Scalable Location-Temporal Range Query Processing for Structured Peer-to-Peer Networks

Rudyar Cortés

UPMC
SORBONNE UNIVERSITÉS

PhD Thesis, Université Pierre et Marie Curie
REGAL team, LIP6/INRIA

Subject: Computer Science
Option: Distributed Systems

Presented by: Rudyar Cortés

Scalable Location-Temporal Range Query Processing for Structured Peer-to-Peer Networks

Traitement de Requêtes Spatio-temporelles pour les Réseaux Pair-à-Pair Structurés

Presented on 06/04/2017 in front of the following jury:

Anne-Marie Kermarrec	Directrice de Recherche INRIA	Rennes, France	*Reviewer*
Pascal Molli	Professeur, Université de Nantes	Nantes, France	*Reviewer*
Peter Druschel	Scientific Director, MPI-SWS	Saarbrucken, Germany	*Examiner*
Franck Petit	Professeur, Université Pierre et Marie Curie	Paris, France	*Examiner*
Xavier Bonnaire	Professeur Associé, Universidad Santa María	Valparaíso, Chile	*Examiner*
Olivier Marin	Professeur Associé, NYU Shanghai	Shanghai, China	*Advisor*
Pierre Sens	Professeur, Université Pierre et Marie Curie	Paris, France	*Thesis Director*

Acknowledgments

I would like to express my gratitude to the many people who contributed to making this PhD thesis a wonderful experience.

I would like to begin by thanking my thesis jury composed of Dr. Anne-Marie Kermarrec, Dr. Pascal Molli, Dr. Peter Druschel, and Dr. Franck Petit, for assessing my work. The quality of your research is impressive and I feel honoured to be presenting my work to you. I want to especially thank Dr. Anne-Marie Kermarrec and Dr. Pascal Molli for their insightful comments about the thesis manuscript which allowed me to keep improving the quality of my work.

I want to express my sincere gratitude to the people I worked with on a daily basis: Dr. Xavier Bonnaire, Dr. Olivier Marin, Dr. Luciana Arantes, and Dr. Pierre Sens. Thank you for inspiring me to do research in distributed systems, your advice and friendship over the last years. You have been extremely kind and I feel very lucky to have had the opportunity to work with you.

Dr. Xavier Bonnaire, thank you for trusting in my capabilities at the end of the Engineering Program at UTFSM and for your continual professional and personal advice; from my first steps doing research, until today. I started doing research because your impressive knowledge inspired me to follow this path. Thank you for all these years of real fun doing research together, and for your continual help over the last years (and especially during my first week in Paris!)

Dr. Olivier Marin, thank you for your unwavering support from the first day I arrived at Lip6. Even if you knew little about me, you agreed to be my guarantor when I rented my first apartment in France. You taught me, step by step, how to write and present research articles and most importantly, how to discover new research axes and how to propose original research contributions.

Dr. Luciana Arantes, thank you for the great advice and all your help during most of my PhD. You were always available to discuss and have a cup of coffee. You helped me improve the quality of my work and it has been a pleasure to work with you.

Dr. Pierre Sens, thank you for all your insightful comments and your attention to detail which is impressive. I want to especially thank you for all your help in the most difficult parts of the PhD. Every time I left your office I had the answers I was looking for!

Beyond my advisors, I had an incredible experience working with the REGAL team. I would like to thank everyone in the team for contributing to creating a great working environment, and I feel very lucky to be part of it. I met passionate people working on exciting projects, every one with a very good sense of humour that I always appreciate.

I want to especially thank the current and past PhD students: Peter Senna, Denis Jeanneau, Marjorie Bournat, Florian David, Maxime Bittan, Gauthier Voron, Hakan Metin, Damien Carver, Ilyas Toumlilt, Lyes Hamidouche, Benjamin Baron, João de Araujo, Ludovic Le Frioux, Redha Gouicem, Jonathan Lejeune, Antoine Blin, Alejandro Tomsic, Mahsa Najafzadeh, Francis Laniel, Marek Zawirski, Florent Coriat, Lisong Guo, Maxime Lorrillere, Laure Millet and all the others I met here for the great time we spent working at Lip6. I also want to thank CONICYT for their financial support and trust in my work.

During my stay in Paris, I met incredible people who helped me make this wonderful city a new home. I would like to thank you all for all the great times we shared together and we will continue enjoying in the future. Thank you Théodore Bluche, Leticia Goto, Sophie Kranzlin, Marion Blondel, Osman Ahmady, Vincent Girard-Reydet, Julien Senet, Alexandre Planchon, Lamis Shamas, Laëtitia Laurencia, Anne Alice Fievet, Eva Brabant, Sofia Gaete, Jerome Thedrel, Zoe Stillpass, Jake Wotherspoon, Corinne Lee, Clarisse Lemaire, Emma Díaz, Gabriel Díaz, Marine Ress, Stephanie Perreau, Julien Crombé, Nicolas Lewandowski, Aranud-Pierre del Perugia Lewandowski, Jess Wigram, Daan Van Der Wekken and the many others I had the privilege to meet.

I want to thank my family for their unconditional support and love. Thank you for making me believe that you can achieve every dream you have.

I would also like to thank my Chilean friends Javier Olivares, Tanya Pérez, Francisco Gamboa, Cristian Fuentes, Sebastian Muñoz, Gustavo Muñoz, Mariel Villegas, and Sergio Henriquez. Despite the distance, you kept supporting me during this adventure!

Shân and Julio Pinna, I am glad you were able to come from London to share this important step with me. I am extremely grateful for your trust, personal advice and support over the last years. You are my second family here, and I am happy to have you in my life.

I want to finish these words with my most important, beautiful and smart other half, Shân Andrea Pinna-Griffith. There are no metrics or words to measure and describe the infinity of my feelings for you. I love you so much and I enjoy every moment spent together. Thank you for your love, patience, support, and for making me the happiest and luckiest person on earth. I want you to know that I will do everything to stay with you. I dedicate this PhD thesis to you.

Rudyar Cortés.

Contents

Chapter 1

Introduction

Indexing and retrieving data by location and time allows people to share and explore massive *geotagged* datasets observed on social networks such as Facebook [1], Flickr [2], and Twitter [3]. This scenario known as a Location Based Social Network (LBSN) [RH13] is composed of millions of users, sharing and performing location-temporal range queries in order to retrieve geotagged data generated inside a given geographic area and time interval.

A key challenge is to provide a scalable architecture that allow to perform insertions and location-temporal range queries from a high number of users. In order to achieve this, Distributed Hash Tables (DHTs) and the Peer-to-Peer (P2P) computing paradigms provide a powerful building block for implementing large scale applications. However, DHTs are ill-suited for supporting range queries because the use of hash functions such as the Secure Hash Function 1 (SHA-1) [EJ01] destroys data locality for the sake of load balance. Existing solutions that use a DHT as a building block [RRHS04, CRR+05, TZX10, HRA+11, HASB16] allow to perform range queries. Nonetheless, they do not target location-temporal range queries and they exhibit poor performance in terms of query response time and message traffic.

This thesis proposes two scalable solutions for indexing and retrieving geotagged data based on location and time.

1.1 Motivation

The increasing number of GPS-enabled devices such as smartphones and cameras lead to the generation of massive amounts of geotagged data. On a daily basis, millions of people share multimedia files (pictures and videos) and comments on social networks.

In this context, LBSNs [RH13] allow users to share and explore *geotagged* information based on location and time. A key requirement of LBSNs is to support *location-temporal queries* on a large scale. These queries retrieve all the objects that match a given location-

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

temporal predicate. It represents a major challenge for data extraction, exploration and analysis, within both a geographic area and a time frame.

Allowing hundreds of millions of users to share and explore geotagged data can be crucial for life saving efforts by helping to locate a missing child, or for security purposes allowing users to inform themselves about an imminent threat. For instance, users might want to review the latest information about places or geographic zones of interest. In this case, the respective query retrieves the latest n uploaded elements, *per geographic zone*, covered by a location interval. We refer to this type of query as *n-recent location-temporal zone query*.

It can also help people to get more insight into public activities. For instance, people who attended a concert between 20:30 p.m. and 01:30 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 analysing pictures, comments and tags. We refer to this type of query as *location-temporal range query*.

Building an architecture that allows people to share and search data by location and time is essential to help them get more insights about what happens inside a given geographic area and time interval. Creating an index based on location and time allows to perform such tasks with low query response time. Location and time are represented by three core parameters observed in geotagged data: Latitude, longitude and timestamp.

The main challenge presented is to provide a scalable architecture that organises data based on location and time, which is not a trivial task due to the multidimensional and diverse representation of location and time.

Centralised relational databases offer a powerful solution to store and index data. However, a high number of users performing concurrent insertions and location-temporal queries impose scalability constraints to central services as they present the following two main issues:

1. *Memory bottleneck*. Concurrent queries that target a high number of objects can overload the main memory. This issue has a direct impact on query response time.

2. *Bandwidth bottleneck*. Bandwidth is currently an expensive and limited resource that can become a bottleneck when the system experiences a high number of concurrent queries.

