Spatial Data Locality in Scalable and Fault-tolerant Distributed Spatial Computing Systems

Martin Werner
Remote Sensing Technology Institute (IMF)
Deutsches Zentrum für Luft- und Raumfahrt, Oberpfaffenhofen
martin.werner@dlr.de

ABSTRACT

In the last decade, spatial datasets started to grow from small collections of high quality geospatial information into huge collections of data covering the whole planet with varying formats and qualities. Large-scale spatial datasets are about to create significant value in varying application fields including navigation, autonomous driving, urban geography, agriculture, and climate research. Therefore, large datasets are actively acquired. In addition, social networks such as Facebook, Twitter, and Flickr provide text, video, and images with associated geospatial information from the crowd. These sources are highly interesting as they provide near-realtime insights into aspects of human behavior and dynamics. Finally, global and long-running satellite missions such as Landsat, Sentinel, WorldView, or TerraSAR add large amounts of geospatial information. It is a matter of fact that these data collections are putting challenges to the computational infrastructure used for spatial computing. Not only do we need a lot of computation, we also need to think about how to organize and design distributed systems that can help tackle the volume, velocity, and variety of current and future geospatial datasets. Modern big data systems employ data replication for two main reasons: first, for increased fault tolerance, and, second, for higher flexibility in scheduling tasks across a large cluster of machines. This paper proposes and compares novel data replication schemata for scalable spatial computing and analyzes the impact on the communication complexity of global spatial joins of a large collection of tweets collected from the Twitter API and building polygons extracted from OpenStreetMap.

CCS CONCEPTS

• **Information systems** → **Key-value stores**; *Data replication tools*;

KEYWORDS

Spatial Big Data, Data Replication and Distribution; Spatial Join

ACM Reference Format:
Martin Werner. 2018. Spatial Data Locality in Scalable and Fault-tolerant Distributed Spatial Computing Systems. In *7th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial 2018), November 6, 2018, Seattle, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3282834.3282837

BigSpatial 2018, November 6, 2018, Seattle, WA, USA

ACM ISBN 978-1-4503-6041-8/18/11…$15.00
https://doi.org/10.1145/3282834.3282837

1 INTRODUCTION

In the last decade, more and more spatial data is generated every day. In addition, many services scale into serving all users across a global digital world. Therefore, not only large datasets are generated, but also large spatial areas are covered. This trend is reflected in the increasing interest of exploiting such data in research and business. For example, in the context of disaster management, the timeliness of social media is convincing [1, 11], for urban mobility, the trace information from social media network check-ins reveals interesting patterns [10]. There are many more such examples, but especially due to the current satellite missions, research starts focusing on global patterns. For example, the DLR has published the Global Urban Footprint (GUF) [6], which represents the whole world as a raster image in which each pixel denotes built-up or non-built-up area.

This trend poses severe challenges to spatial computing infrastructures. In general, parallelization of data and tasks is a simple yet effective means of scaling earth observation and spatial computing to global coverage. However, there are situations in which the data is not yet spatially condensed. For example, when collecting social media messages, they arrive in a temporal ordering, but their spatial component can be seen as a random sample from an underlying distribution in which cities are hotspots and rural areas emit less tweets. In this situation, the parallelization of operations relating two datasets becomes challenging, because they both are larger than the main memory that is practically accessible. Therefore, distributed computing systems have been extended to deal with the special properties of spatial data [2, 4, 12].

In general, big data systems rely on a mechanism of replication for two reasons: first, fault tolerance, because a failing node does not lead to an application-level failure greatly simplifying administration and operation of large clusters, and, second, for decreasing the communication between different physical nodes in a cluster, because this is often the performance-limiting factor.

There are some standard and basic data distribution schemes that have been applied by the previously mentioned systems including block-based distribution following grids or R*-trees and ordering-based distribution using space filling curves.

However, to the best of our knowledge, mechanisms for spatial data replication have not been studied in this context. Instead, systems often rely on an underlying storage layer such as HDFS for replication and fault-tolerance not taking into account the opportunities of the storage redundancy for later query processing.

With this paper, we want to address the question and propose a viable solution for the following problem:

Problem 1. *Given a large cluster of computers, how can we structure data distribution across nodes such that we obtain a fault-tolerant system in which the replicated data is still useful for processing geographical analysis?*

We propose a fixed distribution scheme for primary data and propose several strategies on where to place replica in order to increase data locality. The overall performance is measured by performing a global spatial nearest neighbor join. Concretely, we assign a large dataset of tweets each to the nearest building in order to analyze the buildings in terms of the social media messages from their vicinity.

It is worth noting that the proposed mechanisms are generic in the sense that they can be integrated into any distributed computing system. However, the gain of these approaches depends on the interplay between the storage engine and the query processing engine, which is sometimes easier and sometimes harder to adapt to increasing flexibility. However, we are not proposing a novel system, but rather a novel algorithm for data replication in fault-tolerant distributed computing systems.

We show that the basic strategy of using a ring-structure together with a Z curve works quite well, but propose two other strategies optimized for different aspects: one strategy is able to reduce the number of incomplete results that need to be shuffled across the cluster to the nodes having the missing data and another one is optimized for data locality generating significantly more refinement need, but each one with a smaller coverage and, therefore, easier to solve.

The remainder of the paper is structured as follows: in the next Section 2, we introduce terms related to spatial big data and related work needed to set the right context for this paper. In Section 3, we discuss some properties of a widely studied space-filling curve known as Z curve or Morton order. These observations build the intuition for the constructed replication schemata. In Section 4, we introduce three important queries that jointly come up in the example workload of a global spatial join: nearest neighbor queries, range queries, and spatial join. Section 5 introduces novel replication strategies, how the replicas of a spatial data object are distributed. In Section 6, we evaluate the various strategies using a realistic workload of joining tweets and buildings. Section 7 draws conclusions and gives hints on future research.

2 RELATED WORK

Spatial computing with big data has received an increasing attention over the last years. Basically, it is the intersection of two active fields of research: spatial algorithms and big data.

2.1 Big Data

Big Data has ever been a very coarse and evolving term and is not clearly defined. However, a widely adopted characterization of Big Data has been given by Laney [8] in 2011. He proposed the three dimensions *Volume*, *Variety*, and *Velocity* (3V). Of course, these three aspects somehow describe the term big data in the sense of fixing dimensions for a characterization. However, there is still and active and ongoing discussion on which aspects are most important and which characterizations of big data are better [9, 14].

The growth in volume and velocity, as well as the complexities induced through variety, lead to situations in which classical database management systems cannot efficiently be used to support data analysis.

