GeoTrie: A Scalable Architecture for Location-Temporal Range Queries over Massive GeoTagged Data Sets

Rudyar Cortés[1], Xavier Bonnaire[2], Olivier Marin[3], Luciana Arantes[1], and Pierre Sens[1]

[1]Sorbonne Universités, UPMC Univ Paris 06, CNRS
INRIA - LIP6, Paris, France
E-mail: [rudyar.cortes, luciana.arantes, pierre.sens]@lip6.fr
[2]Universidad Técnica Federico Santa María, Valparaíso, Chile
E-mail: xavier.bonnaire@inf.utfsm.cl
[3]New York University, Shanghai, China
E-mail: ogm2@nyu.edu

***Abstract*—The proliferation of GPS-enabled devices leads to the massive generation of *geotagged* data sets recently known as Big Location Data. It allows users to explore and analyse data in space and time, and requires an architecture that scales with the insertions and location-temporal queries workload from thousands to millions of users. Most large scale key-value data storage solutions only provide a single one-dimensional index which does not natively support efficient multidimensional queries. In this paper, we propose GeoTrie, a scalable architecture built by coalescing any number of machines organized on top of a Distributed Hash Table. The key idea of our approach is to provide a distributed global index which scales with the number of nodes and provides natural load balancing for insertions and location-temporal range queries. We assess our solution using the largest public multimedia data set released by Yahoo! which includes millions of geotagged multimedia files.**

***Index Terms*—Big Location Data; Location-Temporal Range Queries; Scalability.**

I. Introduction

An ever increasing number of GPS-enabled devices such as smartphone and cameras generate geotagged data. On a daily basis, people leave traces of their activities involving mobile applications, cars, unmanned aircraft vehicles (UAVs), ships, airplanes, and IoT devices. These activities produce massive flows of data which cannot be acquired, managed, and processed by traditional centralized solutions within a tolerable timeframe. The time and geographic analysis of such data can be crucial for life saving efforts by helping to locate a missing child, or for security purposes allowing to know which smartphone and therefore who was present in a given area and during a specific event. An emergent field of research, known as *Big Location Data* [1]–[3], addresses this issue.

The essential characteristic of *Big Location Data* is that it associates every data with meta-data which includes a geotag and a timestamp. *Location-temporal range queries* [4] represent a major challenge for data extraction, exploration and analysis, within both a geographic area and a time window.

Popular multimedia services such as Instagram[1], Flickr[2] or Twitter [3] are examples of already existing applications that generate continuous and massive flows of data, enriched with meta-data including geotags and timestamps.

Other services which do not rely on user interaction, like automatic smartphones location or smart cities, can generate meta-data on an even larger scale. Allowing millions of users to explore these data sets could help get more insight about what happens in particular geographic areas within given time-frames. For instance, people who attended a concert between 20:00 p.m. and 01:00 a.m. may want to review public pictures and comments from people who attended the same concert. The concert manager may want to acquire more feedback about the overall concert experience by analyzing pictures, comments and tags. Or people might want to verify whether the referee of a football match missed a foul by catching multiple amateur videos and pictures of the action.

The main challenge presented by such services is to guarantee scalability, fault tolerance, and load balancing for a high number of concurrent insertions and location-temporal range queries. Relational databases cannot reach such scales: their response time significantly increases when concurrent insertions and queries have to cover billions of objects. Other approaches that rely on a centralized global index [5]–[7] contend with a bottleneck. More recent approaches use *space-filling curves* [8] in order to collapse the latitude and longitude coordinates into a one-dimensional index and distribute it over multiple nodes [9]–[11]. Although they allow scalable location-temporal query processing, the *curse of dimensionality* [8] introduces several processing overheads which impact the query response time.

In this paper, we present a scalable architecture for location-

[1]https://instagram.com
[2]https://www.flickr.com
[3]http://www.twitter.com

temporal range queries. The main component of the architecture is a distributed multidimensional global index which supports location-temporal range querying on a large scale, and provides natural insertion and query load balancing.

The main contributions of this paper are:

- An approach to introduce location-temporal data locality in a fully decentralized system such as a distributed hash table. This approach is based on a multidimensional distributed index which maps $latitude, longitude, timestamp$ tuples into a distributed prefix octree, thus efficiently filtering false positives while providing fault tolerance and load balancing for internet-scale applications.
- A theoretical evaluation of our solution, which shows that its message complexity for insertions and range queries is logarithmic with respect to the number N of nodes involved.
- A practical evaluation, which shows that our solution alleviates the bottleneck on the root node during insertions and range queries. This property has an impact on the query response time. For instance, queries which avoid the root node presents an average query response time up to 1.9x faster than queries which starts at the root level.

The rest of this paper is organized as follows. We detail our solution in Section II, and then evaluate it in Section III using a large public multimedia dataset from Yahoo!. We give an overview of the related work in Section IV, before concluding in Section V.

II. System Design

We aim to propose a scalable architecture for indexing and querying meta-data extracted from geotagged data sets.

