Geographical sharding in MongoDB

Using Voronoi partitioning to geographically shard point data

Ricardo Filipe Amendoa; Ríca Electrical and Computer Engineering Instituto Superior Técnico

Abstract—The growth of dataset sizes in the Big Data era has created the need for horizontally scalable Database Management Systems (DBMS), like MongoDB, Apache Cassandra and many others, generally known as NoSQL DBMSs. These systems need to be able to be configured in ways that best fit the data that they’ll contain, partitioning the data (Sharding) in a context-aware way so that queries can be executed as efficiently as possible. Spatial data (associated with a location) availability has been steadily growing as more devices get geolocation capabilities, making it important for NoSQL DBMSs to be able to handle this type of data. While NoSQL DBMSs are capable of sharding, not all (MongoDB included), can do it based on location or do it in an efficient way. Following the work by João Pedro Martins Graça in his Master’s thesis [1], exploring the various spatial partitioning policies, this work aims to implement and evaluating the practical usefulness of the proposed policy GeoSharding, based on Voronoi indexing, on the popular open-source project MongoDB. Benchmark results show that this policy can have a large positive impact in performance when used with datasets of large amounts of small documents, such as sensor data or statistical geography information.

Index Terms—Big Data, Spatial Data, Sharding, MongoDB, NoSQL

I. INTRODUCTION

Geo-referenced data, data that can be associated with some sort of location, has been growing in availability as more devices like smart-phones get geolocation capabilities and satellite imagery and sensor networks become better more available [2].

Photos posted to social-media, business information that can be searched by customers or navigation devices, weather data, census data among many other examples can all be associated with general (regions) or specific (coordinates) locations.

As these datasets grow they become increasingly more difficult to manage and store while maintaining good performance when querying or being used in computation [2].

A common way to scale databases is via vertical scaling, where the machine that runs the database management system is upgraded with better hardware. Vertical scaling eventually hits a limit when the cost of better hardware becomes uneconomical or there is no possible upgrade path. An alternative way of scaling exists, horizontal scaling, where the system is scaled by adding more machines (usually with commodity hardware to keep costs down) [3]. Horizontal scaling has no defined limit on scalability but can of course be limited by communication overhead within the system or external factors like network capacity.

The most common type of DMBS, Relational DBMSs, are usually very complex to scale horizontally because of the way data is stored, with relations between tables and normalization, that makes tables heavily dependent on each other to complete each table’s information [3].

A more recent kind of DBMS, the NoSQL DBMS, which aim to improve horizontal scalability so that these systems can handle incredibly large datasets or/and very large request loads for web services. They mostly achieve this by lowering some of the requirements/promises of RDBMSs, usually data durability and/or data consistency, and by data de-normalization, which makes tables and rows (or the equivalent data types) mostly independent, allowing for data to be easily partitioned and distributed among several machines.

This type of horizontal scalability via data partitioning is called sharding. Sharding policies are usually one of the main things to worry about when trying to improve the performance of a sharded cluster, since the way data is partitioned and distributed affects the efficiency of operations on the database. The policies should be adequate to both the data itself and the operations that will commonly be done on it, so that data locality can be leveraged. Ideally queries should only require contacting a single machine to be executed, the goal of choosing a good sharding policy is to approximate this as much as possible.

Not many NoSQL DBMSs support sharding policies adequate for geographical data [4]: common operations on georeferenced data usually involve fetching information related to a certain area, or doing interpolation based on neighboring data, so a good distribution policy for geographical data should take this into account to take advantage of data locality. If not, a simple query for data related with a certain small area may have to communicate with all machines in the cluster, degrading performance for any simultaneously occurring database operations.

A previous work compared different sharding policies by means of simulations, and concluded that a new suggested policy named GeoSharding can have better performance than most of the policies that are currently used in NoSQL DBMSs.

An implementation and evaluation of this policy could thus be valuable to measure the practical usefulness of GeoSharding for NoSQL DBMSs.

A. Proposed Solution

An implementation of the partitioning policy proposed by João Graça [1] in the DBMS MongoDB is proposed as a way to shard geo-referenced data without losing the context of the data’s location.

MongoDB was chosen as a base to the work because it’s a popular open-source NoSQL DBMS with API-level support for interacting with geographical data (like range queries) and has existing support for spatial indexing (which improves query performance at the machine level) but doesn’t have an adequate sharding policy for georeferenced data, which would be the next logical step for geographical data support.
on MongoDB. MongoDB can also do distributed computation, possibly making it a one-stop shop for geographical data storage and processing.

The partitioning policy is conceptually quite simple: Each shard is assigned a virtual geographical location in the form of latitude/longitude coordinates (doesn’t have to be related to it’s physical location) and then Geo-referenced data is placed on the shard that is determined to be the closest one, based on the shard’s virtual location. GeoSharding is a form of Voronoi Indexing, a spatial partitioning technique that is used in many different fields [5].