Cluster based solutions can scale out the number of servers, thus alleviating the memory bottleneck. However, output bandwidth is still an issue because it limits scalability and application scenarios.

On the other side, the Peer-to-Peer (P2P) computing paradigm presents an alternative model which allows to build highly scalable applications using resources (memory, bandwidth, and processing power) provided by *nodes* distributed worldwide.

Distributed Hash Tables (DHTs) such as Pastry [RD01] and Chord [SMK+01] provide a simple, but powerful building block for internet scale applications. They provide a *get/put* routing interface that routes a message in $O(\log(N))$ messages, where N is the number of nodes in the network. This interface allows to perform exact match queries

efficiently. However, DHTs are ill-suited for range queries because the hash function they rely on destroys data locality.

Several solutions [RRHS04, CRR+05, TZX10, HRA+11, HASB16, CFCS04, BAS04, SP08, PNT12] provide support range queries over DHTs. These solutions can support multidimensional range queries using *space filling curves* (SFCs) [Sam06]. However, they provide high latency and message traffic for range queries. Furthermore, they provide poor support for *n-recent location-temporal zone queries*.

1.2 Contributions

This thesis proposes two architectures which support location-temporal queries over massive geotagged datasets. As opposed to cluster-based architectures, these solutions rely on resources shared among users instead of extra dedicated servers. The proposed solutions are built on top of a DHT, such as Pastry [RD01] or Chord [SMK+01]. They can take advantage of all the desirable properties of DHTs such as storage load balance and fault tolerance through replication.

The first contribution, called Big Location Hash Tree (Big-LHT) supports *n-recent location-temporal zone queries*. Big-LHT scales with the number of DHT nodes and provides direct data access to the last data items generated inside a geographic zone. It produces a primary index based on location and a secondary index based on upload time. The primary index partitions the input dataset into geographic zones, where the size of the zone is an application parameter. The secondary index sorts the input dataset by *upload time*, similarly to a temporal log.

Big-LHT provides the following main properties:

- *Low insertion and index maintenance cost.* Insertions are performed using $O(\log(N))$ messages, where N is the number of DHT nodes. The index maintenance operations present a low cost in terms of number of messages. For instance, the *split* index maintenance operation takes only requires $O(\log(N))$ messages.

- *Storage data distribution.* Big-LHT provides good storage data distribution that slightly depends on the size of the geographic zone. For instance, using a configuration composed of $N = 100$ DHT nodes, no node stores more than 4% of the total number of data items. This result is close to the ideal data distribution, where every node (out of $N = 100$) stores exactly 1% of the total amount of data items.

- *Insertion load balance.* Big-LHT presents good insertion load balance for applications that need to define small geographic zones such as Points of Interest (POIs). However, load balance deteriorates for applications that need to define big geographic zones. This is because the size of every geographic zone is defined in advance (static data partitioning).

The second contribution, named GeoTrie, supports scalable *location-temporal range queries*. The main component of GeoTrie is a three dimensional distributed index which

indexes data based on (latitude, longitude, timestamp) tuples as key into a distributed prefix octree. Compared to Big-LHT, GeoTrie introduces a dynamic partition strategy that adapts to the actual data distribution, thus providing scalability and load balancing. It comprises a client cache and an optimistic protocol that drastically reduce message traffic and latency compared to Prefix Hash Tree (PHT) [RRHS04][4].

GeoTrie provides the following main properties:

- *Low latency and message traffic.* Results of practical evaluations using a dataset composed of millions of geotagged multimedia files uploaded to Flickr that measures message traffic, load balance, and latency for insertions and range queries. The results confirm that GeoTrie provides a low latency and low message traffic compared to the closest solution (PHT). For instance, GeoTrie performs insertions up to 2.47 times faster than PHT with up to 80.4% less messages. Range queries are performed up to 2.64 times faster with 31.75% less messages.

- *Load balance.* The architecture ensures natural load balance of insertions and location-temporal range queries. This is because GeoTrie maintains the direct access property of *prefix trees*, which allows a client node to contact directly any GeoTrie node.

In summary, the contributions of this thesis are the following:

1. **Big-LHT**. A novel scalable distributed data structure, built on top of a DHT, that supports *n-recent location-temporal zone queries* with low insertion cost and low index maintenance cost. Big-LHT partitions the input dataset into geographic zones and provides direct access to the most recent data items generated per zone.

2. **GeoTrie**. A fully distributed global index, built on top of a DHT, that supports *location-temporal range queries*. GeoTrie dynamically indexes data based on $latitude, longitude, timestamp$ tuples into a distributed prefix octree. It provides properties such as scalability, load balance, and fault tolerance. It introduces a client cache protocol which reduces both latency and message traffic for insertions and range queries. Compared to current solutions, it provides lower message traffic and lower latency.

1.3 Outline

This thesis manuscript is organised as follows;

Chapter 2 introduces P2P networks and presents a critical assessment of the main research work related to indexing over DHTs. The existing solutions are classified into three groups: (i) Over-DHT solutions, (ii) Overlay-dependent solutions, and (iii) Non-DHT solutions.

[4] A well known prefix tree index structure built on top of a DHT

Chapter 3 and **Chapter** 4 present Big-LHT and GeoTrie respectively. They are organised into three main sections: (i) Design, (ii) Theoretical and practical evaluation, and (iii) Discussion about the main advantages and limitations.

Finally, **Chapter** 5 presents a conclusion of this research work and directions for future topics.

1.4 Publications

As result of the different stages of this thesis, the following research work has been presented in international conferences.

1. R. Cortés, X. Bonnaire, O. Marin, L. Arantes, P. Sens. "GeoTrie: A Scalable Architecture for Location-Temporal Range Queries over Massive GeoTagged Data Sets", 15th IEEE International Symposium on Network Computing and Applications (IEEE NCA-16), Cambridge, MA, USA (2016).

2. R. Cortés, O. Marin, X. Bonnaire, L. Arantes, P. Sens. "A Scalable Architecture for Spatio-Temporal Range Queries over Big Location Data" (Big-LHT), 14th IEEE International Symposium on Network Computing and Applications (IEEE NCA-15), Cambridge, MA, USA (2015).

3. R. Cortés, X. Bonnaire, O. Marin, P. Sens. "FreeSplit: A Write-Ahead Protocol to Improve Latency in Distributed Prefix Tree Indexing Structures", 29th IEEE International Conference on Advanced Information Networking and Applications (AINA-2015), Gwangju, South Korea (2015).

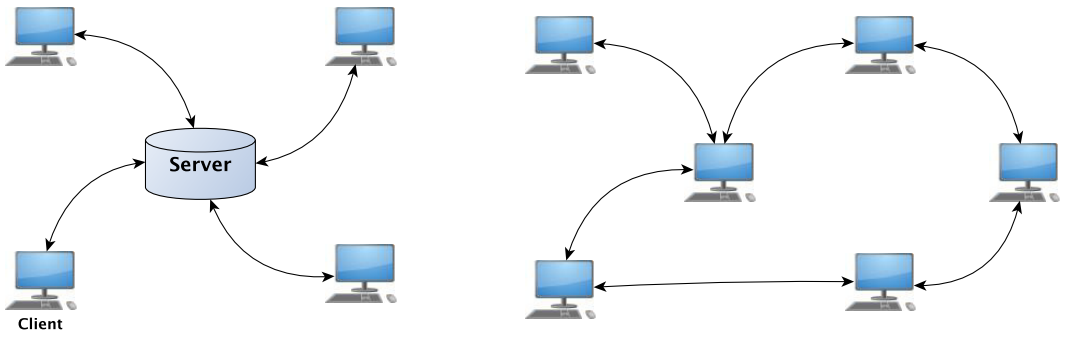
4. X. Bonnaire, R. Cortés, F. Kordon, O. Marin. "A Scalable Architecture for Highly Reliable Certification", The 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-13), Melbourne, Australia (2013).

Chapter 2

State of the Art: Location-temporal Range Queries on P2P Networks

Contents

This chapter presents a review of the main research works that aim to create data locality in structured P2P networks in order to support range queries. We briefly present P2P networks, classified in unstructured and structured networks. Then, we present a definition of location-temporal range queries and the main issues of implementing such queries over structured P2P networks. From this point, we focus on presenting the main research lead in order to overcome these issues. Finally we present a critical assessment of these solutions in terms of latency, message complexity, and load balancing.



(a) Client-server model

(b) Peer-to-Peer model

Figure 2.1: Client-server model versus P2P model

2.1 Peer-to-Peer (P2P) Networks

The continuous and widespread growth of Internet-based applications in terms of number of users and computational resources, has challenged the centralised nature of the *client-server* paradigm. One of the most relevant requirements of such applications is *scalability*, which is defined in [Neu94] as follows.

Definition 1 (Scalability)
A system is said to be *scalable* if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity.

Figure 2.1a presents the general model of the *client-server* paradigm. Services are maintained on a central entity composed of one or more servers. It is widely accepted that this paradigm cannot accomplish the definition of scalability without an enormous amount of effort in terms of resources. Besides, its centralised organisation is prone to resource bottlenecks.

