A GPU-Based Index to Support Interactive Spatio-Temporal Queries over Historical Data

Harish Doraiswamy[*], Huy T. Vo[†], Cláudio T. Silva[*], Juliana Freire[*]

[*] New York University

[†] The City College of the City University of New York

{harishd, huy.vo, csilva, juliana.freire}@nyu.edu

***Abstract*—There are increasing volumes of spatio-temporal data from various sources such as sensors, social networks and urban environments. Analysis of such data requires flexible exploration and visualizations, but queries that span multiple geographical regions over multiple time slices are expensive to compute, making it challenging to attain interactive speeds for large data sets. In this paper, we propose a new indexing scheme that makes use of modern GPUs to efficiently support spatio-temporal queries over point data. The index covers multiple dimensions, thus allowing simultaneous filtering of spatial and temporal attributes. It uses a block-based storage structure to speed up OLAP-type queries over historical data, and supports query processing over in-memory and disk-resident data. We present different query execution algorithms that we designed to allow the index to be used in different hardware configurations, including CPU-only, GPU-only, and a combination of CPU and GPU. To demonstrate the effectiveness of our techniques, we implemented them on top of MongoDB and performed an experimental evaluation using two real-world data sets: New York City's (NYC) taxi data – consisting of over 868 million taxi trips spanning a period of five years, and Twitter posts – over 1.1 billion tweets collected over a period of 14 months. Our results show that our GPU-based index obtains interactive, sub-second response times for queries over large data sets and leads to at least two orders of magnitude speedup over spatial indexes implemented in existing open-source and commercial database systems.**

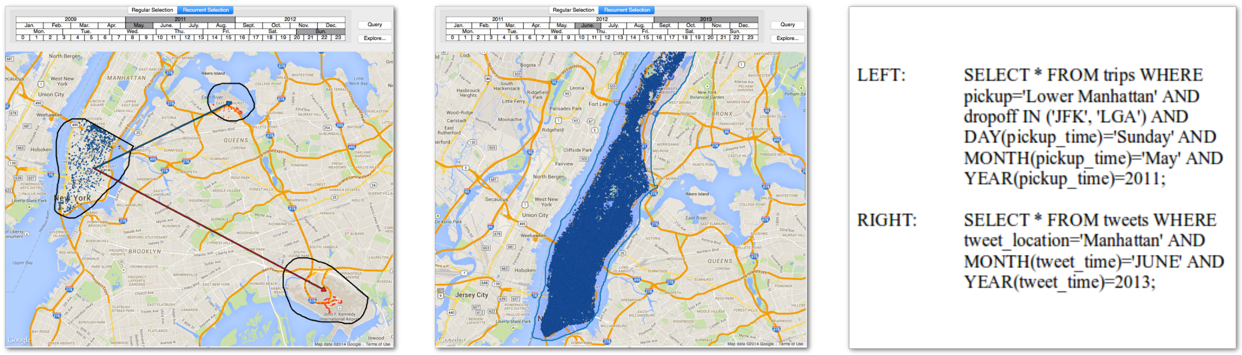
I. Introduction

The availability of low cost sensors such as GPS in vehicles and mobile devices has led to an explosion in the volume of spatio-temporal data. Location and time information is being captured by a plethora of mobile applications such as Twitter and Instagram. Governments and cities all over the world are collecting and making available increasing volumes of data that contain a spatio-temporal component, e.g., 311 complaints, crime, transportation (subway, bus, taxi trips) and land-use data [35]. In fact, a recent study that examined over 9,000 open urban data sets found that over 50% of them contain spatial information [5]. While these data open new opportunities to advance science, to inform policy and administration, and ultimately, to improve people's lives, making sense of them is very challenging. Visualization and visual analytics systems have been successfully used to aid users obtain insights: well-designed visualizations substitute perception for cognition, freeing up limited cognitive/memory resources for higher-level problems [33]. Such systems are therefore increasingly being used to support data exploration. They allow users to formulate queries and visually explore their results. But to be effective, visualization systems must be interactive, requiring sub-second response times [12], [30]. Liu and Heer [30] have shown that even a 500ms difference can significantly impact visual analysis, reducing interaction and data set coverage during analysis, as well as the rate in which users make observations, draw generalizations and generate hypotheses.

Having been designed for batch queries issued through a text-based or terminal interfaces, existing relational database technologies and business intelligence systems used for OLAP analyses are not suitable for interactive tools [51]. Not surprisingly, the problem of providing efficient support for visualization [46] and interactive queries over large data has attracted substantial attention recently (see e.g., [1], [6], [25], [26], [29], [31], [45]). While previous approaches have targeted relational data, in this paper, we address specific challenges that arise with spatio-temporal point set data.

Motivating Example: Taxis and Tweets as Sensors in NYC. The New York City Taxi and Limousine Commission (TLC) collects information about taxi trips in the city. Each trip consists of pickup and dropoff locations and times, along with other relevant data such as the fare and tip. There are, on average, 500 thousand trips each day, totaling over 868 million trips in the past five years [43]. The availability of this data set has generated enormous interest among traffic engineers and social scientists, who are interested in studying various traffic and economic trends (see e.g., [10], [40]). Queries of interest typically span geographical regions over multiple time slices, and include constraints on more than one spatial attribute in addition to temporal constraints. For example, given the flat fare for taxis to get to (from) the NYC airports from (to) Manhattan, economists are interested in understanding taxi trips originating and ending at airports, where the passengers go, how long the trips take, and how these indicators behave at different times of the day and on different days of the week. If the trips take too long, it may be detrimental to drivers to serve the airports, and either the existing flat fare policy should be reconsidered, or the fare should be increased. An example of such a query is shown in Figure 1(a). It asks for trips that occurred between lower Manhattan and the two airports, JFK and LGA, on all Sundays of May 2011.

Visual analytics tools are being developed to help domain experts analyze these data and perform exploratory queries. Users interact with the data through visualizations and their actions are translated into queries issued to a database system or specialized storage manager. Figure 1 illustrates two queries that are visually specified using TaxiVis [13]. As shown in Figure 1(a), a user can draw polygonal regions of interest, place constraints on both pickup and dropoff locations, and select the time period(s) of interest. Figure 1(b) shows a query that selects all tweets posted in Manhattan in June 2013. Users can refine the queries, selecting additional regions, stepping



```
LEFT:     SELECT * FROM trips WHERE
          pickup='Lower Manhattan' AND
          dropoff IN ('JFK', 'LGA') AND
          DAY(pickup_time)='Sunday' AND
          MONTH(pickup_time)='May' AND
          YEAR(pickup_time)=2011;

RIGHT:    SELECT * FROM tweets WHERE
          tweet_location='Manhattan' AND
          MONTH(tweet_time)='JUNE' AND
          YEAR(tweet_time)=2013;
```

(a) Taxi trips (b) Tweets (c) Queries in SQL-like syntax

Fig. 1. Typical spatio-temporal queries visually constructed with the TaxiVis tool [13] to explore (a) NYC taxi and (b) Twitter data. Spatial constraints are specified by drawing arbitrary polygons. For taxi trips, arrows between the polygons are used to specify the pickup and dropoff regions. In (c), we show the queries in a SQL-like syntax: the query on the left selects trips that occurred between lower Manhattan and the two airports on all Sundays in May 2011, while the one on the right selects all tweets posted from Manhattan during June 2013.

through time, or performing parameter sweeps to compare multiple data slices and study how patterns change over space and time. These operations, which are commonplace in visual analysis tools, lead to a high query rate. Coupled with the stringent response times required by these tools, this makes interactive execution for these complex queries challenging, especially for large data sets.

Interactive Spatio-Temporal Queries: Challenges. Spatial constraints can be specified as arbitrary polygons, and thus, they require multiple (often millions of) point-in-polygon (PIP) tests. Because the time complexity for each PIP test is linear in the size of the polygon, these queries are very expensive to compute. Consider the query in Figure 1(a). Assuming that an *optimal* number of point-in-polygon tests are performed, i.e., they are done only on records that lie within the query polygon corresponding to lower Manhattan, over 6.5 million such tests have to be performed even though the query returns only around 13,000 records. Thus, to attain interactivity, it is crucial to reduce the number of PIP tests. For queries that refer to a single spatial attribute, a spatial index helps reduce the number of points to be tested. However, this is not the case for complex queries that contain additional constraints on time or multiple spatial attributes. For example, a selection from a spatial index for trips in Lower Manhattan contains records corresponding to all time steps (not just Sundays in May 2011), and for trips with destinations other than the airports. Performing the selection using an index over time is also inefficient: tests need to be applied to all records returned, since there is no spatial index on these records. Another alternative would be to use multiple indices, one for each attribute. But besides the unnecessary PIP tests from the index selections (as discussed above), this also incurs an additional overhead: results coming from different indices must be joined.