Given the 3V definition of big data, some coarse conclusions can be drawn:

- The sheer volume of data implies that the data needs to be stored in some sort of a distributed system.
- The velocity of the data implies that maintaining consistency and timeliness becomes prohibitively hard.
- The variety and complexity of the data renders preprocessing or optimized data structures largely unrealistic. Instead, processing must work with the data as is without much preprocessing

In this area, two schools of computing are challenging each other: one trend is to use cloud computing and elastic computing in which large amounts of cheap compute resources are orchestrated on demand to build a distributed system that can cope with the given challenges. Another way of dealing with these challenges is to maintain a high-performance computing system that can deal with the challenges.

As the first approach gives higher performance per dollar, it is a preferred way for Internet companies. The second approach is limited to research centers that can afford the extreme cost of buying and operating supercomputers. However, these architectures, in general, can offer better peak performance as compared to cloud infrastructures. And, in addition, they can be cheaper in practice for many researchers, as access to these clusters is often possible for research without payments. That is, for an average scientist, the question, which paradigm to follow, is extremely difficult to answer: while much open source is organized around the cloud computing paradigm, it is not easy for the researchers to access cloud computing resources with more than a few hundred CPUs. However, access to decent supercomputers is available for most universities without cost for the individual researcher including access to thousands of CPUs per job.

2.2 Geospatial Big Data

In geospatial big data, we are facing the question on how to distribute spatial data as well as queries over a large number of computers. This is true for both cloud computing as well as for high-performance computing scenarios. The distribution of data will enable us to split global spatial computing problems into local instances that can be solved by a single computer or a few nodes given only the local data.

Data distribution is related to several other properties of a distributed big data system having significant impact on the overall performance and scalability: *data locality*, *fault tolerance*, *cluster utilization*, and *repair complexity* are some significant examples: First of all, we are interested in a way of distributing data across such that most computations can lead to good results with data that is only local to the machine. In this way, we can reduce network communication in the cluster, which is often a significant bottleneck. To the contrary, however, when data locality is employed,

one consequence is that queries to the data will be served according to this data locality, too. This essentially means that data locality pays off best if the distribution of data and queries is similar and will create significant hotspots and skewed cluster utilization if the distributions differ. Therefore, some big data architectures including Apache Cassandra have adopted a random data distribution by default. This leads to a uniform allocation of resources to randomly incoming data and queries and is clearly the best stateless strategy, when there is no information about the involved distribution of queries and data.

In contrast to the general case, however, we have some principles of geography that vote against a random distribution of spatial data across a cluster. The most basic principle is known as Tobler's first law of geography: "Everything is related to everything else, but near things are more related than distant things" [15]. This translates into the expectation that any query workload will usually have a spatial component making it local increasing the usefulness of data locality and at the same time increasing the risk of hotspots and skewed cluster utilization for queries.

An additional twist on data distribution has been introduced lately by insisting on fault-tolerance of distributed systems. When using cheap off-the-shelf hardware and elastic compute services, we do not expect that compute resources are available from the beginning to the end. Instead, we will lose some of our computers by errors from hard- and software as well as by scaling operations. Such architectures will only work efficiently, if all data is distributed to different locations in the cluster enforcing a sufficient level of replication. In this context, the mapping of data across machines will map each block of data to a set of nodes. This set of nodes can be guided by data locality, but it needs to fulfill additional constraints such that different racks or datacenters are employed.

With this paper, we want to study data distribution schemes of geospatial data in large distributed computing systems. Though fault tolerance might not (yet) be needed in HPC supercomputers as the underlying architectures are largely based on the assumption that there are no faults during a computation, it can still increase performance significantly for spatial workloads by increasing the spatial knowledge available at each node.

For spatial data distribution, we can distinguish two general approaches: *block-based distribution* in which a (logically) central entity takes care of assigning blocks to nodes and managing the replication and *ring-based architectures* in which an ordering of data assigns data to a primary node and replication is managed by this node.

The *block-based distribution* is applied, for example, in Apache HDFS and Google GFS and provide the underlying storage architecture of traditional big data systems such as Apache Hadoop, Apache Cassandra, Apache HBase and others. Blocks form an atomic unit of data management and data access. As these blocks are rather large (i.e., Apache Hadoop propagates a default value of 64 MB), the management overhead is small enough even in large clusters of machines though theoretically bounded by the ability of a central component to manage these blocks and their distribution.

This approach has been extended to spatial data by assigning blocks to nodes from a central entity using various spatial indexing structures. In SpatialHadoop, for example, a master node maintains a global spatial index used to map data and requests to nodes while each node employs a local spatial index to organize and speed up queries with respect to the local data [5]. The underlying indices are a grid index, an R-tree and an R+-tree. This framework is extended by adopting the Z-curve and the Hilbert curve as additional indices for Spatial Join workloads in [4]. In a similar fashion, SparkGIS uses grids, binary space paritioning trees, strips, Sort-Tile-Recursive (STR), and the Hilbert curve for distribution of spatial data and queries.

In contrast to that, the ring-based architectures have been proposed in which all nodes are ordered in a ring-based structure and replica move along the ring to the next node that is possible given the constraints (that is, usually the next k nodes that jointly fulfill the constraints). This setup, in addition to that operations like inserting or removing a node trigger mostly ring-local communication also provides a good basis for data locality in range queries.

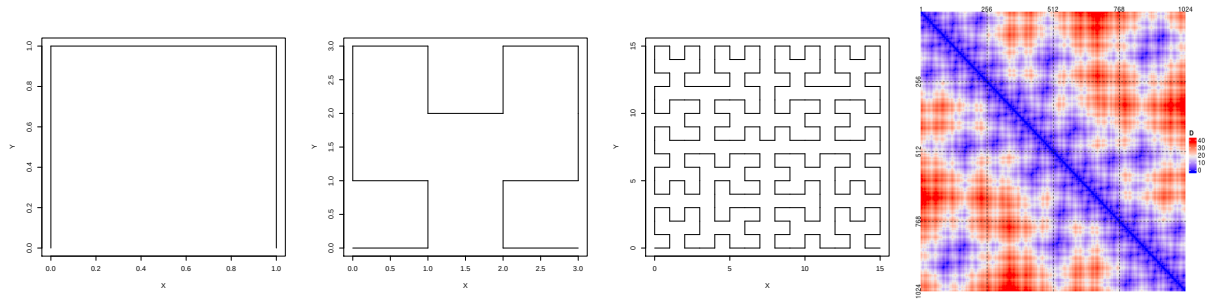
For classical data, Apache Cassandra is a typical reference [7]. Apache Cassandra organizes nodes into a ring by dividing a key space into intervals called *Tokens*. A data object in Cassandra is first assigned a key, either directly or through random hashing. In addition, each ring node owns a token, which represents the range of keys that should go to this node. This node is then called the *primary node* for the data and is responsible for managing operations with respect to this data including replication. Replication is then, basically, performed along the ring. The primary node forwards data objects for replication along the ring and the next node that meets the given set of criteria (e.g., different physical node, rack, or data center) is going to store a copy. If a node fails, the neighboring nodes just extend their tokens to take over the range. An additional twist is to decouple the physical cluster from the tokens by introducing virtual nodes (vnodes) [17]. In this case, each physical node is split into a set of vnodes that each own a token. In random token distribution, this reduces repair time and increases flexibility, when the capacity, performance, or workload are skewed: you can move vnodes across the ring reducing hot spots.

