# A Scalable Distributed Spatial Index for the Internet-of-Things

Anand Padmanabha Iyer
University of California, Berkeley
api@cs.berkeley.edu

Ion Stoica
University of California, Berkeley
istoica@cs.berkeley.edu

## ABSTRACT

The increasing interest in the Internet-of-Things (IoT) suggests that a new source of big data is imminent—the machines and sensors in the IoT ecosystem. The fundamental characteristic of the data produced by these sources is that they are inherently geospatial in nature. In addition, they exhibit unprecedented and unpredictable skews. Thus, big data systems designed for IoT applications must be able to efficiently ingest, index and query spatial data having heavy and unpredictable skews. Spatial indexing is well explored area of research in literature, but little attention has been given to the topic of efficient distributed spatial indexing.

In this paper, we propose Sift, a distributed spatial index and its implementation. Unlike systems that depend on load balancing mechanisms that kick-in post ingestion, Sift tries to distribute the incoming data along the distributed structure at indexing time and thus incurs minimal rebalancing overhead. Sift depends only on an underlying key-value store, hence is implementable in many existing big data stores. Our evaluations of Sift on a popular open source data store show promising results—Sift achieves up to 8× reduction in indexing overhead while *simultaneously* reducing the query latency and index size by over 2× and 3× respectively, in a distributed environment compared to the state-of-the-art.

## CCS CONCEPTS

• **Computer systems organization → Cloud computing**;

## KEYWORDS

Distributed data store; big data; spatial indexing

## 1 INTRODUCTION

The Internet-of-Things (IoT) has lately been gaining a lot of attention. While the idea of connecting millions of devices to the existing Internet infrastructure was originally dismissed by many as far-fetched, it is slowly becoming a reality and we are rapidly moving towards a fully connected world. Recent surveys estimate the number of connected devices to reach approximately 30 billion by the year 2020 [1, 22].

The implications of such an explosion in connected devices is profound. Even with every device generating a very small amount of data, the sheer number of devices suggest that a new source of big data seems imminent: billions of devices in the Internet-of-Things ecosystem. Big data is no newcomer in both academia and the industry, and the field of big data storage and processing has matured over the last few years. However, data from connected devices differ from the existing notion of big data in several ways, and the unique characteristics of the new data source may prove to be a challenge for current big data systems.

First and foremost, the data exhibits a characteristic very unique to it that is different from all present day big data sources: since these connected devices are deployed in the wild, they sense and map the world at unprecedented scales, capturing real-world phenomenon in real-time. Hence, *the data produced by these sources are inherently geospatial in nature*. Current data storage and querying frameworks support many operations on data ranging from simple retrievals to equality operations and more. However, basic geospatial queries include complex polygon containment, intersection and join operations and thus require robust multidimensional indexing techniques for efficient performance. Indeed, spatial indexing has been a thriving area of research in database literature for over a decade [20, 24, 25, 28]. Yet, little attention has been given to the topic of *spatial indexing*[1] *in a distributed setting*. Indexing techniques such as the RTree or GiST are still not amenable to implementation in a distributed environment with high data churn.

Second, the data sources have *high velocity*, either individually or collectively. Today, a typical use case for big data storage and querying systems involve handling human generated data. A familiar example is Twitter, whose record tweet rate currently stands at around 150,000 per second [47]. Compared to this, an average cellular network operator in the United States serves millions of subscribers in each service zone, thus resulting in millions or more machine generated data (e.g., control plane messages) per second. Similarly, a weather service network may generate millions of data records from each scan. Hence, these new data sources will generate an order of magnitude, if not more, data compared to existing sources. While many big data systems can handle the three V's (volume, velocity and variety), and thus may be able to support the velocity requirements in simple scenarios, the complexity of geospatial analysis poses a different set of challenges.

Third, many IoT applications place stringent requirements on data freshness[2]. Emerging applications such as Self-Optimizing Networks (SON) heavily rely on getting the data as soon as it is available for producing meaningful results. Similarly smart cities and connected vehicle applications may be of no use with stale data. In extreme cases, advanced disaster recovery and warning systems may prove disadvantageous if the data is not available for use immediately. Thus, in such scenarios, the value of the data

[1]In this paper, we refer to spatial data as complex polygons, not simple points.
[2]We define freshness as a measure of how long it takes from the time the data enters the system until when it is ready to be served.

*SoCC '17, September 24–27, 2017, Santa Clara, CA, USA*
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3132254

diminishes rapidly and it is critical that the system handling it must be able to serve queries on the data with minimal or no delay.

Finally, the interaction of these characteristics with each other result in stricter requirements. While geospatial data is hard to handle by itself, in-the-wild deployments make the data sources show extreme and unpredictable skews both in space and time. Partitioning and load balancing are critical for performance in the context of databases and key value stores [19, 30, 31, 46], and hence many sophisticated data stores deal with skews by doing post-ingestion balancing. However, continuous ingestion of heavily skewed data would result in frequent balancing operations that may affect the performance of the system. In addition, the freshness requirements precludes such techniques since they are costly. All these clearly underline the requirements of a data storage and querying system for IoT applications—they must be able to efficiently ingest, index and query highly dynamic spatial data having unpredictable skews.

We believe the key to building such a system is a robust, load-balanced distributed spatial indexing data structure. In this paper, we present Sift, a distributed spatial index and its implementation. Sift is designed with skew-resistance as its primary goal. In contrast to many existing indexing proposals that require post-rebalancing operations, Sift tries to mitigate skew at *ingestion time*. The key idea in Sift is that *data distributions exhibiting skews in their native dimensionality can be uniformly partitioned in a higher dimensionality space*. Based on this idea, Sift proposes using properties of the data as additional dimensions to assign .

The basic indexing structure in Sift is a multidimensional tree that stores spatial objects in its nodes. Sift uses additional dimensions derived from non-workload characteristics of the data to create nodes and assign objects to them. By using a recursive space decomposition function with these additional dimensions, Sift ensures that any node in this tree can be computed at any time thus enabling lazy operations. Further, the node assignment strategy ensures a uniform partitioning of objects to nodes, avoiding hotspots. The object assignment strategy ensures a deterministic node location for an object, obviating the need for costly split and merge operations common in tree based spatial indexing techniques. Sift utilizes space-filling curves [26, 38] for node addressing in a distributed setting, making it possible for the data structure to be implementable in a key value store. All these design decisions makes the index distributable across machines at the level of nodes, thus enabling efficient load-balancing and massively parallel operations.

In summary, we make the following contributions in this paper:

- We present Sift, a massively parallel, distributed spatial index that has skew-resistance as its main goal. Sift can easily be implemented on a distributed key-value store.
- We propose a general technique, based on using additional dimensions, for load-balancing spatial skews, and show properties of spatial data that could be used for this purpose.
- Sift's data distribution strategy that provides guarantees on the upper bounds on data duplication in the index.

We have implemented Sift and evaluated it against two common spatial indexing techniques: a GiST index (PostGIS [39]) and a space-filling curve based index (MongoDB [33]). Our evaluations of Sift using openly available and proprietary datasets consisting of over 1.5 billion records in a 20-node cluster on Amazon EC2 has yielded promising results: Sift is able to reduce the indexing time by up to 8× while simultaneously reducing the query latency of even simple queries by over 2× and index size by up to 3×.

## 2 BACKGROUND & MOTIVATION

We begin with a brief overview of spatial indexing techniques. We then discuss the challenges in supporting spatial analytics for IoT applications in a distributed setting, thereby motivating the need for this work. We present a survey of the state-of-the-art solutions in this space, highlighting their shortcomings next. Finally, we list the desirable properties in an ideal solution.

### 2.1 Spatial Indexing Techniques

The primary objective of a spatial index is to organize two[3] or more dimensional objects in order to serve queries efficiently. Fundamentally, this can be achieved by defining *buckets* that group one of two things—the underlying geometric space or the objects. When a query is to be executed, the candidate list of buckets are narrowed down using query constraints and the resulting buckets are searched. For efficiently searching neighborhoods, a spatial index organizes these buckets in such a way that buckets representing spaces close by are placed near by in the index structure. The efficacy of the index heavily depends on the bucket definition. A popular technique for bucket organization is to hierarchically arrange them using a tree [20, 24, 25]. Trees support both space and object grouping, and can provide reasonably efficient query responses, especially for neighborhood queries. For buckets based on space-grouping, other techniques such as using an associative array to represent uniform grids in space [51], can be used. A large number of proposals exist based on both these ideas [42].