The *Peer-to-Peer* (P2P) network and computing paradigm aim to decentralise services. It proposes a different model composed of autonomous entities called *peers* or *nodes* that share computational resources (CPU, storage, and bandwidth) in order to achieve a common task. Figure 2.1b presents its general model. *Nodes* are connected to a set of neighbours through internet links in a logical network called *peer-to-peer overlay network* or simply *peer-to-peer network*.

The term Peer-to-Peer has be defined in [Ora01] and refined in [SW04] as follows.

Definition 2 (Peer-to-Peer)
A self-organising system of equal, autonomous entities (peers) which aim for the shared usage of distributed resources in a network environment avoiding central services.

This basic definition gives peer-to-peer networks the following desirable properties.

1. *Self-organisation.* Nodes are able to interact with each other without any central control. This is the main property of peer-to-peer systems which shifts the paradigm away from the client-server paradigm.
2. *Decentralised resource usage.* The available resources of nodes (CPU, storage, and bandwidth) are distributed and shared with the best effort regarding its load distribution.
3. *Scalability.* It achieves the definition of scalability stated above (see definition 1).
4. *Fault tolerance.* The failure of a single node must not compromise the correct operation of the system.

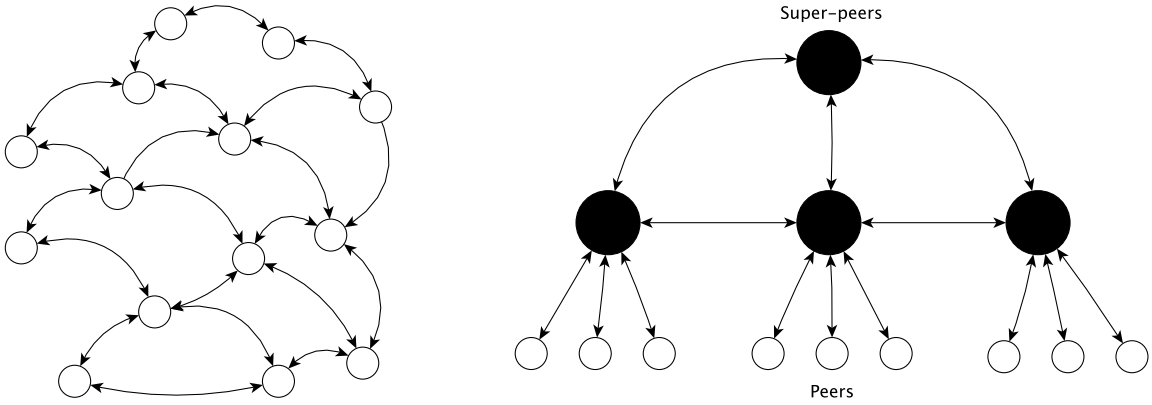
Usually, a resource r is represented by a *resource identifier* $id(r)$ which belongs to a *resource identifier space* $I(r)$. The basic functions supported by most of peer-to-peer networks are the following.

- $Put(n, r)$: Add the resource r on the node n.
- $Get(n, id(r))$: Acquire the resource represented by the identifier $id(r)$ from the node n.
- $Query(q, n_q)$: Get all the node identifiers of n_q which hold resources that accomplish the query predicate q.

Many solution towards achieving these features have been proposed in the literature. They can be classified in two main groups: *unstructured overlay networks* and *structured overlay networks*

2.1.1 Unstructured P2P overlay networks

In an unstructured overlay network, a node can freely join the network independently of its state. These networks typically present a *graph structure* or a *hierarchical structure*. The main difference between these two structures is that in the former all nodes are seen as equal entities, whereas the second adds a layer composed of nodes called *super-peers*. A *super-peer* represents a set of *peers* which interact with the larger system through a central node. Figure 2.2 presents a schema of these two models. A popular example of systems that use this kind of overlay is the first generation of peer-to-peer file sharing applications such as Gnutella [Rip01]. It is possible to characterise this kind of P2P network through the followings properties.



(a) P2P Graph structure

(b) P2P hierarchical structure

Figure 2.2: Graph structure versus hierarchical structure

- *Network Knowledge.* A node knows a set of neighbours (local knowledge). That is, no node maintains the entire state of the system.
- *Resource Knowledge.* There is no implicit knowledge about the location of resources. That is, a node only knows about its own resources.
- *Search algorithm.* Since there is no resource oversight, the search must be propagated through neighbour nodes using *flooding* techniques such as the *breadth-first search* algorithm (BFS) [Cor09]. A search message is forwarded to all neighbours which, in turn, forward it until a *time-to-life* (TTL) number of steps have been completed. In order to avoid cyclic forwarding, every node can remember recent messages and drop duplicate messages.

2.1.1.1 Limitations of unstructured overlays

The main advantage of unstructured overlay networks is their low cost of maintenance. However, the lack of resource oversight induces a high message traffic for performing searches. In the worst case, all nodes may be visited in order to know if a resource exists in the system. Several improvements have been proposed to reduce the number of nodes visited in the BFS algorithm. They include *selective forwarding* [SBR04], *iterative deepening* [YGM02], and *random walk* [LCC+02]. These algorithms significantly reduce the cost of the search, but they still may require to visit all the nodes in order to find a data item.

2.1.2 Structured P2P overlay networks

Structured P2P overlay networks [SMK+01, RD01, RFH+01] aim to address the lack of resource knowledge observed on unstructured overlays in order to reduce the high lookup message traffic cost. They achieve this goal through a strong network topology which

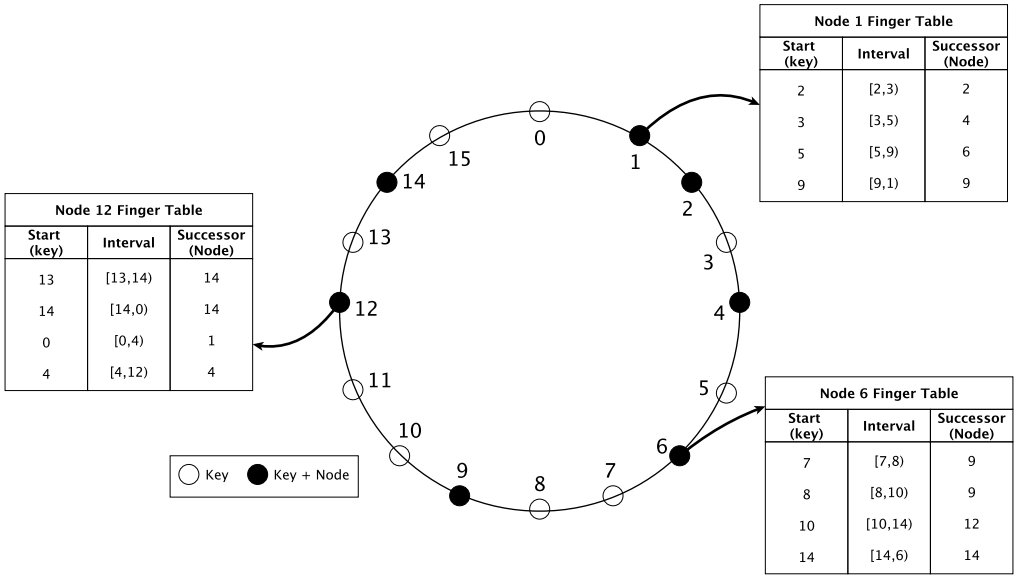


Figure 2.3: Chord ring and finger table associated with nodes 1, 6, and 12

indexes resources to nodes using *hash functions*. Concretely, a hash function $H : id \rightarrow S$ maps node identifiers and resource identifiers to the same domain space S. This class of solutions are called Distributed Hash Tables (DHTs).

A DHT is a distributed data structure which provides scalability, fault tolerance, and load balancing for internet-scale applications. Most DHTs provide support for the followings three operations.

- $Get(key)$.
- $Delete(key)$.
- $Put(key, value)$.

When a node joins a DHT, it receives an identifier called *nodeId*, in the domain $S = [0, 2^s - 1]$. The value of s depends on the number of bits used in the hash function. Every node maintains a *state* composed of a *routing table* and a *neighbour set*. The *routing table* stores the *nodeIds* and IP addresses of a small fraction of nodes compared to the total number of nodes in the network. The *neighbour set* stores the *nodeIds* and IP addresses of close nodes in the identifier space S. This state supports a *routing interface* that can reach any key k in $O(log(N))$ routing hops, where N is the number of nodes in the network.

Fault tolerance is achieved through *replication* on a set of nodes called *replica set*. Several implementations of this distributed data structure have been proposed in the literature. Pastry [RD01], Chord [SMK+01], and CAN [RFH+01] are prominent examples DHT implementations and they will be briefly reviewed in the next sections.

