Efficient QoS-Aware Spatial Join Processing for Scalable NoSQL Storage Frameworks

Isam Mashhour Al Jawarneh, Paolo Bellavista, Antonio Corradi, Luca Foschini, Rebecca Montanari

Dipartimento di Informatica – Scienza e Ingegneria, University of Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
{isam.aljawarneh3, paolo.bellavista, antonio.corradi, luca.foschini, rebecca.montanari}@unibo.it

***Abstract* — Current cloud-enabled NoSQL database frameworks support flexible and scalable storage of huge amounts of data arriving through various and often heterogeneous channels. However, they do not natively provide optimised processing of spatial data, thus making it more difficult to perform accurate data analytics needed in many smart city application scenarios. To improve the performance of spatial data computation in the NoSQL MongoDB storage framework, this paper proposes a novel data partitioning method based on dimensionality reduction. The underlying key idea is to reduce a spatial data representation from multi to single dimensionality, by still maintaining its geometrical meaning and by employing a specific geo-encoding scheme, i.e., a geohash string. In particular, the geohash string is used as a sharding key in order to store geometrically-nearby objects into the same chunks (and consequently into the same shard). In addition, as a distinctive feature, we have extended the MongoDB framework with a custom spatial QoS-aware optimizer that exploits our novel partitioning scheme to support two, typically expensive, types of spatial queries with QoS guarantees. Those queries are containment (and consequently top-N) and proximity. The paper also contributes to the existing literature with extensive experimental results about the performance of both our partitioning method and query optimizer; the reported results show that our solutions outperform baselines by orders of magnitude.**

***Keywords* — Spatial Join, NoSQL, MongoDB, Point in Polygon, Containment, Proximity.**

I. Introduction

Highly dynamic smart city application scenarios require, more than often, to collect massive amounts of geo-referenced datasets from heterogeneous sources [1]. In addition, these data processing workloads typically require storing historical data for future deeper analytics that normally cannot be performed online.

Currently, the most effective way of unifying large and heterogeneous datasets in a consolidated structure is to use NoSQL scalable storage frameworks, such as MongoDB. A drawback, however, is that those systems do not have a native support for effective spatial data partitioning (sharding in MongoDB terms) that enable to distribute spatial data to multiple processing units according to specific application-dependent spatial data criteria. Current NoSQL storage frameworks are general and focus mostly on load balancing by dispatching roughly equal data loads to the storage shards (a.k.a. partitions) regardless of spatial data relationships.

To outline the importance of a proper partitioning support for spatial data, let us consider the case of the collection and analysis of mobility data related to taxicabs, cars, ambulances, and shared bikes within a metropolitan city. That data constitutes a deluge of geo-referenced data (encompassing location representations that incorporate geographical meaning). Interesting analytics include traffic city planning, where decision makers may try to uncover hidden patterns by generating heatmaps (visualized in special dashboards) showing how vehicles move on a daily/weekly/monthly basis during different timeslots. Depending on the analytics results, decision makers can, for instance, "estimate the volume of cars driving in the same city district at the same time" or can "find all taxi trips originated from a given neighbourhood within a city". Such kind of analytics need to know and preserve the real geometrical co-locality of spatial objects during mobility data processing.

Sending those massive amounts of georeferenced mobility datasets to parallel processing and storage computing resources regardless of their spatial characteristic, as in current stock versions of NoSQL storage frameworks, may cause geometrically-nearby objects to be forwarded to potentially far apart processing nodes. As an undesired consequence, aggregating back spatial data for interesting insights (by applying spatial join predicates) requires applying scatter-gather exhaustive searches. On the contrary, sending geographically-close-by objects to the same processing and storage nodes can achieve better time-based QoS goals, i.e., a lower latency and higher throughput, when data need to be correlated for analysis.

In addition, being storage-oriented, NoSQL systems (such as MongoDB) are I/O-intensive. Therefore, minimizing the number of rounds the CPU requires to access the external memory for I/O can significantly improve the overall performance of the spatial query processing algorithm. Other major factors that influence spatial join performance include the availability of spatial indexes.

Along those guidelines, this paper reports the design of efficient methods that act on three factors that may significantly determine the overall performance of spatial joins in distributed

NoSQL (i.e., spatial sharding, indexing, and I/O ratio). Specifically, we have designed an efficient spatial sharding method and a novel query processing optimizer to overcome the difficulty of current NoSQL storage frameworks to efficiently address time-based QoS goals for spatial analytics that intrinsically incorporate spatial joins at scale. Our partitioning method exploits dimensionality reduction, enabling the spatial data structure to be reduced from a multi- to one-dimensional representation (based on geohash encoding). For example, in our solution parametrized longitude/latitude pairs are transformed into geohashes, which are simple strings maintaining the underlying positional information (not precisely but approximately). Spatial points that have the same geohash value are located geometrically in the same location (apart from approximation errors). As a key and original feature of our method, the geohash string is used as a sharding key. Spatial data are split and stored into shards based on their geohash values, thus increasing the probability that geometrically close-by objects end up in the same shard and even chunks (a chunk resembles a bucket that contains many spatial points in MongoDB parlance). This feature contributes to reduce the I/O journeys that the CPU is making to the external memory. An integral part of our sharding scheme is a prefiltering stage that selects the geohash precision that causes a minimal skewness in the sharding key distribution. That geohash precision then will be used for sharding. Another main contribution of the paper is the design of a custom spatial query optimizer specifically for supporting efficient processing of spatial queries that incorporate spatial joins.

Also, we have designed a two-level indexing scheme based on a combined use of geohashes and Google's S2 that significantly improves the performance of spatial joins in NoSQL distributed systems. Our join algorithm resorts to a filter-and-refine approach that is, by far, the most common well-performing spatial join approach for big data at-scale [2, 3]. Our method focuses on optimizing the operation of the 'filter' stage: it is recognized that reducing the 'candidate set resulted from the 'filter' stage' is crucial to improve the overall spatial join query performance [3]. This is exactly one of the main goals of our original proposal. In short, we aim at reducing the 'candidate set' so that we can improve the overall performance of our spatial join. Our optimizer incorporates a method for selecting the geohash that maximizes the 'index selectivity', thus enabling faster targeted scans.

Our novel partitioning, indexing, and query optimizers, originally presented in this paper, support complex classes of spatial analytics that intrinsically incorporate a spatial join predicate. Those optimizations are intended for dynamic application scenarios that, more than often, require reducing data volumes by means of spatial aggregations. A typical scenario is the case where huge volumes of GPS mobility data arrive every day to a distributed storage and computing cluster. The persistence storage of all values is unnecessary (for example, storing GPS coordinates of a taxi every 10 s), whereas what is required is storing data summaries (e.g., ensembles such as Top-N, and statistics such as 'averages', 'sums', and 'counts') showing the trends of how mobility in a city is changing as time ticks forward. Our solution is unique in its ability to support spatial aggregations natively based on a spatial join that is optimized by our retrofitted multi-level indexing-based 'filter-and-refine' method.

Our optimizer outperforms the plain implementations of the MongoDB NoSQL system for answering spatial queries, such as containment and proximity, and, thus, achieves better time-based QoS goals. We have incorporated those optimizations transparently within the layers of best-in-breed NoSQL system (specifically MongoDB). We term our novel system collectively as SpatialNoSQL, indicating the novel support of an optimized QoS-aware spatial data processing in NoSQL systems.

To simplify the description and discussion of our original solutions, in the following we assume that data are two-dimensional and that the goal is to determine whether an object is contained within another object (a 'within' spatial predicate). This applies to both containment and nearness (proximity) predicates. This applies also for proximity predicates because the problem of finding whether an object falls within a specific distance from another object can be reduced to a containment equivalent (as shown rapidly in Section II.C). It is also assumed that one part of the join operation contains spatial points and the other part contains spatial objects with extents (i.e., polygons). Those assumptions are common in the related literature and do not affect the generality of our solution proposals (see Section III.B).

The remainder of the paper is organized as follows. We first provide a focused background on spatial processing and then, we present the design guidelines and the primary implementation insights of our SpatialNoSQL prototype, by reporting an extensive set of in-the-field performance measurements. We conclude by drawing some remarks and recommend future perspectives.

II. Spatial Processing: A Brief Background

A. Data sharding in NoSQL systems

The limitations of single server-based processing for big data has led to the emergence of distributed processing and storage systems that depend on parallelization. Those systems rely on infrastructures that serve internetworked multiple devices connected as computing clusters on-premises or in a Cloud.