Inherent support for non-point spatial data (data which has non-zero size, such as polygons) exist in only a few indexing techniques, namely the Rtree [24], GiST [25] and its variants. R-trees are data structures that store n-dimensional geometries by enclosing them in a rectangular box parallel to the axes commonly called the minimum bounding rectangle (MBR). Nearby objects are grouped into their combined enclosing box hierarchically forming a tree structure. Each higher level in the tree is formed by aggregating lower layers, and the root represents the entire space under consideration. This organization of objects provides efficient querying for neighborhood, containment and intersection. The performance of these data structures is dependent on the explicit assumption that the tree is balanced. Additionally, it is hard to parallelize R-trees, thus making them difficult to use in a distributed setting.

Indexing techniques based on space-grouping are more suited for a distributed environment. In this class, a popular choice is the Quadtree family [20]. A quadtree partitions the underlying space into subspaces, each of which becomes a bucket. Each bucket is given a bounded capacity on the number of objects it can hold. An object is put in the first bucket which has remaining capacity. When a bucket capacity is exceeded, it is split into two or more children (by splitting the subspace), and the objects are moved to the children they fall in. This procedure is repeated recursively. An example quadtree is shown in fig. 1. Compared to R-trees, quad-trees do

[3]In this paper, we focus on 2 dimensional data, since it is more common. However, our work is not limited to it.

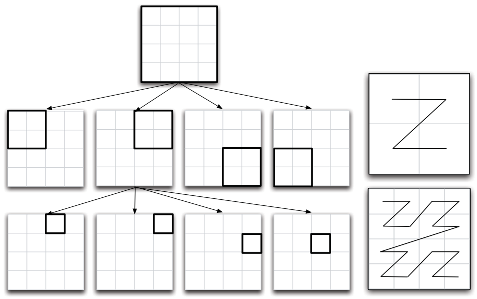


Figure 1: Quad-tree and an equivalent geohash

not depend on a balanced structure. In addition, unlike R-trees, the subspaces of quadtrees are static. This makes it easy to create them beforehand and partition among nodes in a cluster. The major shortcoming with the quadtree is that they are highly inefficient for non-point geometries. To store polygons, the quadtree (or its variants) must store them in all the buckets where the geometry falls in. Due to this reason, they are mostly used for point data.

## 2.2 Challenges

Our goal in this work is propose a robust distributed spatial index that can support emerging IoT applications. There are several challenges in achieving this goal.

***Nature of IoT Workloads & Queries.*** Current spatial workloads are either focused on answering queries on stationary data (e.g., *"find all gas stations near me"*), or on low-volume human generated data that is relatively static (e.g., *"friends near me"*). Emerging IoT applications, such as driverless vehicles, connected cars and connected camera networks could easily produce magnitudes more non-point (polygon) data compared to them. For instance, an existing IoT application, cellular network analytics, produces several million records per second [27]. Enabling spatial analytics on these workloads require a robust indexing structure that can sustain high ingestion rate. Additionally, in comparison with the simple queries supported by existing big data stores, spatial queries range from complex polygon intersection and containment searches to more difficult spatial joins [21, 42]. These operations can only be carried out by parallelizing them which in turn requires that the underlying index be massively parallel.

***Insufficiency of Spatial Indexing Techniques.*** Existing spatial indexing techniques, discussed above, were not designed to support *dynamic* spatial data. As such, they have severe difficulties scaling and/or parallelizing. R-trees are not suited for distributed settings, since the tree structure keeps changing in a highly dynamic environment, and thus balancing them is difficult if not impossible. Quadtrees are more suited for clusters, but are inefficient for non-point data. Many problems arise when quadtree based indices are used for storing non-point data. Since the quadtree must store an object in all the buckets where the geometry falls in, it creates many duplicates of the geometry, thus increasing indexing overhead. Additionally, when doing containment or intersection queries, a large number of leaf nodes need to be searched. More importantly, since each geometry is indexed in many keys, there is no clear way to partition the key space. Grid-based indices that work by statically dividing the space can be massively parallel, but has problems similar to quadtrees for polygon data. In addition, they are are not adaptive. These difficulties point towards the need for a robust indexing technique.

***Skew Handling.*** A more fundamental challenge is to *achieve a load balanced partitioning of the indexed objects in the data structure across multiple machines*. Partitioning and load balancing are critical for performance in a skewed environment [19, 30, 31, 46], and a natural approach to deal with load imbalance is to do the rebalancing after ingestion. However, in environments like IoT where the data is highly skewed and unpredictable, this would mean frequent rebalancing. An alternative approach is to use techniques that try to balance data objects in the cluster by using hash value of a high cardinality field (hash partitioning [15]). However, such partitioning destroys the spatial relationship and queries would result in scatter-gather patterns which are known to be expensive in cloud environments [14]. Thus, it is crucial to maintain spatial relationship between objects while load balancing.

## 2.3 Existing Solutions

To understand the landscape of existing big data stores that support spatial indexing, we conducted a survey of the most popular open-source data stores. Since there are more solutions available today for ingesting and retrieving large datasets than we could possibly list here, we resort to the most popular ones. DBEngines [13] is a popular online resource that ranks open-source databases by their popularity. The popularity is determined via a number of factors such as the frequency of searches, technical discussions on the internet and trends. For the lack of a better alternative, we chose to use this ranking for selecting data stores to review. While analyzing solutions, we avoid commercial solutions and restrict ourselves to the ones that either provide built-in support for spatial datatypes, or are amenable to an implementation. Further, we do not consider solutions that are purely in-memory. This is because the sheer volume of data and the need for historic analysis does not bode well in such solutions. Our findings are presented in table 1.

The most mature database for storing and retrieving spatial objects is PostgreSQL [40] with PostGIS [39] module enabled. Indexing multidimensional objects is done using either R-trees [24] or in recent versions using GiST [25]. PostGIS provides support for complex geometry types and advanced queries, including efficient spatial joins. As a result, it is very popular among GIS applications. However, it is not popular for big data applications due to the shortcomings of Rtrees discussed earlier. Thus, the user base for PostGIS is limited to cartography and visualization.

NoSQL data stores have gained attention in the recent years due to their ability to support extremely large datasets via horizontal scalability. Over time, many different classes of NoSQL databases have emerged, from simple key-value stores to complex document databases. However, we find that spatial data type support in these data stores is lacking or rudimentary.

The common approach to make quad-tree like structures distributable is to use geohashes or space-filling curves (fig. 1). In such solutions, each leaf in the complete quadtree is represented as a geohash which is a one dimensional key. The higher the level, the smaller the length of the hash. Geohashes and space-filling curves

| Database | Spatial? | Non-point? | Data Structure | Limitations |
|---|---|---|---|---|
| MongoDB [33] | ✓ | ✓ | Quadtree, Geohash | No spatial partitioning |
| PostgreSQL [39] | ✓ | ✓ | R-tree, GiST | Difficulties in distributed settings (problems sharding spatial geometries) |
| Cassandra [4, 15] | ✗ | ✗ | None inbuilt | Custom |
| CouchDB [5] | ✓ | ✓ | R-tree | Not suitable for frequent inserts |
| HBase [6, 11] | ✗ | ✗ | None inbuilt | Custom |
| Solr [7] | ✓ | ✗ | Geohash | Expensive updates, experimental |

**Table 1: Only a few of the most popular DB engines support spatial indexing (ranking data from DBEngines [13]). Some datastore have been modified to support spatial datatypes (marked as *Custom*) but their capabilities are unknown.**

are similar in concepts. Thus, instead of storing the tree structure, the data store only needs to store keys, which they are extremely efficient at. A number of data stores utilize this indexing technique, namely MongoDB [33], and Solr [7]. The most sophisticated indexing and querying support is provided by MongoDB—it supports polygon data types and provides inbuilt containment and intersection queries. However, it does not support spatial partitioning. Thus, queries are routed to all partitions and the results are aggregated to produce the answer.

## 2.4 Desirable Properties

Based on our survey, we concluded that there is a need for a better solution for supporting spatial analytics in emerging IoT applications. The main properties for such a solution include:

- *Robustness:* The solution must be able to handle the ingestion requirements of these new applications, which generate magnitudes more data compared to human generated spatial datasets. This precludes any solution that is based on Rtree or its variants. In particular, the solution must be able to ingest and index *dynamic* spatial data at a massive rate.
- *Skew-Resistance:* It must be able to handle unpredictable and unprecedented skews in the data. In other words, it must be able to uniformly distribute highly skewed data along the index without hotspots.
- *Massively Parallel:* The complexity of queries and the amount of data obviates any single machine solution. The resulting solution must be able to scale effortlessly in a cluster.