Contributions. To address these challenges, we propose STIG (Spatio-Temporal Indexing using GPUs), an indexing scheme that supports complex spatio-temporal queries over large, historical data at interactive rates. Interactivity is attained through a two-pronged strategy: (1) *Simultaneous filtering over multiple dimensions*–by using a single index that covers multiple spatial attributes as well as other attributes such as time, data can be simultaneously filtered over multiple dimensions, reducing the number of costly PIP tests to be performed; (2) *Fast spatial operations using GPUs*–the numerous PIP tests are independent of each other. GPUs are used to substantially speedup this *embarrassingly parallel* process.

STIG is a generalization of the kd-tree [7]. While kd-trees have been widely used for in-memory processing, we have extended them to support out-of-core query processing and to leverage GPUs. STIG trees (stg-trees) consist of two components: the tree-nodes that correspond to the kd-tree, and a set of blocks that store the records. The block-based storage has many benefits, notably: it enables out-of-core execution of queries, and it leads to a small memory footprint, making it possible to store the index in memory. As we discuss in Section II, it also reduces the data transfer overheads between the CPU and GPU, which is critical for efficient query execution. We propose multiple query execution strategies that can be used to support different system configurations, including: only GPUs, only CPUs, and a hybrid combination of CPUs and GPUs. The strategies can also leverage multiple GPUs and the newly introduced dynamic-parallelism feature present on newer NVIDIA graphics processors.

We evaluate the efficiency of STIG using a prototype implementation in MongoDB [32][1] and queries over two large data sets: NYC taxi trips and Twitter data. The NYC taxi data consists of over 868 million records having two spatial and two temporal attributes. The Twitter data has over 1.1 billion records, each containing one spatial and one temporal attribute. Our results show that STIG running on GPUs leads to substantial performance gains: it is at least 30 times faster compared to a similar index implemented using just the CPU. In addition, the index is scalable. Not only does it performance increase with the number of GPUs, but also, query execution times increase (almost) linearly with the result sizes. For end-to-end query execution times, our prototype was at least two orders of magnitude faster than existing open-source and commercial database systems.

II. GPU-based Spatio-Temporal Index

In this work, we are interested in spatial queries that involve point-in-polygon tests. These queries are expensive, especially for polygons that have arbitrary structure. A single query may require millions of such tests. Having multiple spatial attributes, in addition to temporal constraints, further increases the cost of executing these queries. As mentioned earlier, visual

[1]Code available at: *https://github.com/harishd10/mongodb.git*

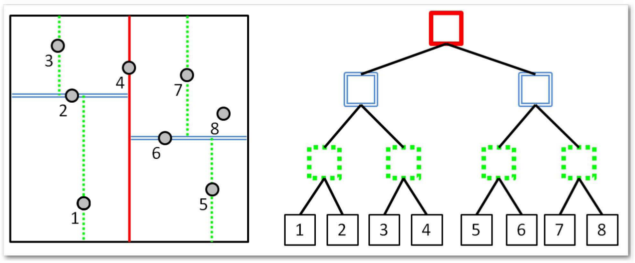


Fig. 2. Kd-tree constructed for a set of 8 points in $\mathbb{R}^2$. We use color to connect nodes in the tree to the corresponding lines that split the plane.

analysis tools typically generate multiple such queries through user interaction. Thus, the operation of resolving whether a data point satisfies the given polygonal constraints creates a bottleneck, hampering the interactivity of these tools.

Leveraging GPUs to speedup spatio-temporal queries. Current generation GPUs consist of thousands of parallel processors. Leveraging the parallelism provided by the GPU to execute a spatio-temporal query can help overcome the above-mentioned bottleneck. However, doing so is challenging. While a brute force parallel search over the entire data can be done, this is inefficient especially for large data. In order to efficiently leverage the GPUs, it is important that the following three properties are satisfied by the indexing scheme:

P1. Minimize data transfer: the volume of data transferred to the GPU must be minimized. GPU memory is limited, and since the memory transfer overhead between the CPU and GPU is significant, it can adversely impact the query execution time. This problem is compounded for large data sizes which require multiple data transfers during query execution;
P2. Maximize occupancy: to best utilize the GPU, the occupancy of the GPU should be maximized, i.e. the number of idle cores of the GPU should be minimized; and
P3. Minimize kernel synchronization: the number of GPU kernel synchronizations should be minimized. GPU code is typically executed as multiple kernels, and the execution of the different kernels is synchronized on the CPU. This synchronization between the kernels is a costly operation, and having multiple such synchronizations can negatively impact the query execution time.

In what follows, we describe the index structure that we designed, keeping in mind the above properties, to support spatio-temporal queries (Section II-A). The index is a generalization of a kd-tree, and through the use of a block-based structure, it can efficiently leverage the GPUs. A possible alternative was to use a R-tree-based data structure for the index. However, we chose the kd-tree for our index to avoid the overhead of unwanted PIP tests induced by the overlapping regions among the child nodes of a R-tree, which grows rapidly with the increase in the number of dimensions (see Section V-D). The index construction process is described in Section II-B, and its space requirements are analyzed in Section II-C.

A. STIG

In this section, we first provide a brief overview of kd-trees, and then describe in detail the structure of the index. For ease of exposition, we assume that spatial attributes of the input corresponds to 2-dimensional points. However, our index can be easily extended to points in three or higher dimensions.

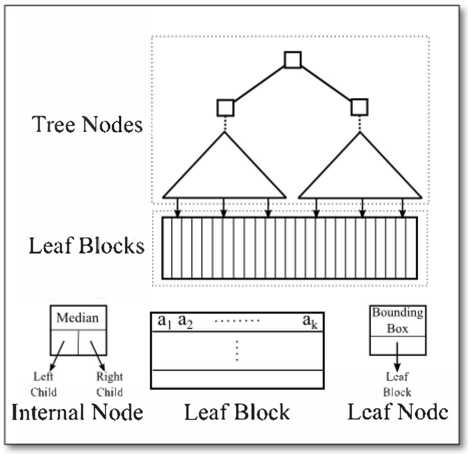


Fig. 3. Structure of the block-based STIG index. The index consists of two parts. The tree nodes store the kd-tree. Each leaf node points to a block containing records that satisfy the constraint given by the path from the root to that leaf. Along with this pointer, each leaf node also stores the k-dimensional bounding box that contains all the records in the corresponding leaf block.

Kd-trees. A kd-tree [7] is a binary tree used to store k-dimensional points. Each non-leaf node in this tree splits the points in its sub-tree along a hyper-plane. All points present in the lower half-space defined by this hyper-plane are present in the left sub-tree, while those in the upper half-space are in the right sub-tree. Let $\{x_1, x_2, \ldots, x_k\}$ denote the k axes of the Euclidean space $\mathbb{R}^k$. In its simplest form, a non-leaf node at depth d splits the points in the sub-tree along the hyper-plane $x_i = c$, where $i = d \bmod k$, and c is the median value of the i^{th} coordinate of the points it splits. Figure 2 illustrates this for a 2-dimensional scenario.

Stg-trees. As illustrated in Figure 3, an stg-tree consists of two parts: the *tree nodes* and a set of *leaf blocks*. The tree nodes correspond to a kd-tree. Given s spatial attributes and m other attributes on which the index is to be built, we create a $k = 2 \times s + m$ dimensional kd-tree over these attributes. Note that the set of m attributes can include one or more temporal attributes. An internal node at depth d stores the median value of the $(d \bmod k)^{th}$ coordinate of the points covered by that node together with pointers to its two children. A leaf node in an stg-tree points to a leaf block which stores data corresponding to a collection of records. This data consists of values of the attributes on which the index is being created along with a pointer to the location of the actual record. Each leaf block satisfies all the constraints specified by the path from the root to the corresponding leaf. Note that this differs from a traditional kd-tree, where each leaf node corresponds to a single record. The size of a leaf block is a parameter specified while creating the index. In addition to storing a pointer to a leaf block, a leaf node also stores the k-dimensional box that bounds all the records in that block.

