Industrial Paper

Scalable Spatio-temporal Indexing and Querying over a Document-oriented NoSQL Store

Nikolaos Koutroumanis
Deptartment of Digital Systems
University of Piraeus
Piraeus, Greece
koutroumanis@unipi.gr

Christos Doulkeridis
Deptartment of Digital Systems
University of Piraeus
Piraeus, Greece
cdoulk@unipi.gr

ABSTRACT

In this paper, we provide an in-depth study of the performance of spatio-temporal queries in document-oriented NoSQL stores. Existing NoSQL stores provide limited support for spatial data and (quite often) no native support for spatio-temporal data. As a result, the performance of query execution over large collections of spatio-temporal data is often suboptimal. We present an approach for indexing spatio-temporal data, which is applicable to *any* NoSQL store that provides key-based access to data *without modifications to its code*, and we show how to generate data partitions that preserve data locality. Moreover, we show the impact of indexing and partitioning on the number of cluster nodes that serve a query, and we discuss the advantages and disadvantages for different applications. We adopt a methodology for the evaluation of spatio-temporal range queries, which can serve as a benchmark. In our experiments, we focus on MongoDB (as a representative NoSQL store that provides spatial support) and we study the impact of indexing spatio-temporal data on performance, using both real-life and synthetic data sets in a medium-sized cluster.

1 INTRODUCTION

Big spatio-temporal data sets are collected every day at unprecedented rates [15, 17], due to emergent applications, such as fleet management solutions, surveillance systems in maritime and aviation, human and animal tracking, IoT sensor feeds, location-based web search, and social networks with geotagged content. These applications generate huge volumes of positional information represented as points, which require scalable storage and retrieval, so that data analysis techniques can be applied to discover hidden spatio-temporal patterns. As a result, scalable spatio-temporal data management is a challenging research topic, and efficient solutions are required for storage, indexing and querying.

NoSQL stores [4, 7] comprise the state-of-the-art in scalable storage to date. However, while support for spatial data is provided recently by an increasing number of NoSQL stores, this is seldom the case for spatio-temporal data. In fact, even spatial data access methods are not always optimized in today's mainstream NoSQL stores. While most relational DBMSs have adopted R-trees [11] (or its variants [2, 16]) for efficient spatial indexing, NoSQL stores with spatial support adopt GeoHashes to map spatial data to one-dimensional (1D) values, which is then indexed using traditional 1D indexes, such as B-trees [6] (see Table 1). Our conjecture is that this decision relates to the cost of building and maintaining a distributed R-tree. Consequently, the performance of existing solutions is suboptimal, when faced with the challenge of efficient and scalable retrieval of spatio-temporal data.

	Database	Spatial Indexing
RDBMS	PostgreSQL (PostGIS extension)	R-Tree
	MySQL	R-tree
	Oracle	R-tree
	MariaDB	R-tree
	SQL Server	B-tree
	SQLite (SpatiaLite extension)	B-tree
NoSQL	MongoDB	B-tree
	Redis (Geo API)	Sorted Set
	DynamoDB	B-tree
	Elasticsearch	BKD-tree
	Neo4J	B+Tree

Table 1: Spatial support in most popular relational and NoSQL data stores

Our work is motivated by real-life applications, revolving around fleet management operators in the urban domain, which collect large volumes of positional data from GPS-equipped vehicles daily. The specific use-cases that are supported by our work relate to exploratory analysis of historical routes, using multiple spatio-temporal queries of varying granularity. The retrieved trajectories are analyzed for fleet cost reduction (by analyzing the fuel consumption of historical routes), intelligent routing, as well as for discovering movement patterns. The challenge is to provide a scalable storage and spatio-temporal querying solution for large volumes of historical mobility data. Unfortunately, existing industrial solutions are not optimized for spatio-temporal querying at scale, thus fleet management operators apply data analysis techniques only on recent subsets of their historical database, while older data is kept in cold storage.

Motivated by these limitations, in this application paper, we provide an in-depth study of querying spatio-temporal data at scale, focusing on a document-oriented NoSQL store, namely MongoDB. The choice of MongoDB is justified due to its wide popularity among big data developers, and its maturity compared to other competitive technologies. We explain the internal details of indexing and sharding, focusing on how spatial data is supported, and eventually design a solution for spatio-temporal data using the built-in indexes of MongoDB. Then, we propose an alternative approach that uses the Hilbert space-filling curve (which has been shown to have nice clustering properties [14]) to generate one-dimensional (1D) keys, which facilitates indexing of spatio-temporal data, and allows to preserve data locality in the nodes of the MongoDB cluster. Moreover, this approach can be implemented *on top of* MongoDB (and other key-based NoSQL stores), thus being directly applicable for any application.

In particular, our approach has an effect on sharding, essentially creating spatio-temporal data partitions that preserve data locality. As we demonstrate in our empirical evaluation, this has a profound impact on performance.

Our contributions can be summarized as follows:

- We propose an approach for efficient storage and querying of spatio-temporal data based on Hilbert encoding, which can be applied to any NoSQL store that supports key-based access to data.
- We show that our approach achieves spatio-temporal data locality across the distributed data partitions, and we discuss the advantages and disadvantages for different applications.
- We present a methodology for evaluating the impact of spatio-temporal access to data at scale, which can also serve as a benchmark for spatio-temporal queries in NoSQL stores.
- We perform extensive experiments over a MongoDB installation on a public cloud, and we study the effect of different metrics (such as keys and documents accessed, nodes involved in query execution) on execution time using both real and synthetic data.

The remainder of this paper is structured as follows: In Section 2, we review related research efforts. Then, in Section 3, we present the internal mechanisms of MongoDB for indexing and handling spatial data. Section 4 outlines our approach for indexing spatio-temporal data. Section 5 presents the results of our empirical evaluation, and Section 6 concludes the paper.

2 RELATED WORK

Spatial data indexing is a long-studied topic, with R-tree [11] and its variants [2, 3, 16] being a prominent data structure in centralized databases. Even though approaches for distributed R-trees have been proposed [8], they have not been adopted by today's NoSQL stores, probably due to the high maintenance cost in distributed settings and due to the gradually diminishing performance after many inserts/updates.

2.1 One-dimensional Indexing of Spatial Data

Space-filling curves have extensively been used in spatial databases in order to map high-dimensional data to one-dimensional values, which can be indexed using standard data structures, such as the B-tree. Although many variants exist, notable examples include the z-order and the Hilbert curve (depicted in Fig. 1). In the context of data management and indexing, the objective of all space-filling curves is to preserve data locality in the one-dimensional space, so that spatial range queries can be transformed to one-dimensional intervals of small length, in order to reduce the number of false positives.

Even though the idea of GeoHashes was first proposed by G.Niemeyer in 2008, it bears similarities with space-filling curves (in particular with z-order), which have been known for decades. MongoDB uses GeoHashes to store spatial data efficiently. The idea of GeoHashing is to use a hierarchical subdivision of the 2D spatial domain, which uses multiple layers, with each layer divided in a set of cells (or buckets). At the top layer, four buckets exist which are derived by splitting each dimension in the middle. Then, each bucket at the top layer can be represented by 2 bits: 00, 01, 10, and 11. The hierarchical subdivision is performed recursively, so at the next layer sixteen buckets exist and four bits are used to address a bucket. As a result, any 2D spatial point

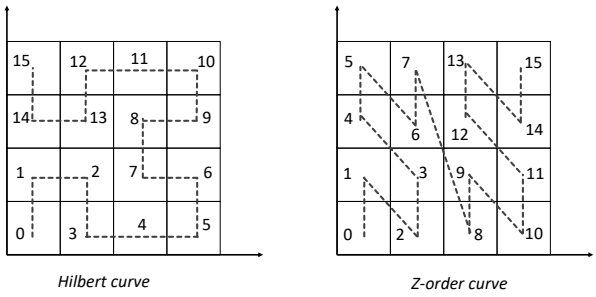


Figure 1: Illustration of the Hilbert and z-order space filling curves

is assigned into a lower-level bucket uniquely, and the bit representation of the bucket can be used as an indication about the location of the point in the 2D space. The more layers, the higher the precision of the respective location. Finally, GeoHashes use a *base32* String representation, instead of a bit representation, which uses a 32-character set comprising the twenty-six letters a–z, and the digits 2–7. As an example, Athens (Greece) has coordinates (37.983810, 23.727539) which is represented as a GeoHash of "swbb5ftzes" for precision of 10 characters. If we used lower precision, the corresponding prefix would be obtained. For instance, the GeoHash of Athens for precision of 5 characters is "swbb5".