We describe each of these in detail and how our proposal, Sift, achieves them in the rest of this paper.

## 3 SIFT DESIGN

We now describe the design of Sift in detail. We begin with the datastructure used to store spatial objects. Then we discuss how the design of data distribution along the datastructure keeps Sift load balanced by mitigating skews. Then we show how the index can be distributed in a cluster for massively parallel operations on it. Finally, we discuss the different operations on the datastructure.

## 3.1 Data Structure

A well-designed index datastructure is crucial for obtaining efficient insertions and retrievals. In many cases, an index often speeds up queries at the cost of degradation in ingestion performance. This overhead is small for many classes of applications, but IoT data presents a different challenge. The high rate of data ingestion requirement combined with complex spatial data type may prove ingestion overhead to grow prohibitively large as the amount of data stored grows. Overheads associated with indexing can be due to two reasons: the objects indexed may occupy more than what is required due to duplication (i.e., write amplification), or the insertion operation may trigger a restructuring of the indexing structure that is costly. Thus, a key design goal for the Sift datastructure is to avoid restructuring operations.

The basic datastructure in Sift is a tree, in which we represent spatial objects by their minimum bounding boxes[4] , similar to Rtrees and Quadtrees. However, unlike Rtrees (and similar to Quadtrees) we organize the underlying space rather than objects in the tree. Since the requirement of having a balanced tree is a deterrent to our goals, we focus on data structures that can remain unbalanced. This aids us in reducing the restructuring operations in a highly dynamic environment. Thus, the tree datastructure in Sift is unbalanced and can remain so without any performance penalties. In the tree, every node is a logical storage buffer for spatial objects and is given a bounding box that is a subspace of the entire space under consideration. Unlike Rtrees, we enforce that node boundaries never overlap in the same level of the tree. Parent nodes enclose the bounding box of their children. Thus, the root node represents the bounding box of the entire space.

An important design decision is in the creation logic for these tree nodes. A simple solution for node creation is to define a bound on the number of objects stored in each node. When a node overflows, it is split to two or more children and objects are moved to the children. This is the approach taken in quadtrees. Unfortunately, doing so leads to tree restructuring during splits and merges. A potential alternative is to statically divide the space equally (e.g., using a grid) and assign an object based on its location. This has the advantage that we can precompute nodes and create them before hand. However, this approach requires rigid grid sizes and does not offer flexibility for load balancing.

In Sift, we take an entirely different approach. We *do not* impose a bound on the number of objects stored in each node. This means that a node in the tree could potentially contain a large number of objects, bounded only by the available memory. Additionally, in contrast to existing approaches of *splitting* a parent node to create children and redistributing the objects among them, we never move objects from a node once it is assigned. New nodes are created by dividing the bounding box of a parent into two or more subspaces. The unbalanced tree structure, along with the unbounded capacity at the nodes and the non-necessity of object movement when nodes are created or destroyed enables Sift in achieving

[4]While we refer to these as bounding boxes for simplicity, they are actually hyperrectangles in a general $N$ dimensional space.

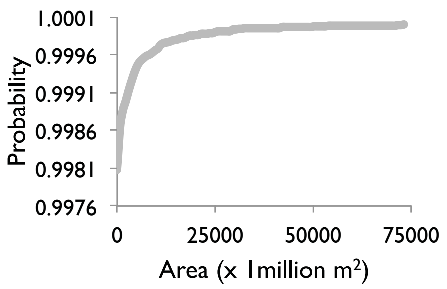


Figure 2: The area of spatial objects exhibit powerlaw distribution. Sift uses such properties as dimensions for skew mitigation.

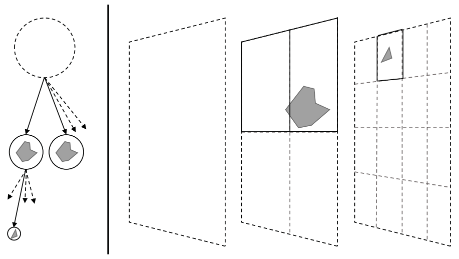


Figure 3: The key idea in Sift is to use multiple layers of indexing, formed using additional dimensions derived from characteristics of the data that are not workload dependent. Here, objects are assigned to layers using their size. The tree on the left, which represents the two layers depicted on the right, is the datastructure used by Sift. Nodes are lazily created (solid lines show nodes that exist).

its goal of supporting extremely high ingestion rate. To insert a new object into the index, Sift doesn't incur *any* tree rebalancing, splitting or merging operation at all.

## 3.2 Data Distribution & Skew Mitigation

While the unbalanced, unbounded tree is very well suited for data ingestion, a spatial index must also support efficient queries. The query efficiency of an index heavily depends on how data is distributed along the index. In particular, hotspots (too many objects in a single bucket) result in poor performance. To illustrate this, consider a simple example of a spatial index consisting of equal sized buckets formed by partitioning the space uniformly. Such an index would perform efficiently only when the objects are uniformly distributed among the buckets. Unfortunately, IoT applications generate data that have heavy skews in them. Thus, load balancing is crucial for performance. In Sift, this boils down to answering the question of *how/when to create nodes* in the tree.

The answer to this question lies in a data distribution strategy that would *avoid* hotspots. To tackle this problem, Sift takes inspiration from Grid file, proposed by Nievergelt et. al. [35]. Originally a solution to avoid clipping issues with techniques such as quadtree, grid file proposed that if an object cannot be added in an existing index without a costly operation, store it separately. In similar spirit, our goal is to propose a data placement technique which would allow us to store objects that should have been placed in the same node in separate nodes while still be able to retrieve them without any additional aids—doing so would mean hotspots do not occur.

To devise such a strategy, we leverage a key insight in Sift—*if a data distribution exhibits skew in its native dimensionality, then it is possible to obtain a uniform partitioning in a higher dimensionality space.* In other words, it is impossible to uniformly distribute $N$ dimensional spatial data in an $N$ dimensional space in a general way. It is easy to see that this is true: for any spatial index, the presence of a large number of objects at the exact same location results in an impossibility of uniform distribution. This means that solutions to load-balancing problems in a given dimensional space can only be achieved by embedding the problem in a higher dimensional space. If we could derive these additional dimensions from the objects to be indexed, we would reach our goal. In Sift, we propose using *characteristics or properties* of spatial data as additional dimensions. We explicitly avoid making any assumptions on the nature of the workload (e.g., knowledge of spatial distribution of objects) in coming up with these dimensions, as it would defeat our goal of supporting unpredictable skews. Thus, we use generic characteristics of the spatial objects to be indexed. We only assume that these dimensions have a reasonable cardinality.

We have analyzed many such characteristics of spatial datasets from different sources [9, 10, 27, 37, 45] and have found that real-world spatial data exhibits skew in many other dimensions in addition to its workload dependent dimension of location. For instance, in almost all of the spatial datasets we looked at, the area enclosed by the polygon (its size) shows power-law distribution. An example of this is shown in 2, which depicts the area distribution of 122,000 landmarks in the United States [9]. Similar trends were observed in a taxi cab dataset of 1.1 billion trip records [45] and data collected from a cellular network [27], where the path of the taxi/network user can be viewed as the polygon object. This observation suggests that we could use these additional dimensions, such as polygon area, to separate the objects that would otherwise have to be stored in the same bucket. For the purpose of Sift, this is equivalent to assigning objects to different nodes. This is intuitive: for example, not all polygon objects at the same location would have the same size. Hence, using the size as a criteria for node creation/assignment would index these objects in different nodes, mitigating skew.

Of course, one additional dimension may not be enough, and the technique might fail if many objects end up in the same node (e.g., too many objects with the same area at the same location). Although such cases are rare in practice, it is easy to add more dimensions. These additional dimensions may even be domain specific. For instance, in the taxi dataset, taxi type could be a property that is workload independent. Similarly, in the cellular network dataset, network usage time is a potential dimension. From our experience, we expect that the number of additional dimensions required to mitigate spatial skew would not be more than a couple, since each dimension drastically reduces the probability of the skew. It is easy to see that as plenty of additional dimensions could be found if necessary, and applying domain specific knowledge helps.

This proposal is readily accommodated in Sift, as its basic datastructure is a tree that can handle multiple dimensions, or a multi-dimensional tree. We create new nodes in the tree by partitioning the space in the bounds of the parent into two or more equal subspaces. We call each level of nodes a *layer*[5], since each layer can act independently as a separate index. To assign objects to nodes,

[5]For the rest of this paper, we use the words level and layer interchangeably.

