A Scalable Architecture for Spatio-Temporal Range Queries over Big Location Data

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

[1]Université Pierre et Marie Curie, CNRS
INRIA - REGAL, Paris, France
E-mail: [rudyar.cortes, olivier.marin, luciana.arantes, pierre.sens]@lip6.fr
[2]Universidad Técnica Federico Santa María, Valparaíso, Chile
E-mail: xavier.bonnaire@inf.utfsm.cl

***Abstract*—Spatio-temporal range queries over Big Location Data aim to extract and analyze relevant data items generated around a given location and time. They require concurrent processing of massive and dynamic data flows. Current solutions for Big Location Data are ill-suited for continuous spatio-temporal processing because (i) most of them follow a batch processing model and (ii) they rely on spatial indexing structures maintained on a central master server. In this paper, we propose a scalable architecture for continuous spatio-temporal range queries built by coalescing multiple computing nodes on top of a Distributed Hash Table. The key component of our architecture is a distributed spatio-temporal indexing structure which exhibits low insertion and low index maintenance costs. We assess our solution with a public data set released by Yahoo! which comprises millions of geotagged multimedia files.**

***Index Terms*—Big Location Data; Spatio-Temporal Processing; Distributed Hash Tables.**

I. Introduction

The proliferation of Location Based Social Networks Services (LBSNS) leads to the continuous and dynamic generation of geotagged data from millions of GPS-enabled devices. Every minute, there are around $216,000$ new pictures uploaded on Instagram[1]. Twitter generates more than 10 million geotagged tweets every day, which represents 2% of the whole Twitter data flow [2]. A recent public data set released by Yahoo! reveals that around 50% of public pictures and videos uploaded to Flickr[3] are geotagged [1]. Sport tracker applications such as MyFitnessPal[4] and RunKeeper[5] generate millions of GPX[6] files shared over social networks on a daily basis.

These applications generate massive flows of insertions (very few deletions) which can not be acquired, managed, and processed by traditional central solutions within a tolerable time. They constitute a new field of research known as *Big Location Data* [2].

Data items generated by these applications have three common core attributes $T = (x, y, t)$ composed of a location (x, y) and a time attribute t. For the sake of clarity, let (x, y) represent the spatial *latitude* and *longitude* of GPS coordinates, and t the number of seconds elapsed since the Epoch: January 1st, 1970 00:00 (UTC).

Performing spatio-temporal range queries over massive and dynamic data sets allows users to extract relevant information around a given location and time. For instance the spatio-temporal range query *"Retrieve all pictures tagged with* {*#Cat, #Lost*} *generated inside a geographic bounding box A and uploaded between* t_0 *and* t_1*"* extracts all pictures of lost cats uploaded by users inside a geographic area and during a given time interval. This kind of query is both I/O intensive and processing intensive because it can cover billions of objects made available via concurrent data insertions.

Recent solutions [3]–[6] extend traditional big data architectures such as Hadoop [7] and Spark [8] in order to provide efficient spatio-temporal data access over Big Location Data sets. However, there are three reasons why these solutions are ill-suited for online spatio-temporal processing. Firstly, they rely on a central master server to maintain spatial indexing structures such as R-Trees [9] and QuadTrees [10]; thus they induce bottlenecks for massive and dynamic input loads. Secondly, these traditional spatial indexing structures lack a time index in order to provide efficient spatio-temporal data access. Finally, most of them follow a batch processing model. It is our belief that distributed spatio-temporal indexing structures can cope better with typical workloads associated with Big Location Data processing.

In this paper, we propose Big-LHT: a scalable architecture that coalesces any number of commodity machines on top of a Distributed Hash Table (DHT) in order to perform continuous spatio-temporal processing. Big-LHT relies on a novel distributed spatio-temporal indexing structure, called Location Hash Tree (LHT); its index maintenance is cost-efficient, and it offers low insertion costs. A full evaluation of Big-LHT, using a real world data set composed of millions of geotagged pictures released by Yahoo!, assesses its scalability for online spatio-temporal range queries.

The main contributions of this paper are :

- a large scale architecture built on top of a DHT which distributes the massive flow from location-aware input

[1]http://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic/
[2]https://www.mapbox.com/blog/twitter-map-every-tweet/
[3]https://www.flickr.com
[4]https://www.myfitnesspal.com
[5]http://runkeeper.com
[6]GPS eXchange Format

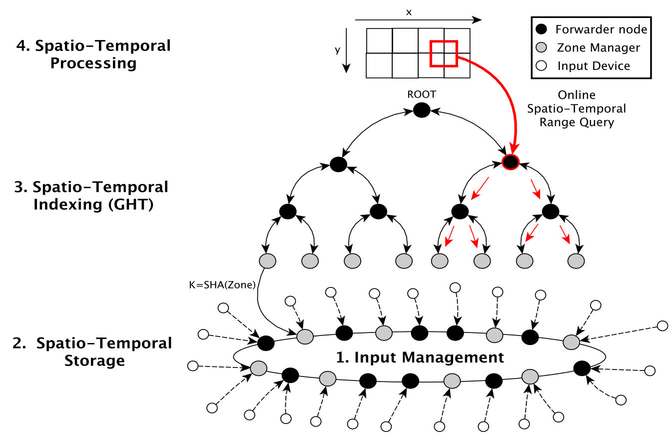


Fig. 1: Architectural overview of Big-LHT

devices in order to perform spatio-temporal storage and processing,
- a location-aware distributed structure which indexes data based on location and time, and incurs low index maintenance costs compared to current spatial indexing structures.

