (left) and two cell coverings with increasingly fine-grained resolution.

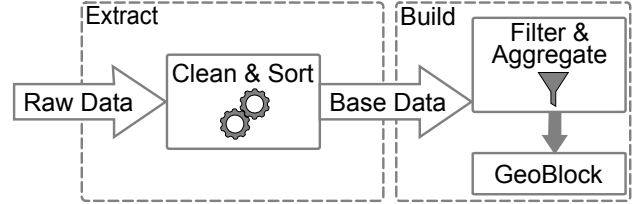


Figure 5: Creation of a GeoBlock in two phases. The extract phase is run once per dataset. The build phase is run for each filter and error bound combination.

be expected to be lower. (2) The cost of reducing the error is not linear. Per each level, the diagonal, and thereby the error bound, reduces by a factor of 2. At the same time, the number of grid cells, and thus the query input, grows by a factor of 4.

3.3 Preprocessing

In addition to transforming the two-dimensional input space to one-dimensional spatial keys, we perform some additional preprocessing steps on the known point data. Our process, outlined in Figure 5, consists of two phases, extract and build, and is similar to the ETL process traditionally applied in OLAP settings. In the first phase, we prepare the raw data by filtering outliers in the often dirty datasets and limiting the columns to those relevant and suitable for analysis. We furthermore sort the data by the generated one-dimensional spatial key. This extract phase is run exactly once per dataset and allows us to cheaply build GeoBlocks from the extracted base data. The second phase, build, utilizes the clean and sorted base data to generate a GeoBlock in a single pass and thus in linear time.

Updates and Filters. An important part of data analysis is filtering to gain insights into the desired subsets of the data. In our process, we could apply filters either before or after sorting the raw data. While the first option seems tempting, as it would reduce the number of tuples over which the expensive sorting has to be performed, we decided to filter the data in the build phase. This way, we can utilize the sorted base data to quickly build GeoBlocks for different filter predicates, aggregates, and grid resolutions in a single pass. Building new GeoBlocks quickly is especially useful in exploratory analyses, where the data and filters of interest might not be fully known a priori. However, the increased cost of sorting all data has to be amortized over multiple GeoBlocks and filter predicates. In reality, the sorting cost might be amortized immediately, as some exploratory queries might need to compare a subset of the data with the total. Consider, for example, a query comparing the tip rate of expensive taxi rides (`WHERE fare_amount > 20`) with that of all rides. In this case, we would need to build a GeoBlock for all rides, and therefore sort the entire dataset anyway.

¹From the table at https://s2geometry.io/resources/s2cell_statistics

Given k different filter predicates with average selectivity s and a total input size of n tuples, we can calculate the runtime of building isolated GeoBlocks with filters before sorting, and incremental builds from sorted base data as follows:

$$k * (O(n) + O(sn * \log(sn)) + O(sn)) \quad (1)$$

$$O(n * \log(n)) + k * O(n) \quad (2)$$

The isolated build (1) has three phases, cleaning and filtering in $O(n)$, sorting in $O(sn * \log(sn))$, and finally aggregating in $O(sn)$. Incremental builds (2) have a fixed component composed of cleaning and sorting in $O(n * \log(n))$, followed by the incremental filtering and aggregation of the GeoBlock in $O(n)$. For incremental builds to pay off, the sorting cost of the regular builds ($k * (O(sn * \log(sn)))$) has to outweigh the initial cost of the incremental builds ($O(n * \log(n))$). As we only have runtime classes for each variant and de-facto runtimes will vary between systems and datasets, we cannot determine when amortization is reached solely depending on k and s . However, we provide an in-depth experimental analysis of the amortization in Section 4.

3.4 Storage Layout

Once the filtering of the base data is completed, we can start aggregating and building a GeoBlock. To build a GeoBlock, for each grid cell in the decomposed space, we compute a number of aggregates over all the tuples that it contains. Empty cells that do not contain any tuples are omitted during aggregation as they would needlessly consume space. We refer to the aggregates of a grid cell as *cell aggregates*. A GeoBlock stores cell aggregates in ascending order of the cell's spatial key, which is the same sorting order as the one applied to the base data. Moreover, a GeoBlock maintains a global header that combines all cell aggregates into a single GeoBlock-wide aggregate and contains additional metadata required for querying, such as the minimum and maximum cell id in the GeoBlock.

Cell Aggregate. Each cell aggregate stores pre-computed answers for spatial aggregation queries at the grid cell level. A cell aggregate consists of the cell's spatial key, the base data offset of the first tuple contained in the cell, and the number of contained tuples. Furthermore, it maintains aggregates for all columns (both numeric and temporal attributes) in the extracted data. The maintained aggregates are the minimum, maximum, and sum of all values contained in the cell. Note that using the sum together with the tuple count allows us to also compute the average as sum/count. Furthermore, the cell aggregate stores the minimum and maximum keys of the spatial column. The table in Figure 1 shows an example of a cell aggregate.

Aggregate Granularity. As described in Section 3.2, the block level (i.e., the granularity of the space decomposition) is defined by the user at build time. However, it is also possible to adapt the granularity at a later time. Building a more coarse-grained GeoBlock from an existing one is rather straightforward and does not require re-scanning the base data. We can easily combine all cell aggregates of the finer-grained GeoBlock corresponding to a more coarse-grained grid cell in a single pass over the aggregates. On the other hand, building a more fine-grained GeoBlock requires scanning and further subdividing the base data.

3.5 Querying

GeoBlocks support two variants of spatial aggregation queries. On the one hand, they support regular SQL SELECT queries that take a query polygon and produce a user-defined subset of the

```

1 lastAgg = 0
2 def selectQuery(polygon):
3     queryCells = s2.coverPolygon(polygon)
4     # Prune search range
5     queryCells.pruneLess(globalHeader.minCell)
6     queryCells.pruneGreater(globalHeader.maxCell)
7
8     lastAgg = 0
9     resultAggregates = initial
10    for qcell in queryCells:
11        # Map qCell to smaller childCells at the block level
12        childCells = s2.childrenAtLvl(qcell, BLOCK_LVL)
13        for cell in childCells:
14            getAggregates(cell, resultAggregates)
15    return result
16
17 def getAggregates(cell, result):
18     # Check the last results successor
19     if lastAgg == 0:
20         # Search initial header
21         aggregate = allAggregates.upperBound(cell).prev
22         if aggregate.cell == cell:
23             combineAggregates(aggregate, result)
24         lastAgg = aggregate
25     else:
26         if lastAgg.next.cell == cell:
27             lastAgg = lastAgg.next
28             combineAggregates(lastAgg, result)

```

Listing 1: SELECT query

available aggregates. On the other hand, they support a specialized efficient implementation of COUNT queries that only report the number of points contained in a query polygon. Such COUNT queries are commonly used in analytics, especially in the context of visualization. Figure 1 shows an example query that extracts a set of aggregates over the Lower East Side region, which is approximated by a cell covering (marked in blue). The answer is calculated by extracting and combining all the aggregates contained in the blue cells.

