Cost-Effective Strategies for Provisioning NoSQL Storage Services in Support for Industry 4.0

Isam Mashhour Al Jawarneh, Paolo Bellavista, Francesco Casimiro, Antonio Corradi, Luca Foschini
DISI, University of Bologna, Bologna, Italy
isam.aljawarneh3@unibo.it, paolo.bellavista@unibo.it, francesco.casimiro@studio.unibo.it,
antonio.corradi@unibo.it, luca.foschini@unibo.it

***Abstract*—The advancement of networking and sensor-enabled devices have motivated the emergence of unprecedented initiatives, including Industry 4.0 and smart cities. Those are entwined in a way that makes their operation duly interconnected. Industry 4.0 will sooner become the biggest consumer of smart city big data. That data is geo-referenced, and its storage and processing need spatial-awareness, which is currently absent within the constellation of biggest big data management players of the market. We aim to fill this gap by providing spatial-aware big data management strategies in support for Industry 4.0 main principles. Our experimental results show that our strategies outperform those of state-of-the-art by orders of magnitude.**

Keywords—Spatial data; Industry 4.0; NoSQL; Big Data; Smart City; Geohash, Spatial Partition

I. Introduction

In the last decade or so, the fusion of Internet of Things (IoT) and cloud computing paradigms have caused big data to grow unbounded, thereby promoted an unprecedented emergence of a constellation of fast-growing big data management systems confronting that data avalanche. Those systems quickly turned into inevitable trends, including storage-oriented systems designed for handling general-purpose workloads. However, real-life application scenarios normally impose transdisciplinary, where they meet at the intersection of two sciences, thus composing a complexity that promotes the integration of solutions into holistic approaches. For example, Industry 4.0 initiatives require the fusion of cooperating technologies, including cyber-physical systems , IoT and cloud computing [1]. One of the main design principles of Industry 4.0 is the real-time capability, featuring an ability of a system to collect, analyze and extract knowledge in a (near) real time fashion. This is helpful in industrial production environments, where for example if one machine in a manufacturing line suddenly failed, the production can proceed normally by rerouting the process into another machine along the path [2]. Smart city is an initiative that has been put forward aiming at providing quality-aware services for citizens by exploiting computing technologies and has reached its tipping point recently. The convergence of the two evolutions occurs in such a way that Industry 4.0 (as predicted for the foreseeable future) constitutes an integral part of smart cities. For example, based on a big spatial data coming from smart cities, factories focus on demand-oriented production, where they tailor their products based solely on recent user flavors.

All that said, smart city big data need to be first stored in reliable distributed storage database systems. Those systems are reliable in the sense that they provide an anticipated degree of elasticity among other QoS features. Giving the fact that Industry 4.0 heavily relies upon Data as a Service (DaaS [3]), where huge amount of IoTs data are stored in autonomous cloud clusters (sometimes heterogeneous), urges the need for improving the storage-oriented QoS so that it utilizes the exploitation of storage resources. In addition, all data received from smart city technologies are geo-referenced, real-time and preserves distance pairwise relationships [4]. For example, two persons exist in a city center for the same purpose such as being friends gathering for a coffee. Preserving such characteristic while storing data in a big data storage system is desirable for facilitating spatio-temporal queries, where most spatial queries require scanning spatial datasets for distance-wise-oriented answers. For example, finding all restaurants nearest to city center. Provisioning NoSQL services requires handling two entwined aspects; spatial data partitioning that accounts for spatial data locality so that geometrically co-located objects are routed to the same cluster computing elements. Also, query optimizers that consider spatial characteristics. For example, adapting current query operators so that they accept spatial dimension predicates. Those optimizations profoundly affect system throughput. Big data storage ecosystems have focused on partitioning data evenly (thereafter referred to as load balancing) among storage elements so that no single element acquire more data than others. However, Geospatial datasets are highly skewed and not evenly distributed in real geometries, thus sending co-located objects to same elements deteriorates the load balance and leaves the cluster lopsided. Therefore, a solution that balances the weights of optimizations is required. All that said, current distributed NoSQL ecosystems do not offer off-the-shelf solutions for geo-coding-based partitioning and query processing. In this paper, we aim to fill this gap by providing novel viable and cost-effective strategies for spatial data partitioning and ad-hoc query processing in parallel computing environments, therefore optimizing the provisioning of storage services within those domains. Our contribution is twofold. We define a geocoding-based partitioning method, aiming at co-locating geo-close objects into same storage elements, thus assisting the scanning access of proximity-alike spatial queries. To take this further, we have tailored baseline spatial queries so that they exploit the new partitioning scheme. Results show that the merits of our partitioning method outperform state-of-the-art methods by orders of magnitude.