We choose to work on top of a distributed hash table (DHT) such as Pastry [12] or Chord [13] because it provides a strong basis for scalable solutions as opposed to centralized server architectures. DHTs provide an overlay with properties such as self-organisation, scalability to millions of nodes (usually routing has a message complexity of $O(log(N))$, where N is the number of nodes) and dependability mechanisms via replication. However, a DHT by itself cannot constitute a satisfactory solution for scalable location-temporal data storage and retrieval because the hash function usually destroys data locality for the sake of load balancing.

The proposed architecture must create data locality over DHTs based on three core dimensions: latitude, longitude and timestamp. We assume that every meta-data entry references data stored on an external storage service. We intent for our architecture to achieve the following properties.

- *Self-organisation:* The architecture must be self-organised. That is, every node must run a local protocol without needing to add a central third party.
- *Scalable query processing:* Meta-data insertions and location-temporal range queries must scale with the number of available nodes in the DHT.

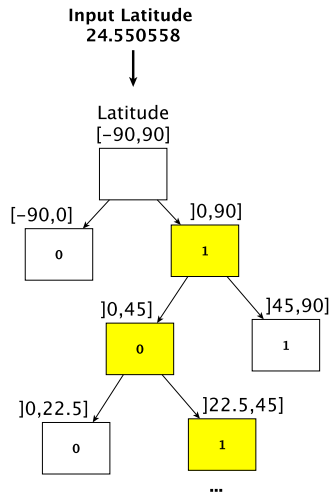


Fig. 1: Latitude mapping example. The latitude value 24.550558 is mapped to the binary string $T_{lat} = 10100010111010101001010111011 01$

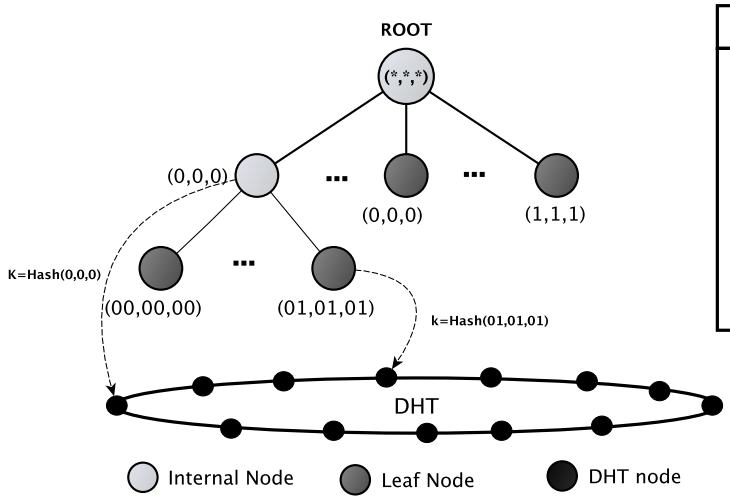
- *Query load balance:* Meta-data insertions and location-temporal range queries must be distributed in the architecture in order to avoid a single bottleneck.
- *High Availability:* The architecture must tolerate node failures.

Geotrie builds location-temporal data locality over a DHT following two main steps: *mapping* and *indexing*. The *mapping* step associates every (latitude,longitude,timestamp) coordinate with a tuple key $T_k = (T_{lat}, T_{lon}, T_t)$, where every coordinate of T_k is a 32-bit word. The *indexing* step inserts every tuple key T_k in a fully distributed prefix octree which allows efficient location-temporal range queries.

A. Mapping

Let $lat \in [-90, 90]$, $lon \in [-180, 180]$, and $t \in [0, 2^{32} - 1]$ be the latitude, longitude and timestamp parameters of a given geotagged object. The timestamp coordinate corresponds to the *Unix epoch* timestamp. We map every coordinate to the multidimensional tuple key $T_k = (T_{lat}, T_{lon}, T_t)$, where every tuple belongs to the same domain $\{0, 1\}^{32}$ (i.e, binary string of 32 bits per coordinate).

Geotrie performs domain partitioning for all the coordinates as follows. First, the space domain $[a, b]$ of every coordinate is divided in two buckets $D_{left} = [a, (a + b)/2]$ and $D_{right} =](a+b)/2, b]$. A bit value of 0 represents a point that belongs to the left sub domain, while a bit value of 1 positions it in the right subdomain. This process continues recursively until all $D = 32$ bits are set. For instance, the first bit of latitude -45 is 0 because it belongs to the first left subdomain [-90,0]. This mapping function reaches its lower bound at latitude value $lat = -90$ represented as the identifier $\{0\}^{32}$, and its upper bound at $lat = 90$ represented as the identifier $\{1\}^{32}$. Longitudes incur the same process, with the lower bound at $lon = -180$ and the upper bound at $lon = 180$.