The two-part structure of stg-trees serves three purposes. Intuitively, the structure clusters points along k-dimensional axis parallel hyper cubes, thus allowing a k-dimensional range search to be restricted to nearby nodes. As we discuss in Section III, candidate leaf blocks are sent to the GPU where they are searched in parallel. Since queries of interest involve spatial ranges in the form of polygons, having nearby points clustered together minimizes the amount of data to be transferred to the GPU (Property P1), and reduces the the number of PIP tests that is performed. Furthermore, because the spatial

Procedure CreateIndex

```
Require: Node parent, Integer depth, Integer startIndex,
    Integer endIndex
 1: if $endIndex - startIndex \leq blockSize$ then
 2:     Create data block $B$ for records from $startIndex$ to $endIndex$
 3:     $parent.leaf = B$
 4:     $parent.bound$ = $k$-dimensional bounding box of the records in $B$
 5:     return
 6: end if
 7: $d = depth(\bmod k)$
 8: sort records from $startIndex$ to $endIndex$ w.r.t. attribute corresponding
    to dimension $d$
 9: $median$ = findMedian($startIndex, endIndex$)
10: $parent.left$ = new TreeNode()
11: $parent.right$ = new TreeNode()
12: CreateIndex($parent.left, depth+1, startIndex, median$)
13: CreateIndex($parent.right, depth+1, median+1, endIndex$)
14: return
```

Procedure SearchLeaf

```
Require: LeafBlocks $LB$, Constraint $C$
 1: for each $block$ in $LB$ do
 2:     for each $record$ in $block$ do
 3:         if $record$ satisfies C then
 4:             $Result = Result \bigcup \{record\}$
 5:         end if
 6:     end for
 7: end for
 8: return $Result$
```

constraints are tested using the GPU on the leaf blocks, having multiple points to test helps maximize the occupancy of the GPU (Property P2). Last, but not least, the size of the stg-tree is reduced by a factor equal to the size of a leaf block. For example, using a block size of 1024, the number of tree nodes is three orders of magnitude smaller than the input data. This enables the internal nodes of the stg-tree to fit into memory, thus allowing fast, in-memory search over these nodes.

B. *Index Construction*

Given the dimension k of the index, the records corresponding to a node at depth i of the stg-tree are first sorted on dimension $d = i \mod k$. The median value of the d^{th} attribute of the these records is then used to split the records into the left and right sub-trees at this node. The nodes of the left and right sub-trees (with depth $i+1$) are then computed recursively. When the number of records corresponding to a node becomes less than a predefined block size, then this node becomes the leaf node, and a leaf block corresponding to this node is created. The pseudo-code to create this index is shown in Procedure CreateIndex.

C. *Space Requirements*

Let the number of records in the data to be indexed be n. Let b be the size of each leaf block in terms of the number of records to be stored. Then, there are $n_b = \frac{2n}{b} - 1$ tree nodes. The index is stored in three contiguous memory regions. The first region stores the $\frac{n}{b} - 1$ internal nodes of the tree. The internal nodes are stored as a linear array. The order of the nodes in this array is obtained through an in-order traversal of the tree. Figure 4 illustrates the linear order of a stg-tree having 32 nodes. The tree is forced to be complete through the addition of dummy nodes to ensure a unique ordering of the nodes in the linear array.

The second memory region stores the $\frac{n}{b}$ leaf nodes, each of which consists of the k-dimensional bounding box along

Procedure SearchTree

```
Require: Node parent, Integer depth, Constraint C
 1: if $parent.leaf \neq$ NULL and $C \bigcap parent.bound \neq \emptyset$ then
 2:     return $parent.leaf$
 3: end if
 4: Set $LB = \{\}$
 5: $d = depth(\bmod k)$
 6: if $(-\infty, parent.median] \bigcap C_d \neq \emptyset$ then
 7:     $LB = LB \bigcup$ SearchTree($parent.left$, $depth+1$, $C$)
 8: end if
 9: if $(parent.median, \infty) \bigcap C_d \neq \emptyset$ then
10:     $LB = LB \bigcup$ SearchTree($parent.right$, $depth+1$, $C$)
11: end if
12: return $LB$
```

with the offset to its corresponding leaf block. The final region stores the set of leaf blocks. For each record in a leaf block, the values of the $s+m$ indexed attributes are stored along with the pointer to the corresponding record in the database.

III. Query Execution

STIG allows for different query execution strategies both in-memory and out-of-core. The execution of a query on one or more of the index attributes consists of two steps:

1) Identify the set of potential leaf blocks that satisfy the constraints specified in the query, and
2) Compute the result set by searching through the identified leaf blocks.

The execution of Step 1, which identifies potential leaf blocks to search, depends on the strategy to be used. Note that for a spatial attribute, the query constraint can be a polygon. When a polygonal spatial constraint is specified, the bounding box of this polygon is used as the constraint for the corresponding spatial attribute to identify the potential leaf blocks *LB*.

Step 2 computes the result set of the query, and is performed on the GPU using Procedure SearchLeaf. This step is embarrassingly parallel, and is therefore accomplished using a parallel brute-force search among all records within the identified leaf blocks in the set *LB*. When a polygonal constraint is specified for the spatial attributes, the point-in-polygon test is also performed. Note that, since this search is on the set of leaf blocks identified in the first step, it can be accomplished independently and in an out-of-core fashion. Thus, instead of just a single GPU, multiple GPUs can be used to improve query performance. Additionally, the presence of multiple records in a single leaf block helps increase the occupancy of the GPUs (Property P2). In what follows, we describe strategies for both in-memory and out-of-core query evaluation. Additional implementation details can be found in the GitHub repository mentioned earlier.

A. *In-Memory Query Evaluation*

We use in-memory query execution when the entire index fits into GPU memory. The implementation of the first part of query execution, that of identifying the set of leaf blocks, depends on the hardware hosting the database. We implement this step for three possible hardware configurations: hybrid CPU and GPU, GPU-only, and GPU with dynamic parallelism.

Hybrid: CPU and GPU. Searching the tree nodes is performed on a CPU by traversing the tree nodes using the recursive Procedure SearchTree. When searching through a given internal node at depth i, this procedure checks the

```
Procedure FindLeafBlocks
Require: Node $leafNodes$, Constraint $C$
1: for each $node$ in $leafNodes$ do
2:     if $C \bigcap node.bound \neq \emptyset$ then
3:         $LB = LB \bigcup \{node.leaf\}$
4:     end if
5: end for
6: return $LB$
```

validity of the query constraint corresponding to dimension $i \mod k$, and recursively searches either one or both the subtrees at that node. The candidate leaf blocks are then searched on the GPU (using Procedure SearchLeaf).

GPU. In this strategy, we use Procedure FindLeafBlocks to perform a parallel brute force search on the leaf nodes of the set of tree nodes to identify the set of leaf blocks. This avoids the synchronization overhead that would be caused by executing Procedure SearchTree in the GPU. SearchTree would require a parallel breadth-first traversal on the tree, which processes all nodes at a given depth in parallel. This would in turn require synchronization between searches across consecutive depths. Such an approach does not satisfy Property P3, and therefore does not provide any advantage for using the GPU.

To implement FindLeafBlocks on the GPU, it is sufficient to load only the leaf nodes, which are stored in contiguous memory, into GPU memory. Given that this operation is embarrassingly parallel, concurrent searches are performed to test whether the given query constraints satisfy the bounding box corresponding to each leaf node. Since there is only a single call to the GPU, the synchronization overhead caused due to multiple kernel calls is avoided. A sparse boolean array is used to store whether a particular leaf block is to be searched or not. A prefix scan is then performed on this array to obtain the result set of leaf blocks.

While a similar parallel brute-force search could be performed by simply storing the entire data set as blocks and checking the bounding box of these blocks in the first step, such an approach is not efficient since locations of applicable records might be far away, requiring more blocks to be searched in the second step. The stg-tree indirectly helps in this situation since it clusters nearby records (in high dimensional space) into fewer leaf blocks.

Alternatively, taking advantage of the storage structure of the tree nodes, the array storing them can be divided into multiple sub-trees, each of which can be searched in parallel. In such a case, a single GPU core is used to search a given sub-tree. For example, the tree in Figure 4 can be divided into either two or four sub-trees as shown, and searched in parallel. The procedure to divide the set of tree nodes into multiple sub-trees is explained in detail in Section III-B. However, since the search time using the brute-force approach is only a few milliseconds even for data having over 800 million records, we choose this simpler approach in our implementation.

GPU-DP. Newer NVIDIA GPUs support *dynamic parallelism* [34], which allows a CUDA kernel to create and synchronize nested kernels. It allows a child CUDA Kernel to be called from within a parent CUDA kernel, and can optionally synchronize on the completion of that child CUDA Kernel. The parent kernel can then use the output produced from the child kernel without the involvement of the CPU.

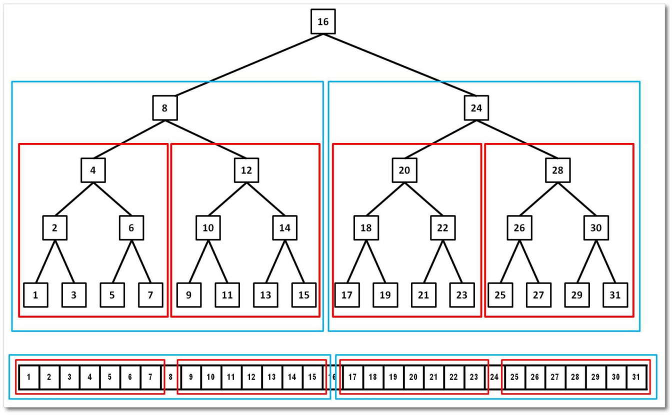


Fig. 4. Internal nodes are stored as a linear array obtained using an in-order traversal of the stg-tree. The red and blue partitions show the division of tree into four and two sub-trees, respectively, that can be searched in parallel.