2.2 Spatial and Spatio-temporal Queries in MongoDB

NoSQL systems are widely used by modern applications for scalability and high performance. For a recent survey on NoSQL stores, we refer to [7] and also to the early work of Cattell [4].

There exists some work on studying the performance of built-in mechanisms for spatial query support in MongoDB. Duan and Chen [9] compare MongoDB against ArcGIS, in order to assess the performance of the spatial extension of MongoDB against a well-established GIS. More recently, Bartoszewski et al. [1] compare MongoDB against PostGIS for spatial data. However, both studies evaluate the systems on a single machine, which is a limitation because it hides the impact of distributed storage on query execution.

In [13], an experimental evaluation of MongoDB against PostgreSQL is performed for spatio-temporal data. This is one of the few studies that try to evaluate MongoDB's capability in terms of querying spatio-temporal data. However, their study has some limitations, most notably the lack of data partitioning in evaluation. Instead, a small cluster is used and all machines contain replicas of the data set. In contrast, our work provides an in-depth investigation of different aspects of spatio-temporal data management, including indexing, data partitioning and load balancing, in a much larger deployment of MongoDB in a sharded cluster.

ST-Hash [10] follows an approach for spatio-temporal indexing on top of MongoDB. The main idea is to extend GeoHashes in a way that time is also incorporated in a string representation of a one-dimensional value. This value can be decomposed to obtain the corresponding spatial and temporal information. Hence, a one-dimensional index is built on this string value, in order to support efficient point and range searches. However, the resulting encoding uses the year as a prefix, which is not

effective for certain query types. For example, queries with high spatial selectivity but low temporal selectivity cannot exploit the encoding, in order to efficiently identify which data blocks need to be accessed.

SIFT [12] is an implementation of a distributed spatial index upon MongoDB. The study focuses on the ingestion, indexing and querying of highly-skewed spatial data. The basic data structure of SIFT is based on a tree where the spatial objects are represented by their minimal bounding boxes. The tree structure follows an approach so as to avoid any rebalancing, splitting and merging operations when spatial objects are inserted to the index. Nonetheless, the index is limited only to spatial queries.

3 BACKGROUND ON MONGODB

MongoDB [5] is a popular document-oriented NoSQL store that stores data in the form of *documents* in *collections*. *Documents* are binary JSON objects (BSON) that consist of a set of fields with associated values. Values can be simple or complex, e.g., an array or even a nested document. Each document is typically associated with a key that uniquely identifies it. *Collections* are containers for conceptually similar documents, however no restrictions are imposed to the schema of each document. In MongoDB, collections are stored in *databases* which are namespaces for physical grouping of collections.

3.1 Indexing in MongoDB

The main indexing structure used in MongoDB is the B-tree [6], which supports both point and range queries. Apart from *single field* indexes, which index documents based on a single field, a *compound* index can be used to combine multiple fields (up to 32 fields), thus supporting queries with predicates on multiple fields.

Compound indexes are organized hierarchically based on the *declared order* of the *index keys*. In the case of two fields A and B, the compound index first organizes the sorted values of A in buckets, and then these buckets keep references to buckets that hold the sorted values of B. An example is depicted in Fig. 2, where A=`hotelName` and B=`price`, denoted as `{hotelName:1, price:1}` in MongoDB. This indexing scheme has some important consequences on performance. First, only queries with a predicate on A can benefit from this index, since the value of the predicate is necessary to start the traversal of the index and locate buckets with relevant keys efficiently. Second, it is beneficial to use as first index key a field that has many distinct values, in order to effectively narrow down a search to few buckets only. As a result, the order of index keys in a compound index determines the performance of searches.

MongoDB holds by default a field for each document, called `_id`. The field represents the identifier of a document and is unique, acting as a primary key. The default type of its value is *ObjectId* with 12 bytes. Its length consists of 4-byte timestamp value based on ObjectId's generation, a 5-byte random value and a 3-byte incrementing counter which is initialized to a random value. For the `_id` field, MongoDB maintains a single-field index which cannot be dropped.

3.2 Indexing Spatial and Spatio-temporal Data

Two variants of spatial indexes are supported in MongoDB; a *2d* index, which manages data on a two-dimensional plane, and a *2dsphere*, which calculates geometries on the surface of the earth.

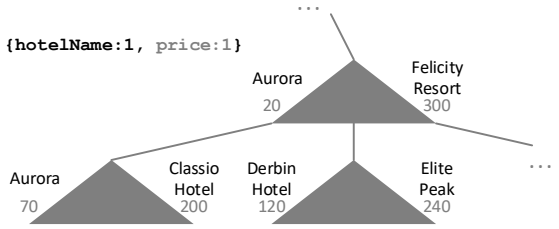


Figure 2: Example of a compound index on fields `hotelName` (string) and `price` (integer). The documents are organized based on `hotelName` and then based on `price`.

Both of them are applied on fields whose values hold spatial data. The values must be either *GeoJSON* objects or *legacy coordinate pairs* which is a representation of the longitude and latitude values either with the usage of two-sized array or by embedding the elements in a document. The spatial indexing mechanism in MongoDB is based on GeoHash[1], where a hierarchical subdivision of the 2D spatial domain using multiple layers takes place. The cells that result from the division of the space are represented by bits. The more bits that represent a space, the higher the precision of the respective location. The GeoHash values that are stored in the index consist of 26 bits by default. They can be set up to 32 bits, performing better for spatial queries, but at the expense of occupying more space in memory. A spatial index can be combined with another field by means of the compound index.

Unfortunately, indexing spatio-temporal data is not directly supported in MongoDB. As a result, our premise would be to build a compound index over the fields storing the spatial information and the temporal information respectively.

3.3 Sharding

Sharding refers to data partitioning and assigning the obtained partitions to MongoDB servers, also called *shards*. Specifically, when sharding a MongoDB collection, its documents are distributed across shards based on a *shard key*. When defining a shard key, MongoDB separates the range of shard key values into smaller non-overlapping ranges with continuous keys. Each of these ranges are associated with a *chunk* and contain a subset of the sharded collection. Also, a chunk has a configurable size which is 64MB by default, and if exceeded, it is split.

The configuration of small-sized chunks leads to a more even distribution of data. However, migrations become more frequent, adding overhead to the network and to the query routing layer (known as *mongos*). Large-sized chunks enforce fewer migrations at the expense of a less even distribution of data. MongoDB achieves load balancing through the (re-)distribution of chunks among shards. The *Balancer* runs in the background so as to migrate chunks across the shards, targeting to achieve an even distribution of chunks in the cluster.