This policy allows for irregularly sized regions to be assigned, as opposed to alternatives like geohashing (z-curves) or quadtree hashing (explained in chapter ??) where the regions are fixed sized. This characteristic makes GeoSharding more flexible and better suited to irregular distributions of spatial data, which is common (business locations are overwhelmingly located in cities, for example).

B. Objectives and results

The main objective of this work is to follow-up on the study done in [1] and implement the GeoSharding partition policy for sharding in an existing database management system and to evaluate its real-world effectiveness.

This boils down to making a working version of MongoDB that can be configured to shard by geographical coordinates and allows the user to issue different virtual locations to different shards, insert and search for Geo-referenced data and issue range-based queries that can be executed in an efficient manner.

These objectives were achieved successfully, with the exception of range queries which, while possible, are not optimized to take full advantage of the data distribution of GeoSharding.

The results of the evaluation from chapter ?? show that GeoSharding is very promising for real-world use, with a good choice of shard locations the performance is equivalent or better than hashed sharding on all tested scenarios, with big performance improvements when using datasets where documents are relatively small, which is common with sensor data and census information.

GeoSharding also has the advantage of distributing data in a way that allows queries and computations to take advantage of data locality, for example, range queries can be implemented efficiently by contacting only a small subset of the cluster’s servers, which is not possible when using hashed-sharding.

II. GEOGRAPHICAL INFORMATION SYSTEMS

Geographical Information Systems (GIS) are systems designed to store, manipulate, analyze manage or present many types of spatial or georeferenced data. These systems can vary much in both intended use and technology stack, but always involve some kind of data store.

In the following sections several features of these systems are analyzed.

A. Spatial data types

Spatial data types are the backbone of GISs, as they are used to represent and store the information in the system. As mentioned in [6], there are two types of spatial data types, which will be somewhat familiar to those used to dealing with image formats: vector and raster data.

Like with images, vector data types represent geometric features, such as points, lines or polygons, while raster data types use point grids that cover the desired space.

The use case will determine the ideal type to use, but the vector model can offer more flexibility and precision. An overview of the most common types of vector data is given below:

- Point Data: This is the simplest data type and it is stored as a pair of X, Y Coordinates. It represents the entity whose size is negligible as compared to the scale of the whole map such as it may represent a city or a town on the map of a country.
- Lines and Polylines: A Line is formed by joining two points in an end to end manner and it is represented as a pair of Point data types. In Spatial Databases, Polyline is represented as a sequence of lines such that the end point of a line is the starting point of next line in series. Polylines are used to represent spatial features such as Roads, Streets, Rivers or Routes, for example.
- Polygons: Polygon is one of the most widely used spatial data type. They capture two dimensional spatial features such as Cities, States, and Countries etc. In Spatial Databases, polygons are represented by the ordered sequence of Coordinates of its Vertices, first and the last Coordinates being the same.
- Regions: Regions is another significant spatial data model. A Region is a collection of overlapping, non-overlapping or disjoint polygons. Regions are used to represent spatial features such as the State of Hawaii, including several islands (polygons), for example.

B. Geostatistics

Geostatistics is the use of statistics with a focus on spatial or spatiotemporal data. It is used in many different industries, like mining (to predict probability distributions or ore in the soil), petroleum geology, oceanography, agriculture [7], forestry and, of course, geography [8]. It can generally be used by businesses and the military to plan logistics and by health organizations to predict the spread of diseases [9].

Kriging: Kriging is widely used class of geostatistical methods that are used to interpolate values of arbitrary data (such as temperature or elevation) of unobserved locations based on observations from nearby areas.

These methods are based on Gaussian process regression, the value interpolation is modeled using a Gaussian process that takes into account prior covariances, as opposed to piecewise-polynomials that optimize smoothness of the fitted values. If assumptions on the priors are appropriate, kriging can yield the best linear unbiased prediction of the interpolated values [8].

Kriging techniques, when used on processing clusters, benefit from good data locality to avoid the slow network transfer of data about surrounding areas on which the computation is dependent.

As can be seen in fig. 1, kriging interpolation chooses more likely results instead of optimizing for line smoothness.

C. NoSQL database management systems

NoSQL, or Not Only SQL, is a category of database management systems (DBMS) that differentiate themselves
Fig. 1: An example of kriging interpolation (red full line). Also shown: smooth polynomial (blue dotted line) and Gaussian confidence intervals (shaded area). Squares indicate existing data.

from the more common SQL DBMSs, that use the relational model, by basing themselves on non-relational or not only relational models. Although NoSQL is a somewhat vague term that also includes niche categories like Graph DMBSs, it’s commonly used to refer to systems that focus on simplifying the design to more easily achieve horizontal scaling and high availability, which usually means dropping the relational model [3].

To achieve simpler scalability and/or availability, NoSQL DBMSs usually relax the consistency guarantee on data they store. This follows from the CAP theorem, which states that a distributed system can only provide two of the following guarantees:

- Consistency: Operations on stored data should always be based on the latest write to the system or return an error
- Availability: Requests should receive non-error responses, although not necessarily from the latest write
- Partition tolerance: The system continues to function correctly even if parts of it become unreachable or messages get delayed

Since many NoSQL DBMSs prioritize horizontal scalability and high-availability, partition tolerance and availability become hard requirements of these systems. Since all three guarantees can’t simultaneously be achieved, a common solution is to degrade consistency to what is known as eventual consistency: After a write the system might become inconsistent for a short period of time but eventually achieves consistency as the changes propagate to all machines.