A notable example of scalable big data storage framework is MongoDB that is a NoSQL document-oriented framework that operates on such deployments. In MongoDB terms, a collection corresponds to a table in relational database management systems (RDBMSs) and a document is analogous to a record or a tuple in RDBMSs. MongoDB relies on *data parallelization*, which simply means splitting the big data into chunks, based on a key (known as *sharding key* in MongoDB), and sending them to various partitions (hosted in various computing machines of the Cloud known as *shards* in MongoDB terms). The process is known as *sharding*. A collection may span several shards and, therefore, is referred to

as a *sharded collection*. A collection can also fully reside in one single shard. Documents within a collection are represented in a JSON format. Upon receiving a query, a MongoDB component (hosted normally in a separate machine), known as router, forwards the query request to specific shards depending on the query plan that has been selected based on a sharding key, if specified. Otherwise, if the query does not contain a sharding key as an index, then the router exhaustively broadcasts (i.e., scatter-gather) the request to every shard in the cluster, waiting for receiving a response from every shard, then merges the results. The stock version of MongoDB supports two advanced sharding schemes. Hashed sharding generates a hash key for a single filed and use it as a shard key. This guarantees more even distribution. However, it reduces the targeted scans, where most queries need to scan most shards (broadcast scan). This is attributed to the fact that post-hash documents that have 'close' values for the shard key are forwarded to different chunks or shards. Having said that, a range search will mostly need a broadcast scan. On the contrary, range-based sharding depends on aggregating data of a contiguous range on the values of the shard key, thus guaranteeing to a good extent that documents with 'close' values on the shard key will end up in the same chunk or shard. It is worth mentioning that load balancing is enabled by default in MongoDB and is performed by a balancer that periodically monitors the number of chunks in every shard, then it automatically migrates chunks from overloaded shards (based on a set threshold), until it reaches roughly equal number of chunks per shard. Most importantly, selecting a shard key with enough variations (i.e., high cardinality) is pivotal.

The choice of an appropriate sharding key is crucial to optimise spatial query processing. The sharding key should help, on the one hand, the MongoDB router to route query requests only to the specific shards that contain chunks with the shard key value and, on the other hand, to prune the search space within each shard according to the specified key in the query.

The current MongoDB stock version allows to use only number values or text values as a sharding key, but not spatial fields that are indexed with spatial indexes, such as 2d and 2dsphere. The negative effect is that geometrically nearby objects can be stored into different chunks and even different shards.

B. Spatial Join

Spatial join is a crucial primitive in dynamic application scenarios that normally require intermixing geo-referenced datasets for deeper analytics. It is specifically becoming more important to improve the performance of spatial join processing because spatial data are typically stored in separate files. For instance, in our smart city example scenario, taxi trips in New York city may be stored, for privacy purposes, in one file as GPS coordinates without exhibiting to which neighbourhood (i.e., polygon) each trip belongs. On the other hand, a separate file can contain the municipality administrative neighbourhoods of New York City. This means that in order to find to which polygon a taxi trip belongs, it is necessary to join the two files and to solve a computationally-intensive spatial predicate known as the Point-In-Polygon (PIP) test for each trip (i.e., spatial point).

In its general form, a spatial join is a set obtained by pairing two geo-referenced datasets while applying a spatial predicate (e.g., intersection, inclusion, and nearness) [3]. The two participating sets can represent multidimensional spatial objects. An example of spatial join query in our application example is "finding boroughs in NY city to which each GPS-represented spatial point (e.g., taxi pickup) belongs". This is an example spatial join with a within (i.e., inclusion) predicate that requires joining spatial points with a master table representing boroughs (an administrative synonym for geometrical polygons that divide the city).

Spatial objects are typically represented by using a specific structure that reflects the way they exist in real geometry. For example, if we consider the Earth as a planar object, each spatial point is represented by the two longitude and latitude coordinates. Even spatial objects with extents (e.g., an area forming a polygonal-alike shape or a line representing a river for example) can be represented as set of several spatial points. For example, for polygonal areas, the set consists of those points representing the vertices of the polygon. Hence, a point is a primitive type that can be used to represent other more complex spatial objects. In this way, spatial data can be parametrized and saved in tables as normal fields (e.g., longitude and latitude coordinates that may be assigned values of a float data type). This is a common practice normally because transferring parametrized values over the network (in addition to storing them) is cheaper than transforming the full objects (shapes). A limitation, however, is that this transformation leads to losing the inherent geometry of spatial objects. In addition, ordering spatial parametrized data in a way that preserves their proximity is intractable [3]. Therefore, it is necessary for computer programs to reconstruct those points into their original formats at query run time. This requires solving expensive spatial predicates and geometrical equations (e.g., PIP test). Therefore, the related literature recognizes that spatial joins run into complexities that do not normally affect standard relational joins. Relational join methods, such as sort-merge or equijoin are inapplicable in our context because, for example, sort-merge join is a sorting-based method: given that parametrized IoT spatial data is two-dimensional, it cannot be sorted in both dimensions (longitude and latitude) [3]. Similarly, also equijoin is generally inapplicable because it depends on grouping objects that have equal values, which is impossible in cases where one side of the join has spatial objects with a multi-dimensional representation (a.k.a. spatial extent). Moreover, in equijoin, GPS coordinate accuracies may differ for different participating devices. What is instead needed is a spatial join operator that can join spatial points and objects within a tolerable mutual distance. Furthermore, other more complex approaches such as plane-sweep technique are inapplicable. Interested readers can refer to a reference comprehensive seminal survey on spatial join processing for more detailed and extensive explanation regarding the inapplicability of the join methods mentioned above [3].

Calculating the spatial predicate on multidimensional spatial objects is a compute-intensive and I/O-dominant operation, which requires query processors to make several rounds in/out the memory (for example, to bring complex spatial objects, such as polygons represented with thousands of points). Having said that, minimizing the number of such operations can significantly improve the overall performance of a spatial query [2, 3]. Thus, to address the significant performance overhead of join operations on spatial objects, most well-performing geospatial-oriented algorithms employ a two-stage approach known as filter-and-refine [2-5]. In particular, the first filter stage aims at pruning the search space by first applying a quick-and-dirty filter, then performing a spatial join on approximations of the objects (normally the MBRs of spatial objects with extents); this generates a candidate set that contains false positives (those with MBRs that render the join condition true, but geometrically do not). Filter-and-refine is a general approach that can be tailored depending on the spatial data structures that underly the spatial join processing algorithm. This is a more tractable and scalable approach as it means joining on MBRs (considering also that spatial points from the other join side are approximated to geohashes that cover the MBRs). In the refinement stage, incorrect results (i.e., false positives) caused by the approximations are removed using the exact geometry processor (i.e., the expensive predicate) that is applied on the original objects. This predicate is also known as Point in Polygon test (PIP hereafter for short), which is a spatial predicate that seeks whether a spatial point is contained within the boundaries of an embedding space (often known as polygon), an expensive operation that is also referred to as 'within', 'inclusion' or 'enclosure' predicate.

Filter-and-refine is a general approach that can be tailored depending on the spatial data structures that underlie spatial join processing algorithm [6]. As an example, in the case of Earth flattened out and overlayed with a uniform grid, an ordering structure such as Z-order curves can be imposed on the grid cells, aiming at specifying the direction and ordering of visiting the cells during query processing. As a special case, each grid cell can be represented by a string that is resulting from a geohash encoding. All spatial points that are fenced within the boundaries of each grid cell then share the same geohash value. Geohash can be considered a quick-dirty filter. It is quick as it does not have to apply a costly PIP test. This is a more tractable and scalable approach as it means joining on MBRs (considering also that spatial points from the other join side are approximated to geohashes that cover the MBRs). However, it is dirty as it may not be accurate for all spatial points in the input set. This is because geohash values for neighbouring cells overlap (a phenomena known as 'edge cases' or 'false positives'). For those false positives, the costly PIP test is then necessary to verify to which exact cell a spatial point in real geometries belongs, which is part of the refinement stage. Spatial *refinement* dominates the cost of the whole join procedure; thus, designers should consider minimizing false positives to reduce the cost induced by applying it [7, 8].