The rest of the paper is organized as follows. In the first section, we provide an insight on partitioning and querying types and challenges in big data context. In the next two sections, we describe our novel spatial-informed strategies and report their experimentation. In the last section, we provide a review of related works together with a conclusion and future research frontiers.

II. BACKGROUND

The performance flaws of relational databases led to the emergence of NoSQL *big data storage engines*. Examples include DynamoDB [5], Cassandra [6] and Bigtable [7]. We herein focus on MongoDB as a representative for document-oriented NoSQL databases [8], as it established itself as a de facto standard in big data management.

A. Big Data Partitioning and Related Challenges

In simpler terms, storage-oriented data partitioning means disseminating datasets to multiple nodes in a distributed computing environment [9], thus facilitating the management of data in parallel. Beside the classical partitioning challenge known in the relevant literature as load balancing, we consider spatial locality as a de facto standard for spatially-tagged data loads. Preserving such pairwise relation while distributing data loads in distributed storage systems is specifically challenging.

Traditional data partitioning approaches include range, hash and round-robin. MongoDB mainly supports range and hash [10]. However, traditional data partitioning methods were designed for general data structures, voiding specific spatial features, which drive the emergence of spatial data allocation methods, including quadtrees [11] and space-filling-curves (SFC) methods [12](such as Hilbert-curves [13] and Z-curves [14]). SFC is a set of dimensionality reduction methods for mapping space into a linear dimension. Over-the-counter, MongoDB supports quadtrees and Hilbert curves. As per 2.6 release, this applies only on indexing, not partitioning. Apart from SFCs, other methods do not provide native handling for spatial co-location.

B. Big Data Query Processing

Storing data is a mean-to-an-end that is of scanning and query processing, which is specifically challenging in distributed contexts, accounting for compute-intensive operations and network communication overheads induced by shuffling data trans-data-nodes. Query optimization in distributed context includes identifying slow queries and improving inter-related factors aiming at providing a speedup.

Quality of Service (QoS) means reducing response time while preserving the correctness and dependability of results obtained [15]. Traditionally, query processors iterate through the whole storage elements while searching for results in a fashion known as scatter-gather, which counteracts some benefits of parallelization. However, some systems (such as MongoDB) apply a query routing method that dispatches query scans to relevant nodes only based on a predefined partitioning key [10].

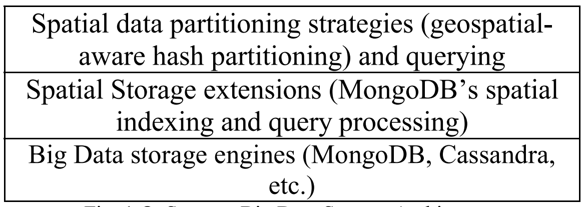
In addition, most NoSQL systems exploits in-memory caching, aiming at avoiding server contact during similar subsequent query requests. Furthermore, indexing facilitates query routing. MongoDB indexing scheme includes single, compound and geospatial Indexes. However, spatial indexing is not supported for partition keys.