III. GEOGRAPHICALLY AWARE SHARDING ALGORITHMS

In this section a quick overview of the relevant algorithms for geographically aware sharding is given. An in-depth analysis of these algorithms can be seen on the work of João Graça in his Master’s thesis [1], for which this work is a follow-up to.

A. Geohashing (Z-curve)

Geohashing is a hierarchical geocoding system that subdivides the space into several equal-sized grid-shaped regions that can be assigned to different machines. A Z-curve is used to reduce the dimensionality of longitude/latitude coordinates by mapping them to a location along the curve, with closely located coordinates being mapped to [1].

Some advantages of Geohashing are its simplicity and easily tunable precision (by truncating the code). Z-curves are, however, not very efficient for certain types of queries (like range-queries) since points on either end of a Z’s diagonal seem to be closely located, which is not the case. Since this method involves projecting the spherical earth into a plane, points that are close by may end up on opposite sides of the plane and appear far away to the curve.

Figure 2 shows a map of the Earth partitioned by a Z-curve.

B. Quadtree hashing

A Quadtree is a tree data structure that recursively subdivides a two-dimensional space into four quadrants, as shown on fig. 3. The subdivision occurs per cell as its capacity (which can be an arbitrary function) is reached, allowing the tree to be more detailed where appropriate [1].

Quadtree hashing allows large empty sections to be quickly pruned when querying.

C. Geosharding (Voronoi partitioning)

GeoSharding is the name of the policy used in this work, proposed by [1]. Based on Voronoi partitioning, which is a
method for dividing a space into several non-uniform regions based on distance to a given set of points in that space. The number of regions is equal to the number of points in the given set as each point gets assigned the region of the space that no other point is closer to. By computing the boundaries between regions a Voronoi diagram can be drawn, such as the spherical one shown on fig.4, which has the benefit of being easily understandable visually.

![Fig. 4: A spherical Voronoi diagram of the world’s capitals](Image 72x433 to 277x632)

Unlike GeoHashing, Voronoi partitioning doesn’t suffer from the same disadvantages: points that are close to each other, regardless of where they are on the map, are correctly considered close and vice-versa.

GeoSharding is a very versatile and efficient space partitioning policy, as more points can be added where more detail/capacity is required and the lack of regularity in the partitions.

A big drawback for GeoSharding can be it’s complexity in implementation for range or similar queries. In terms of computational complexity, although computing the Voronoi diagram can be slower than some alternatives, it can pre-computed and cached when changes to the cluster configuration are made, so that the impact is minimal when actual queries are performed.

IV. EXISTING SOLUTIONS

In this section some viable existing solutions for either storage and/or computation of data are presented.

A. Spatial Hadoop

Hadoop is an open-source framework for distributed storage and computing of large scale datasets. It’s one of the most mature and popular solutions for Big Data processing, it appeared at a time when large scale computation was still done with powerful computers [10].

Hadoop’s design is based on taking advantage locality, it gets around the problem of moving very large datasets to the processing nodes by instead moving the (small) processing software to storage nodes and executing it there, in parallel. This design has two major parts: the Hadoop Distributed File System (HDFS) and the MapReduce programming model, which allows the programmer to easily write massively parallel programs at the cost of flexibility.

The Hadoop Distributed File System is capable of storing very large files, even up to the range of terabytes, with horizontal scalability and high-availability via data redundancy. These files are as flexible as regular files, they can be structured, unstructured, json, images, etc.

The recommended use of Hadoop is large-scale data analysis or other types of long-running computations that don’t need to be done in real-time, since Hadoop does not support streaming computation as new data comes in.

Spatial Hadoop [11] extends Hadoop with spatial capabilities: data types, indexes, operations and a new high-level language called Pigeon for use with spatial data. The use of a dedicated language can be seen as downside since it makes existing Hadoop programs harder to reuse and might make spatial programs incompatible with other Hadoop extensions. Experienced Hadoop programmers might appreciate the high-level nature of the language and its familiar MapReduce paradigm, as well as the several tutorials are available on the project’s website.

B. GeoSpark

GeoSpark [4] is a recent and promising project built on the popular Apache Spark. First released in July 2015 and still currently in an experimental stage as of writing, it’s an open-source cluster computing system designed for large-scale spatial data processing. GeoSpark inherits many of it’s characteristics from Spark.

Compared to Hadoop, Spark provides parallel, in-memory processing, whereas traditional Hadoop is focused on MapReduce and distributed storage.

Spark doesn’t have a native data storage solution, so it’s often used with the Hadoop distributed file-system or other compatible data-stores like MongoDB or Cassandra, making it’s configuration more complex than other alternatives.