C. Spatial Join Processing in MongoDB

MongoDB employs mostly two kinds of spatial indexes for processing spatial queries, i.e., 2d and 2dsphere, where the former is used for flat geometric queries, whereas the latter is used for spherical ones (i.e., an Earth-like sphere) [9] . Several geospatial queries are supported, including proximity (through the $geoNear and $nearSphere operators) and containment (through the $geoWithin operator utilized to search for geospatial points within a shape represented on a flat surface, such as a rectangle, polygon, or a circle). Those queries are supported for geospatial points and shapes (i.e., line, polygon). Containment and proximity queries in MongoDB require applying a 'within' predicate. For operators that require a 'within' predicate, if no spatial index is imposed on the data, MongoDB needs to perform a more expensive exhaustive spatial join, and the costly PIP test needs to be calculated for all points in each shard exhaustively by applying a computationally-intensive algorithm known as *ray casting*.

To cut off such performance penalties to some degree, MongoDB natively allows exploiting spatial indexing (2dsphere and 2d) but only locally within each shard independently. More interesting for our work is the 2dsphere index as it is the indexing structure that we exploit for the spatial specifiers that we are optimizing. 2dpshere is based on google S2[1] and relies on generating non-equal-sized cells that together cover a geometry indexed in 2dsphere (i.e., the embedding space). Thereafter, a B-tree access structure is imposed on the non-uniform grid-cells specifying the order at which grid cells will be visited upon query time, thus speeding up the access. More in details, spatial join in MongoDB is performed by using the $geoWithin operator with a *polygonal geometry specifier*. The join with a spatial index in MongoDB resembles a filter-and-refine approach, where an S2 list is computed for the cells that are covering the geometry specifier. Thereafter, for each cell, B-tree is used to retrieve points that interact with the covering cells. This works as a dirty-and-quick sieve, which returns a list of interacting spatial objects that potentially contain false positives. The refinement step then is responsible for applying the costly PIP test to each object in the false positives list to exclude them from the result set.

However, there are still two limitations with the current MongoDB design. On the one hand, 2dsphere is not allowed to be used as a sharding key for sharded collections. Spatial data locality (SDL) preservation is not achieved during the partitioning stage, thus missing an important optimization. Spatial data is distributed randomly, which typically results in sending geometrically nearby objects to different chunks, and consequently shards. This increases the probability of false positives (i.e., BSO objects), requiring thus to perform more expensive PIP spatial joins within each shard. On the other hand, it is not possible to use the 2dsphere as a cross-shard

[1] https://s2geometry.io/

indexing excluding the possibility for the MongoDB query router to route the query request to specific shards only (those that contain the values in the query specifier). Let us recall that sending geometrically co-located objects to the same shards can boost up significantly the system performance. This is in part due to the fact that spatial queries mostly depend on real geometrical proximity. Then, being able to clump geometrically-nearby objects in same shards increases chances that a local copy of the query processor (within each shard) will be applied to an increased number of proximate objects that are hosted within each shard independently, thus reducing the overall running costs.

Both containment and proximity searches in MongoDB require spatial join predicate. For example, a containment query that seeks to "find all taxi trips that have been originated in a given neighborhood in NY City during a two months period" requires joining two collections (recap that collections in MongoDB are analogous to tables in RDBMSs), the first one containing the spatial points of taxi trips as set of pairs of longitudes and latitudes, and the second one including the neighborhoods in NYC in USA, served as polygons.

Proximity queries in MongoDB (such as those applying $near and $nearSphere operators) also require applying a spatial join predicate. They basically perform a 'within' search (spatial join predicate) on circular areas (regularly shaped polygons). This is because for example $nearSphere (a proximity operator in MongoDB) requires a 'centre point' and a 'radius'. Then MongoDB will construct a circle with the specified radius centred around the centre point. This circle is then considered a polygon (regularly shaped polygon) and the task would be then applying a containment operator (for example, $geoWithin) to find all points contained within the circle. This way, proximity resorts to a special kind of containment, which then requires applying a spatial join.

We have selected MongoDB in this paper as a representative baseline because of the spatially-oriented overarching support that it offers natively. We have stacked-up our SpatialNoSQL prototype (described below) specifically over MongoDB.

III. SpatialNoSQL: A novel spatial-aware framework for NoSQL systems

Our SpatialNoSQL system provides a novel sharding scheme and an optimised spatial-aware join query support for NoSQL solutions. SpatialNoSQL basically comprises two components: a spatial aware novel sharding scheme that is based on dimensionality reduction (specifically geohashing), which we term as geospatial sharding scheme (GSS for short); and a custom spatial query optimizer that exploits GSS in addition to a novel two-level indexing scheme. Our indexing scheme adopts a geohash index at a cross-shard level to operate over different shards and a 2dsphere index at an intra-shard level to operate locally in each shard independently. We have designed this scheme in order to optimize the execution of costly spatial queries that incorporate a spatial join predicate (such as containment searches based on arbitrarily-shaped embedding areas (i.e., polygons), which requires solving the costly PIP test). We explain those two components thoroughly in the next two sub-sections.

A. Geospatial Sharding Scheme

Accounting only for load balancing while distributing big spatial data to computing cluster shards is not enough. Spatial data loads often show co-location continuum relations that need to be considered. Therefore, it is also necessary to address SDL *preservation* to improve geospatial data analytics performance [10]. By achieving SDL preservation while splitting data, the sharding strategy enables the spatial data query processing system to send requests potentially to a reduced number of shards (and consequently chunks within each shard).

Current version of plain MongoDB does not achieve SDL preservation. As MongoDB does not allow using spatial indexes as sharding keys.

To overcome the limitations of spatial support in the current version of MongoDB, we have designed a novel simple, yet, effective sharding scheme that we dub as GSS (short for Geohash Sharding Scheme) based on a dimensionality reduction approach that represents multidimensional spatial points as strings. Specifically spatial objects are represented in terms of geohash[2] strings and geohashes guide the sharding strategy. The underlying idea is that every set of points sharing the same geohash value (intrinsically meaning that they are co-located in real geography) can be sent to the same shards. The fact that a geohash is a string that encompasses a geographical meaning allows us to use it in MongoDB as a sharding key, thus achieving the SDL preservation goal while respecting the underlying MongoDB sharding engine rule.

Algorithm 1 explains how works GSS as it follows. It first receives geo-referenced tuples and applies a mapper on them to

Algorithm 1. GSS Sharding scheme

```
/* input: pointsUpdate (longitude, latitude) collection*/
chunk[max_chunks] = {}
//selecting geohash precession that minimize skewness
1:  min-skewness = INF
    geoPrec = 30 //initial geohash precision
    Foreach gp in geohashes
        skew = calculateSkewness(gp)
        If (skew < min-skewness)
          geoPrec = gp
    //geocode points
2:  Foreach point p in pointsUpdate collection
3:    geoPoint ← geoEncode(p, geoPrec)
4:    groupID ← mapper (geoPoint)
5:    chunk [groupID]. add (geoPoint)
6:  End foreach
7:    bulk_load_chunks (shards [1…i])
```

[2] http://geohash.org/

inject a geohash field, transforming the parametrized GPS coordinates (specifically longitude/latitude) into a one-dimensional geohash value. Thereafter, we specify to MongoDB that the geohash field is the sharding key. MongoDB then proceeds by clumping documents that have the same geohash value into same chunks since geohash has been used as a sharding key. This is possible because the default sharding in MongoDB is range sharding, where documents with similar (or equal) sharding key values end up in the same chunk. Contiguous chunks have more probability to be inserted into same shards. By doing so, we preserve SDL at a low-cost. The only cost associated with our method is the geohash encoding for which we apply a cheap algorithm with a constant complexity. Having said that, the extra overhead introduced by our method can be easily mitigated by the benefits we can reap thereafter from being able to preserve SDL. It is worth mentioning that spatial data coming from IoT is normally highly skewed. Stated another way, spatial instances are clumped into few patches (being city neighbourhoods, districts, boroughs etc.,). This means that specific geohashes will have frequency that may far exceed other geohashes. The extreme case may happen when spatial instances gather only in specific areas of the city during rush hours (for example taxis and human mobility data). This may lead to congest specific shards by sending more chunks to them, leaving the cluster lopsided and may devalue the gains from distributed storage and analytics. This can be solved in two directions, first, an important tweakable parameter in our algorithm is the geohash precision. To increase the probability that better load balancing is achieved, and this extreme worst-case scenario is avoided by design, we have incorporated a prefiltering stage that calculates the skewness of the distribution of data based on the following equation:

$$N.\sum_{k=1}^{N}(y_k - \bar{y})^3 \Big/ ((N-1).(N-2).S^3)$$

Where y_k is the count of each geohash distinct value, $\bar{y}$ is the average count of geohash values, N is the data size and S is the sample standard deviation. Our sharding algorithm then selects the geohash size with the minimum skewness. We calculate the sample skewness instead of the population skewness because the data we have is considered a sample (data keeps arriving from the IoT sources, so the term of population vanishes). This way, we guarantee that the geohash values are more normally distributed and we have fair amount of variations. The imbalance however will persist, but it will have less effect on the migration during the auto load balancing. The other direction is based on enabling the auto-balancing by MongoDB. This will be able to achieve a plausible balance of load balancing even with skewed spatial data distribution as the following. Our sharding scheme is based on the idea of dimensionality reduction where geometrically-nearby spatial share the same geohash string value. Bordering geohashes have 'similar' values because the geohash value will slightly change, so they are considered 'close' in their shard key values. Since we specify the geohash key as the shard key, we recover the range-based sharding which will act as follows. It will take instances with same (or 'close', thus geometrically-nearby) geohash values and clump them in same chunks. What's more, since we are enabling the auto-balancing, MongoDB migrates data as 'chunks' between shards to achieve load balancing. Since spatial-co-locality is already preserved within the boundaries of chunks, then migrated chunks are already optimized for this dimension, in addition to the load balancing that will be achieved automatically.

Algorithm 2. NoSQL spatial join optimizer workflow

```
1:  Input: two versions
    Either Query: q, points: p, r: radius, qp (longitude,
    latitude): query point for proximity through $nearSphere
    OR Query: q, points: p, neighbourhoods: nb for
    containment-PIP through $geoWithin with a geometry
    specifier
2   maxIS = 0 // maxIS: maximum index selectivity
    geoPrec = 30
    //select geohash precision that maximize the index
    selectivity
    If (latency-awareness == true)
      Foreach gp in geohashes
        is = indexSelectivity(geohashValue)
        if (is > maxIS)
        geoPrec = gp //selected geohash precision
    /* List of geohashes covering region (irregular
    polygon),
3:  coverGeo ← getCoverGeo (embedding_area, geoPrec)
4:  coverGeoSp = "geoPrec": {"$in": [coverGeo]}
5:  //adding the geohash specifier to the plain MongoDB
    operator
    newOperator = add (coverGeoSp, MongoDB_operator)
    //enforcing our multi-level indexing scheme
6:  p.createIndexes({"geoPrec",2dsphere"})
    /*Query q is a spatial query (containment-PIP,
    proximity, Top-N) that intrinsically requires a spatial
    join */
7:  executeQuery (q, newOperator,p) //execute the query
    using the new operator
```

B. Spatial NoSQL join Optimizer

We have optimized the plain MongoDB spatial join query optimizer along two directions. First, we incorporate a prefiltering stage as a new specifier that is based on the geohash key, taking thus full advantage of the fact that the geohash key has also been used as a sharding key. Second, we have designed a two-stage indexing scheme that works at a global cross-shard level and at a local inter-shard level.

Our global index is the geohash that enables to select the shards that contain the specific geohash list specified in the query. The geohash acts also on a local level as a pruner within each shard independently because it enables MongoDB to retrieve only points within each shard that interact with the geohash covering (considering that those shards may contain other geohashes that are not included in the query prefiltering geohash specifier). The second level index is a local index that is provided by MongoDB, which is the 2dSphere, that is applied on the result obtained from the higher indexing stage.

Algorithm 2 (which is shown also in Figure 1) explains our spatial join query optimizer for NoSQL based on a filter-and-refine approach. More in details, if the user expresses latency-awareness as a QoS goal, then we apply an index selectivity measurement to calculate the best geohash precision (e.g., 25, 30, and 35) that has a higher index selectivity, by narrowing the search for values during a query scan. We specifically apply the following equation:

$$indexSelect = {}^{dist\_geo\_keys}/_{total\_number\_rows}$$

where $dist\_geo\_keys$ is the 'distinct geohash keys' and $total\_number\_rows$ is the 'total number of rows in the collection'. Having selected the appropriate geohash precision. The algorithm proceeds as it follows. For proximity queries, the optimizer first constructs a circle (given the radius and a query point), then a grid structure known normally as Minimum Bounding Rectangle (MBR) for the circle is imposed and thereafter a list of covering geohashes is generated based on the MBR. For containment query processing with irregularly shaped polygons (e.g., neighbourhoods), the filter-and-refine approach works as follows. The optimizer works on precalculated geohash coverings for the embedding space. The embedding space (the space that is hosting spatial points) is firstly divided into administrative polygons (districts, neighbourhoods, or counties in city management terms) and each polygon is overlayed with an MBR. Obtained MBRs may naturally overlap in the bordering areas between several polygons. Corresponding geohashes coverings are generated and stored into disk.

After generating the geohash covering list, for both query types, our optimizer injects the list as a value in a new specifier, which then will act as a prefiltering frontstage. This will force MongoDB to forward the query request to only shards that contain geohashes that are part of the covering list. In other words, shards that contain points that may interact with the covering.

At a local level, within each shard from the selected shards, the geohash indexing is used to select locally only those points that interact with the geohash covering. Thereafter, within each shard locally, a 2dsphere index is applied as in the following. First, a minimum set of cells is imposed that fully cover the geometry represented by the previously generated geohash covering. Let us refer to this as S2 covering to distinguish it from the coarser level geohash covering. Then a B-tree index is imposed on the S2 covering to speed up the access. false positives, which are already then minimized, are eliminated by applying the exact expensive geometrical spatial join operation. It worth noticing that by indexing on geohash, we were able to minimize the number of possible points that interact with the S2 covering within each shard independently and also on a global level since geohash has been used as a sharding key.

The new geohash specifier is crucial to support a quick-and-dirty prefiltering stage, thus resembling a pruning machine that aggressively prunes the search space before applying the expensive PIP test and even before applying the 2dsphere indexing search. This complies with the filter-and-refine approaches. Stated differently, geohash and 2dsphere together are a compound filter, whereas the following PIP test is a refinement. This way, geohash and 2dsphere reinforce each other without their drawbacks.

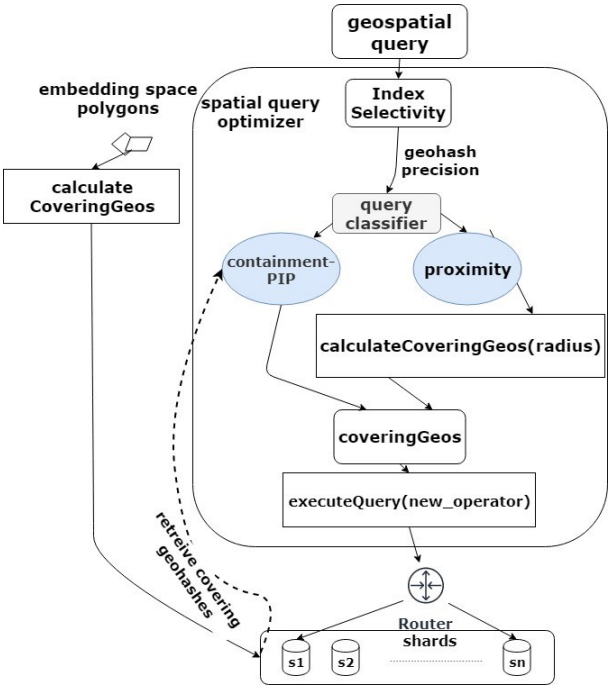


Fig. 1. Spatial join query optimizer for NoSQL

It is worth mentioning that being able to explode multidimensional spatial data polygons into lists of geohashes is an original and significant feature that, to the best of our knowledge, has not been explored yet in the current literature about NoSQL distributed spatial data management. Reading spatial (possibly large) polygons from disks is a dominant overhead factor in the 'refinement' stage [3]. Because our optimizer is able to reduce the 'candidate set' that results from the 'filter' stage, we reduce the number of comparisons that need to be performed in the 'refinement' stage, which applies an expensive PIP operation that needs to bring polygons from disks, especially for very large polygons that are not suitable to be kept in main memory.