2.1.2.1 Chord

Chord [SMK+01] defines a circular identifier space using a hash function that generates a uniform distribution of *nodeId's* such as the *Secure Hash Algorithm Version 1* [EJ01] (SHA-1) of $s = 160$ bits. For the sake of clarity SHA-1 will be refer as SHA. Every node provides its IP address and locally computes its own identifier (i.e, $nodeId = SHA(IP)$). This identifier belongs to the domain $S = [0, 2^{160} - 1]$. A given key k is assigned to the node whose *nodeId* is equal or follows k clockwise in the identifier space. This node is called *successor* of the key k and is denoted as $successor(k)$. Chord provides a routing service which can locate the successor node of k in $O(\log(N))$ routing hops. This routing service relies on *node states* that are defined as follows.

Node state. Every node maintains links to its direct successor and predecessor node. Additionally, every node maintains a routing table called *finger table*. This table maintains references with up to m nodes in the ring. The i^{th} entry in the table of a node with identifier n is the node whose *nodeId* is greater or equal than the key $n_i = ((n+2^{i-1}) \mod 2^m)$, $1 \leq i \leq m$. The size of this table is $O(log(N))$ entries. Figure 2.3 presents a Chord ring composed of $N = 16$ nodes with identifiers in the domain $S = [0, 2^4 - 1]$.

Routing algorithm. The routing algorithm ensures that a message with key k is delivered to the node $successor(k)$. The main idea is to use the *finger table* in inverse order (i.e, from $i = m$ to 1) to reach the farthest predecessor of k that maintains a link to the node $successor(k)$. Following the example presented in figure 2.3, the key $k = 13$ starting at node 6 is first routed to node 12, which in turn forwards the query to its successor (node 14) that is the successor of k. This algorithm has a cost of $O(log(N))$ sequential routing hops.

Network maintenance. When a node joins the network it should initialise its state and at the same time some nodes may have to update their state in order to be aware of the new node. Chord assumes the existence of *bootstrapping nodes* that can help new joining nodes find the direct successor and predecessor of a given *nodeId*. The cost of a join operation is $O(\log^2(N))$, where N is the number of nodes in the network. Additionally, every node runs periodically a *stabilize* protocol that checks and corrects the state of the local node. For further details about the network maintenance used in Chord, please refer to [SMK+01].

Fault tolerance. In order to provide fault tolerance, every node replicates its data in a set of the first r successor nodes. The links to these nodes are stored in a list called the *successor list*. Thus, if a node n notices that its successor has failed, it replaces it with the first live node in its *successor list*.

2.1.2.2 Pastry

Pastry [RD01] defines a DHT composed of a 160-bit identifier space using the SHA-1 [EJ01] hash function (i.e, $nodeId = SHA(IP)$). Similarly to Chord [SMK+01], it provides a routing interface that delivers a message in $O(log(N))$ routing hops. It associates a resource identified with a key k to the node whose *nodeId* is closest to k. The Pastry [RD01] *node state* is defined as follows.

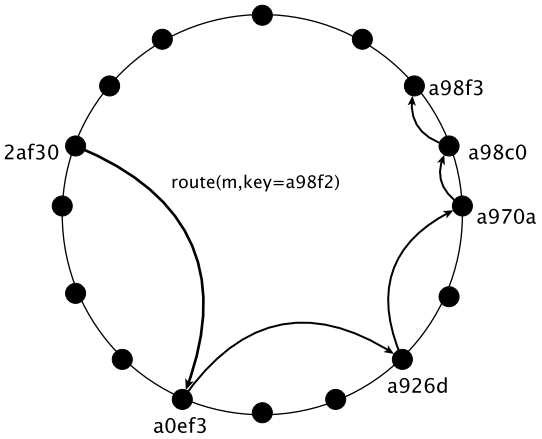


Figure 2.4: Pastry routing example

Node state. The node state is composed of three sets of nodes: the *leaf set*, the *routing table*, and the *neighbourhood set*. The *leaf set* consists of the set of L numerically closest nodes ($\frac{L}{2}$ clockwise and $\frac{L}{2}$ counterclockwise). The *routing table* contains the IP addresses and the *nodeId's* of different nodes in the network. It consists of $\log_{2^b}(N)$ rows with $2^b - 1$ entries. Every row n contains the reference to the node whose common prefix size with the current *nodeId* is n, but whose $n+1$th digit has one of the $2^b - 1$ possible values other than the $n+1$ digits in the present $nodeId$. The nodes of this table are chosen to be close to the local node according to a geographic proximity metric such as the number of IP routing hops. It can be obtained using network services such as *trace route*[1]. The *neighbourhood set* consist of ns nodes that are closest to the local node according to the proximity metric. This set can be used to maintain locality properties.

Routing algorithm. The routing algorithm ensures that a message to a given key $k \in S$ is delivered to the node whose *nodeId* is closest to k in less than $\lceil \log_{2^b}(N) \rceil$ routing hops, where b is typically 4. This is achieved using a recursive prefix based strategy which works as follows. A node which receives a routing message uses its *routing table* in order to forward the message to the node whose $nodeId$ shares $n+1$ digits with k. This sequential process finishes when the node with the closest nodeId is reached. Figure 2.4 presents an example of the routing of the key $k = a98f2$, starting from the node with *nodeId* $2af30$. Thus, using this routing algorithm, a data item identified with key k is stored on the node whose *nodeId* is closest to k.

Network maintenance When a new node with nodeId X joins the network, it asks an existing node A to route a *join message* to the node whose *nodeId* is closest to X. Let this node be node Z. The join message is routed through $O(log(N))$ nodes which send back their state in order to help X build its own table. A single entry of the table may have multiple candidates. In this case, the closest node according to the proximity metric is chosen. In return, X informs the nodes its *leaf set* of its arrival in order to update their states.

The departure or failure of a node X impacts the state of all nodes that have infor-

[1]http://linux.die.net/man/8/traceroute

mation about X. When a node detects a failed node on its *leaf set* it asks for the *leaf set* of the live node with the largest nodeId on the side of the failed node. Then, it uses this *leaf set* in order to repair its own state. Similarly, a failed node detected on row i in the routing table is repaired with the entry of a node j in the same row.

Fault tolerance. In order to provide fault tolerance, a subset of the *leaf set* called *replica set* is used to replicate data. This set also provides load balancing because the routing algorithm always passes through different nodes of the *leaf set* before reaching the target node. Thus, replica nodes can handle the requests when a node is overloaded.

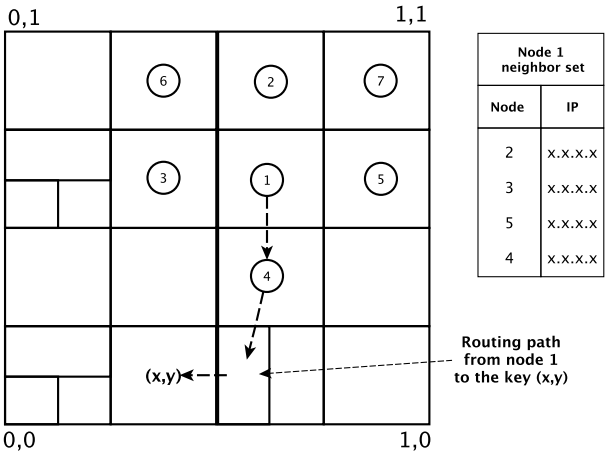


Figure 2.5: CAN structure and routing example

2.1.2.3 CAN

Content Addressable Network (CAN) [RFH+01] is a DHT that defines a virtual *d-dimensional* cartesian space on a d-torus. Every dimension can be represented as a circular space that belongs to the domain $[a, b]$. This virtual space is dynamically partitioned into zones based on node arrivals and departures. Figure 2.5 presents an example of the CAN structure of a $2d$ coordinate space in the domain $[0, 1] \times [0, 1]$. In this example, zones are represented as squares of different sizes. Every node owns only a single zone and it provides an access through links with its neighbours. Using this schema, every key k is assigned to a unqique zone.

Node state. A CAN node maintains a *routing table* comprising the virtual coordinate zones and the IP addresses of its own neighbours. Two nodes are defined as neighbours if their coordinate spans overlap along $d - 1$ dimensions and abut along one dimension. The number of neighbours of a given node is $2d$. For instance, the neighbours of node 1 in figure 2.5 are the nodes $\{2, 3, 5, 4\}$.

Routing algorithm. CAN [RFH+01] uses a *greedy* forwarding strategy to route messages to the neighbour that has coordinates closest to the destination coordinate. This procedure works as follows. First, using multiple hash functions a key k is mapped to a d-dimensional point P. Then, the local node uses its neighbours to route the message

to the zone that holds P. For instance, figure 2.5 presents the routing path from node 1 to the zone (x,y). The average routing path length is $(d/4)(n^{1/d})$ and its message complexity is $O(d \times n^{1/d})$.

Network maintenance A new joining node must be assigned to a new zone. For this purpose, an existing node must split its own zone in half, retaining half of the zone and delivering the other hanf to the new node. The new joining node randomly chooses a point P in the space and sends a JOIN request through an existing *bootstrap* node X. X routes this message to the node Y that holds the zone associated with P. Finally, the new node build its neighbour set based on the neighbour table of Y. The insertion of a new node impacts the state of $O(d)$ existing nodes.