The main advantage of Spark (and GeoSpark) is it’s performance, being up to 100 times faster than Spatial Hadoop’s MapReduce [10] which is, in part, a consequence of its much more aggressive memory usage for it’s in-memory computation. The heavy memory usage makes Spark a very costly option since RAM capacity is much more limited and expensive than SSD or HDD capacity.

GeoSpark adds spatial partitioning (including Voronoi diagram) and spatial indexing to Spark along with geometrical operations, most notably range queries for searching inside regions of space [12].

Since spatial/geometrical support isn’t native to Spark but is added by GeoSpark, the API for these operations is GeoSpark specific, meaning that programs have to be re-written to take advantage of these features. Developing for GeoSpark is complicated not only by the underlying system’s complexity, Spark, but also some consequences of it’s experimental status, like the lack of comprehensive and organized documentation for it’s API.

C. Cassandra

Cassandra [13] is an open-source NoSQL database designed for use with multiple, geographically distributed, data-centers,
with high availability and large total throughput at the cost of extra read and write latency.

It’s what’s known as a wide column store, data is stored in tables with a big number of columns of different data-types. While it may seem similar to relational DBMS’s, Cassandra does not support join operations or nested queries.

Cassandra’s recommended use is as a storage back-end to online services, with large number of concurrent interaction, possibly in different geographical locations, with small data involved in each request. The lack of transaction support makes Cassandra unsuitable for applications that require atomic and durable write operations.

It does not support native server-side computation but it can work as a storage back-end for Hadoop, replacing the Hadoop distributed file system [3].

D. MongoDB

MongoDB is a very popular open-source NoSQL DBMS that was designed to be very easily scalable. To achieve simple horizontal scalability it avoids the complexity of Relational DBMS, lowers the consistency to eventual consistency and provides built-in support for sharding, replication and load-balancing.

It’s a document database, meaning that instead of rows in tables it has JSON documents in no-schema collections. The JSON format allows for the serialization of common programming data-types like maps, strings, numbers and arrays. A simple JSON document example is given on fig.5.

```json
{
  name: "sue",
  age: 26,
  status: "A",
  groups: ["news", "sports"]
}
```

Fig. 5: JSON document example

Source: https://docs.mongodb.com

High Availability (HA) is an important characteristic of MongoDB, data replication and automatic fail-over are built-in and easy to set-up by using replica sets.

MongoDB supports server-side distributed computation via MapReduce and also an abstraction of MapReduce called Aggregation Pipeline, which aims to be a simpler way of using MapReduce. It supports other types of server-side computation that exist mostly as a convenience for small computations like stored procedures, since they are not distributed.

Overall MongoDB is mostly designed for low-latency storage and querying but is also quite flexible, with support for server-side computation, good scaling mechanisms and simple replication [3].

V. GeoMongo Implementation

Most of the time spent on implementing GeoMongo was to get an understanding of the relevant code for sharding in MongoDB, which is a very large project with over 5000 files and over 1.2 million lines of source code.

The MongoDB’s code-base has some internal assumptions on the properties of sharding keys and chunks that are broken by the GeoSharding partition policy. This lead to a much more complex and time-consuming implementation than originally predicted.

There are five major code sections that had to be changed to add support for a new partition policy:

- User interface for sharding
- Chunk creation
- Sharding initialization
- Write operations
- Read operations

There are also several other minor code sections that required updates, however most of these are very similar and are either related to chunk validation or assumptions about chunk object structure and not worth mentioning.

In the following sections the major changes are discussed.

A. User interface

From the user’s perspective not much changed. Two new commands were added to the client shell and the Router process, a command to shard a collection with the GeoSharding policy and one to add/change chunks locations or assigned shards.

The user interface remains very similar to btree and hashed sharding with the difference that the user gets manual control over chunk ranges and what shards they’re assigned to. Although the choice of shard can be automated via the balancer, dynamically choosing good locations to assign to each chunk is a complex problem that should be studied in a future work.

Since MongoDB has native support for geographic queries, like geoNear, the user can possibly get immediate performance benefits without changing existing application code, just by choosing a new sharding policy.

B. Chunk creation

The chunk creation process is different for geographical chunks but also simpler in it’s current form, as the chunk ranges (the Voronoi diagram) aren’t calculated at creation time, which could be a valuable optimization.

Apart from verifying if a chunk with the same position already exists, creating a geographical chunk involves using a higher version than the last created chunk, storing longitude and latitude instead of min and max, setting the chunk’s namespace (what database and collection it belongs to) and what shard it will initially be assigned to. In practical terms this boils down to creating a document in a special collection that is stored on the config server, representing the chunk. These documents can be fetched and/or updated by other routers or shards when stale caches are detected or chunk modifications are made.

Unlike what’s done for the standard btree and hashed sharding mechanisms, automatically selecting uniformly distributed locations for chunks when setting up sharding on a collection isn’t desirable, since expecting a dataset to be uniformly distributed over the entire sphere is unreasonable.