Section II describes our system in detail. Section III assesses our solution by computing the message complexity of every operation, and by conducting an experimental evaluation with a real world dataset. Section IV discusses related work, and finally Section V presents our conclusions and future work.

II. Big-LHT Design

This section details the architecture we propose for continuous scalable spatio-temporal range query processing of *Big Location Data*. The goal of our approach is to provide a scalable architecture which distributes massive and dynamic input flows on a network of commodity machines (cluster, Cloud, Desktop Grid, peer-to-peer) and supports efficient spatio-temporal data access for continuous range queries.

Figure 1 presents the overall architecture of Big-LHT and its protocol stack as detailed in the next sections. In order to distribute the input data flows, the *input management* layer associates the GPS-enabled devices that upload data with a DHT service, thus preventing input bottlenecks in the system. The *spatio-temporal storage* layer creates data locality by partitioning the input data set and distributing it on nodes that control geographic zones. Every node in charge of a zone maintains a *time index*. The *spatio-temporal indexing* layer introduces a novel spatio-temporal indexing structure, called Location Hash Tree (LHT), that supports spatio-temporal range queries and parallel data access at a low index maintenance cost. Finally, the *continuous spatio-temporal processing* layer provides an interface for continuous spatio-temporal range queries.

We assume that all nodes involved in Big-LHT support the Network Time Protocol, and thus maintain their local time within a small deviation from the Coordinated Universal Time (UTC). Thus every newly uploaded data item comes with a timestamp corresponding to the local clock value of the node that stores it.

A. Input management

The *input management* layer breaks down massive input flows of spatio-temporal data generated by GPS-enabled devices. We refer to such devices as *input nodes*, and to nodes that participate to the input management layer as *DHT service providers* (DHT-SPs). To avoid bottlenecks, this layer enforces a uniform distribution when assigning DHT-SPs to input nodes. For this purpose, the *input management* layer relies on a distributed hash table (DHT).

DHTs [11], [12] use a cryptographic hash function to map keys and node identifiers in the same namespace. The hash function usually guarantees natural load balancing. DHTs generally provide very efficient routing on a large scale, with a message complexity commonly of $O(log(N))$ for any operation, where N is the number of nodes involved. Most DHT implementations also offer dependability via replication.

Let I_i be an *input node*: $S^i = \{S_1, ..., S_n\}$ is the set of n *DHT nodes* that provide I_i with an access to the input management service. Every node gets an identifier computed by applying the SHA-1 hash function to its IP address: an *inputId* identifies an *input node*, a *nodeId* identifies a *DHT node*. By construction, S^i is the set of n *DHT nodes* whose *nodeIds* are numerically closest to the *inputId* of I_i. S_{root} is the node whose *nodeId* is closer than the *nodeId* of any other member of S^i.

When an *input node* I_i enters the system, it uses the DHT to route a *service request* message to S_{root}. Upon reception of this message, S_{root} replies directly to I_i with the IP addresses of all nodes in S^i. Thus every *input node* I_i maintains a list with n DHT-SPs which provide a routing service inside the DHT. The hash function ensures a uniform distribution of *input nodes* among *DHT nodes*. The maintenance of the DHT-SPs list is optimistic: an input node that fails to get a reply from a *DHT node* routes a new service request via the DHT to acquire an updated list.

B. Spatio-temporal storage

The spatio-temporal storage layer reintroduces spatio-temporal data locality over a DHT. To this end, this layer combines Geohashes and time indexing.

Geohashes[7] map a bi-dimensional GPS coordinate (x, y) into a single one-dimensional binary key $z(x, y)$. They use a z-ordering space filling curve [13] which loosely preserves data locality. The z-ordering function $z(x, y)$ interleaves the bits of every coordinate. For instance, if $x = 000$, and $y = 111$ then $z(x, y) =$ 01 01 01. This is a core technique used by Big-LHT in order to create a spatio-temporal index over a DHT.