Apart from the definition of the shard key, the sharding operation of a collection requires the strategy type which can be either *range* or *hashed*. With *range sharding*, it is highly probable that documents with similar shard keys will be in the same chunk or shard, as depicted in the example in Fig. 3. This enables

[1] https://docs.mongodb.com/manual/core/geospatial-indexes/

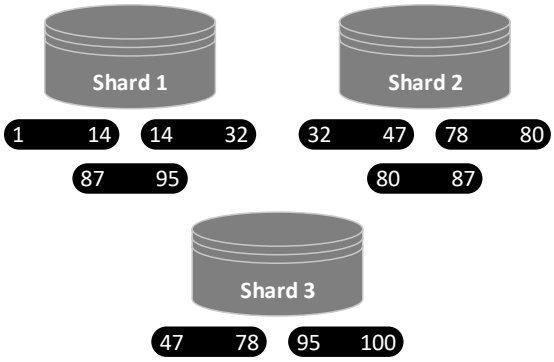


Figure 3: Instance of chunks' distribution that result from a range sharded collection, on a field that contains values from 1 to 99.

routing range queries to specific shards only. On the other hand, in *hashed sharding*, chunks or shards are unlikely to contain documents with similar shard key values. This may serve well for cases where broadcast operations are preferable.

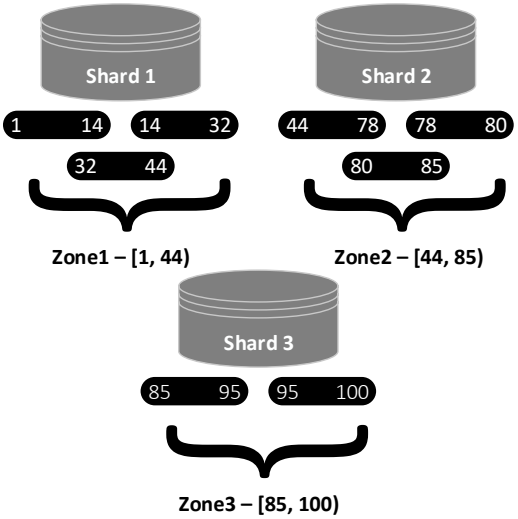


Figure 4: Instance of chunks' distribution that result from the definition and assignments of zones: $[1, 44)$**,** $[44, 85)$**, and** $[85, 100)$ **on the shard key.**

Sharding imposes the creation of a single or a compound index on each shard, based on the field/s of the shard key. MongoDB supports manual grouping of documents based on ranges of shard key values through the concept of *zones*. A zone can be associated with any shard. Similar to chunks, zones have lower inclusive and upper exclusive boundaries and their covering ranges do not overlap. By associating a zone with a shard or shards in an already sharded collection, the cluster migrates the affected data to the respective zones, if a collection is already sharded. Division of chunks may occur so as to follow the data distribution scheme determined from zone(s). If zones are set up before sharding a collection, chunks are created for the defined zone ranges (and additional chunks if necessary) to cover the entire range of the shard key values. Fig. 4 shows an instance of storing the data using zones. In the figure, each shard is assigned with one zone, thus shards maintain more contiguous key ranges.

4 INDEXING AND PARTITIONING SPATIO-TEMPORAL DATA

In this section, we present two alternative approaches for indexing spatio-temporal data, with different implications on partitioning and eventually on the performance of query execution. Our work focuses on point data, which covers many life real-applications, and we leave other data types (such as polylines and polygons) for future work. The first approach relies on the built-in features of the spatial extension of MongoDB for indexing and time-based sharding, therefore we consider it as a baseline solution. Then, we propose an alternative approach that adopts the Hilbert curve to map data to 1D values that will be stored and indexed as a single field, also enabling sharding based on spatio-temporal criteria.

4.1 Baseline Solution

4.1.1 Indexing. The baseline solution for efficient querying of spatio-temporal data stored in MongoDB is via the usage of a *compound* index that is built on the spatial and temporal fields of the documents. Consider the following document structure:

```
{
    "_id": 1,
    "location": {
        "type": Point,
        "coordinates": [37.983810, 23.727539]
    },
    "date": ISODate("2018-10-01T08:34:40.067Z"),
    ...
}
```

The spatial field (`location`) is supported by the built-in *2dSphere* index, and is combined under the *Compound* index with the `date` field. As a result, two possible solutions for spatio-temporal indexing can be designed by exploiting directly the built-in features of MongoDB, which differ on the order of fields: (`{location, date}` and `{date, location}`). The former approach favors the spatial dimension and organizes the buckets of the temporal dimension under the ranges encoded by GeoHash values. The latter approach favors the temporal dimension and organizes the buckets of the encoded GeoHash values under the ranges of the temporal values. As a result, the first approach works well for queries with high selectivity in the spatial dimension, whereas the second approach is more suitable for queries with high selectivity in the temporal dimension.

4.1.2 Sharding. In order to adapt the baseline solution to a distributed environment, a sharding key needs to be defined. We opt for setting as shard key the `date` field. In this way, the spatio-temporal queries are expected to access only specific shards based on the temporal constraint, thus avoiding broadcasting, i.e., routing the query to all shards. In MongoDB, broadcast operations occur if a query's field constraints are not found in the shard key[2]. It should be mentioned that the spatial field cannot participate in the definition of the shard key, since the current version (4.4) of MongoDB does not support a 2dsphere indexed field to be a shard key or part of it.

[2]Broadcast operations may occur also for queries that include the shard key, depending on factors such as the data distribution on nodes and the query selectivity.

By setting the date field as the shard key, it is very likely that the documents will be distributed evenly among the shards, thus resulting in a load-balanced cluster. Chunks are unlikely to become *jumbo* (i.e., this refers to chunks that cannot be split, despite becoming too large), because the values of the documents that represent the temporal information ordinarily are of high cardinality. However, it is still disturbing that the spatial dimension cannot be used for sharding, since this would benefit queries with spatial constraints. Instead, such queries will inevitably be routed to all nodes that contain data that overlap with the temporal constraint of the query.

It should also be noted that an index is automatically constructed on the shard key in MongoDB. Thus, the choice of the date field for shard key results in two indexes: an index on the temporal information and the compound index on space time. During query processing, the query optimizer is responsible for deciding which index is going to be used.

4.1.3 Data locality and zones usage. The baseline solution can only guarantee data locality at the temporal level. This means that documents associated with locations that exist near in space will only be stored on the same node if they have similar timestamps. Still, neighboring temporal intervals (corresponding to chunks) may be stored in different nodes, due to the allocation of chunks to nodes. To improve this shortcoming, we can define zones on the sharding key (date), which will force neighboring temporal intervals to be stored on the same node, thus improving data locality with respect the temporal dimension.

In summary, sharding on the date field has the following two drawbacks: (i) spatio-temporal queries may be routed to nodes that fulfill the temporal range constraint but do not contain any data that satisfy the spatial constraint, and (ii) queries that are selective in the spatial dimension but refer to a large temporal interval are forwarded to many of nodes of the cluster.

4.2 Solution based on Hilbert SFC

4.2.1 Indexing. Instead of relying on the built-in spatial indexes, our approach is based on the 1D mapping of data by means of the Hilbert space-filling curve. Specifically, for each document, the 1D Hilbert value is determined given its longitude and latitude value, and then it is included as a new field (hilbertIndex) that stores this value (of type long), as shown in the example below.

```
{
    "_id": 1,
    "hilbertIndex": NumberLong(2345),
    "location": {
        "type": Point,
        "coordinates": [37.983810, 23.727539]
    },
    "date": ISODate("2018-09-12T12:15:17.777Z"),
    ...
}
```

Even though any 1D mapping could be employed, we select the 1D mapping values based on the Hilbert space-filling curve, as it has been shown to exhibit nice clustering and locality properties [14]. This means that two documents with values of hilbertIndex that differ slightly correspond to objects that are spatially close to each other.