To answer a spatial aggregation query over a polygonal region (Figure 6a), the polygon is approximated using a cell covering, as discussed in Section 3.1. We compute a cell covering that conforms to the error bound (Figure 6b). Note that the cell covering can have cells at different levels, and some of them might be larger than our grid cells. Such larger cells can be easily mapped to smaller grid cells (Figure 6c) in the GeoBlock and offer further optimization potential, as discussed next. The cell covering, however, cannot contain any cells smaller than the cells of the GeoBlock. Once we obtain the cell covering, we query the GeoBlock for each of the covering cells, as visualized for a SELECT query in Figure 6d. We then combine these partial results to compute the final result for the entire query polygon. In the following, we describe the query process for each cell of the covering. First, we use the GeoBlock's header to check if the cell overlaps with the GeoBlock at all. Thanks to the prefix-based containment checks, this is possible in constant time using the minimum and maximum cell id in the GeoBlock. Only if there is a possible overlap, we continue with the specific checks for SELECT and COUNT queries as follows:

SELECT Queries. SELECT queries have to look at all cell aggregates contained in the query cell. Listing 1 presents the pseudocode of the algorithm. After a query cell has passed the first check, we try to further limit the search space to the overlapping area (Lines 5 & 6). After splitting the query cell to smaller cells that match the GeoBlock's granularity if needed (Line 12), we locate the first intersecting grid cell using an upper-bound binary search (Lines 21 - 24). For all the following cells, we exploit the

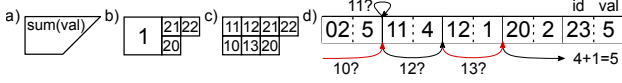


Figure 6: Query overview: Query polygon (a), cell covering (b), grid-cell representation of covering (c), and subquery for covering cell 1 in the cell aggregates (d, Listing 1 Line 12 and following).

```

1 def countQuery(polygon):
2     queryCells = s2.coverPolygon(polygon)
3     result = 0
4     for c in queryCells:
5         f_child = c.firstChildAtLvl(cell, BLOCK_LVL)
6         l_child = c.lastChildAtLvl(cell, BLOCK_LVL)
7         # Get first & last contained aggregate
8         first = allAggregates.lowerBound(f_child)
9         last = allAggregates.upperBound(l_child, first)
10
11         cnt = last.offset + last.count - first.offset
12         result += cnt
13     return result

```

Listing 2: COUNT query

fact that cell aggregates are stored contiguously in ascending order. This allows us to iterate over the cell aggregates (Lines 25 - 28) until we reach a grid cell not contained in the query cell, combining all cell aggregates along the way into the query result.

COUNT Queries. Intuitively, we can answer COUNT queries faster than SELECT queries, as we can exploit the sorted order of the cell aggregates to calculate the count without accessing the cell aggregates of all grid cells that are contained in the query cell. Specifically, COUNT queries can be answered using the count and offset values of only the first and the last cell aggregates that are contained in the query cell, as outlined in Listing 2. Note that here we benefit from having larger query cells. The larger the cells used in the covering, the fewer cell aggregates we need to access overall. To find the first and last cell aggregates, we calculate the id of the first and last child of the query cell at our grid level. We then locate the first child in the aggregates using a lower bound binary search (Line 8). Then, we use the position of the first child as a search start to locate the last child, again with a binary search (Line 9). Once we have located the aggregates of the first and last contained child, we can calculate (Line 11) the resulting count in a range-sum manner as:

$$\text{child}_{\text{last}}.\text{offset} + \text{child}_{\text{last}}.\text{count} - \text{child}_{\text{first}}.\text{offset}$$

3.6 Query-Cache Acceleration

While our cell aggregates can speedup queries significantly, there is further potential in pre-computing aggregates for frequently queried areas. This is based on the following key observations:

- (1) Exploratory analyses are often repetitive in nature. Analysts, e.g., may run consecutive queries for the same area to extract different aggregates (i.e., using a different aggregate function, or aggregating over a different attribute).
- (2) Furthermore, analysts might only iteratively change the shape or size of the query polygon. Consequently, part of the polygon’s interior area remains unchanged.
- (3) Lastly, analytical queries often focus on a geographic subset of the whole data. For the analysis of the NYC taxi data, e.g., the focus lies mostly on Manhattan, Brooklyn, and the airport regions, ignoring most suburbs [40].

In all the above cases, it is reasonable to pre-aggregate small grid cells that are often queried together to avoid costly scans of individual cells. In our example in Figure 1, e.g., we want to keep a single aggregate for the purple region, instead of having to consult all 64 contained cell aggregates.

Determining Relevant Aggregates. We want to determine the relevant areas that are worth being additionally pre-aggregated and cached, without making any assumptions about the expected query workload or the semantics of the indexed data. To achieve that, we use all previously seen queries as hints. Precisely, to determine whether an area is worth aggregating, we consider (i) the number of times it was queried, and (ii) its cell level.

For each query cell that intersects with the GeoBlock, we keep track of the number of times it was queried in a trie-like structure. We then use these statistics to calculate cell scores. The score of a cell is the sum of the cell’s hits and the hits of its parent. This score takes into account that child cells can be used to speed up queries for parent cells. We then sort all cells by descending score. When scores are identical, we sort by ascending level (coarser-grained cells come first). As the last criterion, to ensure determinism, we sort by spatial key. We chose the above metric as it is sufficient to properly and repeatably represent the skew in the experiments in our evaluation while being easy to understand and implement. However, we also identified some weaknesses of our metric:

- Smaller cells might overshadow slightly less frequently queried bigger cells. Consider, for example, the green and purple cells of Figure 1 and assume that the green cell is queried just once more than the purple one. Based on our metric, we would then aggregate the green cell even if the purple cell could have an up to 64× bigger impact.
- The parent-child relationship is simplified: Children only cover parts of their parent but are treated as equally useful. Furthermore, we do not consider calculating aggregates by combining the aggregates of the parent and siblings of a cell. For example, the count for a cell could be calculated by subtracting the count of its sibling cells from the count of its parent cell.

Our evaluation showed that these shortcomings have a minor impact, but we plan to investigate them further and address them, if needed, in our future work.

Aggregate Storage. We cache aggregates in a trie-like cache, which we call AggregateTrie. Further, we allow the user to control the maximum size of the storage available for caching, and we store the AggregateTrie in-place with our cell aggregates and the filtered base data. As the cells are strictly ordered, we can simply insert the most relevant unaggregated cell until the reserved area is filled. Figure 7 shows an example AggregateTrie.

The storage for the aggregates is split into two parts. The first part (up until 0x90) contains the trie structure, while the second part stores the actual aggregates. The root of the trie corresponds to the cell level that can enclose our input data, which is typically just a small fraction of the possible earth-wide input space. Each following trie-level encodes exactly one cell level, resulting in a fanout of 4. Since we store the AggregateTrie in-place, we chose a compact encoding storing all nodes contiguously. Nodes consist of just two 32-bit integers. The first one is the pointer to the first child in the AggregateTrie. The second one is the pointer to the corresponding aggregate in the aggregate storage (e.g., 0xb8). Pointers are encoded as 32-bit offsets from the start of the allocated memory region. Both aggregates and nodes can be sufficiently encoded with an offset, as they are of fixed size.