There is, however, an opportunity for automatic adjustment of chunk locations after the dataset is loaded or dynamically as it grows by perhaps using statistical analysis or machine learning algorithms.
C. Sharding initialization

When initializing sharding for an empty collection MongoDB still expects at least one chunk to exist at the end of the process. Either a default location can be chosen, which would better fit the existing interface of MongoDB, or the user can be asked for the location of the first chunk.

The implemented solution uses a default location that can later be changed by the user, since this fits the existing interface and can transparently support a possible future work that dynamically selects new locations for chunks as the database is used.

D. Write operations

When writing to the database, whether it’s a single document or a batch of multiple documents, each document’s shard key goes through a method of the ChunkManager object, called findIntersectingChunk, that is responsible for mapping the document’s shard key to the correct chunk. Thus the necessary changes here involved adding a new code path, for when the shard key is of type 2dsphere, that extracts the coordinates and iterates through the existing chunks to find the closest one. A new method, findNearestGeoChunk, was created for this since it can be reused for reading queries. That method is described in detail in GeoSharding routing, at the end of this section.

Note that depending on the database configuration documents might not be required to have a shard key, in which case they get placed into the primary shard.

E. Read operations

Reading operations can be made up of multiple filters.

Read operations, unlike write operations, aren’t handled document by document since there’s no way to know how many documents will be involved in a query before it’s executed.

This means that multiple chunks and shards may need to be contacted in order to execute the request. ChunkManager objects have a method, getShardIdsForQuery, responsible for interpreting the query’s filters and returning a list of shards that may contain documents that fit the filters.

In the case of GeoSharding the changes are similar to what was done for write operations: A new code path was added for when one of the filters included coordinates to search for. The same process of matching the relevant chunk is done and the respective shard is returned.

When sharding is enabled an additional filter is added to every read query that instructs shards that are contacted to filter out any documents that are found but don’t belong to chunks they own, since during/after a migration documents might still exist on the old shard for a while.

If none of the filters are related to a document’s coordinates all shards have to be contacted.

F. Minor changes

There were also other, smaller, changes necessary for GeoMongo to work:

- Chunk ranges: MongoDB includes a mechanism for binary tree and hashed sharding to cache chunk ranges. In its current form GeoSharding makes no use of this feature but it could be very useful for range based queries, allowing the Voronoi diagram to be computed at chunk insertion/modification time rather than query time.
- Chunk ID’s: Chunk’s were previously identified by their minimum key, since the complete shard key space had to be assigned and chunk ranges couldn’t overlap. Longitude and latitude don’t have the same characteristics so both are needed to identify a chunk.
- Chunk validation: chunks ranges and min/max values are validated in all server types and several parts of the code-base. These checks range from verifying that the minimum key is smaller than the maximum key to verifying that the whole key space is accounted for with no overlaps. For GeoSharding the checks are simpler: make sure that the longitude and latitude values are valid and prevent multiple chunks having the same location.
- Chunk splitting: Chunk splitting was disabled for GeoSharding since, as stated before, choosing a location for new chunks can’t be trivially automated. Furthermore, the policy dictates that each shard should own a single region of space and there’s no clear benefit to allowing more at the cost of extra implementation complexity.

Internal changes: Like mentioned before, the GeoSharding policy breaks several of the assumptions that MongoDB makes with regards to sharding keys and chunks, the biggest one being that it’s not 1-dimensional.

A new shard key type was created, 2dsphere, matching the key type used for geographical indexes. Unlike other shard key types, for GeoSharding 2 values are needed, the coordinates.

Most of the changes that had to be made were, unsurprisingly, to code that handles chunks.

Because MongoDB is designed with 1-dimensional sharding in mind, chunks include a minimum and maximum key. Since GeoSharding is 2-dimensional policy based on locations the minimum and maximum keys have no meaning, so when using GeoSharding the chunk stores it’s virtual longitude and latitude instead. This change affected many parts of the code-base and all three types of server because the minimum and maximum keys are assumed to exist and are validated in several different components responsible for the sharding mechanism, instead of relying on an abstraction for chunk identification and validation.

Chunk creation, validation and chunk ID’s also required updates to remove limitations imposed by the lack of an abstraction over chunk identification and validation.

The auto-splitting feature is disabled under GeoSharding because it was outside the scope of this work to dynamically create and choose locations for chunks, which would have been required to support the feature.

GeoSharding routing: When inserting data or making queries the router calculates what shard it should forward the request to. While it is conceptually simple (select the shard that owns the chunk closest to the relevant document) computational precision must be taken into account. Since the number of chunks should be equal to the number of shards, and therefore very small, simply iterating over the chunks and calculating the great-circle distance to the document’s coordinates is expected to have a negligible impact in performance.

Vincenty’s formula was used to compute the great-circle distance, as it’s more accurate than common alternative, the Haversine formula, and is computationally inexpensive [14]. Although the Earth is not a sphere but an oblate spheroid
and Vincenty’s formula still has a small amount of error, these details aren’t problematic since precision isn’t crucial for GeoSharding, as it only affects the locations of chunk boundaries, points that are close still end up in neighboring chunks. Performance is a concern because this calculation is done multiple times for each write or read on the collection but Vincenty’s formula is a relatively simple computation and should have a negligible impact.