Geohashes recursively partitions the (longitude,latitude) spatial domain. For instance, the first prefixes $\{0*, 1*\}$ divide the longitude dimension in two: Geohashes beginning with

[7] http://geohash.org

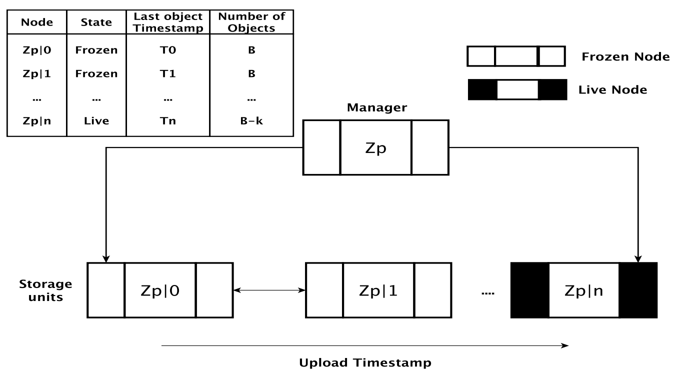


Fig. 2: Management of storage units

'0' define the northern hemisphere. The next four prefixes $\{00*, 01*, 10*, 11*\}$ divide the latitude dimension for every longitude division, resulting in four equal spaces, and so on. This recursive prefix space division guarantees two properties. *Recursive domain division*: a Geohash of size $i+1$ defines a rectangle contained by a Geohash of size i. *Spatial Locality*: shared prefixes imply closeness. For instance, all keys with the same common prefix 000* are necessarily contained in a single rectangle represented by this prefix.

Geohashes alone are ill-suited to perform spatio-temporal range queries because they lack a time index. Our solution, described below, remedies this deficiency.

Let $Z_p(x, y)$ be a prefix of size p for a given Geohash which covers a rectangular geographic area of size R. $Z_p(x, y)$ partitions the surface of the earth in 2^p rectangular zones of area R, where p is a system parameter which depends on the application. A high value of p generates small rectangular zones whereas a small value generates big zones.

We dynamically associate a rectangular geographic zone $Z_p(x, y)$ with the *DHT node* whose *nodeId* is closest to key $K = SHA(Z_p(x, y))$. We refer to this node as the *manager* of the geographic zone Z_p. From here on, we will note $Z_p(x, y)$ as Z_p.

In order to distribute storage resources and to improve spatio-temporal data access, every *manager* holds the label $l = Z_p$ that identifies its assigned zones, and handles a dynamic storage structure composed of nodes that act as *storage units*. Figure 2 depicts our dynamic storage structure. *Storage units* form a double linked list sorted by upload time. The last *storage unit* of the list stores the most recently uploaded data items. This structure scales horizontally, and allows sequential access to temporal data as well as efficient extraction of the most recent data uploads in a given zone. The *manager* also maintains a *storage unit table* which maps the identifier of each *storage unit* to (i) the upload timestamp t of the last item it stores and (ii) the number of objects that it stores. This table provides low latency parallel data access extended with a time index to allow searches over given time intervals. The maximum size T_{max} of this table is a parameter of the system.

The assignment of zones to *managers* is dynamic. When a *DHT node* receives an input data item inside a new zone Z_p, it sets its state as *manager* and its label as Z_p. Then it forwards the message to the first *storage unit*: the *DHT node* whose nodeId is closest to key $K = SHA(Z_p|0)$. The receiving node sets itself as a *storage unit* for Z_p. It follows that the system only creates *managers* for geographic zones where input data has actually been generated. This prevents node provisioning for zones that are empty.

The label $l = (Z_p|i)$ of a *storage unit* determines the *DHT node* it gets assigned to: the node whose *nodeId* is closest to key $K = SHA(l)$. Every *storage unit* holds: its *label* that identifies the zone Z_p for which it stores data, its *state* which can be either *frozen* or *live*, in-memory user-generated metadata such as tags and comments, a persistent data storage space which stores up to B data items, and references to its immediate predecessor and successor nodes. Both *frozen nodes* and *live nodes* store data, but only *live nodes* can receive new incoming data. Once a data item arrives to its assigned *live node*, it is associated with a *timestamp* t corresponding to the local clock value.

Storage unit index maintenance. Big-LHT provides two main operations, *split* and *merge*, to maintain the index of the storage units structure. The *split* operation allocates a new storage unit when a *live node* reaches its maximum storage capacity B. Conversely, when two consecutive *frozen nodes* store less than B items they *merge* into a single *storage unit*.

The *split* index maintenance works as follows. First, the *live node* changes its state to *frozen* and notifies its manager. Upon reception of a storage request at position $B+1$, the *manager* forwards the request to a new *storage unit* labeled $l_{new} = (Z_p|i+1)$. The new *storage unit* sets itself to *live* and sends two ACKs: one back to the *manager* and one to its now *frozen* predecessor in order to update the double linked list structure.

A *merge* maintenance operation occurs when two consecutive *storage units* have less than B data items. First, the *manager* node sends a *merge* message to the two consecutive *storage units*. The *manager* selects the node which has the lowest objects counter to move the data items and provides the link to its merge in order to update the double linked list structure. Upon reception of this message the storage unit transfers all stored data to its sibling node, removes its label $l = Z_p|i$, and sends back an ACK to the *manager* in order to remove its entry from the *storage unit table*. This strategy reduces the data movement cost of the *merge* operation.