We take advantage of this feature in our implementation to make the best use of such GPUs. When using dynamic parallelism, if a leaf node satisfies the input query constraints, the SearchLeaf is launched for that leaf block within the GPU itself from Procedure FindLeafBlocks. Note that this is different from the GPU strategy, where the CPU waits (synchronizes) for FindLeafBlocks to complete before launching SearchLeaf.

B. Out-of-Core Query Evaluation

When the index does not fit into GPU memory, then an out-of-core approach is used to execute queries. We consider three possible implementations for this scenario: GPU, hybrid and multi-GPU.

GPU. First, leaf nodes are divided into blocks of nodes, such that each block fits into GPU memory. This is straightforward since the leaf nodes are stored in contiguous memory locations. These leaf nodes are then searched one block at a time to identify the leaf blocks that satisfy the query constraints. The resulting set of blocks is then transferred to the GPU, again in blocks that fit into memory. Note that in an out-of-core implementation, it is not possible to use the dynamic parallelism feature, since the leaf blocks are not present in GPU memory during the first step.

Hybrid. For the hybrid execution, the CPU is used to search the tree nodes as described earlier. If the internal nodes of the index do not fit into CPU memory, the tree nodes are first divided into a set of sub-trees, each of which fits in memory. Let the height of the tree be h. Then the tree nodes are divided into a set of sub-trees at depth i, where i is equal to the smallest value such that $2^{h-i+1} - 1 \times \text{sizeof(node)} \leq \text{Memory}$. This is illustrated in Figure 4, where dividing the tree at depth 1 results in two sub-trees, and dividing at depth 2 results in 4 sub-trees. Since, the tree nodes are stored as a linear array using an in-order traversal, each of these sub-trees correspond to a sub-array as shown in the figure. Each sub-tree is then processed, one at a time, to identify the set of valid leaf blocks. As in the GPU implementation, the second step is executed by transferring the leaf blocks to the GPU in block sizes that fit into GPU memory.

Multi-GPU. When multiple GPUs are present, then similar to the out-of-core GPU implementation, the leaf nodes and leaf blocks of the index are uniformly distributed among the GPUs, and the search is performed on each of the GPUs. If

```
        db.trips.ensureIndex({type: "stig", pickup_time: 1, dropoff_time: 1,
                              pickup: "2d", dropoff: "2d"}, {name: "taxiIndex"})
```

Fig. 5. Command to create an index on the taxi data set.

```
db.trips.find(
{ $or: [{pickup_time: {$gt: start1, $lt: end1}}, {pickup time: {$gt: start2, $lt: end2}},
        {pickup_time: {$gt: start3, $lt: end3}}, {pickup time: {$gt: start4, $lt: end4}} ],
  $pickup: {$geoWithin: {$polygon: [ [mx1, my1], ..., [mxn , myn ], [mx1, my1] ]} },
  $dropoff: {$geoWithin: {$polygon: [ [jx1, jy1], ...,[jxk, jyk], [jx1, jy1],
                                      [lx1, ly1], ...,[lxp, lyp], [lx1, ly1] ]} } })
```

Fig. 6. MongoDB query to find all trips from lower Manhattan to JFK and LGA airports during all Sundays of May 2011. In this query, the time intervals corresponding to the four Sundays are specified using the $or identifier. The polygon corresponding to lower Manhattan is given by the vertices $\{(mx_1, my_1), \ldots, (mx_n, my_n)\}$. The polygons corresponding to JFK and LGA airports are specified by the vertices $\{(jx_1, jy_1), \ldots, (jx_k, jy_k)\}$ and $\{(lx_1, ly_1), \ldots, (lx_p, ly_p)\}$.

after splitting, the index fits into the collective memory of all the GPUs, then an in-memory based search is performed.

IV. MongoDB Integration

MongoDB [32] is a widely-used NoSQL database which stores data using the binary JavaScript object notation (BSON) format. In this section, we discuss the design choices made to integrate STIG into MongoDB. Our code is open source and available at *https://github.com/harishd10/mongodb.git*.

Command interface. MongoDB does not provide an explicit interface to add a custom index. However, the JSON format used for its queries and commands allows for flexibility in defining new commands. We enable the use of STIG by modifying the default *ensureIndex()* command that is used to create an index in MongoDB. This command requires the user to specify the attributes on which the index is to be created together with the type of the attribute. An example command that creates the 6-dimensional STIG index on the taxi data is shown in Figure 5. The first identifier, *type: "stig"*, informs that the index should be a STIG index. The two spatial attributes, namely the pickup and dropoff locations, are specified using the "2d" type identifier. It is assumed that these spatial attributes are stored as an object which consists of two attributes: x and y. For example, the coordinates for the pickup attribute are stored as pickup.x and pickup.y, respectively. We chose to store the temporal attribute as an integer representing the Linux epoch time. This helps in quick comparisons of the temporal attributes.

Query execution. The query optimizer in MongoDB was modified as follows. When a query is issued, if the query constraints correspond to a subset of the attributes on which the index is created, then STIG is used to execute the query. The command in MongoDB to execute the sample query from Figure 1 is shown in Figure 6.

The result of the query is a collection of disk addresses corresponding to the records that satisfy the query constraints. MongoDB supports a cursor interface, which is used to return the results of a query to the user.

Multi-CPU query execution. In addition to the three query execution approaches presented in Section III, we also include a CPU-based implementation in the prototype. Depending on the underlying hardware, the appropriate setting can be used. This allows STIG to be used on machines that do not have NVIDIA cards. We also support the use of multiple CPU cores to improve the query execution time in such scenarios. Here, the final step of searching through the leaf blocks is performed in parallel on multiple CPUs. Note that when an NVIDIA GPU is not available, it is advisable to reduce the size of the leaf blocks. This reduction in block size, while increasing the height (and size) of the kd-tree part of STIG, enables further pruning of the tree during the identification of potential leaf blocks. Additionally, since the number of records in each leaf block is smaller, it also reduces the number of costly point-in-polygon test needed in the second step, thus improving the efficiency of CPU-based query execution.

Index storage. A single STIG index is stored as three binary files corresponding to the three components of the index – internal nodes, leaf nodes, and leaf blocks respectively. Also, given our focus on OLAP-type queries, our current implementation does not update the created index when new records are added to the corresponding collection. Users can schedule index creation to be performed periodically, as new data snapshots are made available.

V. Experimental Results

In this section, we discuss the performance and scalability of STIG. The experiments were performed on a workstation with dual 12-core Xeon E5-2695 processors clocked at 2.40 GHz, 256 GB of RAM, 8 TB of disk storage, and three NVIDIA GeForce TITAN graphics cards – each having 6 GB of GPU RAM. We first describe the data sets used in our experiments in Section V-A. In Section V-B, we report results on the scalability of the index with respect to the number of GPUs present and query selectivity. Finally, in Section V-D, we discuss the rationale in using kd-trees over R*-trees and compare their performance for querying high-dimensional data.

A. Data Description

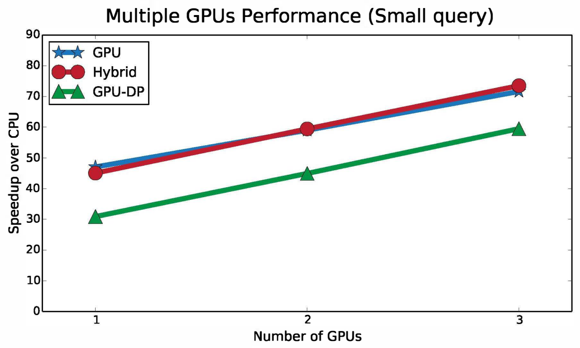
We use two real-world data sets: the NYC taxi data and Twitter data. Both data sets have the property that the distribution of the data points is skewed spatially, while they are uniformly distributed over time.

NYC Taxi data. The NYC taxi data set consists of records corresponding to over 868 million taxi trips that happened over a period of five years [43]. The size of the input data, which was made available as csv files, is 250 GB. Each trip has attributes corresponding to its pick-up and drop-off locations, pick-up and drop-off times, along with other relevant data such as the fare, tip, and distance traveled for that trip.

Twitter data. Twitter provides live public feeds that stream a subset of the tweets posted during a given time period. We collected data corresponding to 1.11 billion geo-tagged tweets over a period of 14 months. Each tweet has information about time and location where it was posted, and the actual text together with other statistics such as the retweet count and favorite count. The data is available as JSON objects and takes about 440 GB space.