Vincenty’s formula for a sphere (which the Earth closely approximates) is shown on 1, where \( \lambda_p \) and \( \lambda_q \) are the longitudes of two points, \( P \) and \( Q \), and \( \phi_p \) and \( \phi_q \) are the latitudes, \( \Delta \lambda \) and \( \Delta \phi \) are the differences in longitude and latitude between the two points and \( \Delta \sigma \) is the great-circle distance between them. Some of these angles are depicted in fig. 6.

\[
\Delta \sigma = \arctan \left( \frac{\cos \phi_q \cdot \sin(\Delta \lambda)}{\sin \phi_p \cdot \sin \phi_q + \cos \phi_p \cdot \cos \phi_q \cdot \cos(\Delta \lambda)} \right)
\]

(1)

For computation the function \( \arctan(y, x) \) should be used in place of \( \arctan(y/x) \) because it can calculate the quadrant of the computed angle and return valid results when \( x = 0 \), as it receives the arguments separately and can thus know the values and signs of \( x \) and \( y \).

In the case of ties when computing the closest chunk the one with the smallest Id is chosen.

VI. PERFORMANCE EVALUATION RESULTS

In this section the benchmark results are presented and discussed. As said before, two versions of the dataset were used, with remarkably different, but expected, outcomes: a text-only version and a version where every document had a moderately sized image attached to it. The same benchmarks were used for both versions.

A. Attached image documents

For this set of tests an image of roughly 0.5MB (473.35KB) was attached to each document, bringing the total size of the dataset to about 2.4GB. Because of the size of each document the tests became network/communication driven and the difference between GeoSharding and hashed sharding was hidden by the transfer time of each document.

Read performance: Analyzing fig. 7, both policies had very similar results, performing equally within margin of error. Because of how large each document was, the test became communication driven: the transfer time of each document was much larger than the time taken to find it, which significantly reduced the impact of the sharding policy.

In fig. 8 the horizontal scaling for reads on both policies can be seen. There is very little improvement, which seems to be a consequence of even 1 shard being able to get very close to theoretical maximum throughput of the network connection: at 1Gb/s, 2.5GB of data should take 20 seconds to transfer. 80% of that throughput (800Mb/s for a running time of 25 seconds) is achieved with a single shard so by adding any additional machines one can only expect at most a 20% improvement in this test. Unfortunately the equipment needed to remove this bottleneck was not available but as the running time was dominated by document transfer it’s unlikely that the two policies would show a significant difference between each other.

Write performance: Writing larger documents is the only test in this evaluation where increasing the number of threads on the client has a very small performance benefit, as seen on fig. 9a. This points to a limitation on the write performance of a single shard. Writing 2.5GB in 125 seconds means a throughput of 20MB/s, which is not unreasonable for a 7200RPM drive when not writing serially (as it’s writing several thousand documents and not a single large file). The storage layer of MongoDB features compression, which might also slow down write performance.

Horizontal scaling of writes was pretty much linear with the number of shards, as shown in fig. 10, unlike with reads (which were limited by the network), writes seem to be directly limited by the write capability of each shard, hence the linear scaling. With 4 shards the network throughput approached the theoretical limit but with a big enough margin that it’s safe to assume it wasn’t limited by the connection.

Performance was pretty much the same for both policies, which is expected on insertions and even more in the case of larger documents, as the transfer time (and in this case, disk-write time) overshadows their impact.

B. Text-only documents

The documents in the text-only version of the dataset were quite small, on the order of hundreds of bytes. GeoSharding performed either on par or significantly better than hashed sharding in both reads and writes.

Because the text-only version of the dataset is quite small, at roughly 0.5MB total, the running time of some test runs took less than 1 second. To prevent this, the text-only benchmark was changed to read the dataset 10 times, raising the running time of the fastest test runs to an acceptable level, where startup overheads and initial network latency should have negligible impact.

Read performance: As can be seen in fig. 11a multi-threading the client (or using multiple clients) can significantly improve the read times, even with a single shard in the cluster.
This is because MongoDB is capable of opening multiple threads to respond to clients, parallelizing the data transfer and reducing the effect of round trip latency between requests. Beyond 16 threads the improvement was small to insignificant, however.

By default, MongoDB will accept up to 65536 simultaneous connections and the storage engine, WiredTiger, has a limit of 128 concurrent simultaneous read/write operations. Thus the bottleneck appears to lie elsewhere, perhaps in the 8 CPU cores of the machines, although the CPU’s didn’t seem over-utilized during the test runs.

Still in fig. 11a, GeoSharding can be seen to perform on par or even slightly better than hashed sharding, showing that GeoSharding routing does not introduce significant read overhead when compared to hashed sharding.