4.2.2 Sharding. Given this new spatial field (hilbertIndex) and the date field, we set the shard key as {hilbertIndex, date}, thus imposing spatio-temporal partitioning of data to nodes. Consequently, the formation of the chunks is based both on spatial and temporal information and each of them contains documents that exist in specific spatial cells (1D values) for a certain time period. In case of spatio-temporal skewness in the data, chunks are unlikely to become *jumbo* because of the cardinality of the temporal field. The hilbertIndex field is more prone to skewness, as it consists of finite long values and some of them may appear with high frequency (i.e., correspond to frequently visited locations). Thus, when a chunk surpasses its configured size due to a unique hilbertIndex value, it is split on the temporal dimension. The two new chunks will refer to the same 2D region, covering different and non-overlapping time spans. The definition of {hilbertIndex, date} as the shard key creates by default a compound index for these fields on each shard. The index constitutes a spatio-temporal index that uses the spatial field first and then the temporal.

When querying such an index, not only the expected spatial and temporal constraints are included in the query as in the baseline approach, but also a constraint is added on the hilbertIndex field. This constraint practically includes the set of spatial cells that need to be examined, because they intersect with the query. More concretely, such a query has the following document representation in MongoDB:

```
{ $and: [
    { location: { $geoWithin: { $geometry: { type : Polygon,
        coordinates: [ [ [23.7397867, 37.9698929],
         [23.7492228,37.9698929], [23.7492228, 37.9761557],
        [23.7397867,37.9761557 ], [23.7397867, 37.9698929] ] ]
    } } } },
    {date: { $gte: ISODate("2019-04-18T12:15:14.002Z") } },
    {date: { $lte: ISODate("2019-04-24T07:34:43.777Z") } },
    { $or: [
        { hilbertIndex: { $gte: 7865, $lte: 12869 } },
        { hilbertIndex: { $gte: 13192, $lte: 13210 } },
       { hilbertIndex: { $in : [7794, 7799, 7856, 12911] } }
    ] }
] }
```

Similarly to the baseline approach, the spatio-temporal query uses the MongoDB *$geoWithin* operator on the field that stores the location as GeoJSON object. Also, the query specifies a specific temporal range using the *$gte* and *$lte* operators on the temporal field. However, in the Hilbert-based approach, an additional constraint is posed on the specific spatial cells. Consecutive values of cells are expressed as ranges, whereas non-consecutive cell values are included as individual values. For this purpose, an additional disjunctive operator (*$or*) is used in the query that contains *$gte* and *$lte* operands to represent the ranges, and an *$in* operator to include the individual cells.

4.2.3 Data locality and zones usage. The exploitation of both spatial and the temporal information of documents for sharding leads to a distribution of data to nodes that preserves the *spatio-temporal* data locality. It is highly probable for the documents that are both spatially and temporally near to be stored on the same node. However, as already explained, this is not guaranteed for data that belong to different chunks. This is because MongoDB does not guarantee (by default) for each of the shards to store chunks with continuous ranges.

Thus, data locality can be improved by defining zones. With zones, documents with consecutive Hilbert values (disregarding the temporal dimension) are likely to be placed in the same shard; this is applied by defining and assigning zones to shards, handling documents with specific ranges of Hilbert values. The

fact that the zones are defined only on the spatial information of the documents cannot guarantee temporal locality, but only for documents with nearby spatial locations. In contrast to the baseline solution, in this approach, the spatial dimension affects the number of the nodes that are to be accessed during query processing.

4.2.4 Zones configuration. In order to assign a pre-defined range of key values to a shard, a zone is specified by means of a range of shard key values or a prefix of shard key values. Zones are then assigned to certain nodes. This enables manual partitioning of data in a controlled manner. In order to achieve data locality regarding the spatio-temporal information of the documents, we define as many zones as the number of available shards, and assign one zone per shard.

The configuration of zones for the Hilbert-based approach is performed on the `hilbertIndex` field (whereas the shard key consists of the spatial and temporal field). Specifically, by using the *$bucketAuto* aggregation pipeline stage of MongoDB, we get the ranges of n buckets, where n is the number of shards. The boundaries of buckets are formed with the goal of even distribution of documents to buckets. The ranges of the zones contain documents that have a specific Hilbert value, without taking account its temporal part. In other words, the zones contain documents that are located in specific cells for the whole timespan in which they belong to. The resulting zones may not contain exactly the same number of documents due to spatial data skew, but we manage to preserve spatial locality.

For the baseline approach, the configuration of zones is performed on the `date` field that constitutes the shard key, using again the *$bucketAuto* aggregation pipeline stage. Each of the buckets, contains the same number of documents. Based on these ranges, the respective zones are created, and each one is assigned to a single shard.

5 EXPERIMENTAL EVALUATION

In this section, the results of the experimental evaluation using both real-life and synthetic data are presented.

5.1 Experimental Setup

Platform. All experiments were performed in Okeanos-Knossos[3], an IaaS platform which is built and supported by GRNET[4] for research purposes. Okeanos-Knossos is a cloud service that provides virtual computing and storage services to the Greek research and academic community. We have engaged from the cloud service the following resources: 17 virtual machines, 68 CPUs, 136GB RAM, and 2.00 TB disk space.

MongoDB deployment. We deployed MongoDB (v 4.0.12) on 17 virtual machines. From this set of nodes, 12 of the VMs were used as (primary) shards, 3 of them as configuration servers and the remaining 2 as query routers. Replica shards were not used since the availability feature which mainly relates to node failures is beyond the scope of this experimental study.

Each VM is equipped with 8GB RAM, x4 CPU cores and 30GB system disk, running Ubuntu 16.04.6 LTS operating system. The VMs that serve as shards have a mounted disk drive of 102GB size. The VMs that serve as query routers have a mounted disk drive of 145GB size. The offered attachable disk of Okeanos-Knossos platform is based on the Ceph[5] storage system, which is a distributed block level storage.

By default, MongoDB uses the WiredTiger storage engine that integrates compression for collections and indexes. The compression on the collections is achieved through block compression, supported through the snappy library. At the level of indexes, prefix compression is used.

Data sets. Two types of data sets are used for the experimental evaluation; real-life (R) and synthetic (S). The R set used in the experimental evaluation is a subset of a proprietary data set that is provided from a fleet management provider in Greece. The subset used in our experiments is 13.7GB in the form of CSV files, containing $15,210,901$ records in total. The data set contains trajectories of vehicles within Greece for a period of five months (July to November 2018). Particularly, each record constitutes a GPS trace of a specific vehicle and is comprised of 75 values. The values represent information about the vehicle, its position in space and time, the prevailing weather conditions, the road network and the nearest points of interest. The coordinates of the minimum bounding rectangle of the data set are: [(19.632533, 34.929233), (28.245285, 41.757797)]. We also use larger portions of the same data set for our scalability study in Section 5.4, by including more vehicles in the same spatio-temporal bounding box.

The S set is a synthetic spatio-temporal data set, which contains twice as many records as the R data set. It consists of two CSV files where each one contains 4 columns (`id`, `longitude`, `latitude` and `date`). The values of each column are generated at random within predefined ranges and following the uniform distribution. Its size is 1.6GB. The timespan of the synthetic data set is the half of the covering time period of the R data set (i.e., it spans 2.5 months) and covers spatially a smaller area than R. Specifically, the minimum bounding rectangle of the synthetic data set is approximately 1.54% of the minimum bounding rectangle area of the R set. The lower and upper coordinates of this rectangle are: [(23.3, 37.6), (24.3, 38.5)].

Queries. In order to assess the performance of the approaches, two categories of spatio-temporal queries are specified; those with small and big spatial constraint, respectively. The categories will be stated henceforth as Q^s (small) and Q^b (big) correspondingly. Each category contains 4 queries with increasing temporal constraint: Q_1^x covers 1 hour, Q_2^x 1 day, Q_3^x 1 week and Q_4^x covers 1 month, where $x \in \{s, b\}$. The queries do not overlap on the temporal dimension; instead, each one pertains to a discrete time span.