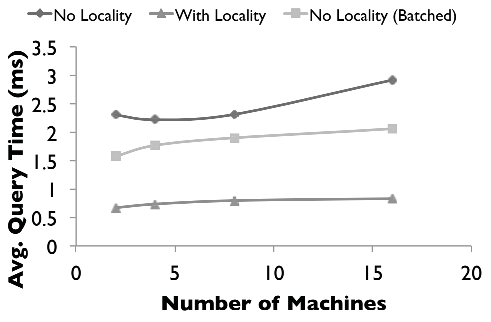


**Figure 4: Average query time from running 1 million queries in Amazon EC2 with different locality settings (Batched indicates a coordinator returns the data compared to the client querying servers individually). Data locality plays a huge role in query response time in cloud hosted machines.**

we use the object's geometric hypervolume as the criterion. Specifically, the hypervolume we consider is the volume occupied by the object's bounding box. We assign an object to a node whose sub-space volume is equal to the object's. We do so by *sifting* the object through layers, starting from the root, until it cannot be furthered because the next layer is made up of nodes whose volumes are lesser than the object's. We then add the object to the node which encloses the spatial dimension. It is possible that there is no single node in the layer that *encloses* the object due to its spatial location. In this case, we assign the object to all the nodes which encloses the object partially (we will later show that this replication is constant-bounded). It is important to note that it may not be optimal to add unrelated dimensions to the same tree, such as incorporating time as a dimension. In such cases, we index them in separate trees. Thus in a general setting, SIFT's indexing structure is not a single tree, but a collection of multidimensional trees. In each tree, we use hypervolume to assign objects to nodes.

For ease of understanding, we will illustrate the basic idea using two dimensions as an example. In a two dimensional space, the hypervolume is equivalent to the area. Each layer in this tree contains nodes with a specific area. To insert an object, we sift the object through layers until we reach a layer whose area is less than the object's. We then assign the object to the layer before it (i.e., the object is assigned to the layer whose area is closest to the object's). To assign the object inside the layer, we use its location. Figure 3 depicts the key idea. The use of multiple multidimensional trees and using the hypervolume as the object distribution function enables SIFT achieve its second goal, skew mitigation.

## 3.3 Partitioning & Load Balancing

The final goal of SIFT is to support *massively parallel* operations. For achieving this goal, the index structure should be distributed among machines in the cluster. We made two explicit design decisions that makes it easy to do this task: first, the nodes in the index are lazy and are not created unless an object is to be indexed in it, and second, each node in the tree can potentially be assigned to a different machine due to independence in operation. However, there are two remaining challenges in making SIFT work in a distributed

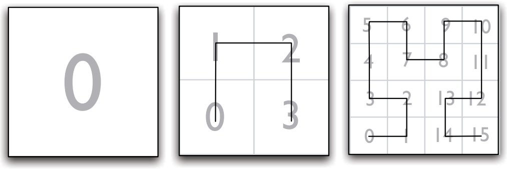


**Figure 5: A Hilbert curve showing the first two iterations.**

environment: how do we address tree nodes, and how do we assign nodes to cluster machines.

One possibility for addressing is to have a globally unique identifier for each of the nodes in the tree, and then have each machine keep a lookup table that maps a node to the machine that own it. However, this indirection is costly when the data is dynamic requiring high rates of insertions and queries. Instead, we need a node addressing scheme that is content addressable. Towards this, we enforce that the SIFT's node creation logic follow binary space partitioning. That is, when a tree node is to be created, it is done by binary partitioning the space represented by the parent's bounding box. We can then use the bounding box of a node, represented by two coordinate tuples[6], as the address for the node. When there is only one tree, this address is globally unique. When there are multiple trees, we can easily make the address globally unique by assigning a static identifier for the tree and incorporating it in the node's address. This recursive partitioning scheme enables us to compute *any* node in any layer in the tree at any time, without the need to create them beforehand.

A straight-forward way to assign the nodes among multiple cluster machines is to use a hash partitioner on the node address. The hash key of the address is used to determine the location of the node in the cluster. Since tree nodes are created on-demand, this provides great flexibility and load-balances the assignment of nodes to machines. However, it also negates one unique characteristic of spatial data: queries in these applications often depend on neighborhood computation. A hash partitioner would place nearby data in different machines with high probability, thus resulting in queries requiring data from multiple servers. The effect of this in cloud hosted environments is well-studied [14], and our own experience confirms these findings [7](fig. 4). Thus, SIFT should enable spatial partitioning on these datasets.

To solve this problem, we leverage a well studied topic in literature, space-filling curves. Our use of binary space partitioning to create tree nodes lends itself naturally to the decomposition property provided by space-filling curves. At each level in a tree, all the possible nodes can be traversed using a space-filling curve of an order that is appropriate for the node's bounding box granularity. In other words, every level in the tree index represents one level in the space-filling curve's decomposition hierarchy. Thus, the levels can be viewed as different iterations of a space-filling curve as illustrated in fig. 5[8]. Unlike the normal usage of space-filling curves (for indexing objects), SIFT uses it just to address the tree nodes and load-balance them in a cluster.

[6]Consisting of $min$ and $max$ for each dimension, for $N$ dimensions.

[7]With the advancements in RDMA, we expect this to be less of a problem in the future, and thus will benefit SIFT as it gives greater flexibility in fine-grained load-balancing.

[8]Note that although we show a Hilbert curve for illustration, it is not fundamental. SIFT can also use Z-curves which are faster to compute.

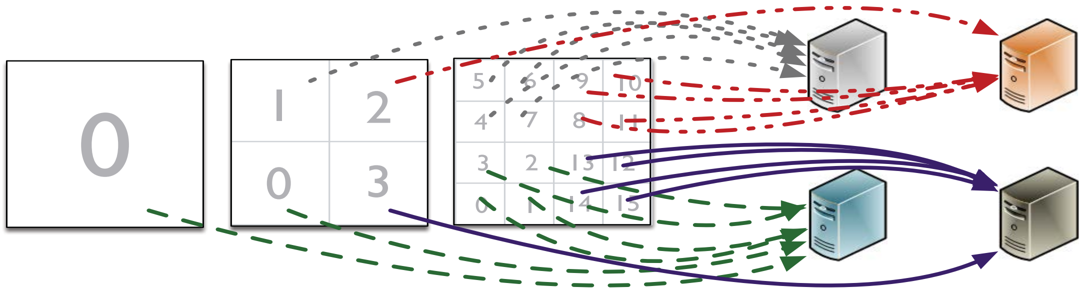


**Figure 6: An example of how SIFT might distribute tree nodes in a cloud-hosted cluster. Here, there are 21 nodes in 3 levels containing data. Range partitioning these nodes using lexicographically ordered binary keys results in 5 nodes in each of 3 machines and 6 in the 4th. For lookup, SIFT can easily prefix match keys. For instance, all keys beginning in 01 are assigned to the same machine in this example.**

A level in SIFT's tree index is defined by a space filling curve of order $n$. For a space-filling curve in $d$ dimension, this means that the level would consist of $2^{n \times d}$ subspaces (nodes) that can be addressed by the curve. To distribute the subspaces preserving spatial properties, we simply generate the 1 dimensional key for the subspaces and then order them lexicographically. Range partitioning the key space then would result in a spatial partitioning that preserves locality. To illustrate this, consider again the curves in fig. 5. Since the maximum order is 2, the maximum number of subspaces is 16, and can be represented using 4 bits. A unique property of the space-filling curve is its recursive nature: from the highest order, any lower order subspace's key can be generated by simply dropping bits from the key. For instance, the key for the bottom-right subspace in each of the levels is 0, 3 and 15 which in binary are `0`, 11 and 1111 respectively. By sorting them in that order, SIFT can assign these three keys to the same partition thus preserving spatial proximity. In case of multiple trees, we derive the final key by prepending this key to the tree identifier. For instance, if we choose to add a new tree based on time as a dimension (e.g., when indexing taxi trip datasets), we can simply append the time identifier of the tree to the key thus retaining the spatial proximity characteristics. It is clear that lexicographical ordering works here too and nearby objects in space *across* levels of different tree can potentially be assigned to the same machine if required.