B. Scalability

We use the taxi data to study the scalability properties of the GPU index. The out-of-core performance was assessed using the entire data set. To test in-memory and weak scalability



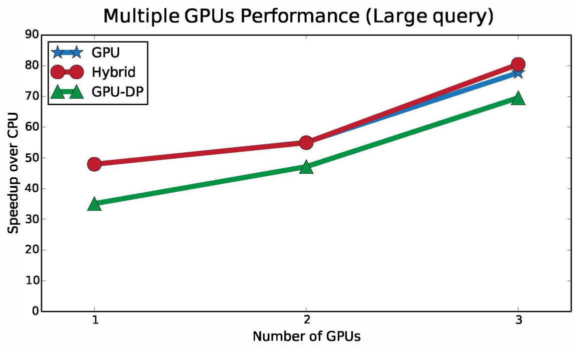
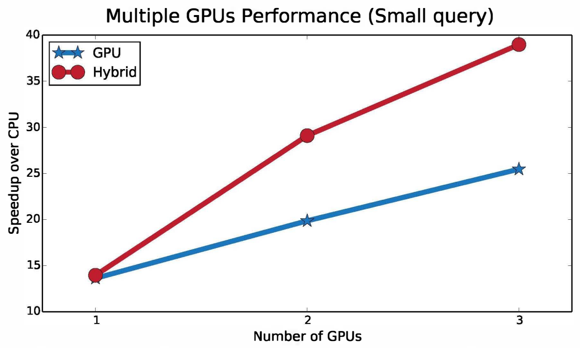


Fig. 7. Strong scalability of multiple GPUs for in-memory query execution. Using three GPUs leads to a speedup of around 70 times over a CPU-based implementation of the index. Note that each of the GPUs has 2688 cuda cores.



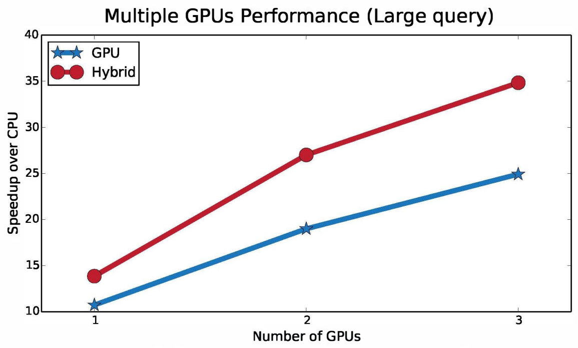


Fig. 8. Strong scalability of multiple GPUs for out-of-core query execution. The hybrid strategy, using three GPUs, leads to over 35 times speedup.

performance, the index was created for a subset of the data uniformly sampled from the entire data set.

The index was created on two spatial attributes—pickup and dropoff locations, and two temporal attributes—pickup and dropoff times. Thus, the dimension of the kd-tree used in the index was six. The leaf block size was set to 4096 for the GPU-based query execution. For all experiments in this section, we used a query that retrieves the set of trips from Midtown to Lower Manhattan. In order to generate different result sizes, we varied the time constraints of the query.

Varying the Number of GPUs. In what follows we study both strong and weak scalability behaviors of STIG.

Strong scalability: The first experiment studies how the speedup attained by the different query execution strategies varies with an increasing number of GPUs. Note that each GPU has 2688 cuda cores. The speedup was computed against a CPU-based implementation of the index, where a single core of the CPU is used for querying. This experiment serves two purposes. First, it demonstrates the benefits of using the GPU when compared to CPU. Second, it helps verify the scalability of the technique with increasing processing power. In order to obtain the best performance for CPU execution, we constructed a traditional kd-tree index (leaf block size = 1).

Figure 7 plots the speedup for two queries, having small and large selectivity, with increasing number of GPUs when the index fits into GPU memory. The queries resulted in 10^5 and 5×10^5 records respectively. The GPU and hybrid implementations are over 70 times faster than the CPU implementation, while the GPU-DP implementation leads a speedup of over 60 times.

Figure 8 shows the speedup for queries when the index does not fit in GPU memory. The full taxi data was used

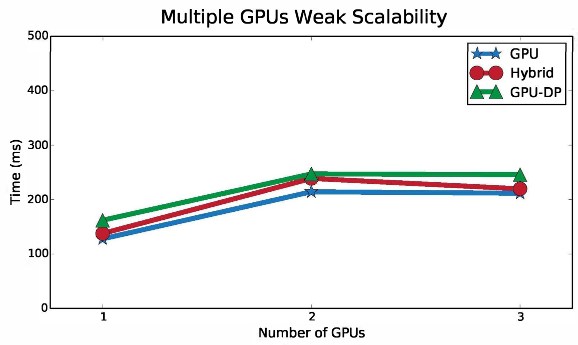
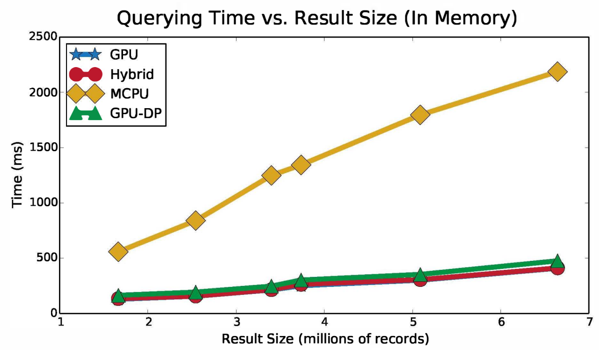


Fig. 9. Weak scalability. The size of data used for querying was increased with the number of GPUs. Note the consistent performance of all the approaches.

for this experiment, and the small and large queries returned 3 million and 13 million records, respectively. Note that for the out-of-core case, hybrid performs better than the GPU-only strategy. This is because, when dealing with data that does not fit in memory, the GPU strategy has an additional overhead of transferring leaf nodes from the CPU to the GPU for identifying potential leaf blocks, which is more expensive than the CPU-based search of the internal nodes that can be accomplished in-memory because of its size (see Section V-C).

An interesting feature that the plots above show is that we obtain a higher speedup for large queries when the index fits in-memory, while the speedup is higher for small queries when the index does not fit into GPU memory. For the in-memory case, fewer leaf blocks are processed by the GPU, and thus, the time to set up the GPU kernel calls takes up a significant fraction of the total query execution time. When more leaf blocks are processed, the GPU occupancy increases resulting in the initial setup time becoming insignificant, thus improving the speedup. In the out-of-core scenario, when more leaf blocks are processed, there is a larger memory transfer overhead, thus impacting the speedup.



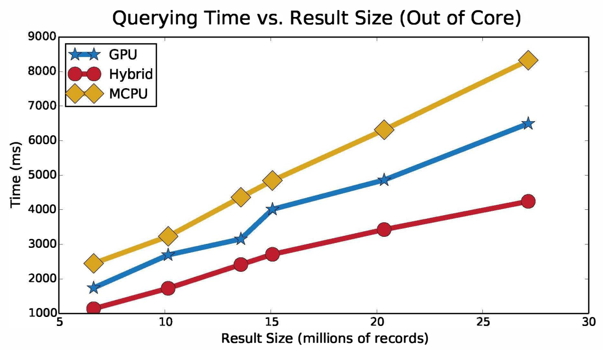


Fig. 10. Comparison of query execution times of the different approaches with increasing query selectivity. We obtain an almost linear performance with increasing query selectivity. Note that GPU and hybrid strategies perform almost the same when the data fits in memory.

Surprisingly, the GPU-DP strategy is marginally slower than GPU. This is because the time taken to setup and execute multiple leaf search kernels directly from the GPU is more than the time taken to launch the second step from the CPU. Unlike our scenario, we believe that the dynamic programming feature will be useful when multiple synchronizations are involved. As mentioned in Section III-B, we do not use the GPU-DP strategy for out-of-core queries. In order to make efficient use of the dynamic parallelism feature, the GPU-DP implementation requires the leaf blocks to be in memory during the first step. This amounts to transferring the entire index, that is all the leaf blocks, into the GPU memory. Since in most queries, the majority of the leaf blocks is not used, this causes unnecessary memory transfers resulting in no advantage over the GPU or CPU-based hybrid implementation.

Weak scalability: We compared the performance of the different strategies with an increasing number of GPUs where the data and query size is fixed per GPU. Figure 9 shows the query execution times for the different strategies. The (almost) constant time for an increasing number of GPUs demonstrates that our approach is scalable and can perform efficiently in a multi-GPU environment.

Varying Query Selectivity. We now compare the time taken to execute queries using the various strategies while increasing query selectivity. As mentioned earlier, the queries used identify all trips that originate in Midtown Manhattan and end in Lower Manhattan. The selectivity was controlled by adjusting the time interval constraints of the queries, i.e., pickup and drop-off times.

In addition to the strategies described in Sections III-A and III-B, we also compare the query execution time of a parallel CPU implementation (Section IV), where the leaf blocks are searched in parallel on multiple CPU cores. For the parallel CPU implementation, we created the STIG index with a leaf block size of 1024, which provided the best performance. For the GPU-based strategies, we report results obtained when using three GPUs.