The paper targets dynamic application scenarios of smart cities and urban computing, where avalanches of geo-referenced parametrized mobility traces of moving spatial objects (vehicles, human, animals, etc.,) reach NoSQL storage frameworks and need to be managed, rapidly and efficiently, to reduce time-to-insight (for example, in smart cities , participatory health care [11]). Those objects are points with negligible spatial extents. On the other side of the join, spatial objects with extent (such as polygons, e.g., a district in a city, a river, a forest, etc.,) are slowly changing dimensions that rarely modify their shapes, thus are statically residing in disks. In short, the most computing-intensive operation, then, is joining spatial points (dynamic IoT-generated traces) with polygons (static shapes). Other spatial joins, such as joining static spatial objects together (polygon with polygon, line with polygon, etc.,) are less common in dynamic application scenarios because they are normally solved once and then stored for successive usage.

For example, "finding through which districts of a city a street passes", "finding to which forest a lake belongs" are only need to be solved once (or with rarely changing results). However, our spatial join method is general and can be easily applied to any combinations of spatial objects (e.g., line-string, multi-point, multiline-string, multi-polygon, and even a very complex geometry-collection). Since the abstract level of MongoDB representation is based on GeoJSON, many object types are supported natively including MultiPoint (array of points), lineStrings (representing streets in a city, for example), MultiLineString (an array of lineStrings), MultiPolygon (array of polygons), and even the most complex spatial shapes such as a GeometryCollection (e.g., a city with all static objects it contains, schools, streets, districts, etc.,). With these elements in mind, it should be clear that the problem of joining a spatial point (dynamic geo-referenced trace coming from IoT) with a spatial shape (static polygons, lines, or any other geometrical shape) can be reduced to the problem of finding the geohash coverings that completely cover the spatial shape region. This can be easily achieved by writing simple code patches at the application layer, such as the one described in the paper as an example for computing the covering geohashes of polygons.

Consider a more complex use case where points at the intersection of two arbitrarily shaped polygons need to be found. For example, consider a region C at the intersection between regions A and B. Two join predicates can describe the case: the first to find the intersection boundaries (region C) and the second to determine the spatial points that belong to the intersection region C. This complex problem can be easily solved by applying our methods in two ways. The first way is by simply using a two-stage aggregation as the following. The first stage computes all the points that fall within the first polygon (region A), which can be achieved efficiently by applying our method; we can call the resulting set of points of this stage as 'candidate set'. The 'candidate set' then will be fed to the second stage, where we can again apply our join method efficiently to select only those points from the 'candidate set' that belongs to region B, thus obtaining only the points in the intersection region C. The second way the user may choose is to write a simple glue code at the application layer to calculate the intersection boundaries and serve the result as a new polygon to our method. Our method then takes care of all the rest by calculating the covering geohashes and applying our join method to find points belonging to the new polygon (the intersection area). Therefore, having the geohash coverings at hand, our methods originally described in this paper are applicable as-they-are; they represent the abstract foundational layer above which a pyramid of spatial analytics, e.g., the most common ones in the literature about smart cities and urban planning, can be seamlessly stacked up.

Having said so, the methods that we are presenting in this paper are novel, unique, and significant and they can have a direct impact on industrial exploitation. Specially, for businesses and practitioners that are interested in shortening the total time-to-insight by managing and analysing terabytes of geo-referenced datasets that arrive continuously from IoT scenarios, where even tiny optimization of a spatial join operation can assist the system in remaining alive during burst spikes in the workloads (which is a common case in dynamic scenarios that is not unheard of), thus preventing the system from coming into halt situations.

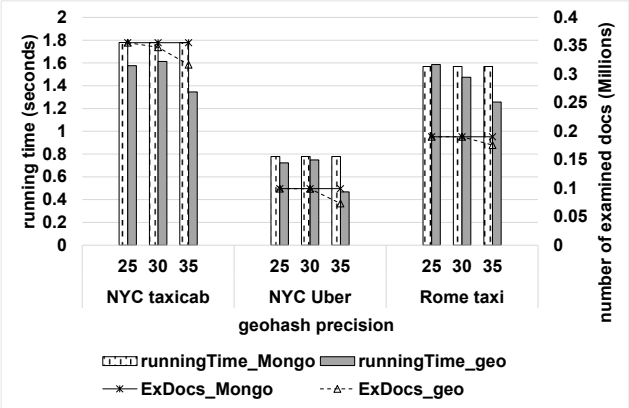


Fig. 2. Comparing the performance of our new spatial join query optimizer on containment-PIP queries (with a $geoWithin operator with a geometry specifier) against the plain MongoDB optimizer for all datasets. 'Mongo' in the legend means the plain MongoDB, whereas 'geo' means our new geohash-based optimizer. ExDocs and ExKeys mean the number of examined documents and keys, respectively.

IV. Experimental Performance Results

To validate our novel query optimizer, we have implemented a prototype of our solutions on top of MongoDB, following the trending layered-up software stack. We implemented a patch of a Scala code over Apache Spark for calculating covering geohashes given an input of a GeoJSON file containing polygons representing the city administrative neighbourhoods. Each polygon consists of an array of locational parametrized points, where each point represents a vertex of the polygon.

For query optimizers, we use JavaScript snippets executed directly in the Mongo shell. By delving into finer details, our solution rewrites queries as the following:

1) Proximity with a $nearSphere specifier. We specifically rewrite $geoWithin with point and circle geometry specifier.
2) Top-N and containment with irregularly shaped polygons (we term this category as containment-PIP) We specifically rewrite $geoWithin with polygon geometry specifier. Top-N is then a special case of containment-PIP, where we find and group the points by the polygon to which they belong, count them, and sort the counts in a descending order. Listing 1 shows an example by applying our optimizer.

C. Deployment Settings and Benchmarking

Dataset. For benchmarking, we have tested our methods using three datasets coming from scalable big data application scenarios in smart cities and urban informatics. NY City taxicab

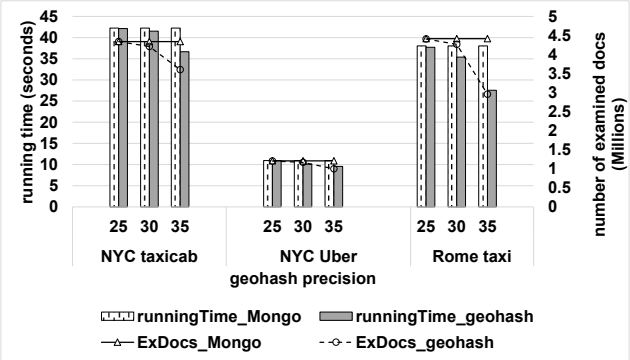


Fig. 3. Comparing the effect on performance of our new containment-PIP query optimizer on ensembles (specifically Top-N queries) against the plain MongoDB optimizer for all datasets. Mongo in the legend means the plain MongoDB, whereas geohash means our new geohash-based optimizer. ExDocs and ExKeys mean the number of examined documents and keys, respectively.

trips datasets [12] [3], which is considered one of the state-of-the-art benchmarking datasets for spatial queries that require solving a spatial join predicate. The second dataset consists of a one-month (February 2014) mobility logs of taxi cabs in Rome, Italy [13], where positions are represented as spatial POINT (latitude, longitude) objects. The third dataset is about Uber pickups in New York City [4] for August 2018. We have selected mobility data because Global Positioning System (GPS) data is normally captured with high spatial accuracy, thus is considered a pivotal source for exploring mobility patterns in smart cities and urban computing. Even more, taxicab mobility data can even be used for exploring human mobility dynamics [14]. Since we are providing spatial optimizations over a de facto standard NoSQL system (MongoDB) that is widely used for urban computing and smart cities, we decided to select mobility data from two big cities in Europe and USA (Rome and NYC, respectively). For NYC taxicab data, we choose a cohort of two months dataset (around three million units) representing data captured through taxi rides for the first two months of 2016. We choose the green taxi trip records, which include interesting fields capturing, most importantly, pick-up/drop-off locations and trip distances. For Rome data, we have selected around two million and a half records that represent the first weekend of February 2014. For NYC Uber mobility traces we have selected logs representing Uber pickups in NYC for the month of August 2014 (around 850k records). The reason for selecting those sizes is twofold. First, we aim to diversify the scenarios we apply and validate the applicability of our methods for various dynamic applications scenarios of smart cities and urban planning. Those scenarios are discussed in subsection D. Second, larger, or smaller sample data sizes have similar data characteristics including the data distributions, so those data sizes that we have selected are good enough to reflect the real-world scenarios.

Deployment and experimental settings. We run our optimizer on a MongoDB Atlas cluster deployed on Microsoft Azure, hosting MongoDB version 4.0. It consists of 4 shards. Each shard is an M30 tier with 32 GB storage, 8 GB RAM and 2 vCPUs.