GeoTrie Node	State	Tuple Keys
(*,*,*)	Internal Node	--
(0,0,0)	Internal Node	--
(00,00,00)	Leaf Node	00,00,00
(01,01,01)	Leaf Node	01,01,01
(0,0,1)	Leaf Node	00,00,10
...	...	...
(1,1,1)	Leaf Node	10,10,10

Fig. 2: GeoTrie Example: Distributed data structure for keys of D=2 bits size

Input: Tuple key T_k
Output: Leaf node
$lower = 0$;
$higher = D$;
while $lower \leq higher$ **do**
  $middle = (lower + higher)/2$;
  // Extract the prefix of size middle of every coordinate of T_k and route the message
  $node = DHT\_lookup(T_{middle}(T_k))$;
  if *node is a leaf node* **then**
    return $node$;
  else
    if *node is an internal node* **then**
      $lower = middle + 1$;
    else
      // node is external node
      $higher = middle - 1$;
    end
  end
end
return $failure$;

Algorithm 1: LocateLeaf pseudocode

The mapping of the timestamp coordinate follows the same principle. It results in the binary representation of the *Unix epoch*. Figure 1 presents an example of the mapping strategy for latitude value 24.550558.

Every tuple T_k encloses coordinates into a location-temporal cell which covers an area of 0.46 cm $\times$ 0.93 cm at the equator, and ensures a one second time precision. It also induces the following properties.

- *Recursive prefix domain partition*. The mapping function presented above recursively partitions every latitude, longitude, and timestamp coordinates into eight areas represented by prefixes. For instance, the tuple key $T_k = (0, 0, 0)$ represents all the latitude,longitude, and timestamp tuples within the interval latitude $([-90, 0])$, longitude $([-180, 0])$, and timestamp $[0, (2^{32} - 1)/2])$. The next subdomain $T_k = (00, 00, 00)$ represents the domain latitude $([-90, -45])$, longitude $([-180, -90])$, and timestamp = $([0, (2^{32} - 1)/4])$, and so on.
- *Prefix data locality*. Shared prefixes imply closeness. That is, all keys with the same prefix necessarily belong to the single and same area represented by this prefix. For instance, all keys that share the same prefix key (00,00,00) belong to the interval covered by this prefix. Note that the converse is not true; for instance 011 and 100 are adjacent but they do not share a common prefix.

Input: Tuple key T_k
Meta-data m
node = LocateLeaf(T_k);
node.insert(leafAddr,T_k,m);

Algorithm 2: Insertion pseudocode

B. Indexing

The GeoTrie structure exploits both properties presented above. It indexes every tuple key $T_k = (T_{lat}, T_{lon}, T_t)$ into a distributed prefix octree-like indexing structure built on top of a DHT.

Every GeoTrie node holds a label l, a state s, and the range it covers. The label l is a prefix of T_k and the state s can either be *leaf node*, *internal node*, or *external node*. Only *leaf nodes* store data; *internal nodes* stand on the path to leaf nodes that do hold data, while *external nodes* stand outside any such path. A DHT node declares itself to be an *external node* for a label l, when it receives a query asking for a non existing label l on its local data structure. The range covered by a GeoTrie node is locally computed by using its label l and the *recursive prefix*

domain partition property presented above. Furthermore, every *internal node* stores links to its direct children nodes and the range they cover.

Every GeoTrie node is assigned to the DHT node whose identifier in the ring is closest to the key $k = Hash(l)$.

Upon start-up, GeoTrie consists of a single root node with label $l = (*, *, *)$ and *leaf node* state. When a node becomes full, GeoTrie scales out by switching it to internal and dispatching its load onto eight new *leaf nodes* created via recursive prefix domain partitioning.

Figure 2 shows a representation of the GeoTrie data structure for tuple keys of size $D = 2$ bits. GeoTrie can take advantage of any replication mechanism used by DHTs in order to provide fault tolerance. For instance, in Pastry [12] Geotrie can use the *leaf set* in order to replicate and maintain the state of every Geotrie node.

Insertions and deletions. Storing and deleting an object with key T_k consists first in locating the *leaf node* whose label is prefix of T_k, and then in carrying out the insertion or deletion operation directly on this node. For instance in Figure 2, tuple key $T_{k_1} = (00, 00, 00)$ is stored on the *leaf node* with label $l_1 = (00, 00, 00)$ and tuple key $T_{k_2} = (10, 10, 10)$ is stored on the leaf node with label $l_2 = (1, 1, 1)$. Algorithm 2 presents a pseudocode of the insertion operation; it uses the *locateLeaf* procedure detailed in Algorithm 1. *locateLeaf* receives a tuple key T_k as input and returns the *leaf node* whose label is prefix of T_k. It does so by performing a binary search over different possible prefixes of T_k until a node with state *leaf node* is reached. For every tuple key T_k, there are exactly D possible prefix labels plus the root node, and only one them can designate a *leaf node*. The client starts by sending a lookup message to a label prefix of T_k of size $D/2$ (i.e, the middle of the space of possible candidates). This label is computed by extracting the first $D/2$ bits of every coordinate. If the state of this node is *external*, it means that the target *leaf node* keeps a label of shorter prefixes. In this case, the binary search cuts the space to prefixes of higher size $D/2 - 1$ and it propagates the lookup to shorter prefixes. Instead, if the target node is an *internal node* it means that the *leaf node* keeps a longer prefix. It cuts the space to a lower prefix of size $D/2 + 1$ and propagates the search down GeoTrie. This process continues recursively and ends upon reaching a *leaf node* whose label is prefix of T_k. If the *locateLeaf* algorithm fails to return a leaf node due to index maintenance, the client must retry the insertion at a later time. This binary search procedure benefits of the prefix property of GeoTrie which allows queries to start at any node, and thus prevents the root node from becoming a bottleneck.