The lazy creation of tree nodes, and the use of space-filling curves to address tree nodes enables SIFT to load-balance at a very fine-granularity during *ingestion* with minimal need for rebalancing. In fact, the use of higher dimensionality for assigning objects to nodes ensures that nodes in the tree are evenly loaded, greatly reducing the chances of a heavily loaded node. Additionally, because the space-filling curve in each level is independent, SIFT has the flexibility to range partition them independently, but still achieve spatial proximity. This means that even when rebalancing becomes necessary, they are minimal and can be achieved by simply moving nodes at the edge of the range window. The SIFT data structure, due to its use of binary space partitioning and object assignment algorithm, can guarantee that any object in its index will not be duplicated more than $2^n$ times for an $n$-dimensional index (appendix A).

## 3.4 Operations

To insert a data object, SIFT first finds the layer in which the object is to be inserted. To do this, SIFT traverses the tree from the root, checking in each layer if the object can be fully contained in any of the nodes in the layer. SIFT might rotate or translate the object in space to make it fit in a node since the objects original spatial location may not allow it to be contained in any nodes. Once a layer is found, SIFT then finds all the nodes in that layer that either contains or intersects the object's original spatial location and adds the object to all of those nodes (Algorithm 1).

**Algorithm 1** Inserting a geometry $g$

```
procedure GetKeys($g$)
    $k \leftarrow empty$                          ▹ Initialize empty key set
    $l \leftarrow empty$                          ▹ Layer in which to index g
    for $i \leftarrow layers$ do
        if g fully contained in a node of i then
            $l \leftarrow i$                      ▹ Find layer to index g in
        end if
    end for
    for $n \leftarrow Nodes(l)$ do                ▹ For nodes in this layer
        if $Contains(n, g)$ or $Intersects(n, g)$ then
            $k += Key(n)$                         ▹ Add the node's key
        end if
    end for
    return $k$
end procedure
```

In order to answer containment or intersection queries, we can leverage the properties of the data structure. To find all objects contained within a given search object, we start by finding the keys for the object in the geometry as if it is to be inserted into the data structure. This gives us the initial key set that must definitely be searched. By the property of the data structure, any object that is contained within the search object cannot be at any higher level. However, it could be in any child levels. Thus, for every key in the initial set, we add all children to the candidate set. The containment search algorithm is shown in algorithm 2. Intersection queries are similar, but must also query all parents.

## 3.5 Putting It All Together

To summarize our approach, SIFT's distributed spatial index consists of one or more unbalanced trees, where nodes are created by binary space partitioning. To resist skews, SIFT uses additional dimensions of the data that are workload independent. On this higher dimension

**Algorithm 2** Containment search

```
1:  procedure KeysToQuery($g$)
2:      $k \leftarrow GetKeys(g)$                    ▷ Geometry's key
3:      $j \leftarrow emtpy$                          ▷ Additional keys to search
4:      for $i \leftarrow k$ do
5:          $children \leftarrow Children(i)$          ▷ Find all children
6:          for $child \leftarrow children$ do
7:              $j+ = Key(child)$                    ▷ Add the child node's key
8:          end for
9:      end for
10:     return $k + j$
11: end procedure
```

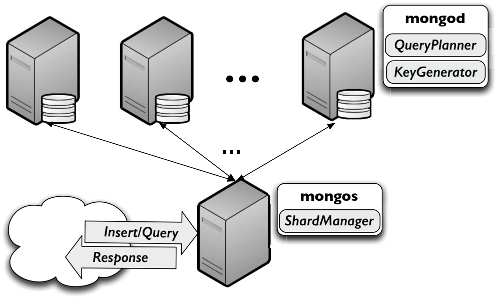


**Figure 7: Sift architecture. Grey boxes are Sift components.**

space, Sift assigns objects to nodes where the hypervolume of the bounding box of the object is fully contained. In two dimensions, we show that using area of the object results in uniform assignment of objects to nodes even under high skews in space. Sift further uses space-filling curves to address these lazily created nodes, and range-partitions them across machines in the cluster.

## 4 IMPLEMENTATION

We have implemented Sift in MongoDB [33], a popular distributed data store. We chose MongoDB as our base because it provides native support for non-point spatial data types, complete with intersection and containment query support. This gives us an opportunity to do a fair comparison of Sift against a well received solution. However, we note that the techniques that we presented in this paper are not specific to our implementation, and hence can be incorporated into any distributed data store.

The MongoDB distributed architecture consists of two types of daemons running on servers. The `mongod` is the database daemon that stores and serves data. Indexing in MongoDB is implemented on B-trees, and are built independently by each server using the data items present in them. `mongos` is the router daemon that interfaces with external applications. It manages the balancing of data across machines and handles requests. Upon receipt of a data item or request, it routes them to the appropriate machine(s), gathers results and returns condensed responses.

Spatial indexing in MongoDB is provided using Google's S2 Geometry Library [23], an open source library that enables geometric operations on a sphere. The library internally implements a Hilbert space-filling curve to achieve this task. It provides in-built functions to convert find the 1 dimensional key(s) in the space-filling curve for polygon geometries. Sift is implemented as library modules and spans both `mongod` and `mongos`. We reuse S2 library for space-filling curve related functionality. Fig 7 shows the architecture of Sift. It consists of three modules: `KeyGenerator`, `QueryPlanner` and `ShardManager`. The first two are implemented in `mongod` and the last is incorporated in `mongos`.

When a data item is received, `KeyGenerator` module examines the contained geometry and finds the Sift tree node(s) in which the geometry would be stored. The result of this operation generates the one dimensional key string associated with the node. The `ShardManager` the finds the partition that is the owner of the node, annotates the data with the node it should belong to, and forwards it to the owning partition. The server responsible for the data item looks up the annotation and determines the location of the item in its index. The item is then saved to a memory mapped file, and the key is inserted into a B-tree index. The update and delete procedures are similar.

For data retrievals, the `ShardManager` first determines the partitions that should be contacted. For this purpose, it examines the contained geometry in the query, just as it were to insert the geometry into the data store. This returns the location of the object in the cluster. It then uses Sift's querying algorithm to find potential tree nodes that need to be queried. The combination of the nodes can be used to determine which partitions need to be contacted to answer the query. The candidate nodes are annotated in the query and is forwarded to the partitions. The individual partitions then can read these annotations and determine which of the nodes that need to be queried are owned by them. These nodes are searched for the query and results returned.

We used MongoDB's production release series version 3.2 to implement Sift. Our modifications to MongoDB codebase consist of approximately 2000 lines of code.

## 5 EVALUATION

In evaluating Sift, we attempt to answer the following questions:

- *How much indexing overhead does it reduce?*
- *What is the query performance?*
- *Does it scale with increase in machines?*
- *Does performance hold with increase in load?*
- *How tolerant is it to spatial skews?*

Our evaluations show that:

- In a single machine environment, Sift outperforms PostGIS's GiST index by 5.18×. Compared to MongoDB, Sift is able to achieve up to 4× reduction in indexing time *while simultaneously reducing* the index size by 33% and the query latency by 30%.
- On a 20 node EC2 [3] cluster, Sift achieves even higher benefits: compared to MongoDB, it is able to reduce the indexing time by 2×-8×. The corresponding reduction in index size and per-query latency is up to 3× and 2× respectively.
- Sift does not require frequent rebalancing; with 1.5 billion objects, the average and standard deviation of the number of object storage blocks in the cluster machines were 176.8 and 4.75 respectively.

| Dataset | Records | Size |
|---|---|---|
| All landmark in USA (Tiger) | 122K | 406 MB |
| All cities in earth (OSM) | 542K | 844 MB |
| All parks in earth (OSM) | 234K | 102 MB |
| All rivers in earth (OSM) | 555K | 945 MB |
| Taxi trip records | 1.1 billion | 280 GB |
| Cellular network (partial) | 500 million | 2 TB |

**Table 2: Real-world datasets used in evaluations (from [27, 45, 49]).**

In this section, we delve deep into these results. We begin by describing our experimental setup and datasets.

## 5.1 Comparisons

We compare the performance of Sift against two existing approaches to storing and retrieving large scale polygon data: a GiST based storage solution (Postgres+PostGIS), and a Quadtree based solution implemented in a distributed data store using geohashes and space-filling curves (MongoDB). These systems have features or shortcomings that are not the focus of this work: for example, PostGIS supports complex join queries while distributed data stores are not optimized for ad-hoc joins. The evaluations presented in this paper only concern the metrics Sift was designed for, and we do not consider other features.

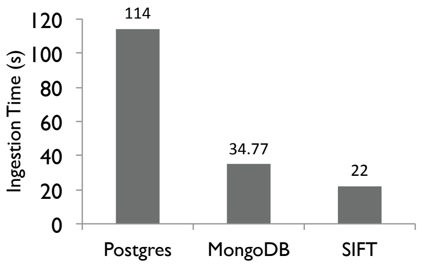
Our choice of these two for comparison stems from our desire to compare two fundamental approaches in spatial indexing and the fact that our selection represents some of the most popular solutions in these categories. A large number of distributed data stores, both open source and commercial, exists today with varying performance characteristics. Through evaluations, our aim is to show the usefulness of our proposed techniques. The ideas in Sift are general and hence can be implemented in other systems.