Parameter settings.

- For containment-PIP (i.e., based on the PIP with polygon geometry specifier) and also for the Top-N queries (which is a special case of containment-PIP), we depend on varying geohash precisions and on total examined documents/keys, in addition to the running time.
- For proximity queries based on $nearSphere operator with a test point and circle geometry specifier, we depend on varying the circle radius and, similarly, on total examined documents/keys, in addition to the running time.

D. Results and Discussion

- **Testing Containment-PIP Query Optimizer**

We specifically focus on containment queries that require PIP test.

Query. We apply the following spatial containment- PIP queries. For NYC taxicab data, we apply "find all taxi trips originated from a given neighborhood in NYC in the last two months". For NYC Uber taxi, we apply the following query: "count the number of Uber orders in a specific neighborhood in NYC during the summer (in August) 2014". Comparing the analysis of the two (Uber and taxicab in NYC can help, for example, determining the districts where Uber orders surpassed those of taxicabs. This can help in sending personalized recommendations to Uber and taxi drivers. For Rome data, we apply the following query: "find taxi trips that have passed through a specific district in Rome in the early morning (at 8:00 A.M.) of a specific weekend (the first weekend of February 2014)". Figures captured though such analytics can help the municipality of Rome at devising better recommendations for tourists who are visiting the city. Figure 2 shows that our optimized version outperforms significantly the plain MongoDB containment- PIP for all the datasets. Note that for geohash 30, our geohash-based optimizer requires scanning 3 shards, whereas the plain optimizer requires scanning 4 shards, which is less efficient. Also, the secondary axis shows that the number of documents examined by each optimizer to answer the same query. It is then evident that our optimizer needs to scan less units for all geohash settings.

- **Testing Top-N Query Optimizer**

As a special case of containment-PIP, top-N query theoretically should act in a similar way. Top-N is possible by checking for each neighbourhood (i.e., polygon) the spatial objects that are contained within it, thus applying the containment-PIP spatial join operator for each object that interact with the coverings.

Query: For NYC taxicab, we apply the following query: "which are the top-10 neighbourhoods in NYC that had the

[3] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[4] Retrieved 16/07/2020 from: https://www.kaggle.com/fivethirtyeight/uber-pickups-in-new-york-city

most pickup taxicab orders in the last two months". For NYC Uber orders, we apply the following: "order NYC districts with most Uber orders during summer (August) 2014". For Rome, we apply the following Top-N: "which are the most congested districts of Rome during the early morning in weekend". Figures obtained from such analytics can show the mobility dynamics and patterns in metropolitan cities, which would help municipalities in improving their urban plans. Figure 3 shows that MongoDB plain optimizer underperforms our optimizer for all geohash settings for all the three datasets. However, the best case occurs at geohash 35, meaning that geohash precision is a configuration that has a paramount importance in our method.

- **Testing proximity queries** (e.g., *k*NN) Optimizer (based on $nearSphere operator with a test point and circle geometry specifier).

Query: We apply the following spatial kNN query for NYC taxicabs: "find all taxicab trip pickup order locations within a specified distance of a test point during the last two months, sorted in order from nearest to farthest". For NYC Uber we apply the following: "find Uber orders locations within a specific distance from the city centre". For Rome data, we apply: "count the number of taxicabs that pass near a specific Point of Interest (POI) in Rome during the early morning of a specific weekend".

As depicted in figure 4, our support outperforms the MongoDB plain support for all datasets. Notice, however, that in cases where a substantial number of documents and keys need to be examined, the difference between running times vanishes. Notice the case of radius 15 where our optimizer needs to examine documents and keys in magnitudes that are roughly equal to those of MongoDB. This is healthy as the number of returned satisfying spatial objects at that distance (i.e., 15 kilometres) is very near to the total number of documents in the points collections that we tested on.

All results shown in this section prove that our framework can satisfy QoS goals, specifically, time-based goals such as low-latency, higher resource utilization, and high accuracy. It does so by applying GSS with retrofitted query optimizers for both proximity (such as *k*NN) and containment-PIP queries. GSS achieves a significant weighted balance between two partitioning goals: SDL preservation and load balancing. The number of BSOs is tweakable through the geohash precision.

V. Related Work

Related methods in the literature are applied to either NoSQL disk-based storage-oriented distributed systems (such as HBase, HDFS and Cassandra [15]) or in-memory processing frameworks (such as Apache Spark [16] and Hadoop).

For example, few systems such as [17, 18] are engineered atop HBase, which is a wide column key-value store that utilizes Hadoop HDFS as its storage layer. However, HBase does not provide a support for multi-level indexing (including secondary indexes), thus degrading reading operations during spatial queries [19]. As a way of contrast, MongoDB supports a native secondary indexing and database aggregations, which increases the productivity levels at the presentation layer, and simplifies the big data access patterns for highly efficient responses to complex queries.

Cassandra-based systems such as the work by [20] run into similar shortcomings as those discussed for HBase counterparts.

By considering distributed in-memory processing, several Spark-based and Hadoop-based frameworks support various data partitioning, indexing, caching, and query optimizers for processing big spatial data at scale. The most visible and popular are GeoSpark [5], spatialspark [21], STARK [22], SparkGIS [23], Simba [24] are all based on Spark, thus not specifically designed for scalable multi-structure storage of rapidly changing data coming from IoT. Also, Simba does not apply the filter-and-refine approach for spatial join processing which deteriorates the spatial query accuracy for multidimensional spatial objects with extents [5]. Other works focus specifically on spatial joins for in-memory batch processing systems such as [4, 25, 26]. In the same vein, few works of the relevant literature focus on exploiting features provided by some of those NoSQL-based and memory-based frameworks in supporting more complex spatial analytics. For example, TrajMesa [27] have extended GeoMesa [28] to support an efficient storage of big trajectory data through novel indexing and query pruning methods. However, it suffers from the same limitations that affect GeoMesa, as it does not support multi-level indexing for spatial data. Also, GeoMesa does not have a post-processing step to deduplicate the additional spatial objects that is introduced by its partitioning model, thus deteriorating the spatial query accuracy [5] . The same fact applies to Simba. Also, GeoMesa employs a simple grid-based local indexing, which is not optimized for processing spatial highly skewed data [5]. Similar work appears in Trajspark [29] that is engineered atop Spark for range and kNN spatial queries on big trajectories data.

Within the same consortium, several works have combined two or more of those frameworks aiming at reusing their features without their limitations. For example, [30] has combined Cassandra with GeoMesa to support spatial storage management and analytics above Cassandra by exploiting those spatial supports from GeoMesa. It however suffers the limitations of Cassandra and GeoMesa altogether, including the absence of a secondary indexing support.

MD-HBase [31] is, by far, the most widely accepted spatial framework stacked above HBase; it does not provide a local index for the contents of the data buckets (analogous to MongoDB chunks), requiring thus an exhaustive scan inside each bucket [19] .

Those spatial frameworks normally do not support multi-level spatial indexing. In addition, in-memory processing systems are designed to handle big data loads in a way that differs significantly from storage-oriented NoSQL counterparts. NoSQL frameworks are designed for managing multi-structured rapidly changing volumes of data coming from various heterogeneous sources such as IoT and content management systems. More in details, NoSQL frameworks have unique ability in being able to ingest data of various

shapes into same unified storage (e.g., table) despite those disparities. For example, a data source is providing level of information (analogous to table fields in relational DBMSs) that is not available in other sources. However, NoSQL is able to ingest from all sources into one collection (table). Their usage is gaining momentum because of the schema-on-read capability they provide for handling the multi-structures of the frequently changing data. On the contrary, in-memory batch processing systems are distributed frameworks designed specifically for real-time analytics. With that in mind, different set of optimizations are required to efficiently handle spatial query processing over NoSQL storage frameworks. It is worth mentioning that emerging analytics, in particular in smart cities, are combining both in end-to-end pipelines so that they reinforce each other synergistically without their limitations. A regular use case is the following. A company that collects millions of spatiotemporally-tagged mobility data daily decides to integrate the powerful real-time analytics pipelines from Apache Spark for running directly on the operation data sitting in MongoDB, where the latter serves as a data lake that stores efficiently (with a unique capability of multi-level indexing) vast amounts of mobility data. This is an efficient mashup as it enables results to be served back to live operational processes without costly expensive ETL processing that would be otherwise needed when integrating Spark with operational databases.