Basic spatial querying includes; i) range (AKA containment) returning objects surrounding a specific spatial object (radius or circumference measures, also known as Minimum Bounding Rectangle (MBR)). ii) *proximity-alike* (for example, *nearest neighbor (NN) query)* for finding objects geo-co-located with a specific object or set of objects (the case of *k*NN). An example is finding nearest hospitals to an accident location. iii) *spatial join query*, joining two spatial subsets. In this paper, we concentrate on testing the performance of two spatial query types; range and join. Complex spatial queries are composable of some of those basics.

To the best of our knowledge, MongoDB is the only NoSQL supporting queries on geo-encoded big datasets. It basically supports proximity-alike and containment queries. The flipside however is that it natively does not support spatial indexing on partition keys, every request is thus forwarded to all partitions and consequently heavily taxing system performance.

III. BIG DATA STORAGE STRATEGIES IN SUPPORT FOR INDUSTRY 4.0 INITIATIVES

Fig. 1 depicts a high-level architecture of optimizations we have designed. Our work resides basically within top layer, setting at the core of MongoDB, providing a layer that is transparent to the user and application layer. Our strategies include co-location spatial partitioning strategies. Also, query optimizations to exploit partitioning benefits, including geo-coding proximity-alike queries. We mainly aimed at achieving an utmost QoS for storing and processing big geospatial data in dynamic contexts such as smart cities and Industry 4.0. We next report our novel strategies for data partitioning and query optimization in geospatial contexts. All aimed at achieving desired QoS in the language of response time and throughput.


Spatial data partitioning strategies (geospatial-aware hash partitioning) and querying
Spatial Storage extensions (MongoDB's spatial indexing and query processing)
Big Data storage engines (MongoDB, Cassandra, etc.)
Fig. 1 QoS-aware Big Data Storage Architecture

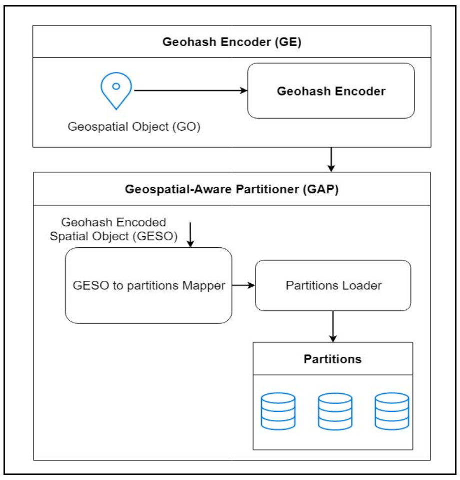

Geohash Encoder (GE)
Geospatial Object (GO)
Geohash Encoder
Geospatial-Aware Partitioner (GAP)
Geohash Encoded Spatial Object (GESO)
GESO to partitions Mapper
Partitions Loader
Partitions

Fig. 2. Custom geospatial-aware partitioner

A. *Spatial-Aware Partitioning Strategies*

Aiming at improving QoS in NoSQL systems, we have designed a custom spatial partitioning method for MongoDB. A comprehensive explanation follows herein.

Geospatial-Aware Partitioner (GAP). Aims of this partitioning strategy include preserving spatial characteristics and load balancing. As the time of this work, MongoDB does not support geospatial-key-based data partitioning. To circumvent this, we have enriched spatial objects with geocoding elements (specifically *geohash*) employing a novel method that we integrated transparently within MongoDB – we term our scheme as *Geospatial-Aware consistent hash Partitioner* (GAP for short). Geohash is better described hereafter, Fig. 2 clarifies the operational mechanism of our method. Coordinates of an arriving spatial object (represented as GeoJSON document) are transformed before they become terminal, passing them to a geohash encoder that calculates and returns a Geohash Encoded Spatial Object (GESO for short), which is then mapped to an appropriate partition using GESO-to-partitions mapper and loaded consequently to its partition through partitions loader.

Geohash is a geocoding system, consisting of a hierarchical subdivision of a two-dimensional map into a grid, where each cell (hereafter bounding box) is represented by a string that represents all points within that box. For more details about geohashing the reader may also refer also to this website http://geohash.org/site/tips.html.