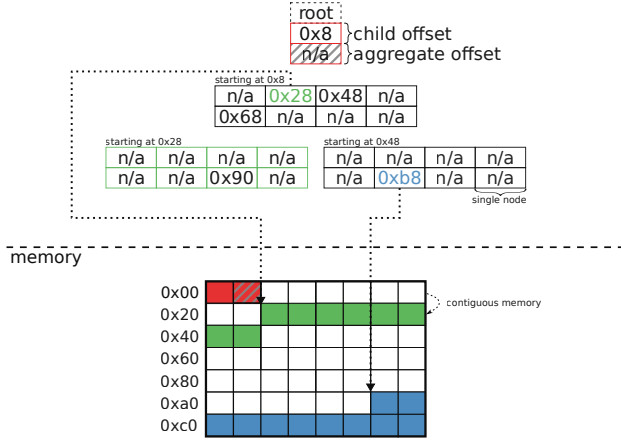


Figure 7: AggregateTrie with 40 byte aggregates and in-memory representation. Non-existent children (or aggregates) are marked with *n/a* and are encoded as *0x0*.

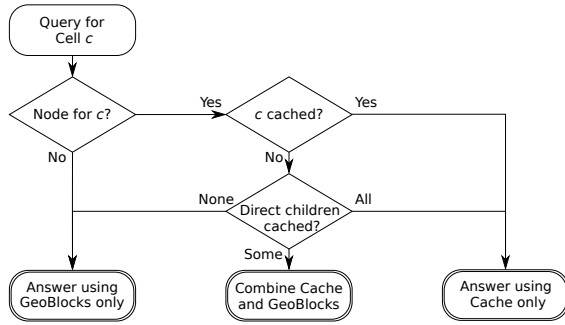


Figure 8: Overview of adapted query algorithm.

Nodes occupy 8 bytes, while the size of the aggregates depends on the schema. Since we store only the offset to the first child, we need to always allocate space for all children in a node, even for children that do not exist in the cache. This can be seen for the node starting at *0x28*, where only one child has an aggregate and no other children or aggregates exist. While this seems wasteful at first, the alternative would be to store four individual child offsets per node. As children are only created and stored if they are needed, our encoding never occupies more storage than this alternative. In fact, our design is more space-efficient in all cases, except for this worst-case in the example above, where only one out of four children is required.

Adapted Query Algorithm. We integrate the cached aggregates into the query algorithm (cf. Section 3.5). As the runtime of COUNT queries is mostly independent of the cell level since only the first and last grid cells are relevant, we do not expect noticeable speedups for them. Therefore, the adapted process, highlighted in Figure 8, is only used for SELECT queries.

Once the pre-query checks are completed, we first probe the query cache and resort to the old algorithm only when necessary. For each query cell, we traverse the AggregateTrie to locate the corresponding node. If there is no node for this cell, we abort probing and answer the query with the old algorithm. Once the node corresponding to the cell is reached, there are two possible ways forward. If the cell is cached, i.e., if it has a valid aggregate offset, the aggregate is extracted as a result. If the cell is not cached, there has to be at least one child at any level

residing in our cache, as nodes are only created on demand. While, theoretically, all children could be used to reduce the number of grid cells of the GeoBlock to query, the number drops with each level, while keeping track of the missing children gets increasingly expensive. Therefore, we only consider direct children for this optimization. If some of the direct children are cached, we combine their aggregates with the results of the old algorithm for the non-aggregated ones to obtain the final result.

4 EXPERIMENTAL EVALUATION

We compare GeoBlocks with on-the-fly aggregation approaches on real-world data. To show that our advantage is not dependent on the indexing strategy, we use different strategies to index the base data of the on-the-fly approaches. We also compare GeoBlocks against a pre-aggregating approach, the aR-tree [30, 31]. However, we do not include the aR-tree in all experiments, as it is designed for rectangular queries and does not directly support polygonal ones.

4.1 Experimental Setup

Baselines. To keep the experiments as fair as possible, we use the mapping from geospatial space to linear space for the baselines as an index key unless specified otherwise. Furthermore, we keep all data in a columnar layout. Below, we describe the three strategies that we use to index the raw data, as well as our pre-aggregating baseline:

BinarySearch: This is the simplest baseline. Instead of indexing the data, we use binary search to locate the first and last contained raw tuple in the data. Afterward, we loop over all tuples in between and compute the requested aggregates. GeoBlocks use binary search to locate the cell aggregate in a similar way.

BTree: We use the BTree as a secondary index over the raw data. For the experiments, we use an open-source B-tree implementation by Google [7]. We probe the tree for the first child and scan the sorted raw data until no further tuple qualifies.²

PHTree: Our last non-aggregating baseline is a multidimensional point index structure, the PH-tree [56]. Instead of the one-dimensional spatial key, we use the latitude and longitude of the points to index the data. As the PH-tree only supports rectangular range queries, we use S2 to get the interior rectangle of the query polygon and use this as a query region. This way, we hope to keep the comparison fair, if not favorable for the PHTree, as this interior rectangle covers fewer points than our approach. As a consequence, the PHTree’s query results differ from the results of the other approaches. For the measurements, we use an open-source C++ implementation [36].

aRTree: We implement the aR-tree [30, 31] based on the boost R-tree [5]. To minimize overlaps between nodes and thereby optimize the query performance, we use the *R** algorithm. In our implementation, each node covers a region *r* and has up to 16 child nodes, which further subdivide *r* into smaller areas. For each node, we store the aggregates in a cell aggregate corresponding to the region covered by the node, and reference it with an offset (cf. Figure 9). That way, we can modify the RTree query logic by adding *early abortion* exactly like in the aR-tree. Given a search area *s* and a node of the aR-tree that covers a region *r*, we distinguish three cases, as shown in Listing 3: (a) If *s* is completely contained by the covered region *r_c* of one of *n*’s

²We first tried the PointIndex of the S2 library (<https://s2geometry.io/devguide/cpp/quickstart.html#s2pointindex>) that uses the same b-tree as point storage. Initial measurements showed that our optimized BTree implementation outperformed the PointIndex by 3×, so we opted for our implementation.

```

1 def queryARTree(node, searchArea, result):
2     partiallyOverlappingNodes = []
3
4     for child in node:
5         if child.contains(searchArea):
6             return queryARTree(child, searchArea, result)
7         if searchArea.contains(child):
8             result += child.aggregatedResult
9         else if searchArea.intersects(child):
10            partiallyOverlappingNodes.append(child)
11
12    for child in partiallyOverlappingNodes:
13        result += queryARTree(child, searchArea, result)
14    return result

```

Listing 3: aR-tree lookup query

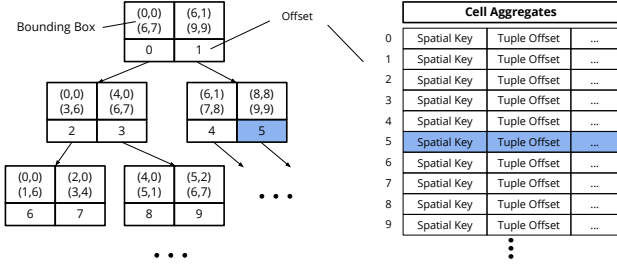


Figure 9: Illustration of aR-tree with node size two and offsets into the cell aggregates.

child nodes, we recursively continue the search at the child node and do not consider other overlapping child nodes as this would result in counting values multiple times. (b) If the region covered by a child node is completely contained within the search area, we add its aggregated value to the overall result and continue processing the next child node. (c) If s and the child node region intersect, we *mark* the child node to be processed later *iff* no other child node fulfills criterion (a).