On fig.11b, with 2 shards, GeoSharding shows significantly smaller running times, especially when the client uses a large amount of threads. When using 64 threads GeoSharding is 3.1 times faster than hashed sharding. This is likely because with hashed sharding the router, when given a pair of coordinates, can’t calculate which shards might have documents with those coordinates and has to contact both shards. Because there are multiple concurrent requests this can actually slow down the test more than expected, as each request causes an unnecessary slowdown to other shards, impacting other requests as well.

GeoSharding allows the router to contact only the relevant shard for each query, lowering the running time for the query and minimizing the impact on other concurrent queries.

With 4 shards (fig. 11c) the same effect can be observed, this time with an even bigger difference, with GeoSharding being 5.1 times faster than hashed sharding.

Fig. 12 shows the best case scenario for each policy and each number of shards, so that the horizontal scaling capability of both policies can be analyzed. The advantage of GeoSharding over hashed sharding is quite large, with GeoSharding even getting more than linear scaling: by moving from 1 shard to 2, performance increases by a factor of 4.1, and then by a factor of 2.8 when going to 4 shards.

Besides the reasons mentioned on the discussion of fig. 11, read locks might also explain the big improvements in performance. With MongoDB, as with most DBMSs, reads and writes attempt to lock the database to prevent other requests from using incomplete/incorrect document. MongoDB allows multiple reads to share a single lock but there is still a performance impact, as locks are released with their respective read queries and a new lock must be obtained for the next group of queries. When using sharding on MongoDB read locks happen at the level of each shard, essentially multiplying the number of possible concurrent query batches by the number of shards.

Write performance: With writes (fig. 13) a similar trend can be seen in all cluster configurations in regards to threading on the client: Performance steadily improves until somewhere between 4 and 16, probably 8 as that is the number of threads on the CPU’s of all machines involved.

As expected, GeoSharding is very much on par with hashed sharding during insertions, since both policies only contact one shard when inserting a document.

With regards to horizontal scaling, in fig. 14, both policies behave similarly: a modest improvement can be seen when going from 1 to 2 shards but there are no significant gains when going beyond 2 shards.

The bottleneck might be the router, since it acts as a proxy between the client and the cluster. The default connection limit for this component is 10 thousand connections, however, so at least it doesn’t seem to be a configuration issue.

Another possibility is that, with a total running time of less than a quarter of a second, the dominant factors might simply be the initial connection latency and the time it takes to start the threads on the server-side. Unfortunately the text-only dataset is small enough for this to be a possible issue.
Inserting the data-set multiple times was not a reasonable option because MongoDB’s storage layer features compression, which might produce inconsistent results, as the insert order was randomized for each test run.

C. Network usage

Network usage during the tests was observed but not accurately recorded. It was fairly consistent for the duration of each test, except for the very small ramp-up and ramp-down times. The following observations could be made:

- The 1Gb/s connections bottlenecked the tests on the bigger dataset.
- For text-only files the network was not a bottleneck, none of the machines had a total throughput close to the Gigabit limit of the connection.
- The router’s role as a proxy was clear, as it’s network throughput in each direction was symmetrical to the sum of the rest of the machines’ throughput.
- The config server used very little bandwidth on all test runs, rarely reaching 15Kb/s and below 1Kb/s most of the time.

VII. EVALUATION OF MONGODB AS A BASE FOR GEOmongo

The decision to use MongoDB came with some unfortunate setbacks, due to lack of knowledge about the project’s limitations. In this section the pros and cons of this decision are mentioned.

A. The pros

The native support for geographical queries on MongoDB meant that most of the client interface for the work would already be in place. It also makes it possible for code already written for this API to get a performance boost for free, by changing the cluster sharding configuration, which means that existing libraries like PyMongo (the Python3 client for MongoDB) and many others would be able to take advantage of the new policy from day 1.

Another big advantage for basing the work on MongoDB was the existence of geographical index support. Indexes can be very important to query performance by avoiding collection scans and should come before sharding when attempting to improve query performance, since sharding makes the system more complex and comes with added costs for hardware. With both existing API for geographical operations and geographical indexes, geographic sharding was the next logical step in terms of geographical data support for MongoDB.

Besides the technical aspects, MongoDB’s excellent user documentation was of great help when trying to setup a cluster, writing the benchmarks and also understanding the impact of various things on the performance of a MongoDB cluster.

B. The cons

The biggest issues were the sheer size and complexity of MongoDB’s code-base, with over 5000 files of source code, and the very limited documentation about the code-base. Because of this, getting a decent understanding of the source code organization took a long time, which delayed GeoMongo’s implementation much more than expected.

The way sharding is implemented in MongoDB assumes too much about partition policies, with many types of checks and validations in several parts of the code-base that have to be worked around when implementing a policy that does not fit those assumptions. A more general implementation, using the object-oriented features of C++, could allow for different policies to define their own validation rules for chunks and chunk ranges. It also has some other limitations, like forbidding the assignment of a document to more than one chunk, that may make some of the future work suggestions hard to implement.