Furthermore, our ability to execute containment queries efficiently (by applying our multi-level indexing scheme) is strongly novel. Spatial aggregations (such as Top-N) are natively supported in our method. Utilizing our support, which is transparently incorporated within the layers of the MongoDB codebase, means that developers at the application layer can write complex aggregation queries without worrying about the underlying logistics. Despite applied generally for spatial joins in NoSQL, this work focuses specifically on non-irregularly shaped spatial objects that requires PIP test (which is more resource-intensive).

A related contribution in this direction is our previous work [32]. In that work, we have designed a sharding scheme (that was termed as GAP, a short for geospatial aware partitioner) that depends on compound sharding key that is comprised of the geohash value of a spatial point and a timestamp of the collection time (GPS sensing time) specially for handling spatiotemporal big data loads. The intention was to achieve a plausible balance between SDL preservation and load balancing on a granular level at the tuple insertion time. However, we have found that such a compound key would rather compromise the SDL preservation. Moreover, in our previous method, we assumed that load balancing should be considered on a document-by-document arrival time basis by engaging the timestamp in the game. However, this may leave the system unstable, because documents are coming from streams and they may arrive out-of-order, rendering the timestamp incorporation inefficient for striking a balance between load balancing and SDL preservation. Also, the workloads we were handling at that time was that of 'ingesting spatiotemporal data from IoT in real-time. However, we found that this may easily turn a bottleneck when write-to-read ratio increases significantly for the collection that is hosting the mobility data. This is so because every insertion must update any indexes in the mobility collection, which normally contains millions of mobility traces, which negatively impacts the write operations. In this paper instead, GSS is different in the sense that we treat new documents (tuples) as collections (appended tables) added to MongoDB untouched as if they were raw updates. Then we convert all new arrived documents at once into geohash-tagged counterparts. Thereafter, offline, we join this collection to the already-sharded collection, update the multi-level spatial index and rebalance the complete collection overnight. Another limitation of our previous work [32] is that we have supported containment queries only for regularly (specifically concentrically) shaped areas (circles in this case). Let us refer to that category as containment-PIC, indicating that it needs a Point-In-Circle (PIC for short) test, which is analogous to PIP test with the exception that the embedding area where we are searching is circular, thus retrieving concentrically located points. However, this kind of queries does not serve all types of spatial analytics in smart cities that incorporate containment spatial predicates. One would then be more interested in finding points that belong to arbitrarily shaped polygons, which is the new type we are supporting in this paper (that we have termed as containment-PIP). One more weakness of our previous work [32] is that we have supported proximity searches by using a MapReduce approach. At the time, $nearSphere MongoDB plain operators were not operating on sharded collections, a drawback that prohibits them from exploiting the benefits of distributed processing. However, starting from MongoDB 4.0, $nearSphere operator has started operating on sharded collections. Consequently, we are employing our novel spatial join optimizer in this paper for enabling an optimized execution of proximity queries over MongoDB.

VI. Concluding Remarks

The abundance of billions of IoT devices have caused the

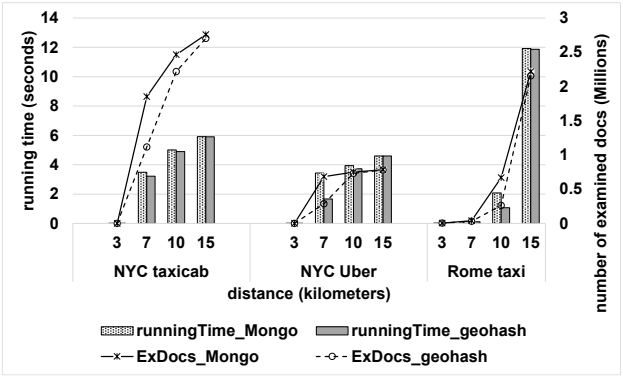


Fig. 4. The performance of our spatial join query optimizer on proximity queries (with a $nearSphere operator) against the plain MongoDB optimizer for all datasets. ExDocs and ExKeys mean the number of examined documents and keys, respectively.

unprecedented accumulation of huge amounts of geo-referenced datasets. Those datasets often need to be analysed by passing through complex spatial analytics pipelines, which most interestingly, encompass spatial join predicates. Spatial join predicate is a core corollary for spatial analytics in smart cities and industry 4.0 and requires normally mashing up multiple views to get deeper insights from spatial data. Proximity and containment spatial queries are two predominant spatial primitive query types in that context. NoSQL scalable storage frameworks, such as MongoDB, tend to provide few spatial analytics supports for proximity and containment searches. However, the spatial join optimizer that those queries depend upon is not optimized for distributed collections of data. This is what specifically this paper is set to solve.

In this paper, we have designed a query optimizer that works specifically for NoSQL queries that involve spatial predicates with an intrinsic join operation (e.g., contain and intersect predicates). Our method outperforms baselines in the field: it is based on dimensionality reduction and on implanting cheap prefiltering stages that significantly prune the search space before applying the costly real geometrical spatial join operators. In summary, we posit that combining spatial partitioning and spatial-aware indexing plays a vital role in the speed of spatial query processing (and most specifically those that incorporate expensive spatial join) in parallel computing environments. We have selected to stack up our optimizations atop MongoDB, because it supports unique features that are not provided by other NoSQL systems. Most importantly, the powerful secondary indexing and native support for grouping/aggregation make it well positioned if compared with its primary competitors in the market. Anyway, the MongoDB support for geospatial data management and analytics has still several limitations and is considered still in its infancy. We have decided to provide a significant original contribution to the community in the field by exploiting the powerful native features of MongoDB and adding layers of spatial optimizations on top of it.

We posit that the novel optimizations presented in this paper are transferable to other similar NoSQL databases such as Cassandra and DynamoDB [33]. However, a further tweaking and a heavy work on the application layers is needed to inject those optimizations. This is so because most NoSQL databases do not offer native advanced indexing, aggregation, and spatial supports. Nevertheless, some systems such as Cassandra and DynamoDB has a potential in implementing our optimizations. For example, Cassandra employs the so-called 'wide partition' pattern, which tries to group related rows together to speed the access at query scans by accessing multiple related rows on same partition [34], which then demystifies applying a dimensionality reduction approach such as geohash. DynamoDB also applies a consistent hashing scheme for partitioning, using concepts like those in Cassandra. Cassandra provides and SQL-alike API for querying, and it would be beneficial to implement our approach on Cassandra in the future. However, the fact that neither Cassandra nor DynamoDB support natively aggregations or secondary indexing requires us to perform more application-layer tweaking.

The encouraging results achieved so far are stimulating our further research work in the area. We are now working on primitives for which we design optimizers for stream-static join processing. In fact, in a case where spatial points arrive from streams only containing GPS coordinates, they need to be joined with disk-residence MongoDB polygons. Also, the primitives introduced in this paper are useful for offline data warehousing view maintenance. Since MongoDB is designed with embedded document structures, it is mostly preferable to keep a beefed-up collection with all documents embedded. For example, keeping the geometrical polygon for each point (which by itself is a document) in an embedded document. This requires solving the spatial join for arriving tuples (could be offline, static-static join as an overnight job for example), thus maintaining the collection regularly.

ACKNOWLEDGMENT

This research was supported by the IDEHA project funded by PON "RICERCA E INNOVAZIONE" 2014-2020 (no. J46C18000440008) and by the SACHER (Smart Architecture for Cultural Heritage in Emilia Romagna) project funded by the POR-FESR 2014-20 (no. J32I16000120009).

REFERENCES

[1] I. M. Al Jawarneh, P. Bellavista, L. Foschini and R. Montanari, "Spatial-aware approximate big data stream processing," in 2019 IEEE Global Communications Conference (GLOBECOM), 2019, pp. 1-6.

[2] I. M. H. Al Jawarneh, "Quality of Service Aware Data Stream Processing for Highly Dynamic and Scalable Applications,". Ph.D. dissertation, Dept. Computer Science and Engineering. alma - University of Bologna, BO, Italy, 2020.

[3] E. H. Jacox and H. Samet, "Spatial join techniques," ACM Transactions on Database Systems (TODS), vol. 32, (1), pp. 7, 2007.

[4] B. Qiao, B. Hu, J. Zhu, G. Wu, C. Giraud-Carrier and G. Wang, "A top-k spatial join querying processing algorithm based on spark," Inf Syst, vol. 87, pp. 101419, 2020.