By accepting that points are counted multiple times in the case of overlapping internal nodes, our aR-tree implementation follows the query algorithm of the original aR-tree that does not consider overlapping children. While the implementation delivers an upper-bound of the result, it visits the internal nodes in the same way the aR-tree does, thus achieving the same performance.

Implementation. We implement GeoBlocks in C++ as described in Section 3. Our implementation, as well as that of all baselines, is single-threaded. Throughout this section, especially in all figures, we will refer to GeoBlocks as Block. Furthermore, we will differentiate between the regular Block and Block^{QC}. Block denotes GeoBlocks without query caching using the basic query algorithm. Block^{QC} is GeoBlocks using query caching with the AggregateTrie and adapted query process outlined in Figure 8.

Hardware. All experiments are run on a server machine with two Intel Xeon E5-2680 v4 processors clocked at 2.4 GHz. The machine is equipped with 256 GiB of DDR4-2400 RAM. All performed experiments fit entirely into main memory.

Dataset. The primary dataset used in the experiments is composed of trip records from 12 million NYC yellow cab rides in the time between January and March 2015, which we cleaned of outliers. It is openly available for download from the NYC Taxi and Limousine Commission (TLC) [49]. It contains data from individual rides like pickup and drop-off location and time, passenger count as well as trip distance.

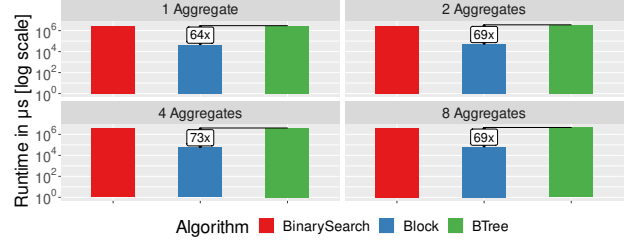


Figure 10: Runtime with increasing number of aggregates.

Unless otherwise specified, the queries consist of polygons representing NYC neighborhoods taken from [25]. As a base workload, we build a query containing each polygon once. For the skewed workload, we select 10% of neighborhoods uniformly at random and query them multiple times. We select 7 aggregates, requesting each column at least once, as query output.

In addition, we use 8 million geotagged tweets from the contiguous US and query them using polygons representing US states. Finally, we use an extract of 389 million OpenStreetMap (OSM) points in the Americas and query them with polygons representing integer countries. Both these datasets have randomly generated integer values as payload. For both, we fix the level at 11 (~7km diagonal). Unless otherwise specified, all experiments are conducted on the primary dataset only.

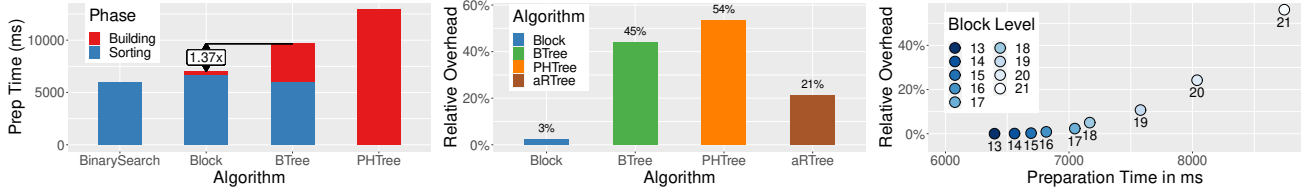
4.2 Baseline Comparison

Impact of Number of Aggregates. To show the impact of the number of aggregates on the performance of the baselines and the Blocks, we use a combined workload consisting of once the base and four times the skewed workload. We query this workload for 1, 2, 4, and 8 aggregates and report the results in Figure 10.

As one can easily see, GeoBlocks outperform both the BTree and BinarySearch baseline in all cases. We omitted the PHTree and aRTree from these experiments, as the imprecise rectangular approximation of the skewed workload lead to a drastic increase in their runtime. Even for the base workload, the PHTree was slower by a factor of about 3× while covering fewer tuples.

Indexing Overhead. We compare the build time, i.e., the preparation time required prior to running any query, in Figure 11a, with the block level set to 17 (~100m diagonal). The reported times for sorting are measured once for the optimized out-of-place sorting for the Blocks and reported for each baseline. This step is completely identical in all sorting baselines. There is a noticeable gap in the sorting phase between the BTree/BinarySearch and the Block. This gap is caused by the collection of grid cell ids to aggregate that we piggybacked on the sorting process to save an additional pass on the data. Overall, the Block is built faster than the BTree and the PHTree, and slightly slower than the BinarySearch, which only needs to sort the input data. We exclude the aRTree baseline from this experiment as we only optimized the implementation for query performance, and the build time was multiple orders of magnitude slower than the others described. Most notably, the majority of the Block preparation is spent on sorting, indicating that once the data is sorted, building additional Blocks with different filter sets is reasonably cheap.

The relative space overhead of each algorithm is depicted in Figure 11b. BinarySearch was omitted as it does not require any additional storage. One could argue that this is not a fair comparison to the BTree and PHTree as they index individual points, but



(a) Build time of GeoBlocks and baselines. (b) Size overhead of GeoBlocks and baselines. (c) Level influence on GeoBlocks overhead.

Figure 11: Index overhead in build time and space.

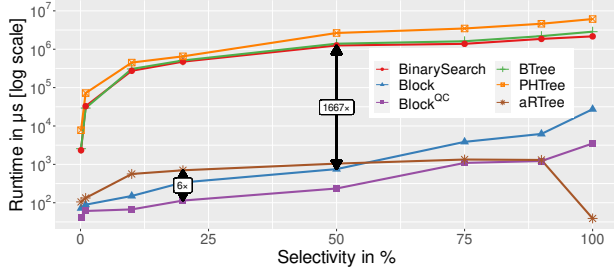
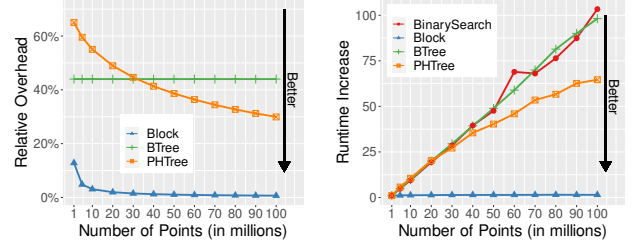


Figure 12: Query runtime for varying selectivity.

as our goal is to provide approximate results, we wanted to show that storing intermediate results is less space-consuming than one would assume for such fine-grained aggregates. While the aRTree is more space-saving when compared to the single-point indices, it still introduces an order of magnitude higher storage overhead than GeoBlocks.

Impact of Selectivity. Selectivity is usually defined based on a single query, but in our context, it is hard to specify what a single query is. We break down query polygons, e.g., the orange bordered Lower East Side in Figure 1, to different-sized cells covering the polygon (e.g., the purple cell), which in turn are broken down into equally sized cells to query (blue cells). While the intermediate cells of the query polygon’s covering are the best representation of individual queries, as each index is probed once for them, they are artificial concepts introduced by our algorithm. Furthermore, these are hard to map to the rectangular query regions of the PHTree and the aRTree. Therefore, we define selectivity based on query polygons. For this experiment, we artificially select polygons covering a part of NYC, which contains a certain percentage of the total rides. Figure 12 reports the runtime of the base workload at different selectivities using a logarithmic scale. PHTree’s and aRTree’s measured selectivities differ slightly from the reported ones due to the less precise covering using an interior rectangle. As this covering contains fewer points, this should slightly skew the experiment in favor of the PHTree and aRTree. Even though GeoBlocks can handle rectangular queries as well, since rectangles are just constrained polygons, we opted for the most-precise covering where possible.