Our method is consistent in the sense that newly arriving data is directed to an appropriate partition without repartitioning. Our scheme guarantees the preservation of spatial data locality. However, we found that using a solo geohash as a partition key tends to cause some degree of skewness (uneven load balance). Therefore, we employed indexed compound fields as a composite partition key, which holds references to timestamp and geohash fields.

B. *Spatial-aware Query Optimizations*

We report our support for an enhanced querying experience in geospatial storage-oriented ecosystems.

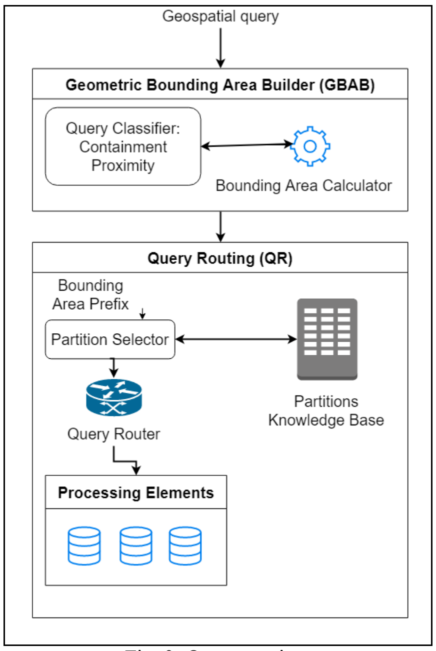


Fig. 3. Query routing.

We first discuss a general methodology for query optimizations in MongoDB. Since geohash is a string, it is accepted both as a partitioning key and a query predicate; In this way, query router employs it for selecting only partitions containing relevant data. MongoDB current scheme for evaluating proximity queries (realized by $near filter) and containment queries (realized by $geoWithin filter) has proved ineffective in big spatial processing context (rationalizing this claim follows in a bit). As a way for accommodating this, we have introduced two functions for efficiently running containment and proximity queries in MongoDB: i) $MR-near (shorthand for MapReduce-near), is a method we introduce as a variant for its counterpart (that of MongoDB, $near), executing proximity queries as MapReduce tasks, thus allowing the work to be done in parallel computing environments (which is not the case for MongoDB version); ii) $geohashBasedWithin, which we implemented as a variation for the native MongoDB operation; $geoWithin. Fig. 3 depicts out general query router. Classifier decides upon query type (containment or proximity) of a geospatial query launched by a user in a normal format, passing it accordingly to appropriate query processor (each taking a different tack). However, both processors start by calculating a list of geohashes covered by the query (for mapping original query into a form employing geohashes and acceptable by MongoDB). Thereafter, modeled request (the new form of the query after mapping) is passed to query router, which is responsible for forwarding request to only appropriate partitions containing relevant result set.

Proximity Query Optimization using MapReduce. This querying scenario has been designed to test or novel partitioning support for MongoDB (GAP method, refer to Subsection III.A for more details**)**, exploiting a specialized querying mechanism developed by us (*$MR-near*).

Our filter works as follows; it first accepts input (point of interest, minimum and maximum distance values), then it constructs a circle around the point of interest, thereafter it calculates a geohash list for points contained within the circle (points within a minimum bounding box, using a general method we have described earlier), it thereafter constructs a MongoDB's compatible query, then applies the query to return all results satisfying query operators (in the form of MongoDB's documents encapsulating spatial objects), and passes results to a MR job. The Map function calculates distance between each object and point of interest (provided in input). Output of Map function is a key-value pair list, where key comprises; distance calculated and object's id (MongoDB's document id), whereas value is a spatial object (as a MongoDB document). Reduce acts as an identity function (returning for each key-value pair, only the value, which is the spatial object itself)

Geohash-Based Containment Query Optimization. This implementation aims at measuring the gain of applying our novel GAP partitioning method. To adapt containment filters provided by MongoDB so that they apply in our case, we have introduced a new filter (we termed as $*geohashBasedWithin*), based on a MongoDB's filter ($*geoWithin*). Our filter receives a circular area for which contained spatial objects are to be found (a filtering operation known as inclusion or containment), thereafter calculates a vector of contained geohashes (using the general method we described in Section III), then it maps original query into a new query by injecting geohash as a constraint (see Excerpt 1 as an example). Under the hood, system scan only partitions containing covered geohashes as specified by the operator.