The departure of a node could merge two zones into a single one. The leaving node looks for a neighbour which manages a zone that can be merged. If such node does not exist, the neighbour node that holds the smallest zone is selected as temporary manager of the zone of the leaving node. This temporary node will hold the zone until a merge operation can be done with one of its neighbours. If the node fails, a takeover mechanism ensures that a neighbour node will take the zone. For further details of this mechanism, please refer to [RFH+01].

Fault tolerance. Fault tolerance is achieved through an improvement of the base architecture called *zone overloading*. The main change consists of allowing a maximum number of nodes (typically 3 or 4) on a single zone instead of a one to one mapping. These nodes act as replicas of the data assigned to zones in order to improve data availability. Additionally, popular data can be eventually replicated on neighbouring node. The main drawback of this schema is that it does not ensure that a single zone will always have more than one single node. Thus, some zones could suffer of data loss upon node crashes.

2.1.2.4 Comparison of DHTs architectures

Pastry [RD01] and Chord [SMK+01] achieve similar properties. They provide a lookup interface that can route messages in $O(\log(N))$ routing hops with a space cost of $O(\log(N))$ links per node. However, the former presents advantages in terms of network latency and fault tolerance. This is due to two strategies used in the design of the routing table. (i) the distance metric used to choose an entry of the table, and (ii) the diversity of routes of the Pastry *routing table* compared to the *finger table*. On the other hand, CAN [RFH+01] proposes a different approach which provides a lookup interface that can route messages in $O(d \times N^{1/n})$ with a cost of $O(d)$ links. The main advantage of this overlay is that the number of links maintained per node does not depend of the total amount of nodes N. Furthermore, choosing $d = \log(N)$ the lookup latency and cost can be comparable to Pastry [RD01] and Chord [SMK+01]. However, the architecture proposed in CAN [RFH+01] assumes a fixed value of d (defined at startup) and therefore d cannot vary as N does. Therefore, a variation of the number of nodes can degrade the latency of lookups.

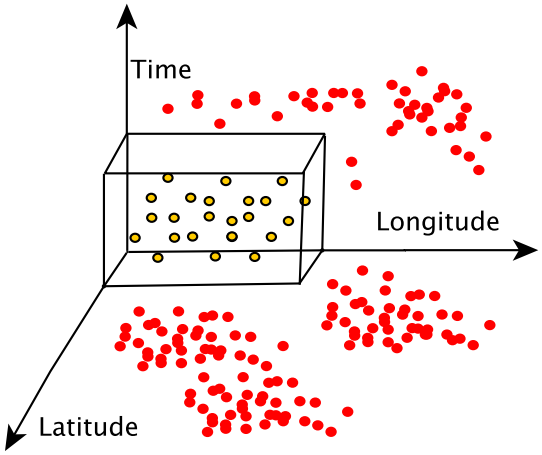


Figure 2.6: Location-temporal range query representation

2.2 Location-temporal range queries over DHTs

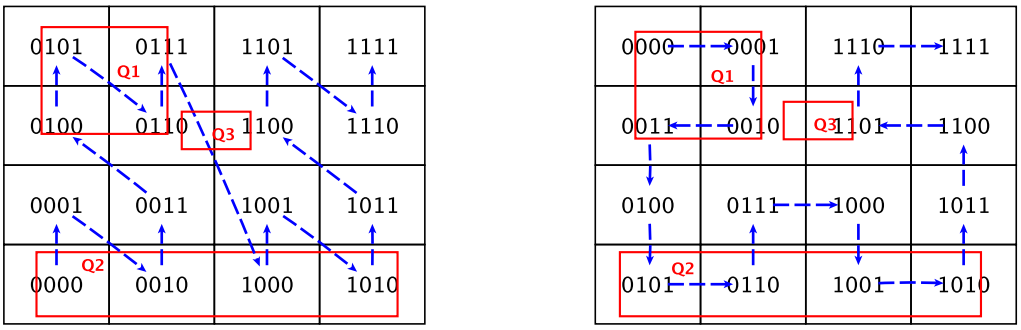
A location-temporal range query asks for all the objects inside of a three dimensional range as presented in equation 4.1.

$$\begin{array}{rcll} \Delta Lat & = & [lat1, lat_2], & lat_1, lat_2 \in [-90, 90] \\ \Delta Lon & = & [lon_1, lon_2], & lon_1, lon_2 \in [-180, 180] \\ \Delta t & = & [t_i, t_f], & t_i, t_f \in [\text{DateTimeStart,DateTimeEnd}] \end{array} \quad (2.1)$$

ΔLat represents the *latitude* starting and ending point ($[lat1, lat_2]$), ΔLon represents the *longitude* starting and ending point ($[lon_1, lon_2]$), and Δt represents a time interval. This query can be represented as ilustrated in figure 4.1. Yellow points represent data items that fulfil the location-temporal query predicate, red points represent data outside of the interval.

2.2.1 Limitations of Distributed Hash Tables

DHTs provide a simple but powerful building block that supports a highly scalable *get/put* interface for internet-based applications. This interface has allowed the implementation of diverse services such as publish/suscribe systems [CDKR02], file systems [SAZ+02], indirection systems [SAZ+02], among many others. However, more complex queries such as location-temporal range queries are difficult to implement because the hash function destroys *data locality* for the sake of load balancing. Building *data locality* in a multi dimensional space while maintaining the base properties of *scalability*, *fault tolerance*, *random access*, and *self-organisation* is a difficult task. In the next sections, different approaches towards supporting these queries will be reviewed. These solutions can be classified into three groups: (i) Over-DHT solutions, (ii) Overlay-dependent solutions, and (iii) Non-DHT solutions.



(a) Z-ordering curve

(b) Hilbert curve

Figure 2.7: Space-filling curves

2.3 Over-DHT solutions

Over-DHT solutions aim to build indexing structures using the DHT as a building block in order to inherit all its desirable properties. They can be classified into mainly two groups: *Distributed prefix trees* and *distributed binary trees*. Although most of these solutions do not target multidimensional data, they can use *Space Filling Curves* (SFC) [Sam06] in order to process a *n-dimensional* range query as an *one-dimensional* range query.

Space Filling curves(SFC) [Sam06] allow to map an *n-dimensional* space to a *one-dimensional* space while loosely preserving data locality. That is, close points in the multidimensional space are relatively close in the one-dimensional space. Examples of these curves are *z-ordering* curves [Mor66] and Hilbert curves [Hil91]. The main drawback of SFCs is that the locality properties of the curve degrade with the number of dimensions [Sam06]. The consequence of this fact is that a variable amount of *false positives* can be introduced.

Figure 2.7 presents two examples of query processing using these curves. The query Q1 is performed without false positives because it matches the trajectory of the query in both cases, the query Q2 introduces more false positives using *Z-ordering*, and the query Q3 introduces more false positives using a *Hilbert curve*.

2.3.1 Distributed Prefix Trees

Distributed Prefix Trees(*tries*) [RRHS04, TZX10, HRA+11, HASB16] create data locality over DHTs using prefixes to create partitions of the data domain. These equal size partitions are distributed and linked through DHT nodes using the *get/put* interface they provide. The main advantage of this strategy is that it provides a *global knowledge* of the prefix tree structure. In other words, a single *trie* node can be directly contacted through a single DHT routing message.

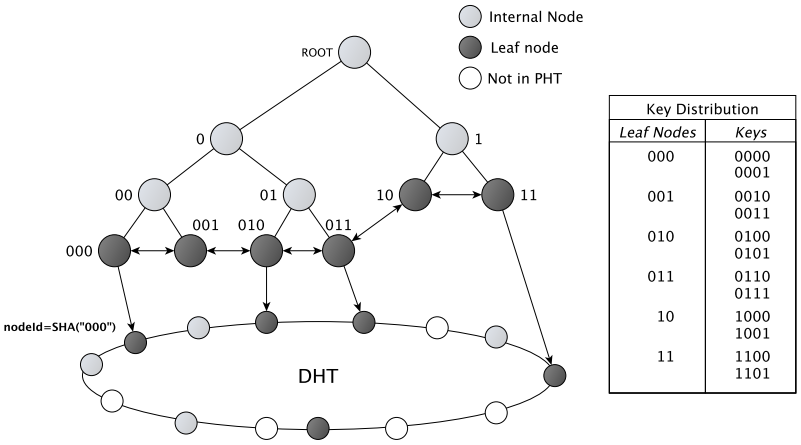


Figure 2.8: Prefix Hash Tree Structure

2.3.1.1 Prefix Hash Tree

Prefix Hash Tree(PHT) [RRHS04] is a one-dimensional distributed indexing data structure built over DHTs such as Pastry [RD01] and Chord [SMK+01].