While runtime rises quickly for all baselines for selectivities above 1%, the increase is much softer for both Block variants. Even though the workload is not skewed, and we only use 2% of additional storage for query caching, Block^{QC} still outperforms the non-caching Block across all selectivities. This is likely explained by the shape of the polygons that are often simple quadrilaterals or pentagons. These can be covered using few cells and, therefore, most of these cells can be pre-aggregated. BinarySearch can keep up with the BTree, reporting similar runtimes



(a) Size overhead of GeoBlocks and baselines. (b) Relative runtime increase of GeoBlocks and competitors compared to 1M points.

Figure 13: Scaling with increasing input sizes.

independent of selectivity, while the PHTree lags behind quickly. Even if the relative runtime gap narrows for higher selectivity, the absolute gap still favors GeoBlocks. The aRTree, our implementation of the aR-tree, outperforms the on-the-fly aggregating benchmarks easily while staying behind GeoBlocks for lower selectivities. However, it can catch up with Block at around 50% selectivity. At 100% selectivity, the aRTree needs to only access the root aggregate, explaining the sharp drop in runtime. Overall, GeoBlocks outperform the non-aggregating baselines by at least two and up to three orders of magnitude, performing on-par with the aR-tree while delivering far more precise results.

Scalability. To study the performance for different-sized datasets, we collect 100M taxi rides spanning all of 2015 and build and query the approaches for an increasing subset of these rides. We omit the aRTree as the build time exceeded reasonable limits upward of 30 million points. As the build time is dominated by the sorting process, which is shared and identical in all approaches, they scale identically in build time. When comparing the size overhead in Figure 13a, we can see that the BTree overhead is constant as expected. For the PHTree, we see the positive impact of the integrated compression strategies for bigger datasets. Still, the near fixed-size grid aggregates - the size of a GeoBlock is determined by the spatial distribution of points, not their number - enables even smaller overheads for GeoBlocks. To focus on the individual scalability for queries, we analyze the query runtime normalized to the runtime of each approach for one million points. As shown in Figure 13b, both the BTree and the BinarySearch scale linearly with the input size, as the on-the-fly aggregation dominates the runtime. We expect a similar behavior from the PHTree, but as the covering is less accurate and chosen deliberately smaller, the increase is not fully linear. For GeoBlocks, the runtime stays nearly constant, since it depends on the number of maintained aggregates, and not on the number of individual points. The number of aggregates is in turn determined by the spatial distribution of the input. Since one million

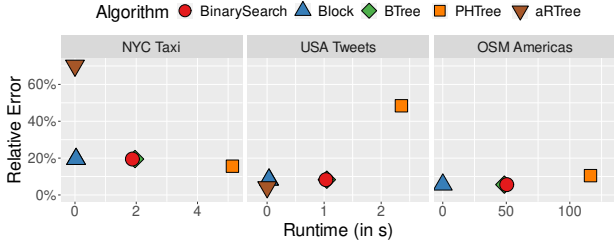


Figure 14: Query runtime and relative error for varying datasets.

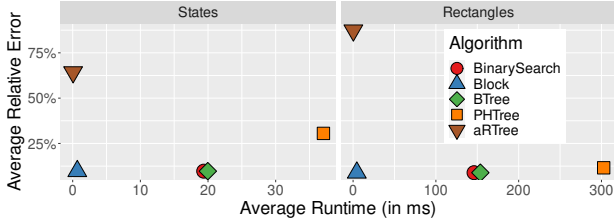


Figure 15: Query runtime and relative error for US states and generated rectangles on the Twitter dataset.

points already cover most areas in NYC, the distribution does not change when further increasing the number of points, i.e., the number of aggregates does not increase significantly. This explains why query latency remains nearly constant for bigger datasets.

Datasets. To show that our approach is not limited to the NYC taxi dataset, we evaluate it on the two additional datasets in Figure 14. We again query the whole area represented by the individual polygons and report runtime, as well as the average error defined as $\frac{|\# \text{ tuples in query result} - \# \text{ tuples in polygon}|}{\# \text{ tuples in polygon}}$. For the OSM dataset, the aRTree again was excluded because of its excessive build time. As the Block, BinarySearch, and BTree use the same covering, the result and error are identical. While the aRTree and PHTree use an identical rectangular representation, the pre-aggregated nodes of the aRTree lead to a different result, and therefore error. Overall, the aRTree and Block are similarly fast with a slight advantage for the aRTree, outperforming the non-aggregating approaches easily. However, the error for Block is far more stable.

Accuracy. Finally, we want to study the influence of smaller individual polygons, as well as rectangular areas, on both runtime and relative error. Therefore, we query all US states and 51 randomly generated rectangles within the US on the Twitter dataset and report the average runtime and error in Figure 15. In contrast to the previous experiment, we query all areas individually. For both polygons and rectangles, the same overall trends are visible. The aRTree is slightly faster than Blocks as the large polygons can be answered in the upper levels of the tree. However, this leads to high imprecision even for rectangular queries as partially overlapping internal nodes might be counted multiple times. Besides, we see that the individual errors canceled out in Figure 14, leading to a seemingly good error bound. While the PHTree error also improves considerably for the rectangular workload, we expected it to be exact. We suspect this is caused by our transformation of the coordinates to integer space, which is necessary for efficient queries. As expected, the performance

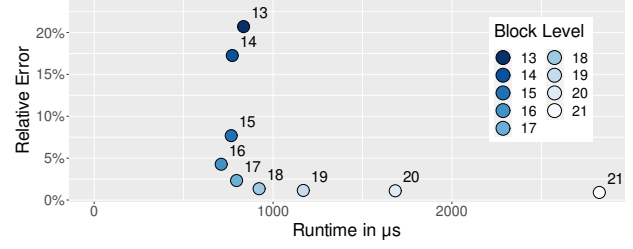


Figure 16: Relative error and runtime at varying levels.

Level	13	14	15	16	17	18	19	20	21
Sorting	6020	6008	6317	6459	6633	6754	7028	7344	7666
Building	376	499	376	356	411	408	538	666	1025

Table 2: Index build times in ms at varying levels.

of Blocks and the other approximating baselines does not degrade for rectangular areas. The aggregating approaches again far outperform the point indexing approaches in runtime.

4.3 Sensitivity Analysis

After showing that GeoBlocks easily outperform all baselines, we study the impact that the configuration of GeoBlocks has on throughput, as well as the impact of data skew on the adaptive Block version. The Block configuration is specified by three parameters: The first setting we study is the level of the Block, i.e., the resolution of the grid overlying the spatial domain. Next, we take a look at the impact of skew on both Block and Block^{QC}. Finally, we examine how the size of the AggregateTrie influences the runtime of unskewed and skewed workloads.