Figure 10 (left) compares the query execution time with increasing selectivity when the index fits in memory. All three GPU-based strategies have similar performance in this scenario–the tree lines overlap. Also, note that the time taken to execute a query resulting in over 6.5 million records is just around half a second when using these strategies. The sample data in this case consisted of approximately 200 million trips (data points). This size was selected so that the memory of all three GPUs was required to store the index. That is, the index does not fit into memory of even two of the GPUs.

Figure 10 (right) compares the query execution times when using the entire taxi data. In this scenario, we observe that the hybrid strategy consistently performs better than the GPU and multi-CPU strategies. Note that the time taken to execute a query is only around 4 seconds even for queries results as large as 27 million records.

Effect of Leaf Block Size. For all the above experiments, we experimented with multiple leaf-block sizes and chose the ones provided the best performance for the different scenarios. For example, having a larger leaf block size when using one CPU might be more cache-friendly, but also increases the number of PIP tests performed which significantly increases the query execution time. On the other hand when using a GPU (or multiple CPUs), the parallel execution of the PIP tests overcomes this trade-off. The size of the tree is inversely proportional to the leaf block size. However, making the leaf block very large offsets the advantage of using GPUs since not only more data needs to be transferred to the GPU, but more PIP tests also have to be performed.

C. Database Performance

We now compare the end-to-end performance of queries across different database systems. In particular, we compare the time taken using STIG on MongoDB with: the freely-available PostgreSQL, and a commercial database system. Due to legal restrictions, the commercial database system is anonymously identified as ComDB. MongoDB was configured to use the three GPUs present in the test workstation. In all experiments, we use the Hybrid strategy for query execution since it was shown to have the best performance for the out-of-core configuration. We start by discussing the time taken to create indexes on the taxi and Twitter data sets on the different platforms, and then we compare the execution times for queries typically issued by domain experts on these data sets.

Database Setup and Index Creation. For the taxi data, the STIG index in MongoDB was created on four attributes, namely, pickup time, dropoff time, pickup location and dropoff location, respectively. Note that a location consists of latitude and longitude coordinates, and correspond to two dimensions in the stg-tree. On both PostgreSQL (PostGIS) and ComDB, spatial indexes were created on pickup and dropoff locations, together with B-tree indexes on pickup and dropoff time. Creating the index using MongoDB took a little over 2 hours, while it took roughly 18 hours and 35 hours, respectively, to create the required indexes on PostgreSQL and ComDB.

TABLE I. TIME TO CREATE INDEXES ON DIFFERENT DATABASE SYSTEMS.

Data	# Records (in million)	MongoDB Time	PostgreSQL Time	ComDB Time
Taxi	868	2 h 14 m	17 h 56 m	35 h 31 m
Twitter	1112	2 h 07 m	11 h 06 m	18 h 27 m

For the Twitter data, the STIG index in MongoDB was created on two attributes – the tweet time and location – and took a little over two hours. On the other hand, it took around 11 hours in total to create a spatial index on the tweet location and B-tree index on tweet time on PostgreSQL, and over 18 hours on ComDB. Note that we ensured that all other transactions were blocked when creating the indexes. The time taken to create indexes using the different database systems is summarized in Table I. To fine-tune query performance and to enable the optimizer to make informed plan choices, commands were issued on both PostgreSQL and ComDB to collect statistics on all the attributes on which the indexes were created. The queries used in the evaluation involved constraints only on these attributes.

Index size: As mentioned in Section II-C, the three-part structure of STIG leads to a substantial decrease in the size of the data structure to be searched in the first step. This can be observed from the space occupied by the internal and leaf nodes. For the taxi data, the size of the internal and leaf nodes was 30 MB and 20 MB, respectively. For the Twitter data, the corresponding sizes were 25 MB and 13 MB. Given such small sizes, even for data having around a billion records, it is possible to efficiently perform the tree search on a CPU without incurring any performance penalty. Even though the leaf block sizes for these two data sets are larger (46 GB and 34 GB), since only the filtered blocks are searched, the memory access/transfer overheads are minimal.

Query Execution.

Taxi data: We use the following queries to test the performance of the different database systems on the taxi data.

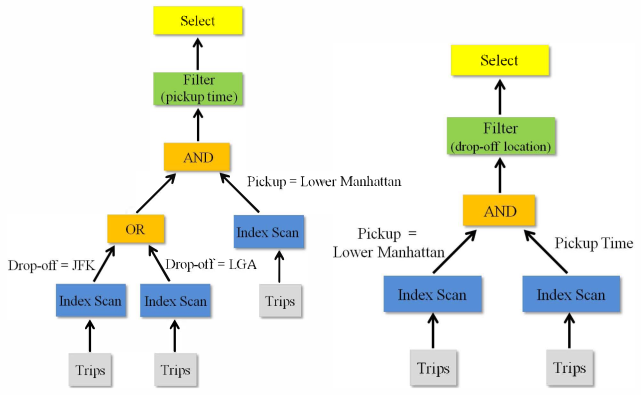
1) Find all trips that occurred between Lower Manhattan and the two airports, JFK and LGA, during all Sundays in May 2011.
2) Find all trips that occurred between Lower Manhattan and the two airports, JFK and LGA, during all Mondays in May 2011.
3) Find all trips that occurred between Midtown and the two airports, JFK and LGA, during all Sundays in May 2011.
4) Find all trips that occurred between Midtown and the two airports, JFK and LGA, during all Mondays in May 2011.

Table II compares the query execution time of the different database systems for these queries. Even though these queries return only around 13,000 records, the results indicate that existing database systems are not suitable for the interactive operations required by visual analytics tools. Note that STIG is over 6000 times faster than PostgreSQL and more than 250 times faster than ComDB.

The significant speedup obtained can be attributed to two important limitations of the query execution strategy followed by existing systems. First, the optimal query plan as identified by the optimizer has to perform one or more costly joins and filtering operations over the records obtained from multiple index scans. Figure 11 shows the query plans used by the two database systems to execute Query 1 on the taxi data. Both Postgres and ComDB perform a join operation followed by a filtering operation. In addition, since index scans are performed on a spatial attribute, a large number of expensive point-in-polygon tests are performed during query execution. This number is further increased if additional spatial filtering is performed during a later stage of query execution. For example, PostgreSQL performs index scans for the three spatial constraints. On the other hand, ComDB performs an index scan to select records satisfying the pickup location constraint, and filters on the dropoff location during the last stage of query execution. Since these spatial constraints are in the form of arbitrary polygons, the large number of containment checks slows down the query execution.

TABLE II. COMPARISON OF EXECUTION TIMES FOR SPATIO-TEMPORAL QUERIES ON THE TAXI DATA ACROSS DIFFERENT DATABASE SYSTEMS.

Query	MongoDB	PostgreSQL		ComDB	
	Time(s)	Time(s)	Speedup	Time(s)	Speedup
1	0.075	503.9	6718	20.6	274
2	0.080	501.9	6273	23.3	291
3	0.067	437.8	6534	21.6	322
4	0.070	437.1	6244	32.6	465



(a) Postgres (b) ComDB

Fig. 11. Optimal query plans (as selected by the query optimizer) used to execute Query 1 on the taxi data.

In contrast, due to the multi-dimensional nature of STIG, we need to perform only a single index scan for such queries. Moreover, since the polygon containment tests are performed in parallel in the GPU, we are also able to obtain a significant speed-up in the query execution time.

Twitter data: We use the following queries to study the performance of the different database systems on the Twitter data.

1) Select all tweets posted from Washington DC in June 2013.
2) Select all tweets posted from Boston in June 2013.
3) Select all tweets posted from Manhattan in June 2013.

These queries result in 1.3×10^5, 1.5×10^5, and 3.7×10^5 records respectively. Table III shows the query execution times for the three database systems. Our approach is at least two orders of magnitude faster than both PostgreSQL and ComDB. When using our index on MongoDB, the actual query execution time is less than 50 ms for all queries—the remaining time is spent on retrieving the result records to be returned. This is reflected in the table, where the times are proportional to the result sizes.

TABLE III. COMPARISON OF QUERY EXECUTION TIMES FOR SPATIO-TEMPORAL QUERIES ON THE TWITTER DATA ACROSS DIFFERENT DATABASE SYSTEMS.

Query	MongoDB	PostgreSQL		ComDB	
	Time(s)	Time(s)	Speedup	Time(s)	Speedup
1	0.246	161.2	655	109.6	445
2	0.288	151.2	525	157.7	547
3	0.558	286.0	512	216.8	388

Compared to the queries over the taxi data, the large query result size for the Twitter queries adversely affects the execution times in ComDB. However, we notice an improved performance by PostgreSQL. This is because, the query plan chosen by PostgreSQL (Figure 11(a)) for a taxi query initially scans for all records that satisfy the spatial constraints, and finally filters on the time constraint. As mentioned earlier, this increases the number of point-in-polygon tests that are performed (in the worst case, for each trip record, 3 tests need to be performed corresponding to one pickup polygon and two dropoff polygons). In the case of Twitter data, there is only one spatial attribute, which not only reduces the number of joins needed, but also significantly reduces the number of costly polygon containment tests since there is only a single polygonal constraint.