This structure creates data locality by using a prefix rule to map *trie* nodes over a DHT ring. This allows to support complex queries such as range queries, proximity queries and MIN/MAX queries. PHT recursively partitions the domain $d = [a, b]$ into equal partitions represented as prefixes in the domain $\{0, 1\}^D$, where D is the amount of bits used to represent these partitions. Figure 2.8 shows an example of the PHT data structure that indexes the data domain $\{0, 1\}^4$ and its respective DHT overlay.

PHT comprises three node states: *leaf nodes* store data, *internal nodes* maintain the trie structure, and *external nodes* belong to the DHT ring but not to the trie. PHT also associates a recursively defined label with every node. The left child node inherits the label of its parent node concatenated with 0, and the right child node does the same with 1. Data storage follows the same prefix rule: looking up an object with key k consists in finding a leaf node whose label is a prefix of k. This strategy recursively divides the domain space $\{0, 1\}^D$, where D is the amount of bits used to represent keys, and delivers an implicit knowledge about the location of every object in the system. PHT also maintains a *doubly-linked list* of all leaf nodes in order to providef sequential access and therefore range query support. The logical map between PHT nodes and DHT nodes is generated by computing the Secure Hash Function SHA-1 [EJ01] over the PHT node label. Thus, the DHT $nodeId$ associated with a given PHT node labeled l is $nodeId = SHA(l)$.

When a leaf node X reaches its maximum storage capacity B, where B is a system parameter, it carries out a *split* operation that creates two new *leaf nodes* following the prefix rule. In the same way, a *merge* operation is carried out when a *leaf node* stores less than $B/2 - 1$ keys.

Lookup. The lookup algorithm is the basic building block for both *insertions* and *range Queries*. Given a key K, this algorithm returns the *leaf node* whose label l is prefix of K.

There are two ways of implementing this algorithm: *linear lookup* and *binary lookup*.

The *linear lookup* algorithm starts at the root node and generates one DHT lookup per prefix of K until the target *leaf node* is located. This search incurs in high latency because it performs $O(D+1)$ sequential DHT lookups. It is possible to reduce this latency by performing all the $D+1$ DHT lookups in parallel. This optimisation reduces the latency to $O(\log(N))$. However, it induces a high message traffic.

The *binary lookup* performs a binary search over the prefix candidates of K until a *leaf node* is reached. The client starts by sending a lookup message to a label prefix of 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 the prefix tree. This process continues recursively and ends upon reaching a *leaf node* whose label is prefix of K. This search reduces the message traffic and latency $O(log(D))$ DHT lookups. The main drawback of this algorithm is that it is inherently sequential and therefore, it can incur high latency searches.

Range Queries. A range query consists in contacting all leaf nodes whose label fits within the given data range. A *sequential range query* over a data interval $I=[L,M]$ starts by locating the leaf node that indexes the lower bound L and crawls through the doubly-linked list until it reaches the leaf node that indexes the upper bound M. For instance, the range query over the data interval $I=[0000,0010]$ in the PHT presented in Figure 2.8 is performed as follows. The client node uses the lookup algorithm in order to find the lower bound *leaf node* (node 000). Then, the *leaf node* uses its double linked list until the upper bound is reached (node 001). The crawling of the doubly-linked list can be avoided by performing a *parallel search*.

A *parallel range query* starts by locating the node whose label shares the largest common prefix with the lower and the upper bound. This node recursively forwards the query until all leaf nodes that cover the range are reached. Following the example presented above, the query over the interval $I=[0000,0010]$ can be resolved by locating the common prefix node (node 00) which then recursively forwards the query until the two target leaf nodes are reached. In most cases, parallel range queries incur a lower latency than sequential range queries.

2.3.1.2 Place Lab

Place Lab [CRR+05] is a large scale system that allows mobile clients to compute their own position based on nearby radio beacons such as 802.11 access points and GSM towers. A client must perform *location* queries in order to retrieve the *latitude* and *longitude* coordinates of nearby radio beacons and estimate its position. Due to the potentially large amount of concurrent users, a central database is prone to become a bottleneck. In order to overcome this scalability issue, Place Lab proposes to implement a PHT on top of a DHT in order to perform *location range queries*. The radio beacon positions are

indexed using Space Filling Curves such as *z-ordering* [Mor66]. It uses the *binary lookup* algorithm presented above in order to support insertions and range queries. This solution could inherit all the properties of PHT such as insertion and query load balancing as well as random access to trie nodes. However, its use for performing *location-temporal range queries* raises the following issues.

- *Data locality.* The locality properties of SFC degrades with the number of dimensions. That is, points that are close in the multidimensional space can be far in the one dimensional space. The main consequence of this degradation is that a variable amount of false positives can be introduced.

- *Latency.* The linearisation of an n dimensional domain into a one dimensional domain increases the maximum height of the prefix tree from D to $n \times D$. This results in longer branches that can increase the latency of range queries. Furthermore, insertions and range queries rely on the *binary lookup* algorithm that presents a latency of $O(log(D) \times \log(N) \times l)$, where l is the latency among DHT nodes. For instance, if $D = 32$ bits, $N = 10^6$, and $l = 100$ [ms], every lookup could incur a latency of around 3 seconds. This latency has severe issues for real world P2P applications.

- *Message traffic.* Similarly to the latency issue, the message traffic generated per the *binary lookup* algorithm is $O(log(D) \times \log(N))$. The large number of insertions and range queries performed by every real world application generates a high message traffic in the network. For instance, if $D = 32$ bits and $N = 10^6$, the insertion of 50 millions objects generates about 1.5 billion messages.

In order to overcome these issues, Place Lab proposes to use a *client cache* in order to improve the performance of the binary lookup algorithm. The main idea is to store the labels of leaf nodes in memory in order to reduce the message traffic and latency of data insertions. Upon a cache hit, the client node uses the cached label in order to route a DHT message to the target leaf node. If the target node is still a leaf node, it process the request. Otherwise, the client node starts a new binary lookup. The main limitation of this optimisation is that when the entry is not a leaf node, the performance of the algorithm is worse than directly performing a single binary lookup.

Although Place Lab cannot directly support *location-temporal range queries* its extension is straightforward. It consists of adding the time dimension in the space filling curve in order to support such queries. In the rest of this work, this solution is referred to as *PHT*.

2.3.1.3 LIGHT

LIGHT [TZX10] proposes a distributed data structure similar to PHT. The main differences come from its structure and the mapping of trie nodes to the DHT. The LIGHT structure introduces an additional root node called *virtual root* and only leaf nodes are mapped to physical DHT nodes. Figure 2.9 depicts the LIGHT data structure. Similarly

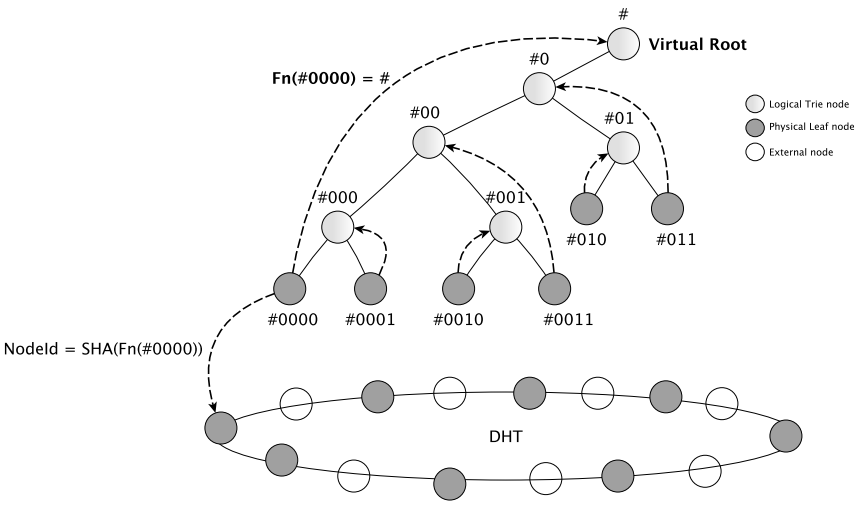


Figure 2.9: LIGHT data structure

to PHT, every node maintains a label λ which is a prefix. Every *leaf node* is mapped to a logical *internal node* using the *naming function* presented in equation 2.2.

$$f_n(\lambda) = \begin{cases} p0, & \text{if } \lambda = p011* \\ p1, & \text{if } \lambda = p100* \\ \#, & \text{if } \lambda = \# \end{cases} \tag{2.2}$$

Put simply, this function removes all the consecutives 0 or 1 at the end of the label. The novelty of this function is that it maps every *leaf node* to a unique *internal node*. For instance, $f_n(\lambda = \#0000) = \#$ and $f_n(\lambda = \#010) = \#01$. A *leaf node* is mapped to a DHT node whose *nodeId* is closest to the key $k = SHA(f_n(\lambda))$. Since different prefixes could map to the same label ($f_n(\#011) = f_n(\#0111) = f_n(\#0111) = \#01$), LIGHT defines a function called the *next naming function* presented in equation 2.3.

$$f_{nn}(u, k) = \begin{cases} p00 * 1, & \text{if } u = p0 \\ p11 * 0, & \text{if } u = p1 \end{cases} \tag{2.3}$$

This function receives the current prefix u and a key k as input, and returns the prefix that produces a different result over the naming function f_n. For instance, if the current prefix is #0 and the key is $k = \#0111$, then the next different prefix is $f_{nn}(\#0, \#0111) = \#011$.