The spatial constraint of the small queries category is defined as a rectangle with the following lower and upper bounds; [(23.757495, 37.987295), (23.766958, 37.992997)]. The spatial rectangle covers 526km^2. The spatial constraint of the big queries category is also defined as a rectangle, approximately $2,603$ times larger than the covering area of the small queries category. Its lower and upper bound coordinates are [(23.606039, 38.023982), (24.032754, 38.353926)], covering $1,369,107$km^2.

Tables 2 and 3 report the number of retrieved documents for each small and big queries, respectively. Clearly, small queries are selective and retrieve relatively few documents. This corresponds to queries for constrained space and time dimensions, aiming at retrieval of very specific records. Instead, big queries, return large result sets, and correspond to analytic queries that retrieve

[3] https://okeanos-knossos.grnet.gr/
[4] https://grnet.gr/
[5] https://ceph.io/

	Number of retrieved documents			
Data set	Q_1^s	Q_2^s	Q_3^s	Q_4^s
R	2	34	877	3,829
S	3	22	239	783

Table 2: The results of small queries for the real and the synthetic data set

	Number of retrieved documents			
Data set	Q_1^b	Q_2^b	Q_3^b	Q_4^b
R	580	5,640	113,890	431,788
S	2,575	61,193	608,685	1,891,291

Table 3: The results of big queries for the real and the synthetic data set

large portions of the data set in order to perform some large-scale data analysis task.

Metrics. The evaluation of the performance of the approaches is based on the following metrics;

- *Average execution time*: corresponds to the execution time required for processing the query and returning the query result (averaged over queries).
- *Documents examined*: corresponds to the maximum number of documents that were accessed on any node during query processing. This indicates the number of documents that need to be fetched from disk during query processing. We use the maximum as it corresponds to the highest cost induced on any node, so it affects the execution time.
- *Keys examined*: shows the maximum number of keys examined in the underlying index over all nodes, in order to find the necessary documents on disk. This indicates the cost paid during query processing induced by accessing the index.
- *Nodes*: reports the number of nodes that were accessed during the execution of a query. This corresponds the subset of nodes that participated in query processing.

Methodology. Based on the description of the subsections 4.1 and 4.2, we evaluate the following individual approaches which cover variants both for indexing and partitioning:

- *bslST*: sharding based on time, and local indexing on shards using compound index (`location`, `date`), where location is based on 2dsphere index.
- *bslTS*: sharding based on time, and local indexing on shards using compound index (`date`, `location`), where location is based on 2dsphere index.
- *hil*: range sharding based on Hilbert curve in 2D (with 13-bit precision) and on time. The applied Hilbert curve covers the entire globe. For local indexing on shards, a compound index (`hilbertIndex`, `date`) is used for each node.
- *hil**: range sharding based on Hilbert curve in 2D (with 13-bit precision) and on time. The applied Hilbert curve is limited to the spatial region of the data set. For local indexing on shards, a compound index (`hilbertIndex`, `date`) is used for each node.

With the *bsl* term we will refer to the sharding of the bslST and bslTS approaches which is common for both of them. These approaches differ only on their indexing part. Furthermore, the hil method competes on equal terms bslST and bslTS approaches, by taking account of the same spatial extent (whole world) and by using 26 bits for storing the geospatial information of each document on the respective indexes. We also tried hil* to investigate if the use of a curve with the same number of bits but on restricted spatial extent has any notable effect on performance.

The evaluation methodology followed in this experimental study is described as follows. Each of the queries of the two categories are executed 30 times, so as to ensure that the measurements of the performance time of queries are in warm state (where indexes have been already loaded in-memory). The execution time is calculated by averaging the last 10 executions of each query.

When switching from one approach to the other, MongoDB is re-installed from scratch so as to have a new deployment which will contain only one sharded collection. For each approach, data loading takes place. Both of the query routers are exploited for this purpose and perform bulk insertion.

The approaches are tested under range sharding with different settings: (i) with the default formation of the chunks, without intervening on the distribution of the data over the shards (Section 5.2, Figs. 5–8), and (ii) with the definition of zones where data is grouped to shards given some pre-determined ranges (Section 5.3, Figs. 9–12).

5.2 Evaluation of Approaches

***hil* vs. *bsl*.** For both of the baseline variants (bslST and bslTS), the number of nodes to which a query is routed increases when the temporal constraint of a spatio-temporal query is increasing, regardless of its spatial extent (Fig. 5c, 6c, 7c, 8c). This is not effective for big queries that cover a large spatial area and refer to a short time period (such as Q_1^b and Q_2^b in Fig. 6d and Fig. 8d). In that case, a small number of nodes are burdened with search operations, as a result of the multiple 1D mapping values that represent the area. This is reflected in Figs. 6a, 6b, 8a and 8b where the maximum number of examined keys and documents are many more than in hil. The same applies for the Q_3^b query, served by 2 and 6 nodes in the R and S set respectively, fewer than the exploited nodes of hil method. The query Q_4^b uses in the baseline approaches 3 nodes more than in hil for both sets, with more maximum examined keys and documents. In summary, hil outperforms the baseline methods in terms of execution time in the case of big queries.

Concerning the small queries, Q_1^s and Q_2^s perform better for hil in R set, since the maximum examined keys are fewer than in bslST and bslTS (Fig. 5a). The same applies for the S set in terms of the performance, but for the Q_2, a few more documents are examined in hil than bslST (Fig. 7a). Still, Q_2^s performs a little better in *hil* than bslST, as 1 node is used, thus without having the small overhead of merging the results from the shards. The opposite happens for the maximum examined documents (Fig. 5b and 7b) for the same queries on the two sets, but the differences are smaller than the respective cases of the examined keys. For queries Q_3^s and Q_4^s, bslST outperforms hil. This happens because more nodes contribute to the execution of the queries. Since the queries are spatially small, fewer nodes are used in the hil method.

hil* vs. *hil*.** The performance of hil and hil* methods differ in favor of hil for the big queries in both sets (Fig. 6d and 8d). The same does not apply for the small queries (Fig. 5d and 7d), excluding Q_1 for both sets and Q_2 for S set. In practice, hil* uses higher precision in the spatial domain than hil, having more

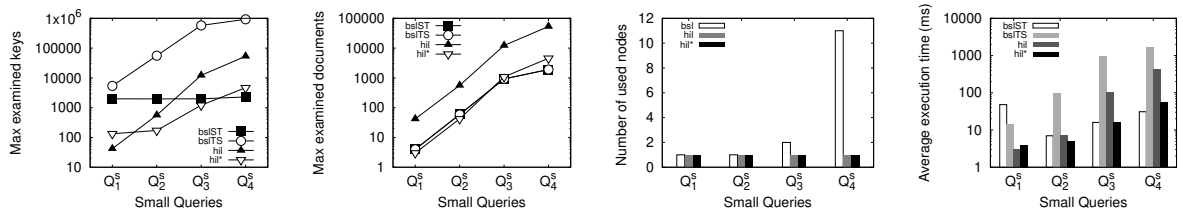


Figure 5: Default sharding ranges: Results for small queries and real (*R*) data

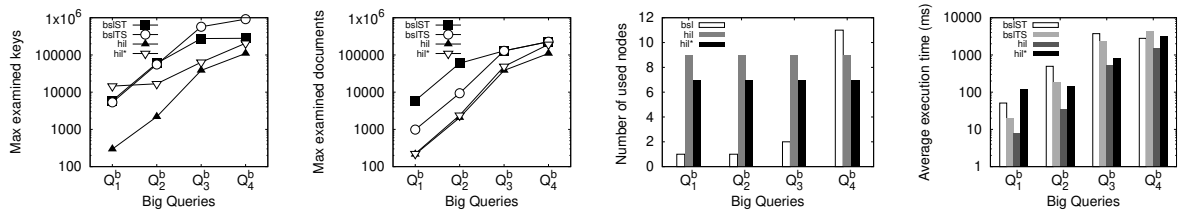


Figure 6: Default sharding ranges: Results for big queries and real (*R*) data

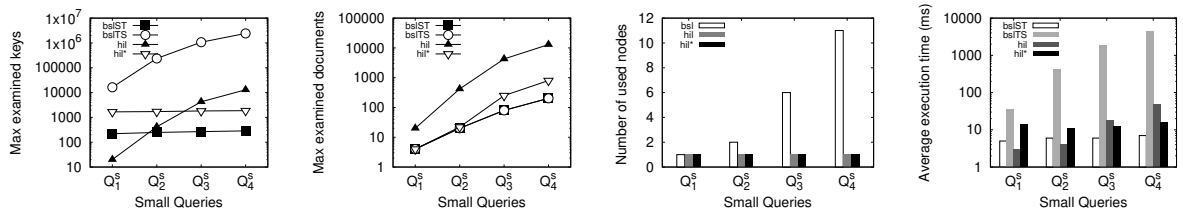


Figure 7: Default sharding ranges: Results for small queries and synthetic (*S*) data