3 SPATIAL REPLICATION

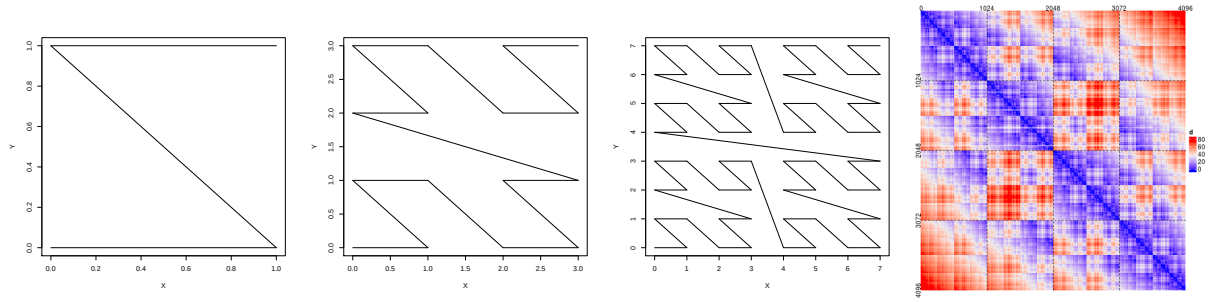
Though advanced distributed systems have been extended for spatial data handling including SpatialHadoop [5], HadoopGIS [2], SparkGIS [3], MD-HBase [12], and others, the general question of spatial data replication has not been discussed in detail. In addition, the raw scalability of these systems has not been evaluated as the current spatial datasets are either not so large in footprint consisting of many very small objects (e.g., points, polygons, etc.) or rather few, very huge objects such as satellite imagery or spatial rasters.

With this paper, we want to study the impact of data replication strategies on workloads such as spatial joins of global datasets of small objects.

We employ a ring-based architecture for distributing the data across the nodes. In this way, we reach a linearly scalable system without the need for a central component. Like in the Apache Cassandra architecture, each node in the cluster can - at any time - compute the *primary nodes* for a given spatial data item. The only prerequisite is to maintain a globally consistent view of the nodes and their respective tokens. In fact, we even don't need to maintain



(a) Hilbert Curve ($k = 1$) (b) Hilbert Curve ($k = 2$) (c) Hilbert Curve ($k = 4$) (d) Locality of the Hilbert Curve



(e) Z Curve ($k = 1$) (f) Z Curve ($k = 2$) (g) Z Curve ($k = 3$) (h) Locality of the Z-Curve

Figure 1: Two widely used discrete space-filling curves for enumerating grid cells preserving locality.

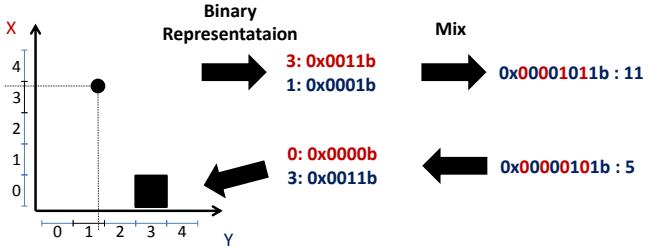


Figure 2: The Z curve encoding and decoding process

this state in all circumstances, because if we detect that a primary node is not responding, we can alternatively contact one of the replica of this node according to the strategy.

Space-filling curves are a perfect match for this ring-based architecture. The two most promising space-filling curves for ordering spatial data are the Hilbert curve and the Z curve depicted in Figure 1. Figure 1(d) illustrates the data locality of the Hilbert curve. Both axes range over all keys and the color indicates the spatial distance of the cells. The triangular structure indicates that pairs of similar keys have a small spatial distance. Figure 1(h) depicts the same for the Z curve and though it is more fractal and - in general - shows higher values, it is still clearly providing good data locality with small values nearer to the diagonal.

The advantage of the Z curve is that it is very efficient to calculate in both directions: from a point into a curve index and from a curve index into the cell.

Figure 2 depicts the process of encoding and decoding in the case of the Z curve for two dimensions. In general, The Z curve in n-dimensional space is constructed as follows: Given a point $p = (x_1 \dots x_n)$, each of the coordinates is discretized according to a previously chosen discretization of the axis and stored as an unsigned integer. The discretized point, therefore, has unsigned integer coordinates $p_d = (d_1 \dots d_n)$ where each d_i is the result of a discretization of the range of x_i. The integer curve rank value is then constructed by mixing and interleaving the bits from these unsigned integer representations.

Examining the properties of the Z curve in more detail, it is easier to study the curve by characterizing the connections of a cell p = (x,y) and the cell of its successor $\phi^{-1}(\phi(x) + 1)$

When the number $\phi(x)$ is an even integer number, its binary representation ends with a zero. Hence, only one bit changes for the increment, namely the last bit. With respect to our ordering of the mixture of bits, this amounts to a horizontal connection of the two neighboring cells (x+1,y). In all other cases, a diagonal edge is constructed. If the number is non-even, the length of the diagonal depends on the first bit location, where the overflow from the increment operation is absorbed by a zero, because this amounts to the number of bit changes for the mixture and, therefore, also defines the number of bit changes for the two coordinates in $\phi^{-1}(x)$. Assuming that we are using a complete, finite ring of all non-negative integer numbers from $0 \dots 2^{l-1}$, the absorbtion of the overflow bit will take place in the first position from the end, where there is a zero. Looking at all bit sequences of uneven numbers, we see that the absorption takes place in place j in 2^{-j} cases.