On the other hand, for a taxi query, ComDB first identifies records that satisfy the pickup spatial constraint and the temporal constraint. This result is then filtered based on the dropoff spatial constraint. The dropoff constraint test is therefore performed on a smaller subset of the data compared to the number when performing the spatial index scan (which indexes the entire data set). This significantly reduces the number of polygon containment tests required (note that there are two polygons specified for the drop-off location). It therefore performs significantly better than PostgreSQL for taxi queries. However, in case of Twitter queries, both ComDB and PostgreSQL have similar query plans, thus showing comparable performance.

D. *Kd-tree vs. R*-Tree*

R-tree-based indexes are known for their robustness against data skew and suitability for disk-based query processing. They are the most widely used index type in spatial extensions available in existing database systems. Nevertheless, they have two critical drawbacks for high-dimensional spaces that makes them unsuitable for our purpose. First, optimizing an R*-tree for high-dimensional data, i.e., minimizing overlapping regions while maximizing coverage of the minimum bounding rectangles (MBR) of child nodes, is both non-trivial and computationally expensive. Typically good splitting strategies [8], [18] have quadratic growth with the number of dimensions, making the index construction process less scalable. Second and more important for our purpose, the overlapping regions among the child nodes of the tree grows rapidly with the increase in the number of dimensions. Thus, query performance is hampered as more false-positive nodes have to be scanned [8].

Different from R-trees and R*-trees, the bounding boxes of sibling nodes in a kd-tree do not overlap. To test the practical effect the overlapping nodes of a high-dimensional R*-tree have, we used a subset of the taxi data having 67 million trips and performed a set of bounding box queries on in-memory CPU implementations of both an R*-tree and a kd-tree. Figure 12 plots the query times obtained with increasing

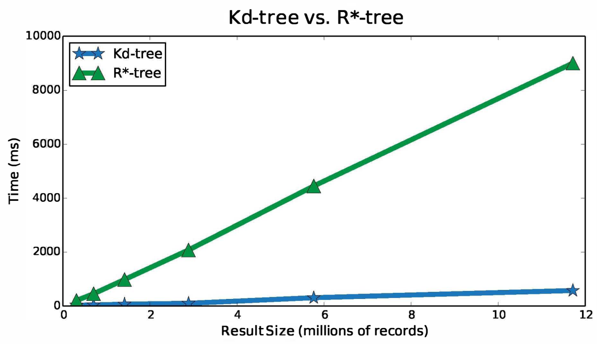


Fig. 12. Comparing the performance between a 6D kd-tree and R*-tree.

query selectivity. Note that the kd-tree consistently performs better than the R*-tree. In fact, increasing the number of query constraints had a higher impact on the R*-tree than on the kd-tree. This is because of the large overlap in the R*-tree nodes, requiring more nodes to be scanned during the search traversal. Even for a query returning over 11 M records, the kd-tree takes less than 600 ms, while the same takes 9 seconds using the R*-tree. Given the sub-second response times required for our application, we opted to use kd-trees as the basis for our index structure. Note, however, that the R-tree (or any block-based structure) can be transformed into a stg-tree-like structure to leverage GPUs for speeding up spatial queries.

VI. DISCUSSION

Spatial data structures. Other choices of data structures used for spatial indexes are the quad tree [14] and grid index [38]. While these can also be extended to support points in high-dimensional spaces, they quickly become inefficient for large and skewed data sets. The taxi and Twitter data sets used in our experiments are uniformly distributed over time, but spatially skewed. For example, in the taxi data, there is a high density of taxi pickups and dropoffs near transit locations such as airports or train stations, but in parts of upper Manhattan or the Bronx, there are significantly fewer pickups and dropoffs. This skew would result in a high-depth quad tree, thus decreasing its efficiency. In the case of a grid index, while some of the grid cells would contain a large number of data points, others would have very few, making the distribution of the data uneven. More importantly, the resolution of the grid which is predefined could further hamper the efficiency of queries.

Updating the index. Supporting dynamic updates to the index would require two main properties to be satisfied. First, the update has to ensure that the resulting tree is balanced. Second, for effective usage of GPUs, the different components of the index, especially the leaf nodes and leaf blocks, should be in contiguous memory locations. Since the focus in this work is on the analysis of historical data, we decided against supporting dynamic updates to the index. Also, given the relatively short time required to create the index (approximately two hours for indexing 1 billion records), users can schedule index creation to be performed periodically.

A naïve update approach could create a highly-unbalanced tree and negatively impact query performance. In future work, we plan to explore data structures similar to the kd-B-tree [39] to support updates to the index. The kd-B-tree is a multiway tree similar to the B-tree that supports dynamic updates. However, the dynamic nature of this data structure makes it non-trivial to split the search operation across multiple GPUs. We also plan to explore other space division strategies used for kd-tree creation in order to help speedup index creation and to obtain more efficient kd-trees.

Choice of database system. The analysis tools we use to explore urban data are Web based and use MongoDB as their data store. We therefore chose to integrate our index into MongoDB to obtain interactive query execution for these tools. However, since the storage structure of STIG is independent of the underlying database system, it can also be integrated with a relational database system.

VII. Related Work

The advent of programmable GPUs has resulted in several efforts to utilize them for database operations. Due to the geometry involved when using spatial data, it is natural to exploit GPUs for processing and querying such data. Directly related to this work, Bandi et al. [4] proposed an architecture to perform spatial queries using GPU hardware. They use the GPU for computing intersection and containment properties by rendering the polygons and points using the GPU. This was implemented as a stored procedure in Oracle. Zhang et al. [47] proposed a GPU-based point-in-polygon spatial join. They used a grid file based approach to join a set of 2D points with a set of polygons. In [49], they perform nearest-neighbor based spatial joins. Both techniques assume that the entire data set fits in CPU memory. This assumption, however, does not hold for larger data sets such as the taxi data.

Cazalas et al. [9] proposed a GPU-based framework to obtain proximity views on spatio-temporal streaming data. They use the GPU to compute the distance matrix of the set of objects, and use this distance matrix to obtain the required proximity views. Ilarri et al. [21] provide a comprehensive survey on existing techniques used for spatio-temporal query processing. These techniques work on temporal data, where objects are constantly in motion. The queries are performed on the existing state of the objects, taking only the current positions into account. Therefore, queries involving historical states of the objects are not supported.

GPUs have also been used to speed up relational database operations. The work by Govindaraju et al. [17] was one of the first to exploit the use of GPUs for relational database query processing. They evaluated the performance of operations such as selection and aggregation using GPUs on databases having up to one million records. Zhang et al. [48] used GPUs to compute aggregates over spatial data. Govindaraju et al. [16] proposed a GPU-based sorting technique that is capable of efficiently sorting over a billion records using the GPU. Kim et al. [27] used GPUs to perform index search using memory and layout optimized binary trees. He et al. [19] presented and evaluated the performance of different join operations using the GPU. Fang et al. [11] evaluated various compression schemes on GPUs in order to improve GPU-based query processing by decreasing the memory transfered to GPUs. Ao et al. [3] used GPUs to perform index compression and a binary search based list intersection. Krueger et al. [28] proposed a GPU-based dictionary merge algorithm for column-oriented databases.

Most of the the above mentioned techniques were implemented as standalone systems. More recently, Aji et al. [2] introduced Hadoop GIS, a spatial data warehousing system built over Hadoop, and integrated into Hive [42]. While this system can handle large data sets, its performance for spatial containment queries, even on a single spatial attribute, is on par with existing database systems, and is therefore not suitable for interactive environments. Existing, freely-available database systems use traditional spatial indexes to support spatial queries. PostGIS / PostgreSQL [36] use an R-tree based index, while SQLite [41] uses an R*-tree to execute spatial queries. MongoDB [32] uses a geohash-based index [15] to support spatial queries. However, in these systems a spatial index can support only one spatial attribute.

Kd-trees have been used to build multi-dimensional indexes [39], as well as image indexes [22], [23]. The kd-trees used in these applications are CPU-based and do not lend themselves to interactive querying. Kd-trees have also been used for spatial indexes [24], [37]. Again, these indexes support the index only on a single spatial attribute, and are not suitable for interactive query execution. GPU-based kd-trees are commonly used in ray tracing [20], [44], [50]. These approaches build a kd-tree on the triangles of the input geometry, which is then used to identify the triangles intersected by a ray corresponding to a pixel. These approaches are restricted to indexing just the triangles. They also assume that the entire data fits into GPU memory. As mentioned earlier, this is not true for data sets like the NYC taxi data, which do not fit into CPU memory let alone GPU memory.