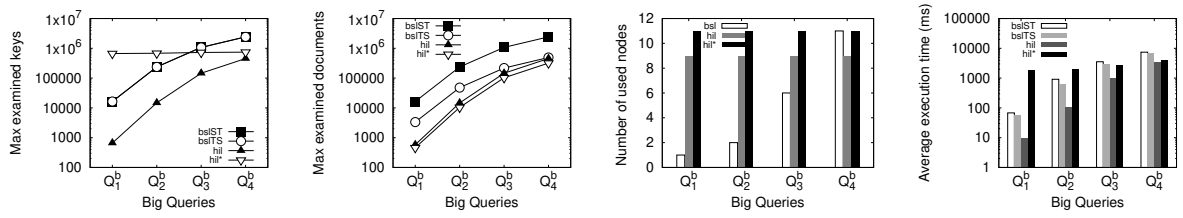


Figure 8: Default sharding ranges: Results for big queries and synthetic (*S*) data

indexed values on its spatial part. This results to a compound index with more buckets but with fewer elements on each bucket. It is expected, that such an index will perform better for spatio-temporal queries that cover a small area, than an index with fewer values on the spatial part. The difference in the performance is greater when the temporal dimension is increasing, since many of the buckets of the hil* that fulfill the spatial criteria will not be accessed, because of their temporal boundaries, thus resulting to fewer examined documents.

Discussion. In general terms, our experiments with the default distribution settings validate our intuition that the integration of the spatial information on the sharding offers performance gain. This is especially evident for big queries where hil outperforms bsl. In the case of small queries, bsl performs better but at the expense of using more nodes for query execution due to the lack of data locality. This is going to have a negative effect in a real system that processes thousands of queries at the same time, since it would require that all nodes need to participate in the execution of each query, which is not scalable.

5.3 Evaluation of Approaches with Zones

In this set of experiments, we only use hil (not hil*) since we did not observe significant performance improvements. Many of the experimental observations of Section 5.2 stand also for the case of grouping data via zones. For instance, for all of the big queries (Figs. 10d and 12d), hil outperforms both bslST and bslTS, because the maximum number of examined documents is smaller. Moreover, the required execution time of small queries

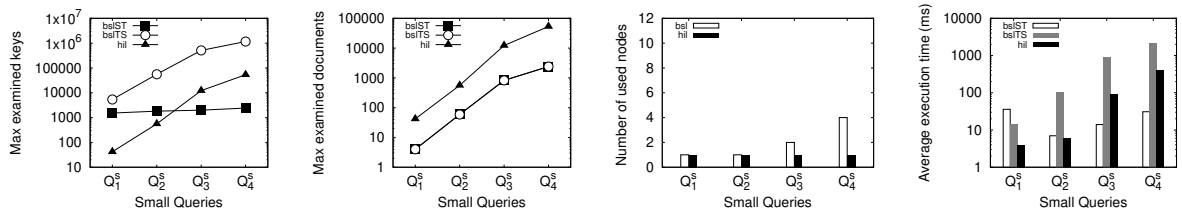


Figure 9: Zone ranges: Results for small queries and real (*R*) data

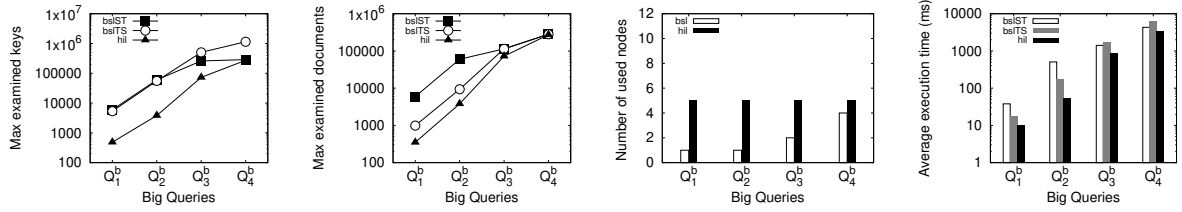


Figure 10: Zone ranges: Results for big queries and real (*R*) data

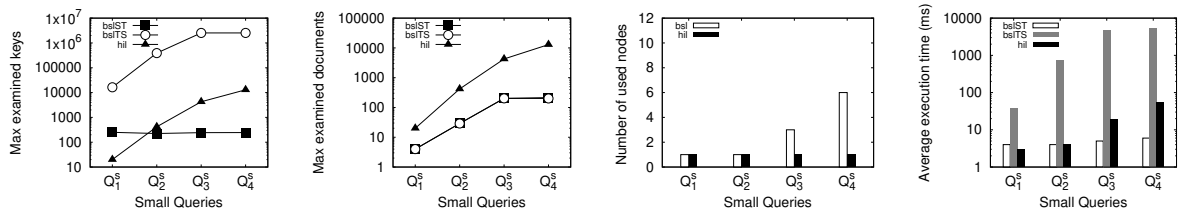


Figure 11: Zone ranges: Results for small queries and synthetic (*S*) data

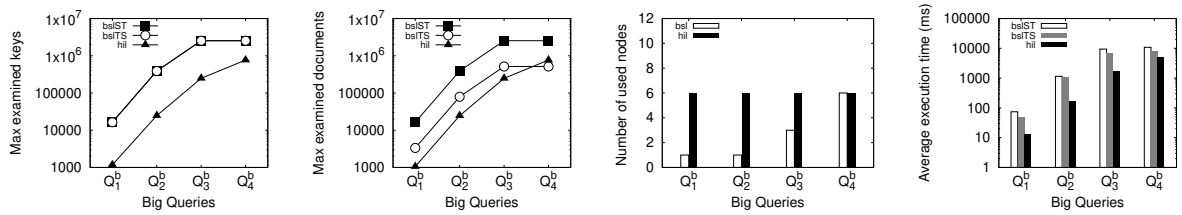


Figure 12: Zone ranges: Results for big queries and synthetic (*S*) data

with a large timespan (Q_3 and Q_4) is smaller for the bslST than hil, since more nodes are exploited, resulting to a smaller number of maximum examined keys and documents for both sets (Figs. 9a, 9b, 11a and 11b).

It should be underlined that in the cases in which more than two node were exploited for the performance of a query in the experiments of Section 5.2, their corresponding application over of the pre-defined zones use fewer nodes. This is expected because of the grouping of zones on shards. For those cases, the query performance drops to a small degree. Additionally, better data locality is offered via zones as data is moved to specific shards, adhering to the specified ranges.

Discussion. The usage of zones for each of the approaches confirms that data locality is better than in the case of default distribution of the documents among the shards. This is demonstrated by the fact that fewer (or, in some cases, the same number) of nodes take part in the processing compared to the case of no zones.