Bit Sequence	Incremented	Offset (x,y)	L_1	Frequency
*******0	*******1	(0,+1)	1	128
******01	******10	(-1,+1)	2	64
*****011	*****100	(+1,-1)	2	32
****0111	****1000	(-3,+1)	4	16
***01111	***10000	(+1,-3)	4	8
**011111	**100000	(-7, +1)	8	4
*0111111	*1000000	(+1,-7)	8	2
01111111	10000000	(-15,+1)	16	1
11111111	00000000	(-15,-15)	30	1

Table 1: Bit Patterns and their Connection in a regular Z-curve (8 bit)

Table 1 lists a concrete example of all numbers with 8 bits (that is 4 bit for X and 4 bit for Y). In this table, an asterisk (*) denotes a bit with unspecified value that does not change between the columns. The second column shows the increment of all bit sequences of this pattern - all asterisk bits are unchanged between the columns. The specified part of the bit sequence already defines the offset in X and Y for ϕ^{-1} given in column 3. In addition, the L_1 distance is shown. In general, depending on whether the overflow absorption takes place in bits related to X and Y, the operation on X and Y is increment (e.g., +1). The effect on the other coordinate is flipping a full 1 sequence of a specific length to zero (e.g., 0b111 is mapped to 0b000 related to a negative of $2^l - 1$). In the last line, a special situation of overflow occurs.

This perspective on the Z-curve helps us to understand, how far the cells associated with neighboring keys are away within the spatial domain. In fact, most increments lead to small movements in the space. Longer movements occur if and only if longer parts of the binary value flip during increment. Another aspect of the Z-curve can easily be seen from the exposition of the last paragraphs and Table 1: During successive increments (e.g., along the order induced by the Z-curve) the movements of L_1-length $l >= 2^k$ are spaced equally across the index space. This is due to the fact that the number of increments between two events of the same bit mask is 2^l, where l is the number of ones that flipped to zero in the event.

4 QUERIES IN SYSTEMS WITH SPATIAL DATA LOCALITY

In this work, we are going to optimize data locality for specific types of queries that often occur in spatial and spatio-temporal database systems. In this context, we are assigning a contiguous range of keys under the mapping ϕ to each of these nodes. These nodes are called *primary nodes* for the specific key range. Data with this key or queries with this key are – at first – mapped to these nodes.

However, due to replication, the same data exists in different places and the nodes holding such replicated data are called *replica*. These replica are usually put into different location in order to solve the problem that a primary node might fail. In this case, one of the replica first takes over the role of the *primary node* for this, answers queries related to this, and supports the repair operations in which the data distribution among nodes is being changed.

This data locality is, however, not only useful for introducing fault-tolerance. Instead, it can be used to reduce communication in typical queries, where several different primary nodes would need to contribute. Three important spatial operations are considered in this work: *range queries*, *nearest neighbor queries*, and *spatial joins*.

4.1 Range Queries

In a range query, all data items within a specific connected region are retrieved. Typically, these connected regions have a shape fitting to the indexing structure being used. Typical choices are disks, that is, a range query is specified with a location and a distance threshold, and rectangles, that is, a range query retrieves all data within a specific rectangle specified with the point of minimal coordinates and the point of maximal coordinates of the rectangle.

4.2 kNN Queries

In a k nearest neighbor (kNN) query, a certain number of k spatial objects are to be retrieved such that no spatial object is nearer to the query location than these k objects. This query typically involves enumerating a superset of the k nearest neighbors based on some criteria using a spatial indexing structure and filter linearily to the set of k nearest neighbors. It is worth noting that the kNN query can be structured by finding k examples, for example, in the local data at the primary node and then performing a refinement operation essentially based on a range query to ensure that the local result is globally correct or is being refined.

4.3 Spatial Joins

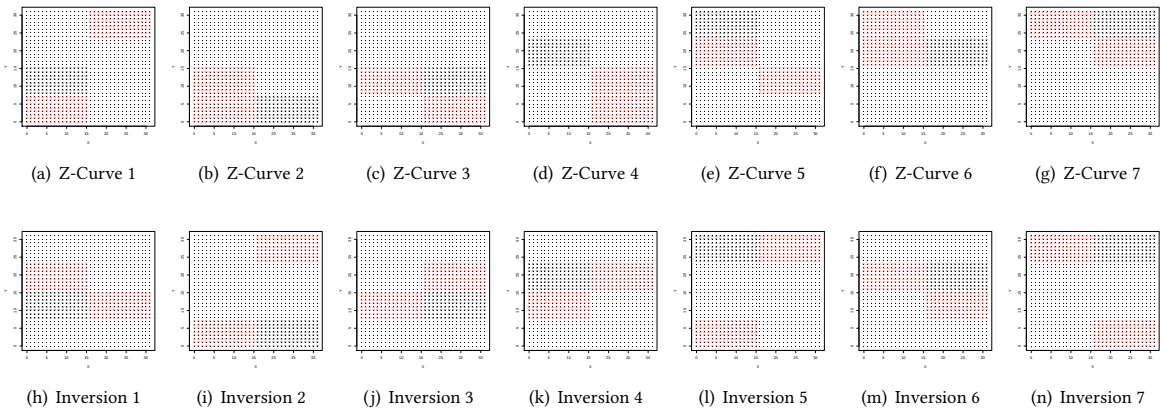
Conceptually, spatial joins are relating multiple larger datasets according to either containment (similar to range queries) or neighborship relations (similar to kNN). We will evaluate our system with a global spatial nearest neighbor join assigning points to the nearest building across the world. Similar to kNN queries, the usual distributed spatial join implementations perform first a local operation involving no cluster communication followed by a global refinement. A detailed discussion of spatial joins in MapReduce has been given by Sabek et al. [13].

5 REPLICATION STRATEGIES

With this paper, we want to discuss several replication strategies based on the Z curve exploiting the observations of the internal structure of the Z curve as explained in Section 2.2. More concretely, in this paper, a *data replication strategy* is a model providing the following functionalities:

- a function `key` mapping a spatial object into one or more integer keys,
- a function `endpoints` calculating the replicating nodes given the key, cluster size, and replication factor.
- a function `circle` calculating the set of keys originating from the set of all points inside a circle.