Index maintenance. GeoTrie provides two index maintenance operations: *split* and *merge*. The *split* operation occurs when a *leaf node* stores B keys, where B is a system parameter. An overloaded *leaf node* scales out its load by creating eight new children nodes via recursive domain partitioning. The dispatch of data follows the prefix rule: a data entry with index T_k gets transferred to the new *leaf node* whose label l is prefix of T_k). After transferring all its data, the *split* node changes its state from *leaf node* to *internal node* and every child node becomes a new *leaf node*.

Input: $(\Delta Lat_b, \Delta Lon_b, \Delta t_b)$
Output: $Meta - data$
$prefixLat = commonPrefix(\Delta Lat_b)$;
$prefixLon = commonPrefix(\Delta Lon_b)$;
$prefixTime = commonPrefix(\Delta t_b)$;
// Common prefix of minimum size
$target$ = minComPrefix(PrefixLat,PrefixLon,PrefixTime);
// If there is not a common prefix
start from the root node
if $target = (,,)$ **then**
  $target = (*, *, *)$;
end
$node = DHT - lookup(target)$;
if *node is a leaf node* **then**
  // process the request
  return LocalRequest($\Delta Lat, \Delta Lon, \Delta t$);
else
  if *node is an internal node* **then**
    // forward the request to the children nodes that covers the range
    $forwardRequest(children)$;
  else
    // Node is an external node
    // Locate the leaf node which covers all the interval using the target label as starting point
    $node = LocateLeaf(target)$ // Forward the request to this node
    $forwardRequest(node)$
  end
end

Algorithm 3: Location-temporal range query pseudocode

The *merge* operation is the opposite to the *split* operation. GeoTrie triggers the *merge* operation on a group of eight *leaf nodes* which share the same *internal node* as parent when the sum of their storage loads becomes less than $\lfloor B/8 \rfloor$ objects. When this happens the *internal node* sends a *merge message* to all its *leaf node* children, thus requesting they transfer back all their stored data. Upon transfer completion, all children *leaf nodes* detach from the prefix tree structure and the parent switches its state from *internal node* to *leaf node*. Typical applications generate far more insertions than deletions, so we expect a low proportion of *merge* operations compared to *split* operations.

Location-temporal range queries. A location-temporal range query is represented as a three dimensional range query as presented in equation 1.

$$\begin{array}{rcll} \Delta Lat & = & [lat1, lat_2], & lat_1, lat_2 \in [-90, 90] \\ \Delta Lon & = & [lon_1, lon_2], & lon_1, lon_2 \in [-180, 180] \\ \Delta t & = & [t_i, t_f], & t_i, t_f \in [0, 2^{32} - 1] \end{array} \quad (1)$$

This query is resolved as follows. First, the sender node uses the mapping function defined above in order to translate every coordinate constraint $(\Delta Lat, \Delta Lon, \Delta t)$ into binary strings belonging to the domain $\{0,1\}^{32}$ as presented in equation 2.

$$\begin{array}{rcll} \Delta Lat_b & = & [lat_1^{32}, lat_2^{32}], & lat_1^{32}, lat_2^{32} \in \{0,1\}^{32} \\ \Delta Lon_b & = & [lon_1^{32}, lon_2^{32}], & lon_1^{32}, lon_2^{32} \in \{0,1\}^{32} \\ \Delta t_b & = & [t_i^{32}, t_f^{32}], & t_i^{32}, t_f^{32} \in \{0,1\}^{32} \end{array} \quad (2)$$

Then, it computes the common prefix label of minimum common size for every coordinate constraint and it forwards the query to the node which covers this label. For instance, query Q which combines constraints $\Delta Lat_b = [00..., 01...]$, $\Delta Lon_b = [10..., 11...]$, and $\Delta t_b = [110..., 110...]$ has a common prefix label of every coordinate $(0, 1, 11)$. This common prefix label has not a common size because the time coordinate is longer than the others. Thus, this common prefix is converted to a label of minimum common size $l_Q = (0, 1, 1)$ which represents a label of GeoTrie. This label covers all data which is inside the interval of Q. If there is not common prefix, the query must start from the root node labeled $l = (*, *, *)$. Algorithm 3 presents a pseudocode for this procedure.