In the baseline approaches, the Q_4 big queries in both R and S data sets consume more time for their execution, as fewer nodes are involved in query processing. The performance degradation is also noticed for the Q_2^b and Q_3^b in the S set, whereas the respective queries gain performance in the R data set. Query Q_1^b gains slightly in performance in all cases. Regarding the small queries, in the bslST approach, query execution remains practically the same. In the bslTS approach, the performance of Q_4^s gets worse for both R and S data sets. The same applies for Q_3^s and Q_2^s, but in

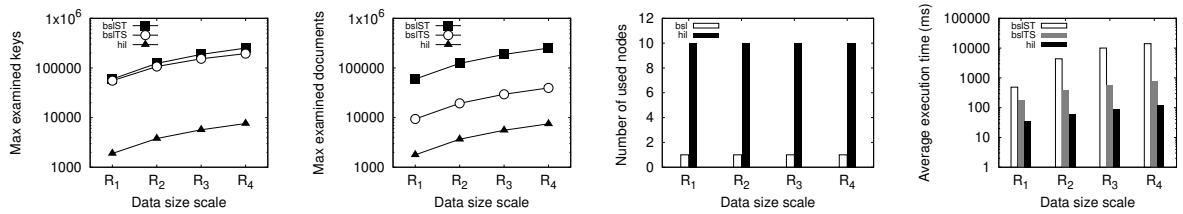


Figure 13: Scalability study for default sharding ranges: Results for Q_2^b query on real data

the S data set. The performance of the remaining queries remains the same.

For the Hilbert-based approach, the usage of zones affects only the execution of Q_2^b, Q_3^b and Q_4^b big queries, increasing their execution time as a smaller number of nodes are involved in query processing. An extra number of keys and documents have to be searched. This is validated in both R and S data sets. The remaining queries have similar performance, except for small queries Q_3^s and Q_4^s in the R data set. Their execution time is slightly decreased despite that the same number of nodes (namely one node) is still exploited with zones. These queries have been favored due to the change in the distribution of data by using zones.

5.4 Scalability Study

In this experiment, we study the scalability of our approach for larger data sets. For this purpose, we use larger portions of the real R data set. In particular, we use R_1 to denote the data set used so far, whereas R_2, R_3, and R_4 correspond to larger data sets, by a scale factor of x2, x3, and x4 respectively. Table 4 reports the size (in GB) and number of documents for each data set. Notice that we keep the same spatio-temporal bounding box for the data set, and we obtain larger instances of the data set by adding data from more vehicles. We select query Q_2^b in order to study its performance when the size of the data set is increased. Also, Table 5 reports the number of results for Q_2^b for the different data sets ($R_1 - R_4$), showing that the same query retrieves more results as the scale factor of the data set increases.

	Data sets with scale factor			
Data set info	R_1	R_2	R_3	R_4
Size (GB)	40.8	83.87	127.27	171.59
#documents (M)	15.2	31.4	47.7	63.9

Table 4: Information for instances $R_1 - R_4$ of the real data set for different scale factor

	Data sets with scale factor			
Query	R_1	R_2	R_3	R_4
Q_2^b	5,640	11,792	17,840	23,854

Table 5: Number of results for query Q_2^b per scale factor of the real data set

Fig. 13 demonstrates that our approach scales gracefully as the size of the data is increased. This is particularly evident when considering the execution time in Fig. 13d. Perhaps more importantly, the gain of hil over the baseline methods increases with the size of the data, which favors the performance of hil for larger data collections. Figs. 13a and 13b show that hil needs to access 1-2 orders of magnitude fewer documents and 2 orders of magnitude fewer keys respectively. When comparing the two baselines, bslTS performs better than bslST, since fewer documents are examined in the query's refinement phase. Recall that query Q_2^b is selective on its temporal dimension (covering only one day) and thus it is expected that bslTS will perform better than bslST.

Discussion. This experiment demonstrates that the proposed approach (hil) sustains its benefits over the baseline methods when the size of the underlying data set is increased. This indicates that our approach is scalable with data size.

6 CONCLUSIONS

In this paper, we provide an in-depth study of spatio-temporal query execution in NoSQL stores, focusing on MongoDB due to its popularity and support for spatial data. Our study indicates that existing NoSQL stores do not natively support spatio-temporal data, despite the ever-increasing number of applications that produce massive spatio-temporal data daily. We also show that a baseline approach based on built-in spatial indexes leads to suboptimal performance in several cases. More importantly, we investigate on the underlying reasons for this impact on performance, and we elaborate on indexing and sharding techniques. Furthermore, we propose an alternative approach which exploits the Hilbert curve to map data to 1D values, which are then: (a) indexed using a standard B-tree, and (b) used for partitioning the data in a way that preserves spatio-temporal data locality in shards. We conclude that this has an effect on the number of nodes storing data that participate in query execution. Our extensive experiments using both real-life and synthetic data in a cluster of nodes support our conclusions.

With respect to future work, we believe that big data developers that work on spatio-temporal data will find interest in our work, especially towards optimizing the performance of their applications. Also, we expect that future releases on NoSQL stores will provide support for spatio-temporal data, therefore our work is a small contribution in this direction. As a result, we intend to extend our approach and investigate other methods for indexing and partitioning spatio-temporal data in distributed NoSQL systems. Also, extending our work towards supporting more complex data types (polylines and polygons) is of interest. Last, but not least, we would like to expand our study using a workload of queries, and propose an adaptive, workload-aware mechanism for indexing and partitioning.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780754 (Track&Know project), and the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT) under grant agreement No HFRI-FM17-81 (Chorologos project).

REFERENCES

[1] Dominik Bartoszewski, Adam Piórkowski, and Michal Lupa. 2019. The Comparison of Processing Efficiency of Spatial Data for PostGIS and MongoDB Databases. In *Beyond Databases, Architectures and Structures. Paving the Road to Smart Data Processing and Analysis - 15th International Conference, BDAS 2019, Ustroń, Poland, May 28-31, 2019, Proceedings (Communications in Computer and Information Science, Vol. 1018)*, Stanislaw Kozielski, Dariusz Mrozek, Pawel Kasprowski, Bozena Malysiak-Mrozek, and Daniel Kostrzewa (Eds.). Springer, 291–302.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 322–331.

[3] Norbert Beckmann and Bernhard Seeger. 2009. A revised r*-tree in comparison with related index structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 799–812.

[4] Rick Cattell. 2010. Scalable SQL and NoSQL data stores. *SIGMOD Rec.* 39, 4 (2010), 12–27.

[5] Kristina Chodorow. 2013. *MongoDB: The Definitive Guide.* O'Reilly Media, Inc.

[6] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.

[7] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Comput. Surv.* 51, 2 (2018), 40:1–40:43.

[8] Cédric du Mouza, Witold Litwin, and Philippe Rigaux. 2009. Large-scale indexing of spatial data in distributed repositories: the SD-Rtree. *VLDB J.* 18, 4 (2009), 933–958.

[9] Miaoran Duan and Gang Chen. 2015. Assessment of MongoDB's spatial retrieval performance. In *23rd International Conference on Geoinformatics, Geoinformatics 2015, Wuhan, China, June 19-21, 2015*, Shixiong Hu (Ed.). IEEE, 1–4.

[10] Xuefeng Guan, Cheng Bo, Zhenqiang Li, and Yaojin Yu. 2017. ST-hash: An efficient spatiotemporal index for massive trajectory data in a NoSQL database. In *25th International Conference on Geoinformatics, Geoinformatics 2017, Buffalo, NY, USA, August 2-4, 2017*. IEEE, 1–7.

[11] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, Beatrice Yormark (Ed.). ACM Press, 47–57.

[12] Anand Padmanabha Iyer and Ion Stoica. 2017. A scalable distributed spatial index for the internet-of-things. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 548–560.

[13] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, and Dimosthenis Anagnostopoulos. 2019. Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data. In *Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference, EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019 (CEUR Workshop Proceedings, Vol. 2322)*, Paolo Papotti (Ed.). CEUR-WS.org.