Upon deletion of the last data item inside a zone, the *manager* node leaves the zone and sends a *leave* message to the last remaining *storage unit* so that it removes its label $l = Z_p|i$.

Manager index maintenance. Our storage management scales outby allocating a new *manager* for a given zone Z_p when the *storage unit table* reaches T_{max} and the last *live* node becomes *frozen*. This procedure works as follows. Upon reception of a new data insertion the *manager* provides a new *manager* by forwarding the insertion request to the *DHT node* whose nodeId is closest to key $k^{i+1} = SHA^{i+1}(Z_p)$. k^{i+1} results from applying the SHA-1 hash function recursively $i+1$ times. That is, $k^{i+1} = SHA(SHA(...(Z_p)))$ $i+1$ times. For instance, $i=0$ generates key $k^0 = SHA(Z_p)$.

Upon reception of this message, the *DHT node* sets itself as *manager* of Z_p at level i and sends an ACK back to the *manager* at level $i-1$ and another ACK to the *manager* at level $i=0$. We refer to the manager of level $i=0$ as the *root manager* of Z_p.

Similarly to storage units, managers form a double linked list: every *manager* maintains a link both to its predecessor and to its successor. Additionally, the *root manager* maintains a link to the *manager* which holds a *live storage unit* for this zone.

When a *manager* leaves a zone Big-LHT provides a *merge* operation which works as follows. First, it sends a *leave message* both to its predecessor and to its successor so that they update the double linked list. The *root manager* leaves a zone only if it is the last *manager* in the double linked list.

Data insertions. Storing a data item consists in locating the *live node* associated with the zone where the data item is generated. This operation is implemented on this layer as follows. First, the *Input node* sends a *StorageRequest* message to one of its *DHT-SPs* nodes. Upon reception of this message the *DHT-SPs* node translates the location coordinates (x, y) to their GeoHash representation and extracts prefix Z_p. Then, it routes the message to the node whose nodeId is closest to key $K = SHA(Z_p)$.

When this node receives this message there are three possibilities. a) The message was generated inside a new zone Z_p: In this case, the *DHT node* which receives this message sets itself as *root manager* for Z_p and forwards the request to the first *storage unit* labeled $l = (Z_p|0)$. Upon reception of the storage request, the new storage unit sets itself as *live node* for Z_p and sends an ACK back to the source *Input node* in order to finish the transaction. b) The message was generated inside an existing zone Z_p: In this case, the manager receives the request and forwards this message to the *live storage unit* which in turn sends an ACK back to the source node in order to finish the transaction. c) The message arrives to a *manager* with a full *Storage unit table*. In this case, the node forwards the query to a new *manager* at level i which forwards the query to the *live* node.

Data deletions. Deleting an object with an identifier id, generated at GPS coordinates (x, y), and successfully uploaded at a time t, works as follows. First, the sender *DHT node* routes the deletion request to the *root manager* of the zone. The request traverses the double linked list until it reaches the *manager* which covers t. This node forwards the query to the *storage unit* whose time index contains the data item.

C. Spatio-temporal indexing

The main goal of this layer is to provide support for scalable spatio-temporal range queries. That is, given any spatial bounding box $B = (s_l, s_h)$ this layer must find all existing zones (i.e. *manager* nodes) which hold zones inside B.

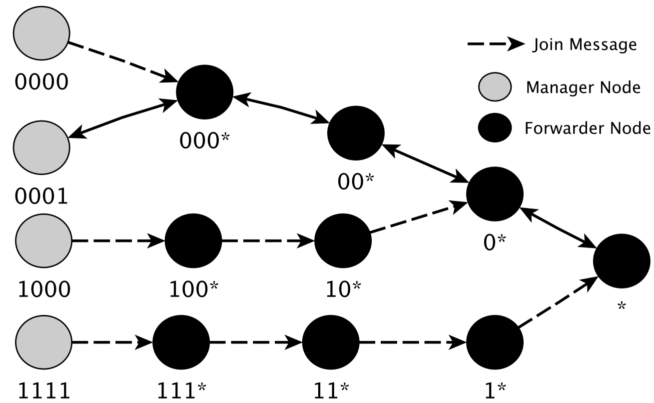


Fig. 3: Example of an index with LHT

Indexing data Structure. We index all zones created by the *spatio-temporal storage layer* in a novel indexing structure named Location Hash Tree (LHT). LHT exploits the *recursive domain definition* property of Geohashes as follows. Upon creation of a new *manager* M with label Z_p, the node routes a *JOIN* message via the DHT to the node whose label l is the prefix of Z_p. We refer to this node as the *forwarder* of M.

Upon reception of a *JOIN* message there are two possible cases. (i) The receiving node does not belong to the indexing structure. In this case, the node sets its state as *forwarder* node and adds the joining node as its child. Then it routes a *JOIN* message to the *nodeId* that is closest to the prefix of its label. (ii) The node that receives the *JOIN* message belongs to the indexing structure. In this case, the node just adds the joining node as its child. If there is no *forwarder* for any prefix of this label, this process gets repeated until it reaches the root node.