IV. EXPERIMENTAL RESULTS

A. Experimental Setup

Our cluster deployment consisted of six virtual machines on a private subnet hosted by three physical machines each having 16 cores AMD @1400 MHZ, 300 GB disk, Ubuntu 12.04, and each running an instance of MongoDB 2.6. Our input spatial datasets consisted of nearly **fourteen million** tuples collected through the ParticipAct project of the University of Bologna. A project that aims at studying potential cooperation between citizens, leveraging smartphones as a tool for interaction and interconnection [16].

Datasets maintain user location beside sampling timestamps, longitude and latitude, among others.

B. Results and Discussion

GAP Performance Test. Input datasets were partitioned using extensions we have added to MongoDB, based on compound key composed of object geohash and sampling timestamp. For the sake of clarity, we adopt standard deviation as a measure for load balancing. We compare standard deviation using a single geohash and a composite geohash-timestamp keys. Speaking a mathematically, the smaller standard deviation implies a more load balancing, meaning that compound keys achieves better data load distribution when compared to single key. This is attributed to the fact that spatial datasets tend to be highly skewed making more spatial objects have same geohash value, and are thus routed to same partition, making congestion in specific partitions (termed as wranglers), thereby deteriorating the overall system throughput. Fig. 4 shows the results.

Geohash-Based Proximity Query Optimization Performance. This test aims at examining the performance of our MapReduce-base implementation for proximity filter, comparing it to MongoDB standard implementation ($*geoNear*). We have selected a point of interest in the city center of Bologna in Italy, thereafter proximity queries have been applied with maximum distances of 500, 2000, 10000 and 20000 meters respectively.

We have adopted Amdahl's law for calculating the performance gain (speedup), with a simple formula; speedup is equal to $T_{geoNear}/T_{mr}$. Where $T_{geoNear}$ is the total time required

by MongoDB traditional operator ($geoNear), whereas T_{mr} is the total time required by our MapReduce-based implementation. As we notice in Fig. 5, we gain a speedup as we increase the covered area.

Testing Geohash-Based Containment Query Optimization Performance. This test aims at comparing the performance of containment (or inclusion) filtering using our version with MongoDB traditional version ($*geoWithin*). We have applied a query that seeks all spatial objects contained within a circular area centered at a location in the city center of Bologna in Italy. For calculating the performance gain (speedup), we use a simple formula; speedup is equal to $T_{geoWithin}/T_{new\text{-}geoWithin}$.

Original query looks like this:
```
{<location field>: {$geoWithin:
{$centerSphere: [ [ <x>, <y>], <radius>]}
}}
```
After mapping it and injecting geo hashes, it turns as follows:
```
{{geohash: {$in: [<hashes>]},
 <location field>: {$geoWithin:
{$centerSphere: [ [ <x>, <y>], <radius>]}}}
```
Excerpt 1. Spatial-aware query mapping