Depending on the state s of the node which receives the query, we identify three cases. (i) If the node is an *internal node*, it uses the *forwardRequest* function in order to recursively forward the query onward to the *leaf nodes* whose location-temporal range intersects that of the query interval. (ii) If the node is a *leaf node*, it is the only node that covers the required location-temporal range and returns the objects that satisfy the query. (iii) If the node is an *external node*, it follows that the common prefix is a label which does not yet exist in the prefix tree (i.e, the query arrived to a DHT node which does not hold this label). In this case, the query is necessarily covered by a single *leaf node*. in order to find this particular *leaf node*, the client node then starts the *LocateLeaf* algorithm from the label of the *external node*.

III. Evaluation

This section presents a theoretical and an experimental evaluation of GeoTrie. The theoretical evaluation assesses the scalability of the solution in terms of the message complexity for insertions and range queries. An extensive experimental evaluation studies the load balancing and the performance of GeoTrie in terms of the query response time. We chose to conduct our experimental evaluation with a real dataset: the Yahoo Flickr Creative Commons (YFCC) dataset [14] released by Yahoo! It comprises $48,469,177$ geotagged multimedia files (photos and videos).

A. Overview

This section presents a simulation-based assessment of GeoTrie to measure its performance, with a large scale real world dataset from Yahoo!

We implemented GeoTrie on top of FreePastry [15], an open-source DHT implementation. In our experiments, we deployed $N = 1,000$ DHT nodes. Every *leaf node* stores at most $B = 10,000$ keys. We assume that DHT nodes are distributed worldwide. We used a baseline latency file provided in the FreePastry distribution to generate latencies among DHT nodes. Table I gives the baseline values, while Figure 3a shows the observed distribution of latencies on the DHT nodes during our experiments.

Distribution	Latency [ms]
Min	2
Quartile 1	178
Quartile 2	225
Quartile 3	269
Max	350

TABLE I: Distribution of latencies among DHT nodes

Query set (QS)	B. Box $M \times M(km^2)$	Time interval (H)
1	2×2	1
2	2×2	48
3	200×200	1
4	200×200	48
5	ALL	1
6	2×2	ALL

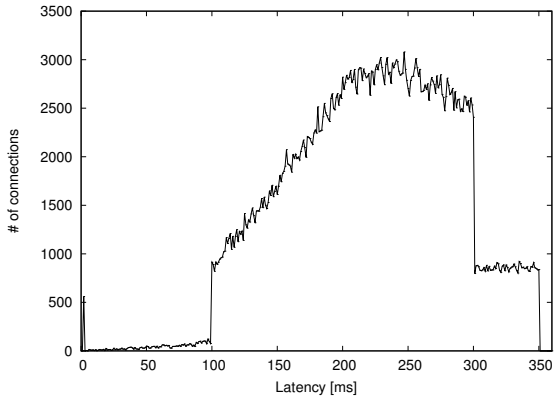
TABLE II: Location-temporal range query sets

1) Query set: In order to assess the performance of GeoTrie under different range query spans, we generated six query sets (QS) which cover different geospatial bounding box sizes and timespans as follows. First, we randomly picked $1,000$ tuples $t_k = (latitude, longitude, timestamp)$ from the dataset. Then, for every query set and for every tuple, we generated a query for a geospatial bounding box which encloses a $M \times M$ kilometres square with a time interval of H hours with centre in t_k.

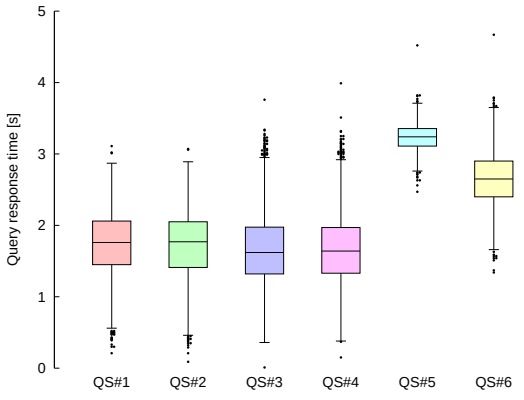
Table II presents the size of M and H we chose for this evaluation and Table III presents the distribution of the data items that match every query set. These queries represent different bounding box sizes and timespans. For instance, query sets **QS1** and **QS2** represent highly selective queries. They target bounding boxes of 2×2 km^2 and timespans of 1 and 48 hours respectively. Query sets **QS3** and **QS4** target bigger bounding boxes of 200×200 km^2, with respective timespans of 1 and 48 hours. The query set **QS5** targets all the space domain with a timespan of one hour. Finally, the query set **QS6** targets all the time domain with a bounding box of two kilometres side size. Query sets **QS5** and **QS6** are specific in that the former covers the entire geospatial domain and the latter covers the entire temporal domain. Therefore, they both always start at the root node because of the absence of a common prefix. We believe that these combinations of queries represent a rich workload to assess the query load balance and performance of GeoTrie.

B. Data insertions

1) Message Complexity: Let N be the number of nodes in the DHT, n the number of keys to be indexed, and D the number of bits used to represent the data domain $\{0,1\}^D$ of every coordinate of the tuple key T_k. The insertion/deletion of a data object identified by T_k involves a binary lookup in order to find the *leaf node* whose label is a prefix of T_k.