The *JOIN* message contains the label of manager Z_p and its level i. Every *forwarder* maintains a *children table* which contains three entries: the *label* Z_p of every child, its direct IP address, and its level i in case the child node is a *manager*.

Figure 3 presents an example of the LHT indexing structure with $p = 4$, that is *managers* have a label Z_p of size 4. The *manager node* with label 0000 joins LHT. In this example, this node routes a *JOIN* message via the DHT to the node labeled $l = 000*$. Since the latter is already a *forwarder node*, the join process finishes. The join process can generate up to p recursive join messages if there is no *forwarder* along the path until the root node is reached. For instance, when the *manager* labeled 0000 joins LHT, its insertion generates $p = 4$ recursive join messages until the root node labeled $l = *$ is reached.

When a *manager* node leaves a zone, it sends a *LEAVE* message to its *forwarder* which deletes the entry from its children table. If the *manager* is the last node in its table, this node leaves LHT by sending a *LEAVE* message to its parent. This process iterates until it reaches a *forwarder* with

// m is the range query message;
// R is the query result
z_i = GeoHash(s_l);
z_f = GeoHash(s_h);
// Computes the common prefix of maximum size p;
shared-prefix = commonPrefix(z_i, z_f, p);
K = SHA(shared-prefix);
node = route(m,K);
if *node is a forwarder node* **then**
    // recursively forward the query until all manager nodes are reached;
    node.forward(z_i,z_f);
end
if *node is a manager node* **then**
    // Forward the query to all storage units which covers the time interval;
    node.forward(t_i, t_f);
end
if *node is a storage unit* **then**
    // Get all the data which match the range query constraints;
    R = getData(s_l, s_h, t_i, t_f);
end
if *node is an external node* **then**
    // There is no data in the spatio-temporal interval;
    $R = \emptyset$;
end

Algorithm 1: Spatio-temporal parallel range query processing pseudocode

at least one entry in its *children table*.

LHT follows a *prefix tree* (trie) indexing structure similar to Prefix Hash Tree (PHT) [14]. Both solutions follow a prefix tree strategy to index data, but they are different in three key aspects: (i) LHT grows along a bottom-up data flow, (ii) LHT generates no leaf nodes (managers) without data, and (iii) LHT defines a grid where every leaf node (managers) holds a space of area A and introduces a time index with horizontal splits.

D. Spatio-temporal range queries

We now detail how spatio-temporal range queries are performed on top of Big-LHT.

Given a spatial bounding box $B = (s_l, s_h)$ and a time interval $[t_i, t_f]$, a spatio-temporal range query retrieves all objects uploaded within $[t_i, t_f]$ and generated inside (s_l, s_h). (s_l, s_h) are the GPS coordinates of the lower and higher limits of bounding box B.

We introduce two algorithms to implement spatio-temporal range queries on Big-LHT. Both algorithms exploit the *spatial locality* property of Geohashes where all spatial items which share the same common prefix are in the same spatial area.

The first algorithm combines parallel an sequential data access. The sender node computes the Geohash of the two spatial bounding box limits (s_l, s_h) and routes the query via the DHT to the node whose label of maximum size p shares the common prefix string between s_l and s_h. By the *spatial locality* property of Geohashes this node covers the whole spatial bounding box (s_l, s_h).

Depending on the state of the node which receives the query we identify three cases. (i) The node is a *forwarder* node. In this case, it recursively forwards the query onward to the *managers* nodes that cover the spatial range. (ii) The node is a *manager* node. In this case, a single node covers the required spatial range. (iii) The node is an *external* node (i.e, neither a *forwarder* node nor a *manager* node). In this case there is no data in the given spatial range. When a *manager* receives the query it reads the time range $[t_i, t_f]$ and forwards the query to the *storage unit* that covers the lower time range t_i. Finally, the query sequentially crosses the double linked list structure until it reaches the *storage unit* that covers t_f.

The second algorithm provides parallel data access. It works exactly like the first algorithm until it reaches all *manager* nodes which cover the spatial range. Upon receiving a range query, every *manager* uses its *storage unit table* to forward the query to all *storage units* that cover the time range $[t_i, t_f]$. Algorithm 1 presents a pseudocode of the parallel spatio-temporal range query algorithm.

III. Evaluation

This section presents theoretical and experimental assessments of Big-LHT both for data insertions and spatio-temporal range queries. Our theoretical evaluation measures the message complexity for every operation of Big-LHT. Our experimental evaluation uses the Yahoo! public dataset [15] that comprises millions of geotagged multimedia files (photos and videos) to assess the impact of Big-LHT parameter settings on system performance.