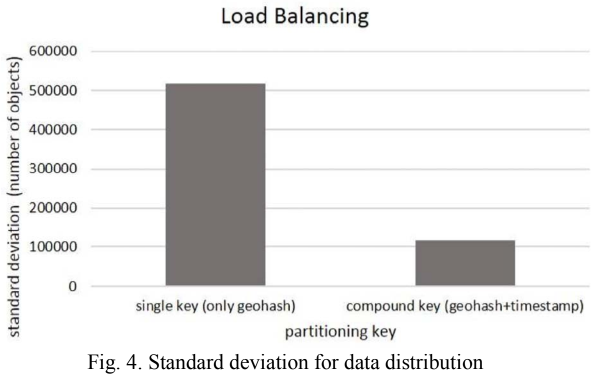


Fig. 4. Standard deviation for data distribution

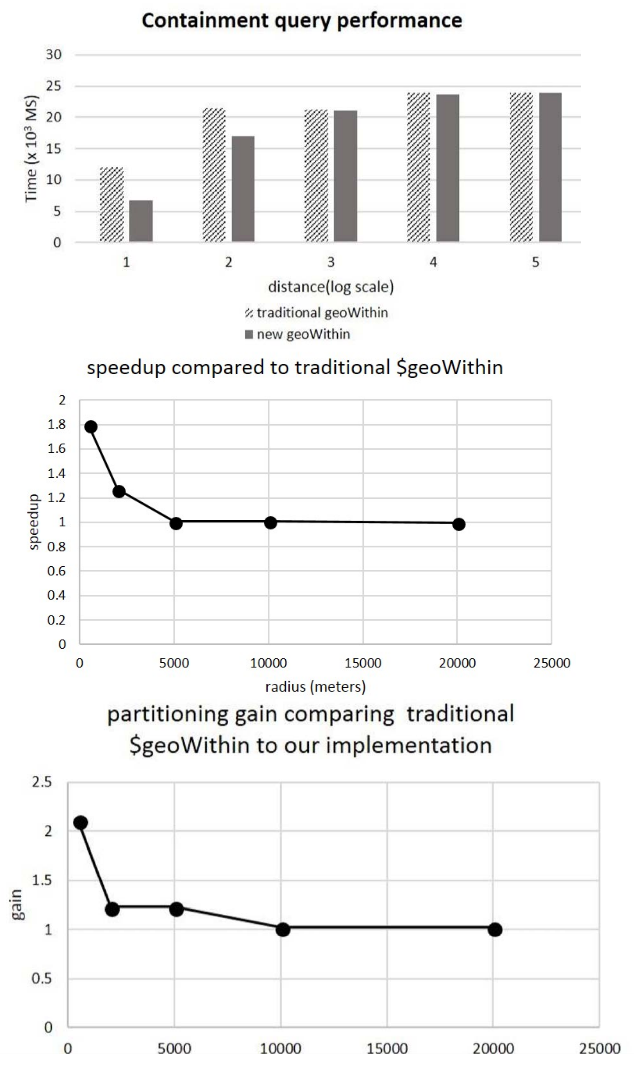


Fig. 6. Speedup by employing our implementation for containment queries compared to traditional MongoDB support

Where $T_{geoWithin}$ is the total time required by MongoDB traditional version ($geoWithin), whereas $T_{new\text{-}geoWithin}$ is the total time required by our novel implementation. Fig. 6 tells the story. It is obvious that as more we increase the radius as less overall speedup we obtain, which is attributed to the fact that increasing circle radius implies scanning more partitions for a result, thus slowing down the system.

In the big picture, we have calculated geohash partitioning gain in containment queries context using a simple formula, $SO_{geoWithin}/SO_{geohashBasedWithin}$, where $SO_{geoWithin}$ refers to the number of scanned objects when using conventional MongoDB method ($geoWithin), whereas $SO_{geohashBasedWithin}$ is equal to the number of scanned objects when using our new implementation. Fig. 6 depicts the gain obtained. Again, there is a declination and stability when increasing the radius of the circle (region to find spatial objects within), which is attributed to scanning more elements, specifically given the fact that our datasets have been collected in Bologna and mostly no more than five kilometers far from the city center (where our focal points are located).