VIII. Conclusion

The explosion in the volume of spatial-temporal data obtained through different sensors, combined with the use of interactive visualization tools to analyze such data, imposes stringent response time requirements on the execution of spatio-temporal queries. In this paper, we proposed STIG, a GPU-based indexing scheme that supports interactive response times for such queries over large data. We demonstrate the efficiency of the index using two real-world data sets: 868 million NYC taxi trips and 1.1 billion Twitter posts. Our experimental results show that STIG is between 35 and 80 times faster than the CPU-based implementation of the index. We have also integrated STIG into the freely-available MongoDB database and compared query evaluation under this implementation against both open-source and commercial databases that use traditional spatial indexes: MongoDB+STIG is between two and three orders of magnitude faster.

Our current implementation has some limitations that we intend to address in future work. First, it does not support updates to the index when new records are added to the database. Users are required to re-create the indexes periodically. Note that a naïve update approach could create a highly-unbalanced tree and adversely affect the query performance. We plan to investigate alternative update procedures that ensure the resulting stg-tree is balanced. Second, because our implementation is based on CUDA, it cannot be used on systems with other graphics cards. We plan to provide an OpenCL implementation that supports all graphics cards. We would also like to integrate STIG into PostgreSQL. Last, but not least, we intend to add support for other spatial queries including nearest neighbors.

Acknowledgment

This work was supported in part by a Google Faculty Award, an IBM Faculty Award, the Moore-Sloan Data Science Environment at NYU, AT&T, DARPA, NSF awards CNS-1229185, CNS-1405927, and CCF-1533564. We acknowledge the support of NVIDIA Corporation with the donation of a Tesla K40 GPU used for this research.

References

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: Queries with bounded errors and bounded response times on very large data," in *Proc. EuroSys*, 2013, pp. 29–42.

[2] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: A high performance spatial data warehousing system over mapreduce," *PVLDB*, vol. 6, no. 11, pp. 1009–1020, Aug. 2013.

[3] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin, "Efficient parallel lists intersection and index compression algorithms using graphics processing units," *PVLDB*, vol. 4, no. 8, pp. 470–481, 2011.

[4] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi, "Hardware acceleration in commercial databases: a case study of spatial operations," in *Proc. VLDB*, 2004, pp. 1021–1032.

[5] L. Barbosa, K. Pham, C. Silva, M. Vieira, and J. Freire, "Structured open urban data: Understanding the landscape," *Big Data*, vol. 2, no. 3, 2014.

[6] L. Battle, M. Stonebraker, and R. Chang, "Dynamic reduction of query result sets for interactive visualizaton," in *Proc. IEEE Big Data*, 2013, pp. 1–8.

[7] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[8] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The x-tree: An index structure for high-dimensional data," in *Proc. VLDB*, 1996, pp. 28–39.

[9] J. Cazalas and R. Guha, "Geds: Gpu execution of continuous queries on spatio-temporal data streams," in *Proc. EUC*, 2010, pp. 112–119.

[10] B. Coltin and M. Veloso, "Scheduling for transfers in pickup and delivery problems with very large neighborhood search," in *Proc AAAI*, 2014.

[11] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *PVLDB*, vol. 3, no. 1-2, pp. 670–680, 2010.

[12] J.-D. Fekete and C. Silva, "Managing Data for Visual Analytics: Opportunities and Challenges," *IEEE Data Eng. Bull.*, vol. 35, no. 3, pp. 27–36, 2012.

[13] N. Ferreira, J. Poco, H. T. Vo, J. Freire, and C. T. Silva, "Visual exploration of big spatio-temporal urban data: A study of new york city taxi trips," *IEEE TVCG*, vol. 19, no. 12, pp. 2149–2158, 2013.

[14] R. Finkel and J. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[15] "Geohash," http://www.geohash.org/.

[16] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *Proc. SIGMOD*, 2006, pp. 325–336.

[17] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proc. SIGMOD*, 2004, pp. 215–226.

[18] A. Guttman, "R-trees: a dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984.

[19] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proc. SIGMOD*, 2008, pp. 511–524.

[20] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha, "Memory-scalable gpu spatial hierarchy construction," *IEEE TVCG*, vol. 17, no. 4, pp. 466–474, 2011.

[21] S. Ilarri, E. Mena, and A. Illarramendi, "Location-dependent query processing: Where we are and where we are heading," *ACM Comput. Surv.*, vol. 42, no. 3, pp. 12:1–12:73, 2010.

[22] U. Jayaraman, S. Prakash, and P. Gupta, "Indexing multimodal biometric databases using kd-tree with feature level fusion," in *Information Systems Security*. Springer, 2008, vol. 5352, pp. 221–234.

[23] Y. Jia, J. Wang, G. Zeng, H. Zha, and X.-S. Hua, "Optimizing kd-trees for scalable visual descriptor indexing," in *Proc. IEEE CVPR*, 2010, pp. 3392–3399.

[24] Y. Jian-si, C. Ya-ling, and L. Peng, "A multi-precision partial kd tree index based on database," in *Proc. CiSE*, 2010, pp. 1–5.

[25] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl, "M4: A visualization-oriented time series data aggregation," *PVLDB*, vol. 7, no. 10, pp. 797–808, 2014.

[26] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi, "Distributed and interactive cube exploration," in *Proc. ICDE*, 2014, pp. 472–483.

[27] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *Proc. SIGMOD*, 2010, pp. 339–350.

[28] J. Krueger, M. Grund, I. Jaeckel, A. Zeier, and H. Plattner, "Applicability of gpu computing for efficient merge in in-memory databases," in *ADMS*, 2011.

[29] L. Lins, J. Klosowski, and C. Scheidegger, "Nanocubes for real-time exploration of spatiotemporal datasets," *IEEE TVCG*, vol. 19, no. 12, pp. 2456–2465, 2013.

[30] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE TVCG*, vol. 20, no. 12, pp. 2122–2131, 2014.

[31] Z. Liu, B. Jiang, and J. Heer, "immens: Real-time visual querying of big data," *Computer Graphics Forum (Proc. EuroVis)*, vol. 32, 2013.

[32] "mongoDB," http://www.mongodb.org.

[33] T. Munzner, *Visualization Analysis and Design*. CRC Press, 2014.

[34] *CUDA Dynamic Parallelism Programming Guide*, http://docs.nvidia.com/cuda/cuda-dynamic-parallelism/index.html, Nvidia, August 2012.

[35] "NYC Open Data," http://data.ny.gov.

[36] "PostGIS: Spatial and geographic objects for PostgreSQL," http://postgis.net.

[37] O. Procopiuc, P. Agarwal, L. Arge, and J. Vitter, "Bkd-tree: A dynamic scalable kd-tree," in *Advances in Spatial and Temporal Databases*. Springer, 2003, vol. 2750, pp. 46–65.

[38] P. Rigaux, M. Scholl, and A. Voisard, *Spatial Databases with Application to GIS*. Morgan Kaufmann Publishers Inc., 2002.

[39] J. T. Robinson, "The k-d-b-tree: a search structure for large multidimensional dynamic indexes," in *Proc. SIGMOD*, 1981, pp. 10–18.

[40] P. Santi, G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti, "Quantifying the benefits of vehicle pooling with shareability networks," *Proceedings of the National Academy of Sciences*, vol. 111, no. 37, pp. 13 290–13 294, 2014.

[41] "SQLite," http://www.sqlite.org.

[42] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *PVLDB*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.

[43] "NYC TLC Trip Data," http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[44] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o(n log n)," in *IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 61–69.

[45] H. Wickham, "Bin-summarise-smooth: a framework for visualising large data," had.co.nz, Tech. Rep., 2013.

[46] E. Wu, L. Battle, and S. R. Madden, "The case for data visualization management systems," *PVLDB*, vol. 7, no. 10, pp. 903–906, 2014.

[47] J. Zhang and S. You, "Speeding up large-scale point-in-polygon test based spatial join on gpus," in *Proc. BigSpatial*, 2012, pp. 23–32.

[48] J. Zhang, S. You, and L. Gruenwald, "High-performance online spatial and temporal aggregations on multi-core cpus and many-core gpus," in *Proc. DOLAP*, 2012, pp. 89–96.

[49] ——, "High-performance spatial join processing on gpgpus with applications to large-scale taxi trip data," http://geoteci.engr.ccny.cuny.edu/pub/nnsp_tr.pdf, CUNY City College, Tech. Rep., 2012.

[50] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Trans. Graph.*, vol. 27, no. 5, pp. 126:1–126:11, 2008.

[51] K. Zoumpatianos, S. Idreos, and T. Palpanas, "Indexing for interactive exploration of big data series," in *Proc. SIGMOD*, 2014, pp. 1555–1566.