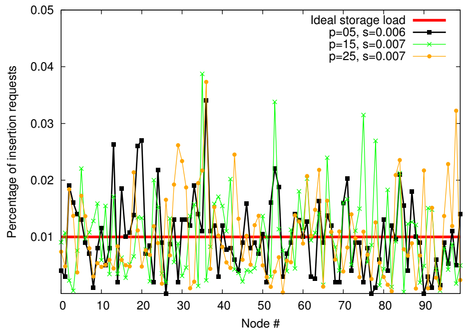
A. Theoretical evaluation

1) Data insertions: Let N be the number of nodes in the DHT. The insertion of a data file directly routes to the manager of the zone Z_p the data belongs to. Equation 1 presents the average message cost of an insertion.

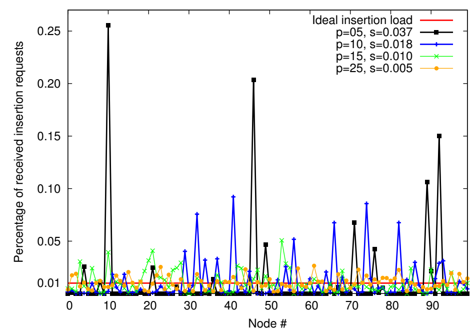
$$C_{insertion}(N) \approx \log(N) + 2 \tag{1}$$

The data insertion cost may diverge from equation 1 in two special cases. a) There is more than one *manager* for zone Z_p. In this case, the *root manager* node forwards the insertion request message to the *manager* at level i which holds the *live storage unit*; this adds one additional message. b) Either a *storage unit* or a *manager* reaches its maximum storage capacity. In this case, the next insertion request is used to dynamically create a new node which adds an average cost of $\log(N)$ routing hops to equation 1. With a message complexity of $O(\log(N))$ for insertions, we feel confident in stating that such operations scale with the number of nodes on Big-LHT.

2) Data deletions: Let i be the number of *manager* levels for a given zone Z_p. A delete operation must go through the list of i managers until it reaches the storage unit which spans the time interval of the item. Equation 2 presents the average cost of a delete operation on Big-LHT. The lower bound of this cost corresponds to cases where the data is stored by the *root*

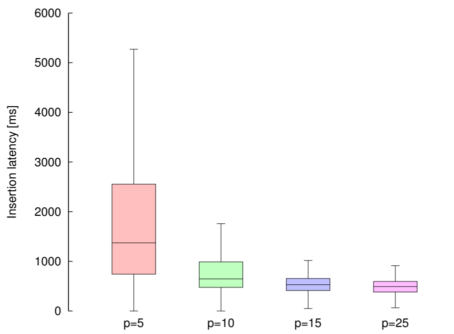


(a) Storage data distribution

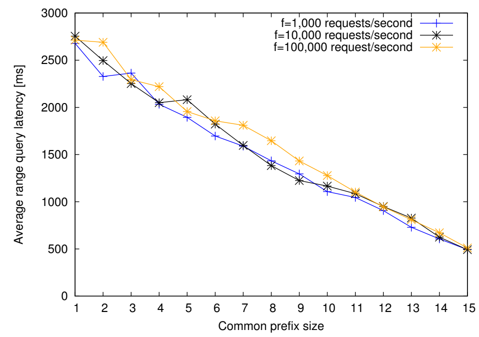


(b) Insertion load distribution on *managers* nodes

Fig. 4: Insertion and storage load balancing for different values of p



(a) Insertion latency



(b) Average range query latency for $p = 15$

Fig. 5: Insertions and range query latency

manager node. The upper bound corresponds to the deletion of data stored by the last *manager* at level i_{max}

$$\log(N) + 1 \lessapprox C_{deletion} \lessapprox \log(N) + i_{max} + 1 \tag{2}$$

Deletions have a message complexity of $O(\log(N) + i_{max})$. The value of i_{max} depends on the maximum number of entries in the *storage units table* maintained by a *manager*, which in turn depends on the memory capacity of every node. In most scenarios this is a very low value because the order of magnitude of an entry in the *storage units table* is in bytes. Delete operations cost more than insertions. In practice, LBSNS applications incur far less deletions than insertions.

The prefix domain space partitioning used by Big-LHT improves the insertion/deletion cost compared to traditional spatial indexing structures such as R-Trees [9] and QuadTrees [10] because it avoids the expensive *root-to-leaf* path. This strategy exhibits the following two benefits (i) It removes the bottleneck on the root node, and (ii) It reduces the insertion cost because data is directly addressed to the target *manager* node.

3) Storage index maintenance cost: The *split* index maintenance operation performed by a storage unit in Big-LHT consists in forwarding an insertion request to the new *live node* without any data transfer. The new node then sends two ACKs in parallel in order to update the double linked list structure. Equation 3 gives the average cost of a storage index maintenance operation.

$$C_{storage-split} \approx \log(N) + 2 \tag{3}$$

The *split* operation on Big-LHT drastically reduces the data transfer cost incurred by traditional indexing structures such as R-Trees [9] and QuadTrees [10] which can be considerable for big indexed objects such as multimedia files.