V. Related works

In the relevant literature, a wealth of optimizations exists for partitioning and query processing in the NoSQL databases context. For example, [17] designed a custom partitioning method termed as SPPS for a column-oriented database. Their method achieves load balancing and spatial locality to some good degree. It basically investigates data distribution by sampling methods, thereby computing the proper number of partitions needed for balancing loads. The method proceeds by employing the so-called spatial longest common prefix (SLCP) mapping that mainly sends each geometrical area to a convenient partition based on the key, thus co-locating geometrically-close objects. they also provide an indexing on a geohash-based key for speeding up range-based queries.
However, they do not support other kinds of more advanced spatial querying.

Contemporary with that, [18] have engineered HGrid and encapsulated it within HBase, which is basically a partitioning method that is a mix of quad-tree and grid partitioning schemes, aiming at sending geometrically co-located objects to same partitions. Along the same lines, [19] incorporated a spatial geocoding method known as GeoSOT [20] within the layers of HBase. GeoSOT works by employing quadtrees for constructing a multi-level coding scheme that spans from a higher level representing the earth into a micro level representing square centimeters, where each level is divided into tiles and each tile respects z-ordering, which guarantees load balancing and spatial locality preservation. This method also supports range-based spatial queries by indexing the GeoSOT fields.
The conclusion to be drawn from those works is that they are ad hoc fixes and as far as we are aware of there is no single

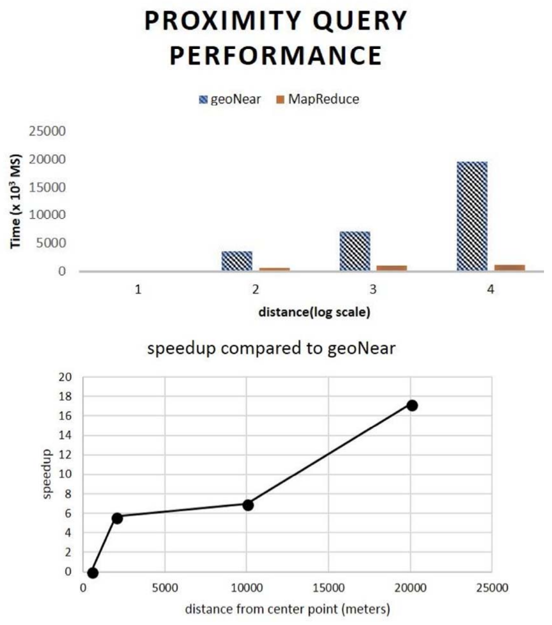


Fig. 5. Speedup by employing MapReduce compared with geoNear

work that adequately addresses fixing spatial data locality in storage-oriented big data management contexts with all its overarching traits, a point where the importance of our work herein stands out.

VI. Conclusions, Limitations And Future Work

Today, data in all fields is big, making its management a challenging task, especially in geospatial application domains. Two main factors play a vital role in determining the obtainable performance gain; those are data load balancing and spatial locality preservation. We always have the traditional trade-off between those requirements, where altering any of them in an exaggerated manner (sometimes counterintuitively even with a tiny factor) may have a negative impact on others. For example, some strategies may achieve better load balancing, but at the expense of spatial locality preservation. With exponentially increasing amount of big geospatial data and wider spectrum of workloads, the problem seems to be computationally intractable. Hence, our recommendation is to seek for a weighted balance for each application scenario, thus accepting a satisfiable overall system performance.

However, our strategies perform favorably against state-of-the-art alternatives. The spacious availability of resources within cloud environments does not directly imply an exacerbation in their usage. Instead, big data management systems should seek to utilize as minimum sources as possible. This is advantageous for many reasons specifically if company is on an economy bound.

There is a work underway to define a mechanism for pre-materializing costliest queries, thus reformulating query structure so that it consumes less time (for example, replace a join with semi-joins). One way to think of that is by introducing a query optimizer that selects the best route for each query depending on many factors, including query type, where a knowledge base is aware of the performance of each type of queries, then selecting the best partitioning method accordingly.