(a) Distribution of latencies among DHT nodes



(b) Location-temporal range query response time

Fig. 3: Latency among DHT nodes and GeoTrie Performance

Dist.	QS1	QS2	QS3	QS4	QS5	QS6
Min.	1	1	1	1	60	1
Quartile 1	6	12	15	95	535	259
Quartile 2	21	37	36	256	771	1,393
Quartile 3	56	113	81	612	1,027	9,089
Max.	5,353	62,333	6,079	77,629	6,694	215,647
Avg.	50	169	68	555	825	14,008

TABLE III: Distribution of the number of data items that match every query set QS

Dist.	QS1	QS2	QS3	QS4	QS5	QS6
Min	1	1	1	1	796	4
Quartile 1	1	1	5	5	1693	15
Quartile 2	1	1	9	9	2023	23
Quartile 3	1	1	14	14	2161	38
Max	31	82	54	108	2227	284

TABLE IV: Query cost distribution.

Equation 3 presents the message complexity in terms of the number of DHT lookup messages generated by a single insertion/deletion operation. Note that the message complexity of every DHT lookup is $O(\log(N)$. So the total number of messages generated by a single insertion/deletion operation is given by equation 4.

$$C_{ins/del}(D) = O(\log(D)) \tag{3}$$

$$C_{ins/del\_total} = O(log(D)) \times O(log(N)) \tag{4}$$

The message complexity of insertions and deletions on GeoTrie does not depend on the number of objects n. A balanced tree index built on top of a DHT requires $O(\log(n))$ DHT lookups. Via binary searches on a trie whose height cannot exceed 32, GeoTrie outperforms a balanced tree for large values of n. For instance, indexing a new entry among 50 million keys of size $D = 32$ bits requires about 25 DHT lookups with a tree, while GeoTrie does the job in about 5 DHT lookups.

Distribution	Time[s]
Min	1 x 10^{-3}
Quartile 1	1.80
Quartile 2	2.84
Quartile 3	3.33
Max	6.12

TABLE V: Insertion latency distribution

# DHT lookups	Percentage [%]	Min. Time[s]	Avg. Time [s]	Max Time [s]
1	0.05	1 x 10^{-3}	0.83	1.62
2	27.79	1 x 10^{-3}	1.47	3.20
3	2.01	0.387	2.12	3.90
4	20.27	0.49	2.71	5.05
5	49.84	0.82	3.34	6.12
6	0.04	2.32	3.88	5.76

TABLE VI: Number of DHT lookups per insertion and its latency distribution

2) Insertion time: In this experiment, we indexed all the meta-data of $48,469,177$ geotagged multimedia files extracted from the YFCC dataset [14] at a rate of $1,000$ items per second. At every insertion, a DHT node is chosen randomly in order to perform the operation. Then, we measured the insertion time as the time elapsed between the moment the client node starts the insertion process until the moment the leaf node successfully stores the item.

Table V presents the overall distribution of insertion latencies. Most of the latencies are concentrated between 1.8 and 3.3 seconds with a median of 2.84 seconds. The insertion time depends of the size of the prefix label of the target *leaf node*. Some insertions can directly arrive to the target *leaf node* through the binary search algorithm presented in section II. Others insertions require more messages to locate the target *leaf node*. In order to understand the behaviour of the binary search we measured the distribution of the number of DHT lookups in the insertion process. Table VI presents our results. Since the binary search algorithm is inherently sequential, the number of DHT lookups has a strong impact on the insertion latency. For instance, most insertions (49.84%) require 5 DHT lookups. They present an average latency of 3.34 seconds. Instead, insertions that require 2 lookups (27.79 %) present an

(QS#)	Min	Quartile 1	Quartile 2	Quartile 3	Max	Avg
1	0.21	1.45	1.76	2.06	3.11	1.72
2	0.09	1.41	1.77	2.05	3.07	1.69
3	0.01	1.32	1.62	1.97	3.76	1.68
4	0.15	1.33	1.64	1.97	3.99	1.69
5	2.47	3.11	3.24	3.35	4.52	3.23
6	1.34	2.4	2.65	2.90	4.67	2.66

TABLE VII: Location-temporal range query response time in seconds

Distribution	QS1 SP	QS3 SP
Min	0	0
Quartile 1	5	3
Quartile 2	6	5
Quartile 3	6	5
Max	12	6

TABLE VIII: Distribution of the GeoTrie level starting point for **QS1** and **QS3**

average insertion latency of 1.47 seconds (2.27 times faster).

C. Location-temporal range queries

1) Message Complexity: A location-temporal query on GeoTrie first reaches the node which maintains the common prefix label of minimum common size, and then branches out in parallel down the tree until it reaches all *leaf nodes*. Equation 5 computes the number of messages for this operation.

$$O(log(N)) \leq C_{range-query} \leq O(log(N)) + D + 1 \quad (5)$$

The lower bound occurs when the query arrives directly to a single *leaf node* which covers the interval, and the upper bound occurs when there is no common prefix of minimum size. In the worst case, the query arrives at the root node labeled $l = (*, *, *)$ and traverses the whole tree in parallel through a maximum of $D + 1$ levels (the root node plus the maximum prefix size D=32). Note that the global implicit knowledge introduced by GeoTrie allows to reach the query subspace directly when it does not include the root node. Unlike queries that cover large geographic areas and timespans, queries which target small geographic areas and timespan share a larger common prefix of minimum size, and therefore directly arrive to higher levels of GeoTrie (i.e., close to the leaves). Indexes relying on a balanced tree usually start queries from the root node, and thus induces bottlenecks.