[5] J. Yu, Z. Zhang and M. Sarwat, "Spatial data management in apache spark: The geospark perspective and beyond," GeoInformatica, vol. 23, (1), pp. 37-78, 2019.

[6] P. Bouros and N. Mamoulis, "Spatial joins: what's next?" SIGSPATIAL Special, vol. 11, (1), pp. 13-21, 2019.

[7] I. M. Aljawarneh, P. Bellavista, A. Corradi, R. Montanari, L. Foschini and A. Zanotti, "Efficient spark-based framework for big geospatial data query processing and analysis," in 2017 IEEE Symposium on Computers and Communications (ISCC), 2017, pp. 851-856.

[8] I. M. Al Jawarneh, P. Bellavista, A. Corradi, L. Foschini, R. Montanari and A. Zanotti, "In-memory spatial-aware framework for processing proximity-alike queries in big spatial data," in 2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2018, pp. 1-6.

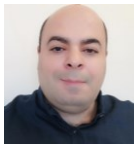
[9] S. Bradshaw and K. Chodorow, Mongodb: The Definitive Guide: Powerful and Scalable Data Storage, 3rd Edn. O'Reilly Media Inc, USA, 2018.

[10] G. Heiler and A. Hanbury, "Comparing implementation variants of distributed spatial join on spark," in 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 6071-6073.

[11] I. M. Aljawarneh, P. Bellavista, C. R. De Rolt and L. Foschini, "Dynamic identification of participatory mobile health communities," in Cloud Infrastructures, Services, and IoT Systems for Smart CitiesAnonymous Springer, 2017, pp. 208-217.

[12] New York (N.Y.). Taxi and Limousine Commission. New York City Taxi Trip Data, 2009-2018. Inter-university Consortium for Political and

Social Research [distributor], 2019-02-20. https://doi.org/10.3886/ICPSR37254.v1
[13] L. Bracciale, M. Bonola, P. Loreti, G. Bianchi, R. Amici and A. Rabuffi. (). CRAWDAD dataset roma/taxi (v. 2014-07-17). Available: https://crawdad.org/roma/taxi/20140717/taxicabs. DOI: 10.15783/C7QC7M.
[14] J. Wang, X. Kong, F. Xia and L. Sun, "Urban human mobility: Data-driven modeling and prediction," ACM SIGKDD Explorations Newsletter, vol. 21, (1), pp. 1-19, 2019.
[15] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," ACM SIGOPS Operating Systems Review, vol. 44, (2), pp. 35-40, 2010.
[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: Cluster computing with working sets." HotCloud, vol. 10, (10-10), pp. 95, 2010.
[17] D. Zhang, Y. Wang, Z. Liu and S. Dai, "Improving NoSQL Storage Schema Based on Z-Curve for Spatial Vector Data," IEEE Access, vol. 7, pp. 78817-78829, 2019.
[18] P. Wang, F. Xu, M. Ma and L. Duan, "Efficient spatial big data storage and query in HBase," in 2019 IEEE International Conference on Smart Cloud (SmartCloud), 2019, pp. 149-155.
[19] J. K. Nidzwetzki and R. H. Güting, "BBoxDB: a distributed and highly available key-bounding-box-value store," Distributed and Parallel Databases, vol. 38, (2), pp. 439-493, 2020.
[20] J. K. Nidzwetzki and R. H. Güting, "Distributed SECONDO: A highly available and scalable system for spatial data processing," in International Symposium on Spatial and Temporal Databases, 2015, pp. 491-496.
[21] S. You, J. Zhang and L. Gruenwald, "Large-scale spatial join query processing in cloud," in 2015 31st IEEE International Conference on Data Engineering Workshops, 2015, pp. 34-41.
[22] S. Hagedorn, P. Gotze and K. Sattler, "The STARK framework for spatio-temporal data analytics on spark," Datenbanksysteme Für Business, Technologie Und Web (BTW 2017), 2017.
[23] F. Baig, H. Vo, T. Kurc, J. Saltz and F. Wang, "Sparkgis: Resource aware efficient in-memory spatial query processing," in Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2017, pp. 1-10..
[24] D. Xie, F. Li, B. Yao, G. Li, L. Zhou and M. Guo, "Simba: Efficient in-memory spatial analytics," in Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 1071-1085.
[25] R. T. Whitman, B. G. Marsh, M. B. Park and E. G. Hoel, "Distributed spatial and spatio-temporal join on apache spark," ACM Transactions on Spatial Algorithms and Systems (TSAS), vol. 5, (1), pp. 1-28, 2019.
[26] C. Rong, X. Cheng, Z. Chen and N. Huo, "Similarity joins for high-dimensional data using Spark," Concurrency and Computation: Practice and Experience, vol. 31, (20), pp. e5339, 2019.
[27] R. Li, H. He, R. Wang, S. Ruan, Y. Sui, J. Bao and Y. Zheng, "Trajmesa: A distributed nosql storage engine for big trajectory data," in 2020 IEEE 36th International Conference on Data Engineering (ICDE), 2020, pp. 2002-2005.
[28] J. N. Hughes, A. Annex, C. N. Eichelberger, A. Fox, A. Hulbert and M. Ronquest, "Geomesa: A distributed architecture for spatio-temporal fusion," in Geospatial Informatics, Fusion, and Motion Video Analytics V, 2015, pp. 94730F.
[29] Z. Zhang, C. Jin, J. Mao, X. Yang and A. Zhou, "Trajspark: A scalable and efficient in-memory management system for big trajectory data," in Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data, 2017, pp. 11-26.
[30] A. Nanjappan, "R*-Tree index in Cassandra for Geospatial Processing," 2019.
[31] S. Nishimura, S. Das, D. Agrawal and A. El Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in 2011 IEEE 12th International Conference on Mobile Data Management, 2011, pp. 7-16.
[32] I. M. Al Jawarneh, P. Bellavista, F. Casimiro, A. Corradi and L. Foschini, "Cost-effective strategies for provisioning NoSQL storage services in support for industry 4.0," in 2018 IEEE Symposium on Computers and Communications (ISCC), 2018, pp. 1227.
[33] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: amazon's highly available key-value store," ACM SIGOPS Operating Systems Review, vol. 41, (6), pp. 205-220, 2007.
[34] J. Carpenter and E. Hewitt, Cassandra: The Definitive Guide: Distributed Data at Web Scale. O'Reilly Media, 2020.



Isam Mashhour Al Jawarneh received PhD degree in computer science and engineering from University of Bologna, Italy in 2020. He is now a postdoctoral researcher at University of Bologna. His research interests cover many aspects of big data stream processing and active data warehousing for highly dynamic application scenarios. He has authored/co-authored many international journal articles and papers for flagship conferences (such as IEEE GLOBECOM and ICC). He has a research and teaching experience at higher-education level for more than 13 years.

Paolo Bellavista (SM'06) received MSc and PhD degrees in computer science engineering from the University of Bologna, Italy, where he is now a full professor of distributed and mobile systems. His research activities span from pervasive wireless computing to location/context-aware services, from edge cloud computing to middleware for Industry 4.0 applications. He is currently the scientific coordinator of a large H2020 big data innovation action called IoTwins about distributed digital twins for the manufacturing industry. He serves on the Editorial Boards of IEEE Communications Surveys and Tutorials, ACM Computing Surveys, IEEE T. on Network and Service Management, Elsevier Pervasive Mobile Computing, and Elsevier J. on Network and Computing Applications, among the others.

Antonio Corradi (SM'19) graduated from University of Bologna, Italy, and received MS in electrical engineering from Cornell University, USA. He is a full professor of computer engineering at the University of Bologna. His research interests include distributed systems, middleware for pervasive and heterogeneous computing, infrastructure for services and network management.

Luca Foschini (SM'19) graduated from the University of Bologna, Italy, where he received a Ph.D. degree in computer science engineering in 2007. He is now an associate professor of computer engineering at the University of Bologna. His interests span from integrated management of distributed systems and services to wireless pervasive computing and scalable context data distribution infrastructures and context-aware services. Currently, he is working on mobile crowdsensing and crowdsourcing and management of Cloud systems for Smart City environments.

Rebecca Montanari graduated from the University of Bologna, where she received a Ph.D. degree in computer science engineering in 2001. She is now an associate professor of computer engineering at the University of Bologna. Her research primarily focuses on semantic-based middleware supports for service provisioning, context-aware services, security solutions for pervasive environments, policy-based service management, and adaptive and scalable middleware solutions for system and service management.