[14] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Trans. Knowl. Data Eng.* 13, 1 (2001), 124–141.

[15] Georgios M. Santipantakis, Apostolos Glenis, Kostas Patroumpas, Akrivi Vlachou, Christos Doulkeridis, George A. Vouros, Nikos Pelekis, and Yannis Theodoridis. 2020. SPARTAN: Semantic integration of big spatio-temporal data from streaming and archival sources. *Future Gener. Comput. Syst.* 110 (2020), 540–555.

[16] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, Peter M. Stocker, William Kent, and Peter Hammersley (Eds.). Morgan Kaufmann, 507–518.

[17] Chaowei Yang, Keith C. Clarke, Shashi Shekhar, and C. Vincent Tao. 2020. Big Spatiotemporal Data Analytics: a research and innovation frontier. *Int. J. Geogr. Inf. Sci.* 34, 6 (2020), 1075–1088.

A ADDITIONAL EXPERIMENTAL RESULTS

We provide additional details on the empirical evaluation, including the bulk loading technique that we use, details on index usage, as well as information of query distribution to nodes and index size.

A.1 Data Loading

Data Loading. Both of the nodes that act as query routers, store half of the CSV files of the data sets on their disks. Data loading to the MongoDB store is carried out by accessing the CSV files record-by-record and converting them directly to documents (binary JSON - BSON format). The conversion process does include the formation of a GeoJSON object as a value of the `location` field based on the longitude and latitude columns of the CSV datasource files. It also includes the addition of the fields with its respective values for all of the rest columns of the CSV files, and the addition of the `_id` field which is handled automatically by the MongoDB client driver. Note that for the Hilbert-based approaches, a further action takes place, related to the calculation of the 1D value. When a document is formed, it is added to a list. The list is used for bulk insertion of documents for performance reasons. The insertion is triggered after a specific batch of elements has been placed in the list, and we use 15K documents as batch size in our experiments.

The size of the R and S set is a marginally smaller in the bsl method than hil and hil* (Table 6), since its documents do not integrate the `hilbertIndex` field, as hil and hil* do. Also, the sizes of the methods in R set are much larger than S set because they incorporate much more information due to the extra fields.

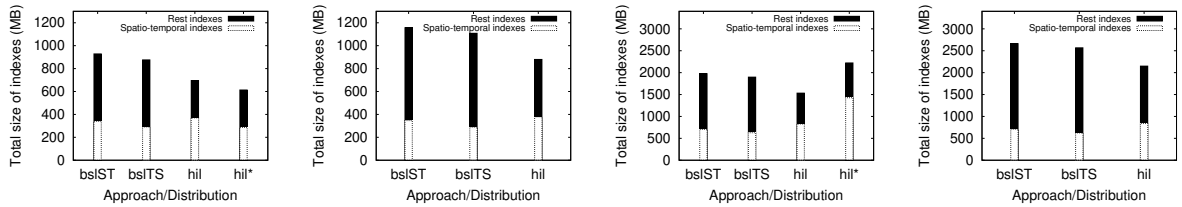
	Approach	
Data set	*bsl*	*hil(*)*
R	40.54	40.8
S	3.62	4.13

Table 6: Data size of real and synthetic data set in MongoDB (in GB)

A.2 Querying the Data

The bslST and the bslTS approaches use as a sharding key the `date` field which results to an additional index per node, apart from the compound index. As a result, there are some cases where the query optimizer chooses to process the spatio-temporal query at hand using the index on `date`, rather than using the compound index. This occurs only in the bslST approach, whereas in the bslTS approach all queries are processed using the compound index. Table 7 reports which index was used during query processing for the bslST approach for all queries and both data sets.

Querying under the hil and hil* approaches, requires the determination the cell of the indexes that cover the queries' spatial extent. Table 8 shows the average time execution of the Hilbert algorithm for the identification of the cell indexes. When applying the hil* methodology, it is reasonable that the algorithm requires more time than hil for specifying the 1D values, as the total searching space is limited to a smaller surface, resulting to increased precision. Furthermore, the execution time in the *S* data set is increased when comparing the respective approaches (except for Q^s for hil) to the *R* set, since the data exists in a smaller 2D space which increases the precision. This adds a burden to the algorithm's operation for identifying the 1D values. The figures that follow in the next subsections, showing query execution time, do not include the time for the determination of the 1D values through the space filling curve.



(a) R set with default distribution **(b) R set with zones** **(c) S set with default distribution** **(d) S set with zones**

Figure 14: Total size of indexes for R and S set with default distribution settings and zones

Distribution	Data set	Q^x	Q^x_1	Q^x_2	Q^x_3	Q^x_4
Default	R	Q^s	●	●	●	●
		Q^b	○	○	●	◐[a]
	S	Q^s	●	●	●	●
		Q^b	○	○	○	○
Zones	R	Q^s	●	●	●	●
		Q^b	○	○	●	◐[b]
	S	Q^s	●	●	●	◐[c]
		Q^b	○	○	○	○

Table 7: Usage of indexes for the bslST approach

●The used nodes exploit the compound index, ○the used nodes exploit the date index, ◐mixed usage of the two indexes among the used nodes - the most delayed node uses the compound index

[a] 3 of the 11, [b] 3 of the 4 and [c] 5 of the 6 used nodes exploit the compound index

	Methodology			
	hil		*hil**	
Data set	Q^s	Q^b	Q^s	Q^b
R	0.05	0.2	0.1	1.8
S	0.05	0.3	0.6	7.6

Table 8: Average time of Hilbert algorithm performance in ms, for finding which 1D values should be searched on the index

A.3 Index Size

The baseline methods have a few more main memory resources requirements than hil for maintaining the indexes among the shards. This is indicated in Figs 14a, 14b, 14c and 14d. Except for the spatio-temporal indexes, bslST and bslTS maintain additionally two indexes which are created by default; one for the *_id* field of every document, and the other for the *date* field, as it acts as the sharding key. Whereas, hil and hil* methods maintain as an additional index apart from the spatio-temporal, only the one that is used for the _id field. Their spatio-temporal indexes preexist by default due to the fact that its fields constitute the sharding key. This, favors especially the hil method, since in all cases, less memory is required for handling all the indexes than bslST and bslTS.

For each of the methods, both in R and S set, the total size of indexes among the shards increases when transiting from the default data distribution to zone ranges. The change in size is affected by the indexes' size that handle the _id field. The insertion of the documents in MongoDB database, takes place under the default distribution settings, and the documents with similar construction time have a larger common prefix part on the _id field. Documents created in similar timestamps that get inserted in the same shard contribute to the occupation of less main memory by the _id index, since MongoDB uses prefix compression. With the definition of zones, the documents that have been already stored in the shards, are shuffled around the cluster, resulting to shards with documents that have a smaller common part. This makes the compression less effective and thus the sizes of _id indexes are increased.

Moreover, the total size of the spatio-temporal indexes does not change significantly when shifting from the default data distribution state to zone ranges. The indexes have approximately the same size. The same applies for the date indexes that exist in bslST and bslTS methods; their size remains approximately the same for both default distribution and zone ranges.

Comparing the hil and hil* total size of indexes, it is noticed that in R set less main memory is required for hil* method (Fig. 14a), whereas the opposite is true for the S set (Fig. 14c). This is reasonable because for the S set, the cardinality of the 1D values for hil* is much greater than hil method when contrasting the same methods for the R set. In R set, many are the documents that have the same 1D value as they are spatially skewed. The values are grouped to buckets, covering each one a specific time period. This saves memory especially when having smaller buckets that still cover a specific time period; this is achieved through hil*. On the other side, in S set, hil* is less efficient in terms of memory occupation than hil because the 1D hilbert values that are handled, are many more as the data is distributed uniformly all over their minimum bound box. This makes the compression less effective either, thus increasing the size of the hil* index.