## 5.2 Evaluation Setup

**Cluster:** We set up a cluster in Amazon EC2 [3] consisting of 20 `r4.xlarge` instances. Each of these machines is equipped with a Intel Xeon E5-2686 high frequency processor, 30.5 GB of system memory. These instances are recommended for high performance databases. We do not enable advanced features such as guaranteed IOPS. All the machines run a recent version of Linux.

We installed Sift enabled MongoDB on all the machines and followed MongoDB's recommendation on setting up a distributed cluster [34]: `mongod` and `mongos` runs on all the 20 machines, while 3 machines act as config servers. To run the experiments, a client connects to a random `mongos` instance. To avoid any additional overhead, we turn off journaling in MongoDB (except for comparisons with PostGIS). Sharding is enabled with no replicas, so data items are not replicated. For comparison experiments, we also installed Postgres (v9.5) with PostGIS.

For performing our experiments, we wrote custom programs that use the APIs of these two products to insert, update, delete and query them. To avoid any effects of disk reads, the program maintains an in-memory buffer of the dataset to be inserted. It then simply uses a thread pool to insert/query the items into the data store. In MongoDB, we turn on write-acknowledgement which indicates successful receipt of the data item at the server end.



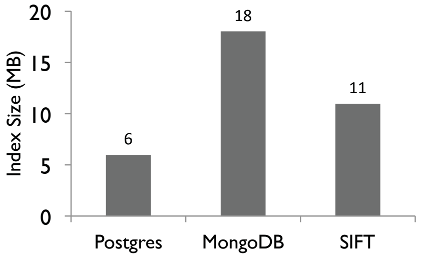


**Figure 8: Object partitioning based indexing incurs high cost in ingestion but lowers overhead, while space partitioning based indexing incurs higher overhead with lower ingestion cost. Sift's benefit in smaller dataset is masked by the time to store objects.**

***Datasets:*** Since IoT is an emerging field, most real-world datasets are proprietary. For evaluating Sift, we used four openly available datasets and one proprietary dataset. For microbenchmarks, we use datasets obtained from OpenStreetMap [37] and US Census Bureau [9], which contain a large number of polygon geometries. For experiments on the cluster, we used taxi trip information obtained from [45] and cellular network data from a tier-1 service provider. The taxi dataset consists of over 1.1 billion trips between 2009 and 2015, with each record noting pickup and drop-off location and a few other metadata. The cellular dataset consisted of several billion records, from which we extracted a partial set of 500 million records with non-negligible network usage duration. A description of different datasets is listed in table 2. Note that the size is the dataset size and may include metadata that are not used.

***Workload:*** A large variety of benchmarking suites exist for big data. However, to our knowledge there exists no spatial benchmarking suite designed for distributed data stores. Hence, we focus on three key metrics that answer the questions we are interested in: indexing time, index size and query latency. The indexing time would reflect benefits of Sift's while the query latency indicates how well Sift deals with skewed data. The size of the index provides information on the usefulness of Sift's technique. For experiments on query performance, we use a mixture of polygon intersection and containment since these are the most common queries.

## 5.3 Single Server Performance

First, we evaluate the benefits of Sift in a single machine setting. Here we are interested in learning the benefits of Sift in terms of indexing overhead and index size.

***Comparison to PostGIS:*.** We extracted a subset of the dataset that consists of approximately 125,000 polygons that represent the important landmarks on earth. These landmarks include airports,

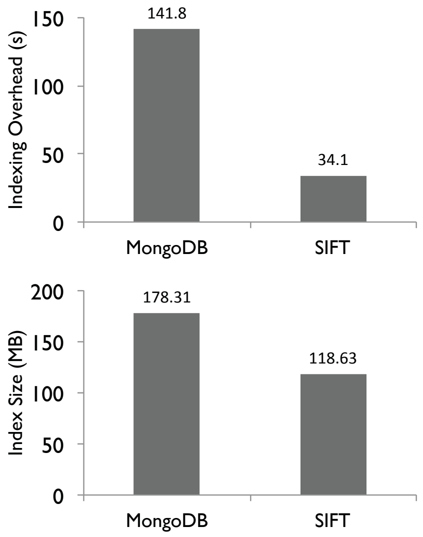


**Figure 9: Sift reduces indexing time by 4X while reducing the index size by about 33% in single machine settings.**

parks, educational institutions etc. Naturally, the dataset exhibits spatial skew, but we do not focus on skew for this particular experiment. Since individual inserts proved to be too expensive in Postgres, we later used the bulk import by providing it with a file with all the records to insert. For MongoDB, we used our client program to insert the data. To obtain a fair comparison, we turned on journaling in MongoDB for this experiment. We start the experiment by creating multi-dimensional indexes in both the systems on the geometry field. Then we warm up the data stores by doing the entire insert and delete 5 times. After that, we measure the insertion time and report the average of 10 such runs in fig. 8.

We find that MongoDB performs 3× better compared to Postgres. This is understandable, since Postgres uses GiST index that incurs heavy insertion costs. In contrast, the approach in MongoDB does not incur rebalancing costs. We note that in smaller datasets such as in this experiment, the time to write the data to disk is significant compared to the indexing time. Thus, the performance of Sift is not significant in this case. Nevertheless, Sift reduces the ingestion time by 37% and the index size by 39%.

These results underline the advantages and disadvantages of the two fundamental indexing techniques for spatial data types: while object partitioning techniques high ingestion cost, they manage index overhead better. Space partitioning based techniques do just the opposite: they incur high cost of index storage while reducing ingestion overhead. Comparing index size of two implementations is not fair, since the numbers depend on the data structures used. Due to the poor performance of PostGIS, and because of its poor support for distributed settings, we drop it from further comparisons.

***Effect of Larger Datasets:*** How do benefits scale with increasing dataset size? To answer this question, we combined a subset of the dataset to obtain 1 million polygons. The last experiment was then repeated with the new dataset. We report the time spent in indexing and the size of the index in figure 9. Sift clearly shows a benefit in this case, reducing the indexing overhead by over 4× and the index size by about 33%. These reductions come from the reduced number of entries that need to be inserted into the index. To confirm this intuition, we examined the average number of index entries created per geometry indexed in both cases. We found that while Sift has a number close to 2, MongoDB's index resulted in roughly 20 keys. Surprisingly, the reduction in the index size does not reflect this number. We attribute it to the storage methodology in MongoDB: internally, it pre-allocates disk space in blocks and represents objects in a custom binary JSON format. Hence, the depicted size may not be the actual disk space.

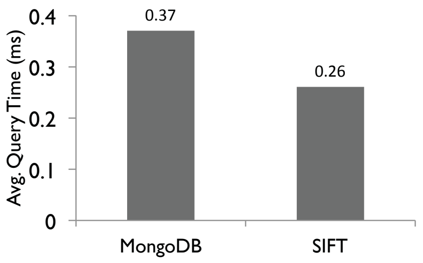


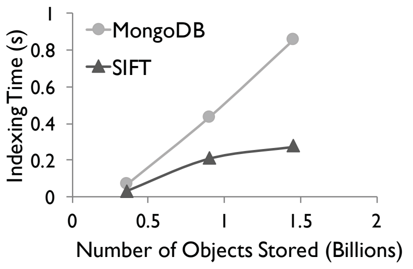
**Figure 10: As expected, the per query time reduces by a small amount in single machine**

***Query Performance:*** Finally we look at query performance in a single machine setting. Specifically we are interested in the reduction in query latency. Intuitively, we do not expect a huge gain because the advantage of Sift is in reducing the number of indices examined. This number may be insignificant when the index is small and is entirely contained in memory. We ran a query set consisting of 100,000 queries on the indexed data and computed the average query latency. The results are in fig. 10. The benefits of not having to scan more parts of the index is clear, and we believe it will increase with the increase in indexed objects.