The *split operation* allows to create two new *leaf nodes* labeled $\lambda 0$ and $\lambda 1$. They are mapped to two DHT nodes $k_1 = SHA(f_n(\lambda 0))$ and $k_2 = SHA(f_n(\lambda 1))$. Note that, due to the property of the naming function, one of these two mappings is the local node. In this way, LIGHT can save a variable amount of bandwidth depending on the distribution of data.

Lookup. The lookup algorithm returns the *leaf node* whose label is prefix of a given key k. Similarly to PHT, this algorithm performs a sequential binary search by routing a

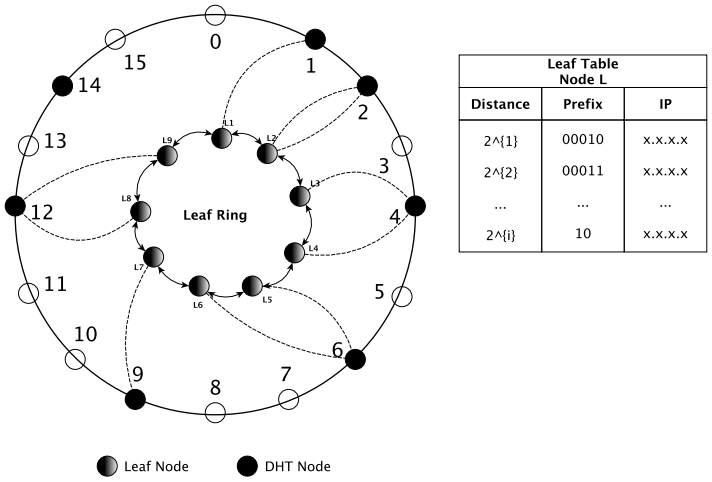


Figure 2.10: Dring data structure

variable number of DHT lookups over the different prefixes of λ. It reduces the message cost to $O(log(D/2))$ sequential DHT lookups.

Range Queries. A range query over the interval $I = [L, M]$ is performed as follows. First, the client node uses the lookup algorithm in order to find the *leaf node* that manages the key of the lower bound interval L. Then, using a recursive process and the DHT interface, it forwards the query to all the *leaf nodes* that cover the interval. The message complexity of this process is $O(z)$ DHT lookups, where z is the number of *leaf nodes* that cover I.

LIGHT decreases the lookup latency compared to PHT. However, range queries generate a high message traffic $O(z \times \log(N))$, where z is the number of leaf nodes. This is because only leaf nodes are mapped to the DHT, and therefore, range queries must use the DHT *get/put* interface in order to forward messages.

2.3.1.4 DRING

Dring [HRA+11] relies on the assumption that decreasing traffic message overhead reduces latency. It aims to reduce the message traffic of both insertions and range queries. The DRING structure is a variation of PHT that suppresses *internal nodes* and maintains *leaf nodes* in a ring structure called *Leaf Ring*. Thus, it changes the *parallel search* used in PHT to a *sequential search* through the Leaf Ring.

Similarly to Chord [SMK+01], every *leaf node* maintains a *Leaf table* that maintains up to m references to *leaf nodes* at a distance 2^i in the ring. Additionally, every node maintains links to its successor and predecessor leaf node in the trie structure. Figure 2.10 depicts an example of the DRING structure.

In order to gain access to the *Leaf Ring*, every node maintains a list of *leaf nodes* called *Leaf List* that contains the labels and IP addresses of *leaf nodes* that have been previously contacted. If the *Leaf List* is empty, the node can contact its neighbours or perform the traditional *linear* or *binary search* algorithms provided by PHT.

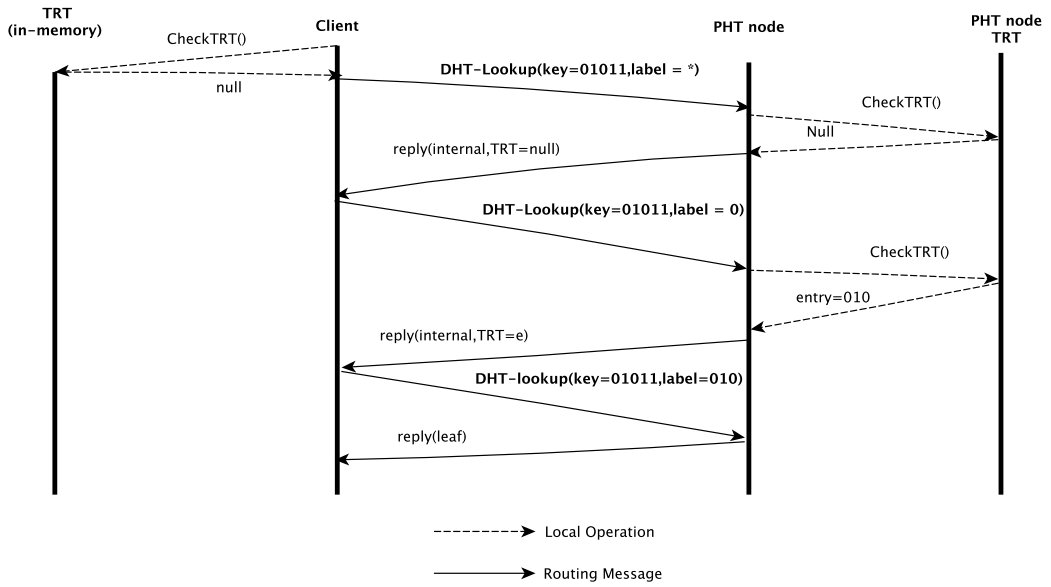


Figure 2.11: ECHO Routing example for key 01011 (leaf node l=010)

Lookup. The lookup algorithm uses the *Leaf List* in order to access the *leaf ring*. Then, the *Leaf Table* is used in order to reach the target *leaf node*. Performance results show that the performance of this algorithm in terms of the number of messages is similar to the *binary search algorithm* provided by PHT.

Range Queries. A range query over the data interval $I = [L, M]$ starts at the lower bound L and sequentially traverses the list of *leaf nodes* until the upper bound M is reached.

DRING reduces the number of messages of range queries compared the parallel range query algorithm introduced in PHT. Unfortunately, it induces a high latency due to the inherent sequential traversal of the Leaf Ring.

2.3.1.5 ECHO

ECHO [HASB16] is a cache solution that extends PHT in order to reduce message traffic and latency. The main idea is the introduction of a linear Tabu Routing Table (TRT) that stores the labels of internal nodes in order to reduce the search space. A client node first checks for a *label* in its TRT and then follows the *linear search algorithm* using the DHT routing interface.

Figure 2.11 depicts an example of the *linear lookup algorithm* proposed in ECHO. The client node wants to locate the leaf node whose label is prefix of the key $k = 01011$. First, the client node checks in its own TRT. As it does not find any entry, it uses the DHT lookup interface to route a message to the root node (labeled $l = *$). The root node checks in its own TRT and replies with its own state. The search continues onto the next node labeled $l = 0$ that has an entry in its TRT that can avoid the *internal node* labeled

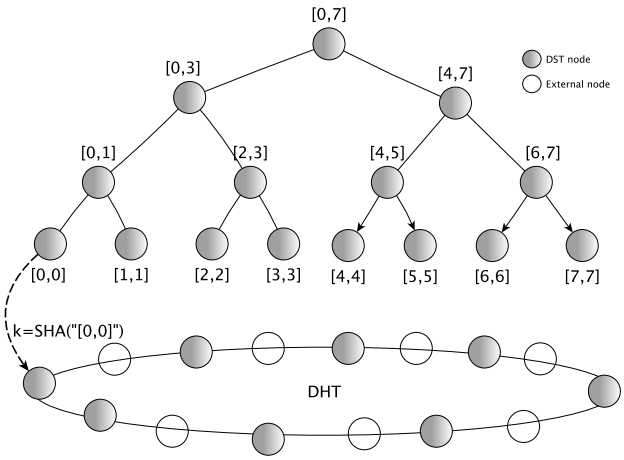


Figure 2.12: Distributed segment tree structure

$l = 01$. This process continues, until the leaf node labeled *l=010* is reached.

ECHO reduces the message traffic and latency of the PHT linear lookup algorithm However, its evaluation shows that, depending on the cache state, it still requires between 2 and 9 sequential DHT routing messages to reach the target *leaf node*. These results are similar to the *binary lookup* algorithm provided in PHT that performs a lookup in $O(log(D))$ sequential DHT routing messages. Therefore, although ECHO reduces the latency and the number of messages, it still delivers high latency and high message traffic lookups.

2.3.1.6 Limitations of Distributed Prefix Trees

Distributed one dimensional prefix trees provide a large scale index layer on top of DHTs. They provide properties such as *random access* to any node in the *trie* and *load balancing* of insertions and range queries. However, these solutions exhibit high latency and high message traffic that have a strong impact on the performance of practical applications.