In this paper, we focused on spatial data partitioning and its implications on spatial query processing performance. Nonetheless, our fixes are generic in such a way that makes them easily tweakable for satisfying specific domain constraints and poised to take a central position, thus easily finds their way to break into big spatio-temporal management filed. For example, depending on dataset domain and query workload. A future work is geared toward taking data as streams and process it, mention semi-stream join processing and mention a scenario for this. For example, we may be joining stream data with master data and test the performance of join queries, maybe injecting novel semi-stream join algorithms here in this context. Also, we want to test the performance of both storage and processing oriented under extremely harsh conditions attributed to stream data coming in a very fast and unpredictable pattern.

Acknowledgment

This research was supported by the SACHER (Smart Architecture for Cultural Heritage in Emilia Romagna) project funded by the POR-FESR 2014-20 (no. J32I16000120009) through CIRI.

References

[1] N. Jazdi, "Cyber physical systems in the context of Industry 4.0," in *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*, 2014, pp. 1-4.

[2] M. Lom, O. Pribyl, and M. Svitek, "Industry 4.0 as a part of smart cities," in *2016 Smart Cities Symposium Prague (SCSP)*, 2016, pp. 1-6.

[3] V. Mateljan, D. Cisic, and D. Ogrizovic, "Cloud Database-as-a-Service (DaaS) - ROI," in *The 33rd International Convention MIPRO*, 2010, pp. 1185-1188.

[4] S. Cho, S. Hong, and C. Lee, "ORANGE: Spatial big data analysis platform," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 3963-3965.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin*, et al.*, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.,* vol. 41, pp. 205-220, 2007.

[6] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.,* vol. 44, pp. 35-40, 2010.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows*, et al.*, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.,* vol. 26, pp. 1-26, 2008.

[8] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. Capretz, "Data management in cloud environments: NoSQL and NewSQL data stores," *Journal of Cloud Computing: Advances, Systems and Applications,* vol. 2, p. 22, 2013.

[9] M. T. Özsu and P. Valduriez, "Distributed Database Design," in *Principles of Distributed Database Systems, Third Edition*, ed New York, NY: Springer New York, 2011, pp. 71-129.

[10] K. Banker, *MongoDB in Action*: Manning, 2012.

[11] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta Inf.,* vol. 4, pp. 1-9, 1974.

[12] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer, "Space-filling curves and their use in the design of geometric data structures," *Theoretical Computer Science,* vol. 181, pp. 3-15, 1997/07/15/ 1997.

[13] I. Kamel and C. Faloutsos, "Hilbert R-tree: An Improved R-tree using Fractals," presented at the Proceedings of the 20th International Conference on Very Large Data Bases, 1994.

[14] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," Ottawa, Canada, IBM1996.

[15] R. Epstein, M. Stonebraker, and E. Wong, "Distributed query processing in a relational data base system," presented at the Proceedings of the 1978 ACM SIGMOD international conference on management of data, Austin, Texas, 1978.

[16] G. Cardone, A. Corradi, L. Foschini, and R. Ianniello, "ParticipAct: A Large-Scale Crowdsensing Platform," *IEEE Transactions on Emerging Topics in Computing,* vol. 4, pp. 21-32, 2016.

[17] K. Zheng, D. Gu, F. Fang, M. Zhang, K. Zheng, and Q. Li, "Data storage optimization strategy in distributed column-oriented database by considering spatial adjacency," *Cluster Computing,* vol. 20, pp. 2833-2844, 2017/12/01 2017.

[18] D. Han and E. Stroulia, "HGrid: A Data Model for Large Geospatial Data Sets in HBase," in *2013 IEEE Sixth International Conference on Cloud Computing*, 2013, pp. 910-917.

[19] Z. Weixin, Y. Zhe, W. Lin, W. Feilong, and C. Chengqi, "The non-sql spatial data management model in big data time," in *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, 2015, pp. 4506-4509.

[20] C. Cheng, X. Tong, B. Chen, and W. Zhai, "A Subdivision Method to Unify the Existing Latitude and Longitude Grids," *ISPRS Int. J. Geo-Information,* vol. 5, p. 161, / 2016.