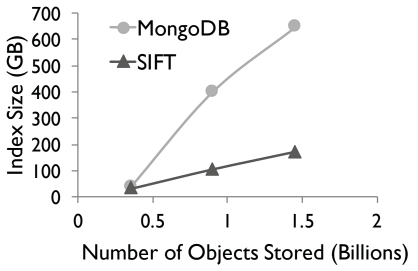
## 5.4 Cluster Performance

Sift was designed to be used in a distributed setting. While it performs well in single machine environments, the real test for its data structure lies in a cluster environment. We do not assume the data from connected devices to be able to fit in a single machine. Even otherwise, for performance reasons, it might be desirable to deploy the data store in a clustered environment. To understand the performance of Sift in a cluster setting, we conducted experiments that focuses on evaluating its design choices. For these experiments, we used the taxi trip dataset and cellular dataset. Since the taxi dataset contains the trip information metadata, we use the trip path as the polygon object. In the cellular dataset, we use the location updates from the user to create a similar path. In addition, we also used trip distance and network usage time as additional dimensions, in the taxi and the cellular datasets, respectively. To emulate real deployment scenarios, we load both datasets simultaneously.

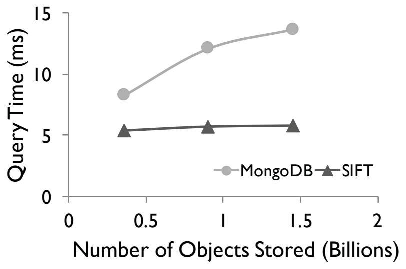
We prime the data store by warming it up using a test dataset. Then we insert the entire dataset into the system in a streaming fashion. At fixed intervals during the ingestion, we stop the process and run a query job that launches 100,000 queries. These queries consist of simple containment and retrievals, due to the size of dataset, and



(a) Indexing time

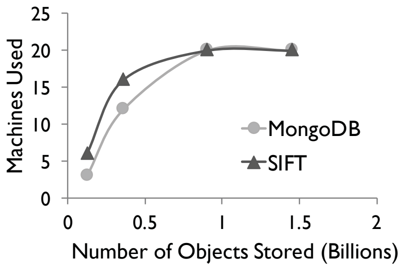


(b) Index size

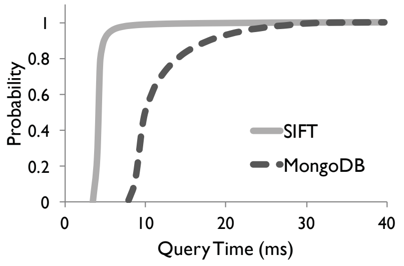


(c) Query time

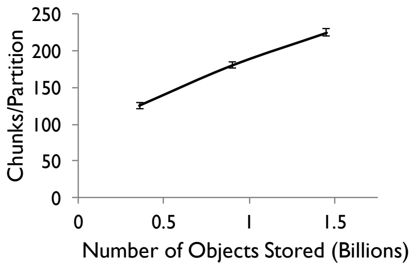
**Figure 11: SIFT reduces indexing time per object by 2×-8× while reducing the index size by upto 3X in cluster environments. These benefits do not come at the cost of queries. In contrast, SIFT reduces per query cost by 2X or more.**



(a) Number of machines used



(b) CDF of query latency



(c) Load imbalance

**Figure 12: The number of machines used does impact SIFT's advantages, the more machines in the cluster, the better it performs. This is because SIFT can handle skews effectively.**

return at least one result. Once the queries complete, we resume the ingestion. We present several results from this experiment:

***Indexing Performance:*** Does SIFT carry over its single machine performance in the clustered environment? Figure 11 presents the answer to this question. The plot depicts the time required for ingesting a single data item as the number of objects stored in the database increases. Both SIFT and MongoDB incur an increase in indexing overhead as the data store size grows. The reason for this behavior can be attributed to the use of a B-tree based index in each partition in MongoDB. The speed up for SIFT ranges from 2X to 8X, with the benefits increasing with data store size.

The index size also shows a similar trend, with SIFT reducing the index size by upto 3×. This came as a surprise to us, because the single server results never provided us with improvements over 40%. Upon further investigation, the reason became clear: since MongoDB partitioned the dataset using object ID, nearby objects end up in different partitions. Since each partition maintains its own index, many of the keys get duplicated multiple times, thus blowing up the total index size significantly.

***Query Performance:*** We now turn our attention to query performance depicted in fig. 12. While MongoDB's query cost increases with increase in the number of objects, SIFT sustains an almost constant latency per query. The performance difference is more than 2×. At the beginning, we do not see much improvement to SIFT. The reason for this behavior can be explained by examining the number of machines used to answer the queries. In fig. 12a, we see that at the start of the experiment, MongoDB used only 3 machines, while SIFT utilized 6. As MongoDB uses hash partitioning on object ID, it has to reach all the partitions to answer a query. But this cost is small when the number of machines is small. Hence, the query latency for MongoDB almost matches that of SIFT. As the data size grows, more and more machines share the load and thus the overhead of querying increases for MongoDB. At around 1 billion objects, both schemes utilize the entire cluster, and at this point the query latency of MongoDB plateaus. In contrast, SIFT sustains the same latency regardless of whether it uses 2 machines or 20. This indicates the effectiveness of the partitioning technique in SIFT.

***Skew Tolerance:*** Finally, we evaluate the skew tolerance of SIFT. One of the goals of SIFT is to be resistant to heavy skews without incurring frequent rebalancing. The basic unit of balancing in MongoDB is a chunk, which is a collection of records. When the number of chunks become too large in a partition compared to others, the rebalancer kicks in. For this evaluation, we modify rebalancer to avoid running unless for heavy imbalances. Then we repeat the ingestion experiment. At every fixed interval, we check the average number of chunks per machine that was used for storage and the variance across machines. Figure 12c shows the results. The variance is extremely small, which indicates that at no point does SIFT incur an imbalance of more than a couple of chunks per machine, thus validating the design of SIFT.

## 6 RELATED WORK

Our work in this paper is related to several areas in the literature as we discuss in this section.

### 6.1 Traditional Spatial Indexes

In general, existing proposals on indexing multi-dimensional objects can be classified into two categories:

**Data partitioning based Indexes:** These indexes organize objects in a tree-like structure. The most common approach is to use the R-tree [24] or its variants such as the R*-tree [8]. The basic idea here is to bind an object in a Minimum Bounding Rectangle (MBR). The MBRs are then organized hierarchically resulting in a tree structure, making sure each bottom level MBRs contain no more than a given number of objects. These indexes are susceptible to data skew, with many MBRs in crowded areas. Examples of R-tree based indexes include the TPR-tree [50] and its variant the TPR*-tree [44]. Although these indexes are efficient for querying, their performance suffer when the indexed objects are highly mobile.

**Space partitioning based Indexes:** This category of indexing relies on partitioning space rather than objects. Proposals in this category are based primarily on Quadtrees [20], $B^+$-trees [28] or grids [16]. Quadtree based approaches build a hierarchical decomposition of space to index objects. Search is then done by representing the decomposition as a tree. However, they suffer from performance degradation under high ingestion rate due to the overhead of splitting and merging. The other two approaches avoid this by partitioning the space under consideration in advance. Each object is then indexed in the grid cell it belongs to. Efficient lookup of cells are done with the help of space-filling curves that gives a unique one dimensional id for each cell. This category of indexing has gained attraction because they result in very efficient CRUD operations. In addition, they can be easily parallelized since operations do not require locking. However, these indexing techniques were designed for simple point spatial types which is not the focus of this paper. A few adaptations of Quadtrees exist for accommodating polygon data [41]. [43] proposes enclosing objects in the first cell they intersect which would lead to many small objects getting assigned to very large areas. None of these proposals are intended at skew mitigation or distributed load balancing.

### 6.2 Workload Aware Indexing

A few recent work has addressed the issue of data skew in spatial indexes. Consequently, techniques have been proposed to modify the indexing based on the workload. Zhang et. al [52] propose $P^+$-tree to handle queries in diverse data distribution settings by partitioning the space into subspaces and applying indexing independently on each of the subspaces. However, they assume stationary objects. [48] proposes QU-Trade, a technique that explores the trade-off between query and updates to achieve adaptability when the workload changes. Finally, $ST^2B$-tree [12] proposes a self-tunable $B^x$-tree index by adapting grid cell sizes in response to workload. However, these techniques do not assume a distributed environment and thus do not provide solutions for load balancing queries and updates across different subspaces.

### 6.3 Distributed Spatial Indexes

Relatively few work has been done in distributed spatial indexing. MOIST [29] builds a recursive spatial partitioning index on BigTable. To reduce the number of updates, they employ school tracking that only tracks leaders and sheds updates from followers. Similarly, MD-HBase [36] builds a spatial index on HBase that can sustain high update rates while delivering query performance. However, none of these proposals consider data skew or workload variation over time. Proposals for a distributed version of RTree exists [17], but it is unclear how well it will scale in an environment with unpredictable skew. Moreover, their technique relies on post-ingestion load balancing. Recent literature has proposed several distributed B-tree based solutions (e.g., [32]). However, these are not designed for spatial data.