We will use this replication strategy for both: assigning data to nodes and assigning workload to nodes. In this way, we might face challenges with respect to hotspots when the query and data distributions are too different. Strategies for removing this assumptions



(a) Z-Curve 1 (b) Z-Curve 2 (c) Z-Curve 3 (d) Z-Curve 4 (e) Z-Curve 5 (f) Z-Curve 6 (g) Z-Curve 7

(h) Inversion 1 (i) Inversion 2 (j) Inversion 3 (k) Inversion 4 (l) Inversion 5 (m) Inversion 6 (n) Inversion 7

Figure 3: Data Distribution Example Using Two Replication Strategies Z Curve and Inversion

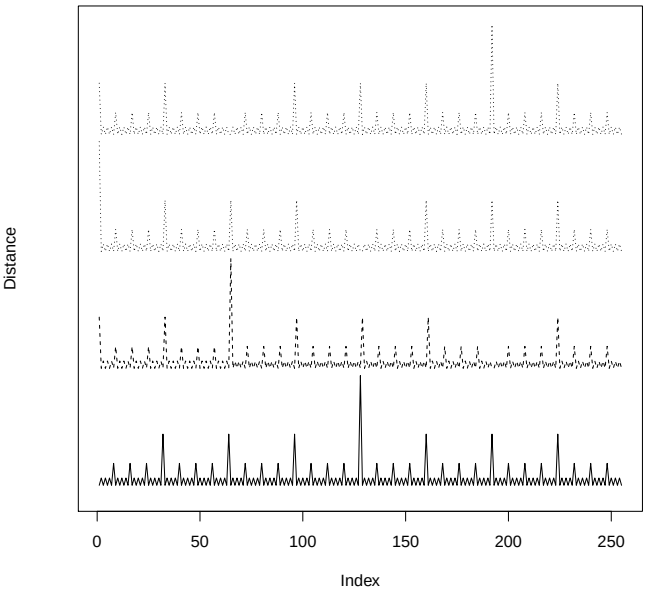


Figure 4: Rotation Strategy

are an interesting direction for further research, but beyond the scope of this paper.

5.1 Z-Curve Strategy

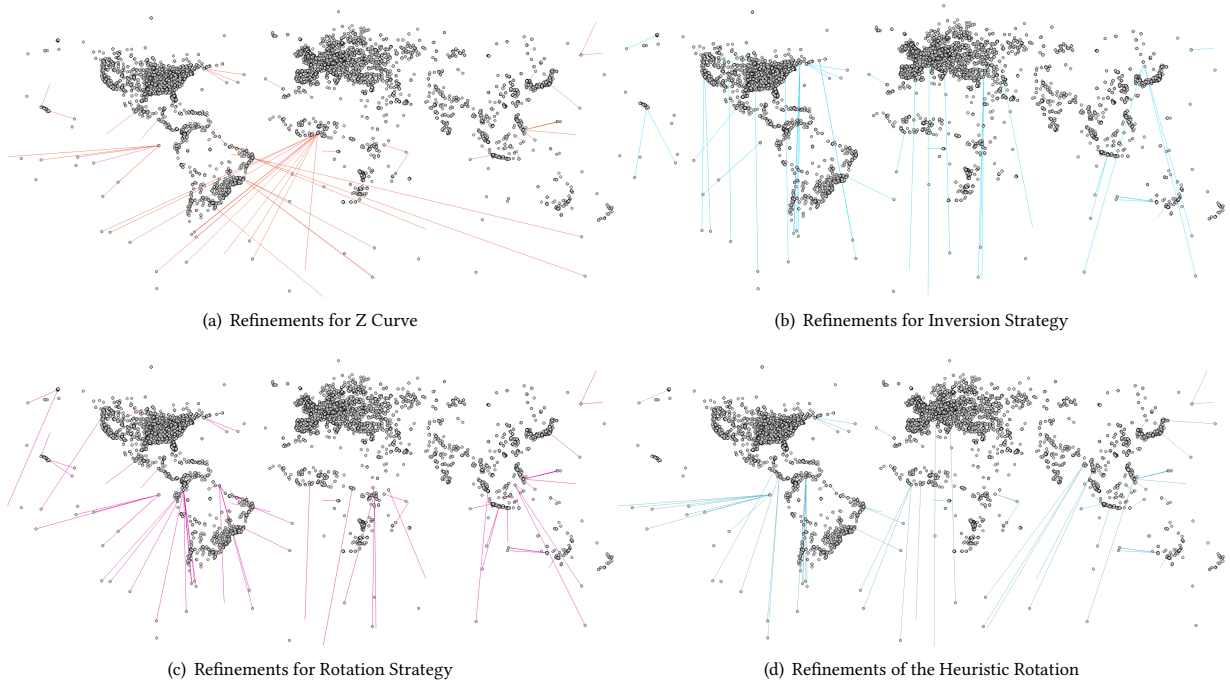
The most basic strategy modelled after Apache Cassandra is to use a Z curve for assigning spatial data to a node. Therefore, we first calculate the key in a chosen Z curve representation and assign the object to the node along the ring that owns this key. Replication is then performed along the ring wrapping over from the last node to the first. In the sequel, this strategy is called **zcurve**.

Figure 3 depicts a sample distribution using the Z curve strategy with 7 nodes. As you can see, the blocks follow a Z curve taking always three steps at a time. This gives good data locality, but not that much better than the original Z curve. The large jumps are for four and five just in the interesting middle of the data space.

5.2 Rotated Z-Curve Strategy

Given that a Z curve enumerates space through a space-filling curve, one simple approach to extending this to replication scenarios is by rotation. We calculate the Z curve index and add a well-chosen integer number to the key for each replica wrapping around the key space via a modulo operation. From the analysis of the Z curve in Table 1, we know that the number of long edges in the Z curve is only a few. Figure 4,depicts the length of the edges of the Z curve along the curve. Rotation can be used to move the large spike away from the center of the key space. This amounts to rotating with combinations of the maximal key value divided by a chosen power of two, e.g., max_key/2, max_key/4, essentially moving the largest spike as far as possible from the original curve in the key space. Note that especially adding or subtracting an additional one keep this effect largely same while also moving all of the very small spikes towards the very small local minima. This leads to two strategies, **rotation** in which we rotate always choosing to move the spike as far away from the spikes of the other replica as possible, and a strategy **rotation+** in which we alternatingly add and subtract +1 from the keys.

We performed a global search using a replication factor of three and eight nodes evaluating all possible rotations. For initial assessment of data locality, we scan the key space using points sampled from a uniform grid and count the number of neighboring cells that are missing at the primary replica of this node. Interestingly, it reports clear preferences: the minimum is achieved for configurations in which the first replica is rotated by 248 or 249 and the second replica between 384 and 499 in a space with 1024 cells. Though this is only loose information on data locality concentrating on only the directly neighboring cells, we derive the following additional rotation strategy: we choose rotation factors of $0.242 \cdot \text{max\_key}$ and $0.4311 \cdot \text{max\_key}$. The rationale of generalizing this to larger numbers of cells is given by the recursive self-similarity of the Z curve.