2) Range query performance: In this experiment, we study queries whose prefix maps to any other node than the root node. More specifically, we assess their impact on the query response time under a workload composed of concurrent range queries. In order to conduct this experiment, we generated a workload of $1,000$ queries per second for every set **QS1** to **QS5**. Every query originates from a DHT node chosen at random. It consists in counting the number of items inside the location-temporal range.

Figure 3b and table VII presents the query response time measured as the time elapsed between emission of the query and the latest reception of results from all the *leaf nodes* that store data relevant to this query. Only about 1% of the query response time corresponds to processing time (i.e, the time it takes a *leaf node* to filter-out all the data outside the query interval). The remaining 99% of the time corresponds to communication latency. Indeed, in our experiment the meta-data is stored in memory, and therefore the local processing cost of a query over a single *leaf node* is much lower than the network latency cost. As expected, queries that span a shorter space present a lower query response time than queries that span a larger space. For instance, the average query response time is 1.9x slower for queries of **QS3** than for queries of **QS5**. Indeed, we tailored queries in this experiment so that they can avoid the root node. Our results show that Geotrie thus allows to alleviate a potential bottleneck on the root node, and this has a considerable impact on the query response time.

In order to understand how queries are distributed among the GeoTrie structure we measured the GeoTrie level starting point (SP) for **QS1** and **QS3**. We exclude both **QS5** and **QS6** because by construction they always start at the root node. Table VIII presents our results. Only 0.3% of queries of **QS1** (3 out of 1,000) starts at the root node, and 75% of the total amount of queries starts at a level greater or equal than 5. In the case of **QS3**, 8.3% of the queries (83 out of 1,000) started at the root node and the 75% of the queries started at a level greater or equal than 3. **QS3** queries target bigger areas, and therefore match shorter common prefixes in the trie structure.

IV. Related Work

Large scale architectures to store, query and analyse *big location data* constitute a fast-growing research topic. Current solutions fall into four groups: (i) Hadoop-based solutions; (ii) Resilient Distributed Dataset (RDD) based solutions; (iii) Key-value store-based solutions, and (iv) DHT-based solutions.

Hadoop-based solutions such as SpatialHadoop [5], Hadoop GIS [6], and ESRI Tools for Hadoop [7], extend the traditional Hadoop architecture [16] with spatial indexing structures such as R-Trees [17] or Quad-Trees [18] in order to avoid a scan of the whole dataset when performing spatial analysis. These approaches employ a two-layer architecture that combines a global index maintained on a central server with multiple local indexes. For instance, SpatialHadoop [5] builds spatial indexing structures over HDFS [6] in order to add spatial support for MapReduce tasks. However, these solutions are ill-suited for concurrent insertions and location-temporal queries because (i) the global index structure on a single node is prone to become a bottleneck, and (ii) they are designed for batch processing of large tasks. Unlike these solutions, GeoTrie constitutes a large scale global location-temporal index which provides random access and fault tolerance for concurrent insertions and location-temporal queries.

Resilient Distributed Dataset (RDD) based solutions such as Spatial Spark [19] and GeoTrellis[4] extend traditional RDD solutions such as Spark [20] in order to support big location data. Similarly to Hadoop-based solutions, these systems are

[4]http://geotrellis.io

designed for batch processing and do not target online processing.

Key-value store-based solutions such as MD-Hbase [9], MongoDB [21], ElasticSearch [22], and GeoMesa [10] require linearization techniques such as *space-filling curves* [8] in order to collapse several dimensions into a single one-dimensional index and to support multi-dimensional queries. Then, the multidimensional query can be reduce to a single dimensional space. However, *space-filling curves* [8] loosely preserve data locality and introduce several I/O overheads when the number of dimensions increase due to the *curse of dimensionality* [8]. These overheads impacts negatively the query response time. Unlike these solutions, GeoTrie follows a multidimensional approach which drastically reduces this overhead.

DHT-based solutions can be classified into two main groups: (i) extensions of traditional indexing structures such as B-Trees, Prefix Trees, R-Trees, QuadTrees, and KDtrees to DHTs [23]–[27], and (ii) overlay-dependent solutions [28]–[31]. Compared to the solutions of the first group, GeoTrie performs *domain partitioning* to prevent bottlenecks on the root node, and introduces global knowledge about the tree structure to balance the load of insertions and range queries. To the best of our knowledge, the only structure which performs *domain partitioning* over a DHT is the Prefix Hash Tree (PHT) [23]. PHT can handle multi-dimensional data using *linearisation* techniques such as *z-ordering*. As mentioned previously, however, dimensionality reduction with respect to space and time introduces query overheads due to the *curse of dimensionality*. GeoTrie reduces this overhead because it uses a multidimensional structure and, unlike the solutions of the second group, it is portable to any DHT.