### 6.4 Spatial Indexing for Big Data

Recent proposals have identified geo spatial capabilities as an important feature in big data processing frameworks [2, 18]. However, they are not geared towards online indexing and hence assume the ability to partition the data offline. The main goal of these systems is to provide analysis capabilities rather than querying. Thus, they are orthogonal to our work and could benefit from it.

## 7 CONCLUSION

In this paper, we presented SIFT, a scalable, distributed spatial index that has skew resistance as its primary goal. SIFT can efficiently ingest, index and query spatial data with unpredictable skews. We showed that using additional dimensions, derived from characteristics of spatial data, for object distribution can help mitigate spatial skews and provide load balancing with minimal rebalancing requirements. SIFT can easily be incorporated into existing distributed key value stores. We implemented SIFT on MongoDB, a popular open-source big data store and evaluated it on real-world data sets. Our evaluation showed that SIFT is able to achieve up to 8× reduction in indexing time compared to the existing state-of-the-art while simultaneously reducing the query latency by over 2× and index size by over 3× respectively, in distributed environments.

## A BOUNDS ON OBJECT DUPLICATION

***Property:*** *The upper bound on the number of nodes storing the same geometry in the data structure is* $2^n$ *for an n-dimensional index.*

**Explanation:** Due to the use of binary space partitioning, any parent node in the data structure contains atmost $2^n$ child nodes. The indexing strategy assures that an object is placed in the first level in the hierarchy where the level's children have subspaces that cannot entirely enclose the object. Imagine that an object is present in $2^n + 1$ nodes. This is possible only if it intersects $2^n + 1$ node's subspaces in the level it is in. In such case, the bounding box of the object has to be larger than that of the nodes in the level. Thus, it cannot be indexed in level $l$ and should be indexed in its parent where it would be indexed in atleast 1 less node.

## ACKNOWLEDGMENTS

We would like to thank the SoCC reviewers and our shepherd, Siddhartha Sen, for their valuable feedback. This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, and gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

## REFERENCES

[1] ABIResearch. 2013. https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne. (2013).

[2] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1009–1020. https://doi.org/10.14778/2536222.2536227

[3] Amazon. 2017. EC2. (2017). http://aws.amazon.com/ec2/

[4] Apache Cassandra. 2017. http://cassandra.apache.org/. (2017).

[5] Apache CouchDB. 2017. http://couchdb.apache.org/. (2017).

[6] Apache HBase. 2017. http://hbase.apache.org/. (2017).

[7] Apache Solr. 2017. http://lucene.apache.org/solr/. (2017).

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. ACM, New York, NY, USA, 322–331. https://doi.org/10.1145/93597.98741

[9] US Census Bureau. 2017. TIGER. http://www.census.gov/geo/maps-data/data/tiger.html. (2017).

[10] National Climatic Data Center. Doppler Radar Data. http://www.ncdc.noaa.gov/data-access/radar-data. (????).

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1267308.1267323

[12] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A Nascimento. 2008. ST 2 B-tree: a self-tunable spatio-temporal b+-tree index for moving objects. In *ACM SIGMOD*. ACM, 29–42.

[13] DBEngines. 2017. Database Engine Rankings by Popularity. http://db-engines.com/en/ranking. (2017).

[14] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[16] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. 2009. Indexing Moving Objects Using Short-Lived Throwaway Indexes. In *Proceedings of the 11th International Symposium on Advances in Spatial and Temporal Databases (SSTD '09)*. Springer-Verlag, Berlin, Heidelberg, 189–207. https://doi.org/10.1007/978-3-642-02982-0_14

[17] Cédric Du Mouza, Witold Litwin, and Philippe Rigaux. 2009. Large-scale Indexing of Spatial Data in Distributed Repositories: The SD-Rtree. *The VLDB Journal* 18, 4 (Aug. 2009), 933–958. https://doi.org/10.1007/s00778-009-0135-4

[18] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce Framework for Spatial Data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. 1352–1363. https://doi.org/10.1109/ICDE.2015.7113382

[19] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, Berkeley, CA, USA, 371–384. http://dl.acm.org/citation.cfm?id=2482626.2482662

[20] R.A. Finkel and J.L. Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta Informatica* 4, 1 (1974), 1–9. https://doi.org/10.1007/BF00288933

[21] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. 2008. *Database Systems: The Complete Book*. Prentice Hall.

[22] Gartner. 2013. Gartner on the Internet-of-Things. http://www.gartner.com/newsroom/id/2636073. (2013).

[23] Google. 2016. S2 Geometry Library. https://code.google.com/p/s2-geometry-library/. (2016).

[24] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. ACM, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[25] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 562–573. http://dl.acm.org/citation.cfm?id=645921.673145

[26] D Hilbert. 1981. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.* (1981).

[27] Anand Iyer, Li Erran Li, and Ion Stoica. 2015. CellIQ : Real-Time Cellular Network Analytics at Scale. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 309–322. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/iyer

[28] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. 2004. Query and Update Efficient B+-tree Based Indexing of Moving Objects. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 768–779. http://dl.acm.org/citation.cfm?id=1316689.1316756

[29] Junchen Jiang, Hongji Bao, Edward Y. Chang, and Yuqian Li. 2012. MOIST: A Scalable and Parallel Moving Object Indexer with School Tracking. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1838–1849. http://dl.acm.org/citation.cfm?id=2367502.2367522

[30] David R. Karger and Matthias Ruhl. 2004. Simple Efficient Load Balancing Algorithms for Peer-to-peer Systems. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04)*. ACM, New York, NY, USA, 36–43. https://doi.org/10.1145/1007912.1007919

[31] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[32] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 451–464. https://www.usenix.org/conference/atc16/technical-sessions/presentation/mitchell

[33] MongoDB. 2017. http://www.mongodb.org/. (2017).

[34] MongoDB. 2017. Sharded Cluster Guide. http://docs.mongodb.org/manual/tutorial/deploy-shard-cluster/. (2017).

[35] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* 9, 1 (March 1984), 38–71. https://doi.org/10.1145/348.318586

[36] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *Proceedings of the 2011 IEEE 12th International Conference on Mobile Data Management - Volume 01 (MDM '11)*. IEEE Computer Society, Washington, DC, USA, 7–16. https://doi.org/10.1109/MDM.2011.41

[37] OpenStreetMap. 2017. http://www.openstreetmap.org/. (2017).

[38] G Peano. 1890. Sur une courbe, qui remplit toute une aire plane. *Math. Ann.* (1890).

[39] PostGIS. 2017. http://postgis.net/. (2017).

[40] PostgreSQL. 2017. http://www.postgresql.org/. (2017).

[41] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (June 1984), 187–260. https://doi.org/10.1145/356924.356930

[42] Hanan Samet. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[43] Kenneth C. Sevcik and Nick Koudas. 1996. Filter Trees for Managing Spatial Data over a Range of Size Granularities. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 16–27. http://dl.acm.org/citation.cfm?id=645922.673466

[44] Yufei Tao, Dimitris Papadias, and Jimeng Sun. 2003. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *In VLDB*. 790–801.

[45] TLC. 2017. Trip Record Data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml. (2017).

[46] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[47] Twitter. 2013. Tweet Record. https://blog.twitter.com/2013/new-tweets-per-second-record-and-how. (2013).

[48] Kostas Tzoumas, Man Lung Yiu, and Christian S. Jensen. 2009. Workload-aware Indexing of Continuously Moving Objects. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 1186–1197. http://dl.acm.org/citation.cfm?id=1687627.1687761

[49] UMN. 2016. Real-world Spatial Datasets. http://spatialhadoop.cs.umn.edu/datasets.html. (2016).

[50] Simonas Šaltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. 2000. Indexing the Positions of Continuously Moving Objects. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 331–342. https://doi.org/10.1145/342009.335427

[51] Darius Šidlauskas, Simonas Šaltenis, and Christian S. Jensen. 2012. Parallel Main-memory Indexing for Moving-object Query and Update Workloads. In *ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2213836.2213842

[52] Rui Zhang, Beng Chin Ooi, and Kian-Lee Tan. 2004. Making the Pyramid Technique Robust to Query Types and Workloads. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*. IEEE Computer Society, Washington, DC, USA, 313–. http://dl.acm.org/citation.cfm?id=977401.978092