(a) Refinements for Z Curve

(b) Refinements for Inversion Strategy

(c) Refinements for Rotation Strategy

(d) Refinements of the Heuristic Rotation

Figure 5: Refinement Need in Various Redundant Data Distribution Schemes

5.3 Binary Inversion Strategy

From the observations of the Z curve, we can conclude that long edges occur if and only if many bits flip due to an increment after mixing. That is, when the key value changes from, for example $7 : 0x0111b$ to $8 : 0x1000b$, the cell coordinates change from $(3/1) : (0x11b/0x01b)$ to $(0/2) : (0x00b/0x10b)$ with a Manhattan distance of 4. We can use this to avoid flips of the curves of the replication by binary inverting one of the coordinates. That is, a novel familiy of replication keys is given by bitwise mixing (x, y), $(\tilde{x}, y)$ and $(x, \tilde{y})$, where $\tilde{x}$ denotes the binary inverse of the unsigned integer representation of x. This replication can be extended to higher degrees of replication by only inverting half of the bits, or a fourth of the bits, and so on.

Figure 3 depicts the inversion strategy for seven nodes. As you can see, we have good data locality when the underlying space is actually wrapping around such as WGS84 spherical coordinates.

While many other strategies can be designed, we think that these basic strategies stand out by observing that they do not need any costly operations or any shared knowledge except the size of the keyspace and the number of nodes in order to assign keys to nodes.

6 EVALUATION

In this section, we present the evaluation of the performance of the system with respect to a global spatial join of tweets and buildings.

The building data consists of 8,482,569 buildings extracted from OpenStreetMap. Each building is associated with its OSM ID, the set of tags including street name, usage, address information, and geometry given as a MULTIPOLYGON in well-known text format. The overall building object is formatted as a single-line JSON object and the dataset is composed of one JSON object per building. In addition, we employ the BZIP2 compression algorithm. The file is a 428 MB compressed file with an uncompressed footprint of more than 2.6 TB.

The Twitter data is given as a set of Twitter API objects, which are JSON objects with fields representing the tweet content, the user, as well as metadata such as language and location. The dataset has been collected from the public Twitter API and is stored as a set of BZIP2-compressed files with one JSON object per line. Its storage footprint is 359 GB which amounts to an uncompressed footprint of 3 TB. It contains 824,295,091 valid tweet objects.

For visualization purposes, we also extract datasets containing only 100,000 buildings randomly sampled from the large dataset and about 160,000 tweet objects randomly sampled from the large dataset.

6.1 Nearest Neighbor Join Workload

In order to get a good performance estimate of the various strategies, we perform the following workload on the cluster: We join all tweets with all buildings by a nearest neighbor left join. That is, the result contains for each tweet the pair of the tweet object and the building object of the nearest building.

Therefore, we perform the following distributed meta-algorithm:

- First, both datasets are distributed according to the same strategy in the cluster.
- Each node now joins all the tweets where the node is a primary node with all buildings that are locally known including those buildings that we only know from the replication mechanism.

- For each local result, the node calculates the keys of cells that are not known locally, but possibly needed to refine the result.
- This information is then used to greedily forward the data object to the primary node of the first missing key.

Some comments on the meta-algorithm need to be given: First, using the same distribution for both datasets might be suboptimal, but as we concentrate on human aspects of buildings and social media activity, we expect that these distributions are sufficiently related. Exploiting more knowledge on the data distribution of both datasets is promising, but left for future research. In addition, it would not be easy to understand effects of data locality, when data mobility during query execution is taken into account. The local spatial join is implemented using an in-memory R*-tree containing all locally-known buildings scanning over the primary tweets. This has two implications: first, the number of nodes as well as their main memory must be large enough to cope with indexing the fraction of the building dataset and, second, the dataset of tweets need not be sorted or indexed and can be of unbounded size. For the given workload, this is a plausible decision as the number of tweets grows much faster over time than the number of buildings and though we have a significant amount of buildings in the dataset, it is feasible to hold in memory even in systems with limited amount of main memory per core. For the refinement step, we chose that the primary node has to calculate the set of keys that are missing. We could instead have calculated the set of nodes that this should visit. However, this would limit envisioned extensions in which spare nodes that are not affected by the workload will actively contribute by observing statistics of cell misses. In addition, the refinement can be seen as a difficult optimization problem. Instead of forwarding to the first node that owns some missing data, we could actually forward to the node which knows most of the missing cells or to a node which knows a few of the cells, but has a low utilization. These aspects, however, raise the general question on how to consistently distribute metadata about the involved nodes (e.g., amount of data, number of tasks, set of known cells) in a very large cluster and go beyond the focus of this paper. In addition, this would directly break compatibility of the algorithm with any HDFS-based big data system in which at least eventual metadata consistency is guaranteed by central components (for example, JobTracker and NameNode in Apache Hadoop).

With the assumption that the data is distributed in the cluster before the workload arrives, the performance is basically bound to the refinement operations. The different strategies forward a different number of objects, each with a different number of cells, each time with a different minimum, average, and maximum distance to a missing cell and, in fact, it is impossible to find a single best replication strategy. For example, we could identify a strategy which forwards a lot of problems, which only involve a single second node. This can be faster in practice compared to a strategy in which forwards might hop around the cluster, thereby breaking parallelization and amplifying work.

For the presented strategies evaluated on the smaller sample datasets, Table 2 gives numbers related to various performance parameters of the strategies to illustrate that there are actually different optimal strategies depending on the type of data and cluster

		Cell Count		Cell Distance	
Strategy	**N**	**Avg.**	**Max.**	**Avg.**	**Max.**
`inversion`	**268**	26.82	297	13.53	134.76
`zcurve`	293	24.41	425	14.32	208.06
`rotation+1`	338	24.46	425	**14.59**	208.06
`rotation_heur`	397	14.51	272	11.04	131.62
`mixed`	398	9.70	**118**	10.01	94.03
`rotation`	432	**8.84**	148	9.07	94.03

Table 2: Number and Distance of Missing Cells in a Spatial Join of 180,000 tweets to 100,000 buildings with a Replication Factor of 3

performance metrics. As you can see by the boldly marked numbers, different strategies excel in different parameters. In general, all the approaches have very good data locality. This is to be expected as the primary node is assigned via a Z curve in all cases leading to very good general data locality even without data replication. With data replication, we see that, for example, the binary inversion strategy `inverse` is forwarding the overall fewest number of objects. When the limiting factor is cluster communication, then this is the best strategy. However, the classical rotation strategy `given` maximizing the distance of the largest spot forwards the highest count of objects for refinement. Each of those has, however, only a few cells missing. The strategy `mixed`, however, was providing the solution in which the maximal count of missing cells in the refinment step is minimal, that is the hardest refinement problem is still significantly smaller than the other problems. From a plain data locality perspective, we also look at the distance of the missing cells. In general, these distances are difficult to use for a ranking and are more of informational value: a large distance to a missing cell can occur due to two reasons: we found only very bad examples leading to a large range of cells for refinement or the only missing cells are far away.