Let B be the maximum number of items stored on a single node. The *merge* index maintenance operation aims to reduce as much as possible the data transfer cost. It moves β files from the node which stores the smallest amount of data items, where $1 \leq \beta \leq B/2$. It involves the emission of three ACK messages. As LBS applications exhibit a very low rate of deletions compared to insertions, *split* operations are likely to be much more frequent than *merge* operations.

4) LHT index maintenance cost: Let p be the size of the Geohash prefix used to define geographic zones; p is a fixed parameter of the system. When a new *manager* joins/leaves LHT it sends a *JOIN/LEAVE* message which gets forwarded recursively, possibly as far as the root node. Equation 4 computes average index maintenance cost. It reaches a maximum value of $p \times \log(N)$ messages when the message reaches the root node.

$$\log(N) \lessapprox C_{LHT-index} \lessapprox p \times \log(N) \tag{4}$$

5) Spatio-temporal range query cost: Let $r = p - cp$ be the number of LHT tree levels that a spatio-temporal range query with a common prefix of length cp must traverse. Let s be the average number of storage units that cover the time range per *manager* node. The upper bound on the message cost of a range query corresponds to a situation where all zones (i.e, all *manager* nodes) concerned by the range query prefix hold data in LHT. Equation 5 computes the upper bound on the average number of messages for a given spatio-temporal range query.

$$C_{range-query} \lessapprox \log(N) + 2^{(r-1)} + 2^r \times (s+1) \quad (5)$$

Note that the parallel range query algorithm goes down through r levels until all *storage units* are reached. Therefore it incurs a complexity latency of $O(r)$ which is much lower than the upper bound on the average number of messages.

Similarly to insertions and deletions, the spatio-temporal range query algorithm of Big-LHT induces a much lower message complexity than traditional indexing structures such as R-Trees [9] and QuadTrees [10] because it directly reaches the node in charge of the subspace, thus avoiding the *root-to-leaf* path when the common prefix does not contain the root node.

B. Experimental evaluation

We implemented a prototype of our architecture on top of FreePastry, an open-source implementation of Pastry [11]. We ran all experiments presented in this section on an intel core i7 2.6Ghz with 8GB RAM, OS X 10.9.1, and Java VM version 1.6.0-65 .

Every experiment indexes 1,000,000 geotagged multimedia files (photos and videos) from the Yahoo! public dataset [15] in a DHT comprising $N = 100$ nodes. Every storage unit has a capacity of $B = 1,000$ data items and the maximum number of entries of the *storage units table* is $T_{max=10,000}$.

We aim to assess the impact of p, the prefix size, on insertions and on spatio-temporal range queries. Table I gives the area size and and the number of zones generated for different values of p. A small value for p generates a small quantity of big zones. Note that the number of generated zones differs from the theoretical number of zones required to cover the entire map, and that the ratio between the two values decreases fast as the prefix size increases. For instance, with $p = 25$ the number of generated zones is only 0.3% of the total number of zones. This is a benefit of our approach towards spatial skewness of data: Big-LHT does not allocate *managers* for zones that contain no data.

Prefix size	≈ Zone area size (R)	Generated zones/Total zones
5	5,004 km × 5,004 km	$32/32$
10	1,251 km × 625 km	$531/1024$
15	156 km × 156 km	$5,189/32,768$
25	4.9 km × 4.9 km	$127,167/33,554,432$

TABLE I: Impact of p on zone coverage of geolocated data

1) Storage data distribution: This experiment analyzes the data distribution of Big-LHT storage. We compare our results with the ideal case using the above configuration when every node reaches exactly 1% of the whole insertion load.

Figure 4a presents the storage distribution for different values of p; s is the standard deviation for the number of insertions on every node. Increasing the value of p from to 5 to 25 only increases the standard deviation s from 0.006 to 0.007. These results suggest that p bears little impact on the data distribution, which is logical because *storage units* are uniformly distributed among DHT nodes.

2) Insertion load distribution: This experiment assesses the impact of zone size on the insertion load distribution, measured as the percentage of insertions requests per *manager* node. Figure 4b presents the insertion load distribution and its associated standard deviation s for different values of p. We compare our results with the ideal case where every *manager* handles exactly 1% of the entire insertion load.

A small prefix ($p = 5$) distributes the load over 32 *managers*, which produces the worst insertion load balancing, measured as the highest standard deviation $s = 0.037$. In this configuration, node 10 handles about 25% of the insertion load. With respect to insertion requests, increasing the value of p improves load balancing significantly because it divides the space in smaller zones, and therefore distributes the load among more $managers$. For instance, $p = 10$ generates 531 zones and decreases the standard deviation to $s = 0.018$, with the maximum load on a single node lower than 10%.

3) Insertion latency: This experiment stresses the system with a high insertion load: $100,000$ insertions per second uniformly distributed among DHT, to evaluate its impact on latency. We measure latency as the time elapsed between the emission of a request and the reception of an insertion ACK from the responding *live storage unit*.