Impact of Block Level. We vary the block levels from 13 to 21 (between ~1.5km and ~6m diagonal) while keeping the other configuration parameters fixed. From a runtime-only point of view, lower-level (coarser-grained) blocks are always preferable, as the query algorithm needs to take fewer cells into account. However, this comes at the price of precision loss. Figure 16 illustrates the connection between the block level, the runtime, and the relative error introduced by the cell covering. The cell covering can introduce only false positive results, i.e., some reported results are not contained in the actual polygon. The figure clearly shows the expected overall trend: the higher the level, the lower the relative error and the higher the runtime. However, after a certain point, decreasing the level further does not pay off. Further, we see that the correlation between error and runtime is not linear, as we already suspected in Section 3.2. The correlation does not even follow the discussed influences completely, which is likely caused by missing sparse children, and the non-uniform distribution of points leading to a gap between the relative error and the configurable spatial error.

The block level influences not only the relative error and the runtime, but also the build time and size of GeoBlocks. Figure 11c depicts the build time and size overhead for GeoBlocks from levels 13 to 21. The build time seems to be only slightly affected by the level, rising slowly with it. Table 2 splits the runtime into two parts: sorting and building. There is a noticeable increase in sort time along with the block level, in addition to the expected increase in build time. This increase in sorting can be explained through our grid cell extraction that we piggybacked to the sorting process, which has to extract more finer-grained

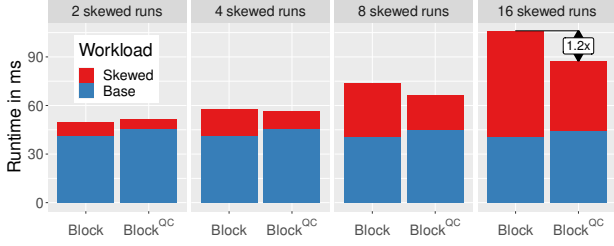


Figure 17: Query runtime with increasing workload skew.

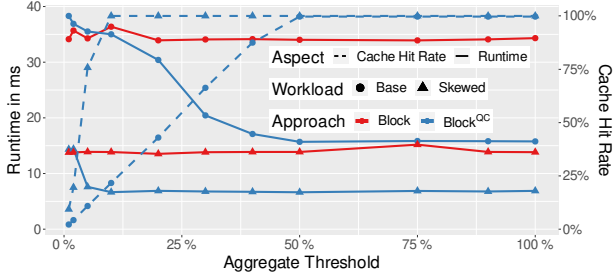


Figure 18: Impact of threshold on workload runtime (solid line) and cache hit rate (dashed line).

cells. The size overhead, however, grows exponentially due to the exponentially growing number of cells along with the level.

Impact of Skew. To study the impact of data skew on the effectiveness of query caching, we measure the query runtime when running the NYC workload once, and the skewed workload multiple times. The number of times we run the skewed workload varies in each experiment. We fix the block level to 17 (~100m diagonal) and the size of the cache to 5% of the cell aggregates, which roughly corresponds to aggregating all cells of the skewed workload. Figure 17 displays the absolute runtime for both the base and the skewed part of the workload. One can see that after four skewed runs, the cached aggregates start to pay off. With even more skew in the total workload, our query-caching Block^{QC} quickly starts to outperform Block. Furthermore, as expected, the runtime for the base workload stays nearly constant, and is always slightly faster for Block. This is easily explained by the overhead of probing the AggregateTrie for each cell, regardless of whether the cell is aggregated or not.

Impact of Aggregate Threshold. Having studied the impact of skew, we want to examine how the aggregate threshold, and thereby the size of the query cache (in Block^{QC}), influences the runtime of the base and the skewed workload. The aggregate threshold denotes the relative size overhead that the query cache, the AggregateTrie, introduces compared to the size of the cell aggregates in the regular GeoBlock. We again fix the block level to 17, and the number of skewed runs to four. Figure 18 depicts the measured runtimes and cache hit rates. The runtime of Block is unaffected by the changed threshold and only acts as a baseline to highlight the influence on Block^{QC} . Up until a threshold of around 5%, only queries from the skewed workload can be answered using the AggregateTrie. The small speedup in the base workload can be explained by the inclusion of the skewed workload in the base workload. Once all cells in the skewed workload are cached, and the cache hit rate for the skewed part reaches 100%, other query cells of the base workload start to get cached

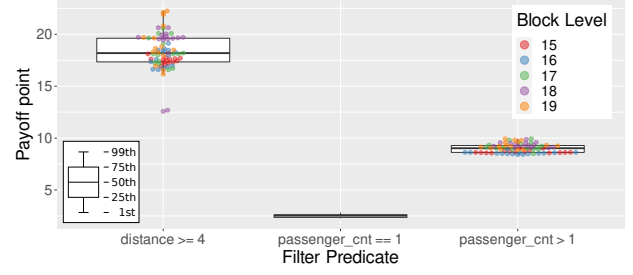


Figure 19: Payoff point: Number of incremental builds required to amortize the cost of sorting the raw data.

as well. While this, of course, leads to further runtime improvements, it is undesirable, especially when memory is scarce. In our experiments, at around 50%, the cache hit rate reaches 100% for both workloads, and there is no further speedup, even when the cache size is doubled. The cache hit rate, illustrated by the dashed line and shown on the right axis, shows the desired effect. The skewed part is cached almost immediately, and the hit rate for the unskewed workload grows linear with increasing cache size. The average lookup time slowly grows from 58ns at 1% to 81ns at 100%. As the lookup time depends on the number of levels (30 in the maximum) and not on the size, this growth is attributable to more complex access patterns for larger cache trees.

4.4 Changing Filters

Finally, we compare our process of Figure 5, wherein we build multiple GeoBlocks from the sorted base data, against building isolated GeoBlocks from scratch. We vary the block level from 15 to 19 (between ~420m and ~27m diagonal), and build 15 GeoBlocks per level using three different predicates of varying selectivity:

- **distance >= 4:** Long taxi trips, selectivity of ~16%
- **passenger_cnt == 1:** Solo taxi trips, selectivity of ~70%
- **passenger_cnt > 1:** Shared taxi trips, selectivity of ~30%

For this, we want to analyze how many different filter and level combinations are required to amortize the initial cost of sorting. Figure 19 shows the payoff point of filter changes for our three filter predicates. The payoff point is the number of incremental builds required to be, in sum, faster by creating incremental builds than building individual GeoBlocks from the raw data and filtering before sorting. We omitted the individual runtimes for the `passenger_cnt == 1` predicate as they would be too densely packed vertically.

As expected, the more selective the filter, the lower the speedup. Once all tuples in the raw data have been filtered according to the predicate, the qualifying tuples have to be sorted. More selective predicates take longer to amortize as sorting few tuples is cheap, whereas for the 70% selectivity query `passenger_cnt == 1`, the more expensive sorting is amortized almost immediately. There is a correlation between the block level and amortization, most notably for the most selective predicate `distance >= 4`. Given that the payoff point drastically rises with lower selectivity, we expect that incremental builds will only pay off when the new filters are less selective. If only a few highly selective queries are expected, building regular GeoBlocks directly from the raw data will still be the fastest option. However, the time to switch to a new filter, and therefore the individual query latency, will always be lower for incremental builds.

5 DISCUSSION

In this section, we discuss the takeaways of the evaluation as well as updates for GeoBlocks.