V. Conclusion

This paper presents GeoTrie, a scalable architecture for massive geotagged data storage and retrieval built on top of a DHT. Our solution indexes the location and temporal dimensions of every meta-data on a multidimensional distributed structure which scales and balances the load of insertions and range queries. A theoretical analysis of the message complexity of every operation on GeoTrie demonstrates its scalability. An extensive experimental evaluation over a Yahoo! dataset comprising about 48 millions of multimedia files shows that our solution balances the load of insertions and range queries on a large scale. In a configuration involving $1,000$ nodes, queries which avoids the root node presents an average query response time up to 1.9x faster than queries which starts at the root level. This property of GeoTrie allows to alleviate a potential bottleneck on the root node. We are currently working on an extension of our proposal to handle n-dimensional range queries over massive data sets.

References

[1] N. Pelekis and Y. Theodoridis. The case of big mobility data. In *Mobility Data Management and Exploration*, pages 211–231. Springer, 2014.

[2] A. Eldawy and M.F. Mokbel. The era of big spatial data: Challenges and opportunities. In *Mobile Data Management (MDM), 2015 16th IEEE International Conference on*, volume 2, pages 7–10, June 2015.

[3] Ranga Raju Vatsavai, Auroop Ganguly, Varun Chandola, Anthony Stefanidis, Scott Klasky, and Shashi Shekhar. Spatiotemporal data mining in the era of big spatial data: algorithms and applications. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 1–10. ACM, 2012.

[4] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30(2):170–231, 1998.

[5] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1352–1363, 2015.

[6] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.

[7] Randall T Whitman, Michael B Park, Sarah M Ambrose, and Erik G Hoel. Spatial indexing and analytics on hadoop. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 73–82. ACM, 2014.

[8] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

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

[10] Anthony Fox, Chris Eichelberger, John Hughes, and Skylar Lyon. Spatio-temporal indexing in non-relational distributed databases. In *Big Data, 2013 IEEE International Conference on*, pages 291–299. IEEE, 2013.

[11] Kisung Lee, Raghu K Ganti, Mudhakar Srivatsa, and Ling Liu. Efficient spatial query processing for big data. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 469–472. ACM, 2014.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[13] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[14] B Thomee, DA Shamma, G Friedland, B Elizalde, K Ni, D Poland, D Borth, and LJ Li. Yfcc100m: The new data in multimedia research. *Communications of the ACM*, 2015.

[15] Peter Druschel, Eric Engineer, Romer Gil, Y Charlie Hu, Sitaram Iyer, Andrew Ladd, et al. Freepastry. *Software available at http://www. cs. rice. edu/CS/Systems/Pastry/FreePastry*, 2001.

[16] T. White. *Hadoop: the definitive guide: the definitive guide*. " O'Reilly Media, Inc.", 2009.

[17] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[18] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[19] Simin You, Jianting Zhang, and L Gruenwald. Large-scale spatial join query processing in cloud. In *IEEE CloudDM workshop (To Appear) http://www-cs. ccny. cuny. edu/˜ jzhang/papers/spatial_cc_tr. pdf*, 2015.

[20] M. Zaharia, M. Chowdhury, M. J. Franklin, and S Shenker. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[21] Kristina Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[22] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide*. " O'Reilly Media, Inc.", 2015.

[23] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M Hellerstein, and Scott Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. In *Proceedings of the 23rd ACM symposium on principles of distributed computing*, volume 37, 2004.

[24] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal*, 16(2):165–178, 2007.

[25] Cédric Du Mouza, Witold Litwin, and Philippe Rigaux. Large-scale indexing of spatial data in distributed repositories: the sd-rtree. *The VLDB Journal?The International Journal on Very Large Data Bases*, 18(4):933–958, 2009.

[26] Chi Zhang, Arvind Krishnamurthy, and Randolph Y Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. *Department of Computer Science, Princeton University, New Jersey, USA, Tech. Rep*, pages 703–04, 2004.

[27] Rudyar Cortés, Olivier Marin, Xavier Bonnaire, Luciana Arantes, and Pierre Sens. A scalable architecture for spatio-temporal range queries over big location data. In *14th IEEE International Symposium on Network Computing and Applications-IEEE NCA'15*, 2015.

[28] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 591–602. ACM, 2010.

[29] Rong Zhang, Weining Qian, Aoying Zhou, and Minqi Zhou. An efficient peer-to-peer indexing tree structure for multidimensional data. *Future Generation Computer Systems*, 25(1):77–88, 2009.

[30] Hosagrahar V Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st international conference on Very large data bases*, pages 661–672. VLDB Endowment, 2005.

[31] Yuzhe Tang, Jianliang Xu, Shuigeng Zhou, and Wang-chien Lee. m-light: indexing multi-dimensional data over dhts. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 191–198. IEEE, 2009.