Figure 5a gives insertion latencies for different values of p. Smaller values of p produce the highest insertion latencies, because the smaller number of nodes is more likely to introduce bottlenecks. For instance, $p = 5$ induces insertion latencies of up to 6 seconds, with a median between 1 and 2 seconds. Increasing the value to $p = 15$ drastically reduces the insertion latency to a maximum value of about 1 second with a median of about 500 milliseconds. Note that increasing the value to $p = 25$ bears little impact, as $p = 15$ already acheives optimal results for this input workload.

4) Spatio-temporal range query latency: This experiment assesses the scalability of Big-LHT under a massive flow of spatio-temporal parallel range queries. In this evaluation we set the prefix size to $p = 15$, which produces the best tradeoff between insertion latency and storage data distribution according to our previous results, and index $1,000,000$ geotagged items. We then measure the average range query latency with different input query workloads: from $r = 1,000$ range queries per second to $r = 100,000$ queries per second. Every query asks for all objects inside a given spatio-temporal range: it reads an input Geohash from our data set and extracts a common prefix cp at random. This strategy generates a workload which follows the input data distribution for different sizes of the spatio-temporal space. For instance, a value $cp = 1$ generates a query which covers half of the spatio-temporal domain. It enters the tree at a high level and then goes down in parallel until it reaches all *storage units*. Choosing $cp = p$ generates a range query which asks for data inside a single zone.

Figure 5b presents the average spatio-temporal range query latency of Big-LHT in this experiment. A first observation is that the average range query latency evolves linearly with respect to the common prefix size cp, a logical result of the parallel sweeps down the tree. Given that different workloads produce similar curves, we conclude that Big-LHT scales gracefully with the workload.

IV. Related Work

The need to store, query and analyse *big location data* has recently motivated the usage of traditional spatial indexes such as R-Trees [9] and Quad-Trees [10] on top of traditional big data solutions such as Hadoop [7], Hbase [16], and Spark [8]. These solutions can fall into three groups: (i) Hadoop-based solutions; (ii) Resilient Distributed Dataset (RDD) based solutions, and (iii) Key-value store-based solutions.

Hadoop-based solutions such as SpatialHadoop [3], Hadoop GIS [4], and ESRI Tools for Hadoop [6], extend the traditional Hadoop architecture [7] with spatial indexing structures in order to avoid a scan of the whole dataset when performing spatio-temporal analysis. SpatialHadoop [3] builds spatial indexing structures such as R-Trees [9] over HDFS [4] in order to perform MapReduce tasks. However, these solutions are ill-suited to perform online *spatio-temporal* processing because (i) they maintain a global index structure on a single node that is prone to become a bottleneck, and (ii) they follows a batch processing model which requires processing the whole data set for every task.

Resilient Distributed Dataset (RDD) based solutions such as Spatial Spark [5] and GeoTrellis[8] extend traditional RDD solutions such as Spark [8]. Similarly to Hadoop-based solutions, these systems are designed for batch processing and do not target online *spatio-temporal processing*.

Key-value store-based solutions support spatio-temporal processing by building spatial indexing structures on top of key-value storage solutions. MD-Hbase [17] extends Hbase [16] with multi-dimentional indexing structures such as Quad Trees [10] and K-d trees [18] over a key-value storage layer through linearization techniques such as z-ordering [13]. It provides support for spatio-temporal range queries. However, the bucket split overhead introduces a data movement cost which limits the peak throughput. Big-LHT overcomes this issue by providing a low split index maintenance cost because it scales horizontally when a *storage unit* is overloaded.

V. Conclusion

This paper proposes a new approach towards continuous spatio-temporal range queries over Big Location Data. Our solution combines a storage architecture which distributes massive flows of data uniformly and a distributed spatio-temporal indexing structure that scales. A theoretical analysis of the message complexity of every Big-LHT operation, as well as an experimental evaluation conducted over a Yahoo! dataset comprising 1,000,000 multimedia files, show that our solution remains cost-efficient on a large scale.

We are currently working on a full scale experimentation of Big-LHT to assess its behaviour on a very large number of nodes. We also plan to explore an alternative solution for the distribution of Big Location Data storage and querying: a storage structure that fully matches the distributed index by introducing the time dimension in the indexation.

[8]http://geotrellis.io

References

[1] B. Thomee, D. A Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L. Li. The new data and new challenges in multimedia research. *arXiv preprint arXiv:1503.01817*, 2015.

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

[3] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. ICDE, 2015.

[4] 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.

[5] S. You, J. Zhang, and L Gruenwald. Large-scale spatial join query processing in cloud. *The City College of New York, New York, NY, Tech. Rep*.

[6] R. T. Whitman, M. B Park, Sarah M A., and E. 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.

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

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

[9] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[10] R. Finkel and J. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[11] 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.

[12] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[13] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[14] S Ramabhadran, S. Ratnasamy, J. M Hellerstein, and S. 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.

[15] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L. Li. The new data and new challenges in multimedia research. *arXiv preprint arXiv:1503.01817*, 2015.

[16] L. George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[17] S. Nishimura, S. Das, D. Agrawal, and A. E. 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.

[18] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.