From a more general point of view, the table allows three conclusions:

(1) The classical Z curve construction with replication to the neighboring nodes along the ring is quite powerful.
(2) The binary inversion strategy exceeds the classical construction in terms of minimizing the number of tasks that are to be reshuffled across the cluster.
(3) All strategies are valid as they present different tradeoffs between various positive and negative effects whose severity is tightly linked to the cluster architecture including interconnect speed, flexibility of task scheduling, etc.x

In practice, an evaluation with respect to each given actual query workload is inevitable for an optimal operation. Still, we propose to use the inversion strategy as a default.

6.2 Qualitative Evaluation

Figure 5 depicts the instances of nearest neighbor join that needed to be forwarded to another node for refinement. We selected the `zcurve`, `inversion`, `rotation+`, and `rotation_heur` strategies. The `rotaton` strategy is quite similar to the `rotation+` strategy. In this Figure, a point is a tweet and a line denotes an assignment of a

tweet to a building that needs to be refined, because the primary endpoint responsible for the tweet was unable to guarantee that this is actually a nearest neighbor.

In general, we can conclude that all strategies are working well, because most tweets are solved without any refinement. In other words, the best node-local example is the globally nearest neighbor. However, a few long lines are given in each of those images. These occur when a tweet was associated to a node that actually does not know a building nearer to the tweet that the depicted line illustrates. While it is counter-intuitive that such a thing happens, it is easily explained: data locality is guaranteed only for the primary nodes and some tweets end up in one worker responsible for the global south-west cell. Now, the dataset does not contain any buildings in this area. Therefore, only the replicated data objects that end up in this rank can provide buildings at all. But, in this very case at the bottom-left of the data space, the replica are actually coming from far away for many strategies. Note that this behavior depends on the number of nodes, too: while the Z curve strategy with powers of two leads to a sort of block-wise assignment with some deviations, other numbers lead to non-rectangular areas along the curve being assigned, sometimes only connected through one of the long edges thereby exploring two connected, compact, but distant areas inside which data locality is given.

6.3 Time Evaluation

The presented algorithms have been carefully implemented in C++ using the MPICH3 MPI implementation. In this situation, the rank is identified with the location in the ring. In this section, we report times for a single PC (Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz, eight cores) with some surprising results compared to the table before. The quality of the implementation should not play a major role in the reported time, but the overall structure (number of vnodes, hard disk access, caching) does. We, therefore, applied each method five times with disk caches being flushed each time leading to the following findings for a nearest neighbor join on the sample of 100,000 buildings and about 180,000 tweets:

The fastest strategy was actually the `rotation+1` strategy with an average join time of 3,188ms (sd=149.7ms). This might be related to the fact that missing cells were quite far away and at the same time the number of cells was only moderate. The `inversion` strategy was runner up with slightly higher average wall clock time of 3,464ms (sd=199ms). An overview of performance values is given in Figure 6.

This deviates clearly from expectations one might have drawn from the data locality analysis. But, many other aspects including the quality of the local spatial indices given the local data and the number of refinements that are triggered in the greedy setting have significant impact beyond plain data locality or refinement count.

6.4 Scalability

We perform experiments with an increasing number of tweets to see the general scalability of the approach. For a dataset of one million tweets, the numbers again show that the inversion strategy has the least number of refinement steps. Only 11,755 tweets are submitted for refinement in a cluster with eight worker nodes (each one with eight cores). The second-best number is achieved by the

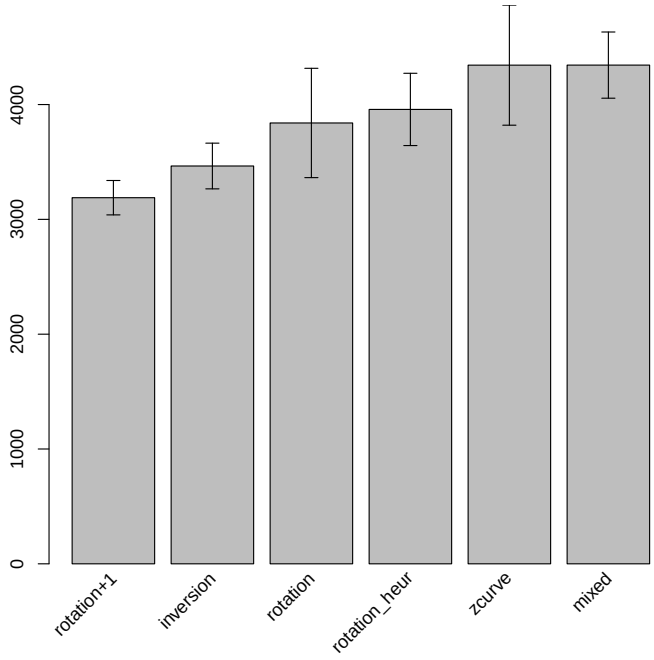


Figure 6: Average wall clock time of joining the 180,000 tweets to 100,000 buildings

`zcurve` strategy with nearly 18,000 refinements, all others lead to refinement counts between 20,267 and 22,808 instances. Considering wall-clock time, however, it turns out to be that the rotation strategies are slightly faster (86.7s and 88.7s) than `zcurve` and `inversion` (91.2s and 94.6s). Only the strategy `mixed` is significantly slower with about 120s.

This is explained from the fact that the number of missing cells for the rotation strategies are small (7.7 on average) while the number of cells for the other strategies are larger (19.2 for `zcurve` and 22.4 for `inversion`). With this medium-size dataset, we can conclude that it depends a lot on the workload, which strategy to choose: if the data comes from a very correlated distribution, then you might want to have fewer missing cells and choose the rotation strategies. If, however, the data is coming from different spatial distributions, one might get longer distances in general and have a better result from the `inversion` strategy, which is significantly better in reducing the number of refinements, though not their complexity. For the presented case of joining buildings and tweets, one should take into account that most tweets originate in an urban context and, therefore, most of the tweets are very near to a building. This property leads to the fact that a strategy with expected lower performance, because it fails on more instances, becomes faster in practice.