Evaluation Summary. First, we showed that pre-aggregation in a spatial context pays off when a limited and bounded spatial error is acceptable, independently of the number of aggregates queried and the selectivity of the query polygons. Furthermore, GeoBlocks can be built fast, introducing only a small overhead compared to the simple BinarySearch baseline. Even when the data is already indexed with one of our baselines (i.e., without taking the index build time into account), GeoBlock’s build time of around 7 seconds can be amortized by fewer than 30 polygon queries with a selectivity of 10% (cf. Figures 11a and 12). In addition, building multiple GeoBlocks once the data is sorted is possible within one second for our dataset, cf. Figure 11a. Building new GeoBlocks for different filters is even faster when using sorted base data, often amortizing the initial extra cost of sorting all data in less than 10 filter changes (cf. Figure 19). Even though not all configurations are optimal for GeoBlocks, there are acceptable error-runtime trade-offs, in our case around levels 17 and 18. While the level does not significantly impact the index build time, the size overhead growth is almost exponential, cf. Figure 11c, indicating that it is wise to think about which error is acceptable for the given query workload when memory is scarce.

Updates. Up until now, we considered GeoBlocks to be read-only as they are designed for historical point data. However, the layout of GeoBlocks allows us to integrate updates easily, as long as a cell aggregate for the region of the newly arriving tuple already exists. For the non-adaptive version, all we have to do is locate the cell aggregate containing the tuple and update all stored aggregates. In the adaptive version, we additionally need to update all cached parents of the grid cell in the AggregateTrie as well. Thanks to the prefix-based indexing property of the trie, we can do this in a single depth-first traversal. Only if tuples arrive for a new, previously unaggregated region, we have to rebuild the aggregate layout, as we rely on the cell aggregates to be sorted. However, as we have shown in the evaluation, recalculating the cell aggregates is often possible within a second, so this operation would not induce too much delay when updates are implemented in batches instead of single tuples. Other indexing approaches on the cell aggregates (e.g., a clustered B-tree) could eliminate the need to rebuild by reserving storage for new aggregates. Preliminary experiments using `std::map` and a B-tree as an index showed similar lookup performance at the cost of increased size overhead.

6 RELATED WORK

Our approach builds on seminal work from decades of research on spatial indexing. Decomposing space into hierarchical grid cells [1, 9, 39], as well as approximating polygons using simpler shapes [18], are all well-known approaches. Likewise, enumerating cells using a space-filling curve such as Hilbert or Z order [26, 27] and storing aggregate information within cells [20, 30, 46] are ideas that have been around for some time. However, while building on these established concepts, GeoBlocks present the first pre-aggregating data structure that supports a *bounded, distance-based error* on the results of *polygonal queries*. Specifically, prior work on pre-aggregation [14, 30, 31, 34] is limited to rectangular queries and requires an expensive post-processing (refinement) step to answer polygonal queries. GeoBlocks, on the other hand, yield error-bounded results and do not require expensive refinement.

Spatial Aggregation. Past work has proposed several approaches for spatial aggregation queries [23]. These approaches mainly rely on pre-aggregation [14, 34]: they pre-aggregate records at various spatial resolutions and store this summarized information in a hierarchy of rectangular regions, maintained using a spatial index like the quadtree or the R-tree [19, 30–32]. For instance, the aRtree [30, 31] enhances the R-tree by storing aggregate information for each node. This allows to directly extract the aggregate of all the records contained in a node, if the node’s MBR is fully enclosed in the query region. Being a variant of the R-tree, the aRtree constrains the supported queries to only rectangular regions. Furthermore, the computed aggregates are approximate and the error *cannot be bounded*, since the accuracy depends on the resolution of the rectangular R-tree nodes. Providing precision guarantees for arbitrary polygons requires accessing the raw data and involves additional processing. There are also approaches that store aggregates inside a data cube [6, 37], or using sketches [48]. Nanocubes [21], for example, store the CUBE operator for spatio-temporal datasets, and are specifically designed for visualization systems. The data cube-based approaches suffer from the same limitations as the aRtree, since they also rely on a hierarchy of rectangular regions. Besides, accessing the raw data to refine the aggregates might require additional indices, as the cube does not store individual records. Vorona et al. [55] approximate the distribution of geospatial points with an autoregressive deep learning model to answer arbitrary polygonal queries, but they cannot provide any error bounds. Pandey et al. [29] propose to use learned indices for query-efficient spatial indexing, albeit limited to range queries. Finally, Raster Join [51] uses GPU rendering to compute aggregates over a point-polygon join. In contrast, GeoBlocks support aggregation over spatial selections.

Prefix sums [10] can be used in addition to pre-aggregation to enable fast range-sums. This is achieved by only inspecting the aggregates in the two corners of a query region, rather than every aggregate inside the query region. An example of this is our COUNT algorithm. However, in contrast to our SELECT queries, these range-sums are unable to extract min and max aggregates. **Materialized Views and OLAP Cubes.** GeoBlocks are essentially materialized views over geospatial data with support for filters and aggregations. In contrast to regular views [12, 44], GeoBlocks are designed for historical spatial data and can adapt to the query workload at a fine-grained level using a trie-like cache. Work on materialized view selection [2] also makes materialization decisions based on the query workload, but at a much coarser granularity (e.g., what columns to aggregate). There has also been a lot of work on data cubes and query caching [11, 15, 42], but these do not support geospatial data as a first-class citizen.

Spatial Point Indexing. Spatial point indexing approaches typically index points using a hierarchy of MBRs, most notably the R-tree [13], or by subdividing grid cells into equally-sized children, e.g., the quadtree [9, 39]. Both of these index structures are queried using the dimension-wise min/max values, i.e., the query regions are rectangular. Other approaches, like the UB-tree [3], assign univariate keys to the indexed regions first and rely on these keys for data access. While the UB-tree does not specify how these keys have to be generated, most approaches use space-filling curves like the Z order [26, 27].

Based on these concepts, more specialized indices have been developed. The PH-tree [56] combines a quadtree with hypercubes to allow splitting all dimensions in each node, providing a space-efficient index structure for multidimensional data. The space efficiency can be partly attributed to the utilization of prefix

sharing, similar to the one used in our trie-like cache. Alternating the indexed dimensions in an in-memory tree structure, the BB-tree [47] offers fast point and range queries for multidimensional data. While these structures require the index to be built a priori, there are others like QUASII [33], where the index is built incrementally as a side product of query execution. As a result, QUASII can adapt to the query workload at runtime. However, QUASII only supports spatial range (window) queries. Recently, Shin et al. [43] proposed integrating grid indices into a tree structure to achieve faster node accesses and point operations.

7 CONCLUSIONS

We have introduced GeoBlocks, a novel pre-aggregating data structure for geospatial data. GeoBlocks pre-compute aggregates over fine-grained grid cells, thereby supporting arbitrarily shaped polygons. Using these aggregates, GeoBlocks can provide fast query results with a user-controlled spatial error. Furthermore, GeoBlocks can speed up aggregate queries for commonly queried regions by dynamically adapting to any given workload using limited additional storage.