In terms of server-side computation MongoDB was a letdown. Although MongoDB supports several options for server-side computation, most of them run only on the router, which means...
that data must be transferred from the shards, processed by a single node and then sent back to update the results. MongoDB does support MapReduce and an easier to use abstraction called the Aggregation pipeline, which uses several rounds of MapReduce to perform a series of computations one after the other. The problem here is that MapReduce functions can’t interact with the database for any reason, meaning that many types of computations, like Kriging interpolation, can’t take full advantage of data locality: the results have to be sent to the router and then sent back to write to the shards [15].

VIII. CONCLUSION

The work presented in this dissertation was aimed at evaluating, with a practical implementation, the usefulness of the GeoSharding partitioning policy as suggested by João Graça’s work, which used simulations to compare different policies in [1].

The very popular database management system MongoDB was chosen as the base for this work, which seemed on paper like a good fit but its internal architecture proved to be limiting or difficult to work with in several ways, for example for distributed computation. Despite the setbacks from this choice, a working version of MongoDB with GeoSharding for point data, named GeoMongo, was achieved.

In the evaluation chapter two scenarios were analyzed: datasets with moderately sized data points, like georeferenced images, and datasets with small data points, like georeferenced sensor data. The results showed that for larger data points MongoDB’s request latency is good enough that hardware limitations can quickly come into play, as well as the large transfer times for these bigger data points making the choice of policy less important. For small data points the outcome was quite different, with GeoSharding performing incredibly well when compared to hashed sharding, making GeoMongo very well suited to these kinds of datasets.

The results show great promise for use with datasets with very large numbers of small data points, such as sensor data [16], and the way spatial-data is distributed in GeoMongo clusters has great potential to be leveraged further by future work.

IX. FUTURE WORK

In this work the GeoSharding partitioning policy was proved to work with point data on MongoDB, and with very good results when used appropriately. Due to time constraints, some interesting ideas (range queries and non-point data) could not be tested and unfortunately had to be left out. In this section these and other relevant ideas are discussed, both on their possible benefits and their complications.

A. Generalizing MongoDB’s sharding internals

As discussed in section VII, MongoDB’s sharding internals are not a good fit for GeoSharding, requiring a ugly duct-tape in several parts of the code-base to work. Generalizing chunks, chunk creation, chunk ranges, chunk id’s and chunk splits so that they can be overridden by subclasses of chunks could allow for a very clean and maintainable implementation of GeoSharding and possibly other policies that don’t be reduced to a single dimension without information loss.

B. Leveraging GeoSharding for range and intersect queries

A major feature of GeoSharding is how it keeps points located closely together in the same chunks/shards. This can be leveraged by the native geographical queries of MongoDB to hit only the necessary shards when searching for documents, instead of the current behavior of hitting every shard because previously there was no way to predict what shards could have data for a certain location.

This is bound to be more complex than searching for specific locations, as the problem becomes about calculating with what chunk regions the range query intersects, which can be more than one. To avoid performance issues caching the chunk regions might be necessary as well.
Fig. 13: Write performance of text-only documents with: a) 1 shard, b) 2 shards and c) 4 shards

Fig. 14: Horizontal scaling of write performance of text-only documents, measured with 64 client threads

C. GeoSharding for non-point data

Sharding non-point geographical data, such as lines or polygons, in MongoDB would be a very complex problem, due to certain limitations imposed by MongoDB on its sharding mechanisms (for very valid reasons).

The two main issues are that in MongoDB each document must be assigned to one and only one chunk and documents must be filtered at the shard level. This means that a range query that intersects a line or polygon may not find the line if the range is contained to a set of chunks than don’t include the chunk that the line is assigned to.

One way to solve this problem is to make range queries (and similar queries) search also on chunks outside it’s range, which is not optimal and can be very inefficient. Another solution would be to place copies of the document on chunks it intersects, however, MongoDB’s requirements don’t allow this, which means at least a heavy rewrite of several parts of MongoDB would be needed or it may even incompatible with other existing features of MongoDB.

Although seemingly complex, implementing this would be very valuable and make GeoSharding on MongoDB much more versatile, as lines are often used to represent roads, rivers and other similar objects while polygons can often be used to represent buildings, land properties, cities, natural parks and many other types of geographic areas.

D. Dynamic allocation of chunk’s virtual coordinates

Dynamic allocation of chunk regions would make GeoMongo much more useful, practical and intuitive, since manually calculating good locations based on the geographical distribution of a dataset can be time-consuming or otherwise impossible if the data-set is still growing. For datasets that grow over time, dynamic allocation can make it possible to keep the cluster balanced, without manual intervention, long after it’s been setup.

A possible method of doing this allocation is using the K-means algorithm, with K equal to the number of shards, implemented as MapReduce, to take advantage of MongoDB’s distributed computation capabilities, as described in [17]. This computation can be used in a similar fashion to garbage collectors in programming languages with managed memory, being done only periodically, when the system load is low or when data distribution imbalance in the cluster rises above a certain threshold.

A study on the characteristics of different algorithms for this purpose, using simulations, could be very valuable not only for GeoMongo but also for other projects that have or plan to have similar features.

REFERENCES