7 CONCLUSION

With this paper, we studied several scalable replication schemata for fast distributed processing of spatial operations. Ignoring the minor variation in the rotation class, we proposed three strategies: one based on the Z curve replicating to the next node along the

ring, one based on the Z curve replicating to carefully chosen binary inversions of the coordinates, and one based on replication through rotating along the Z curve. All three strategies provide data locality, but most importantly, we have seen a tradeoff between the number of refinements that are needed and the complexity of those refinements.

While it is possible to get a very small number of incomplete results by using the strategy based on careful coordinate bit inversions, this approach leads to a significant number of missing cells and, possibly, to a longer wall clock time, because the missing cells need to be located and scanned for refinement. On the other hand, the rotation strategies have nearly twice as many incomplete results, but for each of those, only a few cells are missing. Therefore, the workload used in this paper shows better performance with the rotation strategies.

In general, one should consider the workload and data distributions in order to select an optimal strategy. As we have been working on an MPI-based supercomputer with Infiniband interconnect, the transmission of information inside the compute cluster was significantly cheaper as compared to cloud computing, where a usual network has to be used to transmit the refinements to another node.

Consequently, we can conclude that the `inversion` strategy should be used if the network or the task manager are under pressure while with sufficient ressources and similar spatial distributions, the `rotation` strategy is preferrable.

For future work, we envision to address a few open questions: first, of course, query skew. What happens to the data distribution strategies, when queries become local. Think about having the same datasets as presented here in the paper: a lot of tweets and a lot of buildings. But assume, you are only interested in a single city. In these cases, the nodes should learn to trade their tokens such that the query can be distributed to more nodes than the set of nodes that cover the query region. For example, a node that owns only unrelated data to a query could load some of the cells over the network and start augmenting the search.

This, however, raises an interesting and complex additional problem: when nodes start to trade tokens or to load additional data in order to support a given query, then it is not generally known, which node knows which data. Therefore, the submission of refinement jobs becomes difficult: given a single master, we could of course track the spatial knowledge of each node and do some assignments. Still, it is an NP hard optimization problem to find an optimal schedule for which nodes should load which data in order to answer which subqueries. And in general, of course, we are interested in not having a central entity such as a master.

Therefore, we need to implement a gossip-like protocol in which the nodes can exchange their spatial knowledge and, possibly, also whether the data is available on disk or already in memory. We envision to apply Bloom Aggregated Cell Representations (BACR) for this in the future [16]. A BACR is a small, constant-size sketch of a set of cells. In a supercomputer, we can exploit MPI All-to-All communication as well as neighbor communication, in cloud computing scenarios, we would need to research strategies of how a query related to a certain set of cells reaches a good node for processing.

And even without token trading and excessive usage of virtual nodes, we face the problem of assigning a refinement task to workers, which needs to be balanced given the complexity of the query in relation to the worker (how many cells does he know, what contribution does this computation bring) and the number of tasks that are assigned to each and every node. We perform a greedy refinement in this paper just assigning the primary node of the first missing information, however, this will be suboptimal in some cases.

Another direction of future research is given by asking, how the block-based distribution schemes relate to replication and to come up with strategies of increasing data locality through replication in the case of assigning data based on an R-tree, for example.

REFERENCES

[1] Nabil R Adam, Basit Shafiq, and Robin Staffin. 2012. Spatial computing and social media in the context of disaster management. *IEEE Intelligent Systems* 27, 6 (2012), 90–96.

[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. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1009–1020.

[3] Furqan Baig, Mudit Mehrotra, Hoang Vo, Fusheng Wang, Joel Saltz, and Tahsin Kurc. 2015. SparkGIS: Efficient comparison and evaluation of algorithm results in tissue image analysis studies. In *Biomedical Data Management and Graph Online Querying*. Springer, 134–146.

[4] Ahmed Eldawy, Louai Alarabi, and Mohamed F Mokbel. 2015. Spatial partitioning techniques in SpatialHadoop. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1602–1605.

[5] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 1352–1363.

[6] Thomas Esch, Mattia Marconcini, Andreas Felbier, Achim Roth, Wieke Heldens, Martin Huber, Max Schwinger, Hannes Taubenböck, Andreas Müller, and Stefan Dech. 2013. Urban footprint processor—Fully automated processing chain generating settlement masks from global data of the TanDEM-X mission. *IEEE Geoscience and Remote Sensing Letters* 10, 6 (2013), 1617–1621.

[7] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[8] Doug Laney. 2001. 3D data management: Controlling data volume, velocity and variety. *META Group Research Note* 6, 70 (2001).

[9] Songnian Li, Suzana Dragicevic, Francesc Antón Castro, Monika Sester, Stephan Winter, Arzu Coltekin, Christopher Pettit, Bin Jiang, James Haworth, Alfred Stein, and Tao Cheng. 2016. Geospatial big data handling theory and methods: A review and research challenges. *ISPRS Journal of Photogrammetry and Remote Sensing* 115 (2016), 119 – 133. https://doi.org/10.1016/j.isprsjprs.2015.10.012 Theme issue 'State-of-the-art in photogrammetry, remote sensing and spatial information science'.

[10] Yu Liu, Zhengwei Sui, Chaogui Kang, and Yong Gao. 2014. Uncovering patterns of inter-urban trip and spatial interaction from social media check-in data. *PloS one* 9, 1 (2014), e86026.

[11] Stuart E Middleton, Lee Middleton, and Stefano Modafferi. 2014. Real-time crisis mapping of natural disasters using social media. *IEEE Intelligent Systems* 29, 2 (2014), 9–17.

[12] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, Vol. 1. IEEE, 7–16.

[13] Ibrahim Sabek and Mohamed F Mokbel. 2017. On Spatial Joins in MapReduce. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 21.

[14] Shan Suthaharan. 2014. Big data classification: Problems and challenges in network intrusion prediction with machine learning. *ACM SIGMETRICS Performance Evaluation Review* 41, 4 (2014), 70–73.

[15] Waldo R Tobler. 1970. A computer movie simulating urban growth in the Detroit region. *Economic geography* 46, sup1 (1970), 234–240.

[16] Martin Werner. 2015. BACR: Set Similarities with Lower Bounds and Application to Spatial Trajectories. In *23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL 2015)*.

[17] Brandon Williams. 2012. Virtual nodes in Cassandra 1.2. (2012). available at https://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2.