Comparing GeoBlocks with on-the-fly aggregating indexing baselines, we have shown that we can outperform them for any number of aggregates, in parts by three orders of magnitude. The introduced storage overhead is comparable, and often even lower, to that of traditional indexing structures, while GeoBlocks can be built equally fast. Looking at GeoBlocks' configuration options, we have shown how they can be adapted to the given dataset and workload, and how they influence the runtime, the overhead, and the error in the result. Overall, GeoBlocks are materialized views over geospatial data that support filter predicates and aggregates while enabling fine-grained adaptation to the query workload.

REFERENCES

- [1] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *PODS*, pages 265–272. ACM Press, 1990.
- [2] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, pages 156–165, 1997.
- [3] R. Bayer. The universal b-tree for multidimensional indexing: general concepts. In *WWCA*, pages 198–209. Springer, 1997.
- [4] *How to analyse bike data for urban planning?* <https://www.bikecitizens.net/analyse-bike-data-urban-planning/>.
- [5] *Boost R-tree*. https://www.boost.org/doc/libs/1_69_0/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost_geometry_index_rtree.html.
- [6] F. Braz, S. Orlando, R. Orsini, A. Raffaetà, A. Roncato, and C. Silvestri. Approximate aggregations in trajectory data warehouses. In *ICDE Workshops*, pages 536–545, 2007.
- [7] Google code archive. <https://code.google.com/archive/p/cpp-btree/>.
- [8] N. Ferreira, M. Lage, H. Doraiswamy, H. Vo, L. Wilson, H. Werner, M. Park, and C. Silva. Urbane: A 3d framework to support data driven decision making in urban development. In *Proc. IEEE VAST*, pages 97–104, 2015.
- [9] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.
- [10] S. Geffner, D. Agrawal, A. E. Abbadi, and T. R. Smith. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *ICDE*, pages 328–335, 1999.
- [11] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.
- [12] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112. Springer, 1997.
- [13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM Press, 1984.
- [14] J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. In *PAKDD*, pages 144–158. Springer, 1998.
- [15] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216. ACM Press, 1996.
- [16] A. Kipf, H. Lang, V. Pandey, R. A. Persa, C. Anneser, E. Tzirita Zacharotou, H. Doraiswamy, P. A. Boncz, T. Neumann, and A. Kemper. Adaptive main-memory indexing for high-performance point-polygon joins. In *EDBT*, pages 347–358, 2020.
- [17] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. A. Boncz, T. Neumann, and A. Kemper. Approximate geospatial joins with precision guarantees. In *ICDE*, pages 1360–1363. IEEE Computer Society, 2018.
- [18] H. Kriegel, H. Horn, and M. Schiewietz. The performance of object decomposition techniques for spatial query processing. In *SSD*, volume 525, pages 257–276. Springer, 1991.
- [19] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412. ACM, 2001.
- [20] I. Lazaridis and S. Mehrotra. Multi-resolution aggregate tree. In *Encyclopedia of GIS*, pages 764–765. Springer, 2008.
- [21] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2456–2465, 2013.
- [22] Z. Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *Proc. TVCG*, 20(12):2122–2131, 2014.
- [23] I. F. V. López, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: a survey. *IEEE Trans. Knowl. Data Eng.*, 17(2):271–286, 2005.
- [24] F. Nagel, P. A. Boncz, and S. Vlas. Recycling in pipelined query evaluation. In *ICDE*, pages 338–349, 2013.
- [25] NYC neighborhoods. <https://data.cityofnewyork.us/City-Government/Neighborhood-Tabulation-Areas/cpf4-rkhq>.
- [26] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *SIGMOD Conference*, pages 326–336. ACM Press, 1986.
- [27] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS*, pages 181–190. ACM, 1984.
- [28] V. Pandey, A. Kipf, T. Neumann, and A. Kemper. How good are modern spatial analytics systems? *PVLDB*, 11(11):1661–1673, 2018.
- [29] V. Pandey, A. van Renen, A. Kipf, J. Ding, I. Sabek, and A. Kemper. The case for learned spatial indexes. In *AIDB@VLDB*, 2020.
- [30] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *SSTD*, pages 443–459. Springer, 2001.
- [31] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *ICDE*, pages 166–175, 2002.
- [32] D. Papadias, Y. Tao, J. Zhang, N. Mamoulis, Q. Shen, and J. Sun. Indexing and retrieval of historical aggregate information about moving objects. *IEEE Data Eng. Bull.*, 25(2):10–17, 2002.
- [33] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki. QUASII: query-aware spatial incremental index. In *EDBT*, pages 325–336, 2018.
- [34] T. B. Pedersen and N. Tryfona. Pre-aggregation in spatial data warehouses. In *SSTD*, pages 460–480. Springer, 2001.
- [35] T. Phan and W. Li. Dynamic materialization of query views for data warehouse workloads. In *ICDE*, pages 436–445, 2008.
- [36] *mcxme/phree*. <https://github.com/mcxme/phree>.
- [37] F. Rao, L. Zhang, X. Yu, Y. Li, and Y. Chen. Spatial hierarchy and olap-favored search in spatial data warehouse. In *DOLAP*, pages 48–55. ACM, 2003.
- [38] S2 geometry. <https://s2geometry.io/>.
- [39] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.
- [40] T. Schneider. *Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance*. <https://toddschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/>.
- [41] V. Shah. *Citi Bike 2017 Analysis - Towards Data Science*. <https://towardsdatascience.com/citi-bike-2017-analysis-efd298e6c22c>.
- [42] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic caching of query results for decision support systems. In *SSDBM*, pages 254–263, 1999.
- [43] J. Shin, A. R. Mahmood, and W. G. Aref. An investigation of grid-enabled tree indexes for spatial query processing. In *SIGSPATIAL*, pages 169–178, 2019.
- [44] O. Shmueli and A. Itai. Maintenance of views. In *SIGMOD*, pages 240–255. ACM Press, 1984.
- [45] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, pages 488–499, 1998.
- [46] S. Singla, A. Eldawy, R. Alghamdi, and M. F. Mokbel. Raptor: Large scale analysis of big raster and vector data. *PVLDB*, 12(12):1950–1953, 2019.
- [47] S. Sprenger, P. Schäfer, and U. Leser. Bb-tree: A main-memory index structure for multidimensional range queries. In *ICDE*, pages 1566–1569, 2019.
- [48] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–225, 2004.
- [49] NYC tlc data. <https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page>.
- [50] *TNCs TODAY*. <http://tncstoday.sfcta.org/>.
- [51] E. Tzirita Zacharotou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freire. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *PVLDB*, 11(3):352–365, 2017.
- [52] E. Tzirita Zacharotou, A. Kipf, I. Sabek, V. Pandey, H. Doraiswamy, and V. Markl. The case for distance-bounded spatial approximations. In *CIDR*. <http://cidrdb.org>, 2021.
- [53] *Uber Movement*. <https://movement.uber.com/>.
- [54] F. van Diggelen and P. Enge. The world's first GPS MOOC and worldwide laboratory using smartphones. In *Proc. ION GNSS+*, pages 361–369, 2015.
- [55] D. Vorona, A. Kipf, T. Neumann, and A. Kemper. DeepSPACE: Approximate geospatial query processing with deep learning. In *SIGSPATIAL/GIS*, pages 500–503. ACM, 2019.
- [56] T. Zäschke, C. Zimmerli, and M. C. Norrie. The ph-tree: a space-efficient storage structure and multi-dimensional index. In *SIGMOD*, pages 397–408. ACM, 2014.