2.3.2 Binary Trees

Building *binary trees* over DHTs is a popular solution for indexing one dimensional data at scale. The main difference with *prefix trees* is that the former provide a more flexible way to partition the space. That is: it can be partitioned in equal parts while providing *random access* to tree nodes, or in more complex shapes in order to improve load balancing. In practise, load balancing is sacrificed for the sake of *random access*. Furthermore, *binary trees* usually allow *internal nodes* to store data.

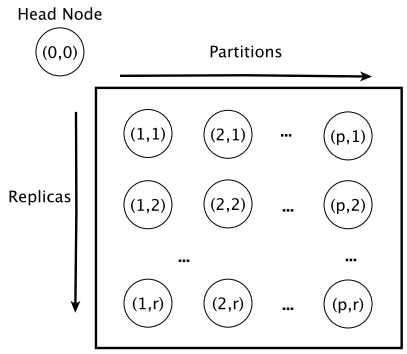


Figure 2.13: Load Balance Matrix (LBM)

2.3.2.1 Distributed Segment Tree

A Distributed Segment Tree (DST) [ZSLS06] is a distributed setting of a *segment tree* [DBVKOS00] extracted from computational geometry. In essence it is a static binary tree of $l+1$ levels that partitions the space into equal segments. The main novelty of DST is that all *DST nodes* are allowed to store data.

At start-up time a setup process starting at the root node assigns every *DST node* a recursive label $l = [a, b]$ that associates a node with a segment. Figure 2.12 depicts an example of the DST structure and its mapping on DHT nodes. A *DST node* is assigned to the DHT node whose label is closest to the key $k = SHA(label)$. Similarly to prefix trees, this strategy allows to provide *random access* to tree nodes.

Data insertions. The insertion of a key k consists in locating all the segments where k is contained and perform $l+1$ parallel insertion operations on every level. When a node stores more than γ keys, where γ is a system parameter, the node becomes *saturated*. A *saturated node* does not store new incoming data.

Range queries. A range query over a data interval $I = [L, M]$ is performed as follows. First, the client node computes the minimum union of subintervals that completely covers I and it forwards the query to all these nodes. If a node is *saturated* it forwards the query to its children nodes until only non saturated nodes receive the query.

2.3.2.2 Range Search Tree

Range Search Tree (RST) [GS04] aims to address the load balancing problem of DST [ZSLS06] presented at two levels: storage and query. The storage level, RST introduces a novel data structure called *Load Balance Matrix* (LBM) which allows nodes to store more than γ keys. Figure 2.13 depicts the LBM data structure. It stores p partitions and r replicas whose indexes are maintained on a *head node* labeled $(0, 0)$.

The query level, RST uses an *elastic band* that forwards queries to lower or higher levels according to the insertion and query load. Only nodes inside of this band are able to process requests. The state of the band is maintained in a structure called *Path*

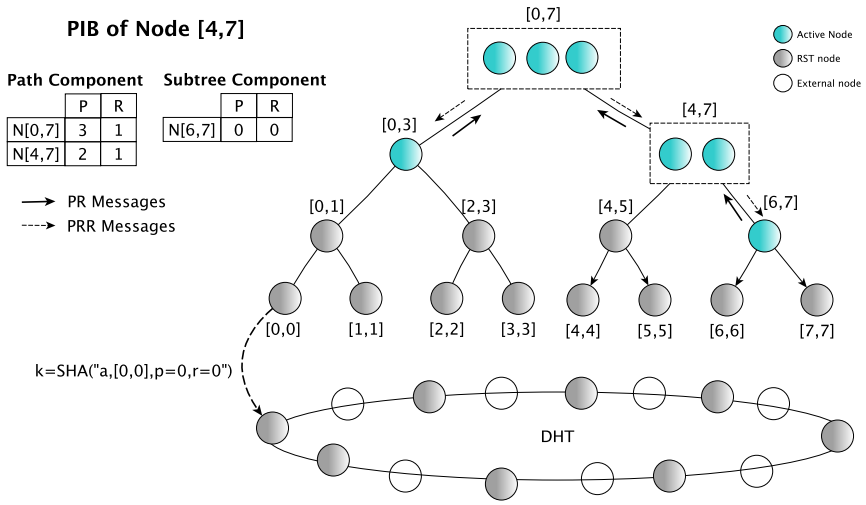


Figure 2.14: Range search tree structure

Information Base (PIB) which is divided into a *path component* and a *subtree component*. The *path component* stores the matrix size (i.e, the values of p and r) of nodes in the path from the current node to the *root* node. The *subtree component* contains the matrix size of the subtree of the current node. Figure 2.14 presents an example of the RST structure. For instance, the *path component* of the node $[4, 7]$ stores the size of the matrix of itself and the root node and its *subtree component* stores the LBS state of its own children (i.e, the node $[6, 7]$).

The *elastic band* is maintained as follows. Every Δt all RST nodes send a *Path Refresing* (PR) message to their parent nodes. This message contains the *subtree component* presented above. In turn, parent nodes send back a *Path Refreshing Reply* (PRR) message that contains the *path component*. Based on a set of statistics about the input load, a node can locally decide to include or detach a node from the *elastic band*. Including a new child node involves the replication of a part of its LBM. This operation is similar to the *split* operation presented in PHT and LIGHT.

The mapping of RST nodes to the DHT uses the matrix coordinates and the label of the node. That is, an RST node that covers the interval $l = [a, b]$ is mapped to the DHT node whose nodeId is closest to the key $k = SHA(l, p, r)$.

Data insertions. A key k is stored in all segments of the *elastic band* that cover k. The client node must search the band of at most one node in the branch that covers k and insert the data at all nodes in this branch.

Range queries. A range query over a data interval $I = [L, M]$ is performed as follows. First, the local node retrieves the PIB of I from at most three nodes at a level $\lceil \log(M - L) \rceil$. Then, the local node reconstructs the logical RST structure and executes a local algorithm to decompose the query range and minimises the number of physical nodes that should be contacted.

RST improves load balancing compared to DST. However, it incurs a high maintenance

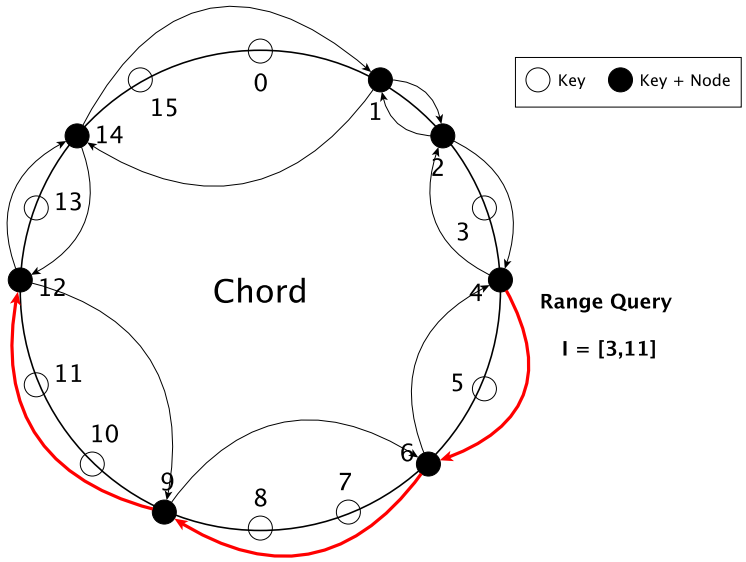


Figure 2.15: Example of MAAN range query processing

cost in terms of message traffic and bandwidth that limits its scalability.

2.4 DHT-dependent solutions

DHT-dependent solutions differ from over-DHT solutions in the sense that they aim to adapt the DHT instead of using it as a building block in order to support range queries. This section presents a brief summary of these solutions as well as their main strengths and drawbacks.

2.4.1 MAAN

Multi-Attribute Addressable Network (MAAN) [CFCS04] extends Chord [SMK+01] in order to support multidimensional range queries. It uses a *locality preserving hash function* in order to create data locality. For two different values v_i and v_j, a hash function H_l is defined as a *locality preserving hash function* if $H(v_i) < H(v_j)$, $v_i < v_j$. Furthermore, if the data interval $[v_i, v_j]$ is split into the intervals $[v_i, v_k]$ and $[v_k, v_j]$, the corresponding interval $[H(v_i), H(v_j)]$ must be split into $[H(v_i), H(v_k)]$ and $[H(v_k, H(v_j)]$.

Data insertions. A key $k \in [v_{min}, v_{max}]$ is mapped to the node whose *nodeId* is successor of $H_l(k)$, where $H_l(k)$ is a *locality preserving hash function* in the domain $[0, 2^m - 1]$ defined as follows.

$$H(k) = (v - v_{min}) \times (2^m - 1)/(v_{max} - v_{min}) \tag{2.4}$$

Range queries. A range query over the interval $I = [L, M]$ starts at the successor node of the lower bound L. Then the query traverses the successor links until the upper bound

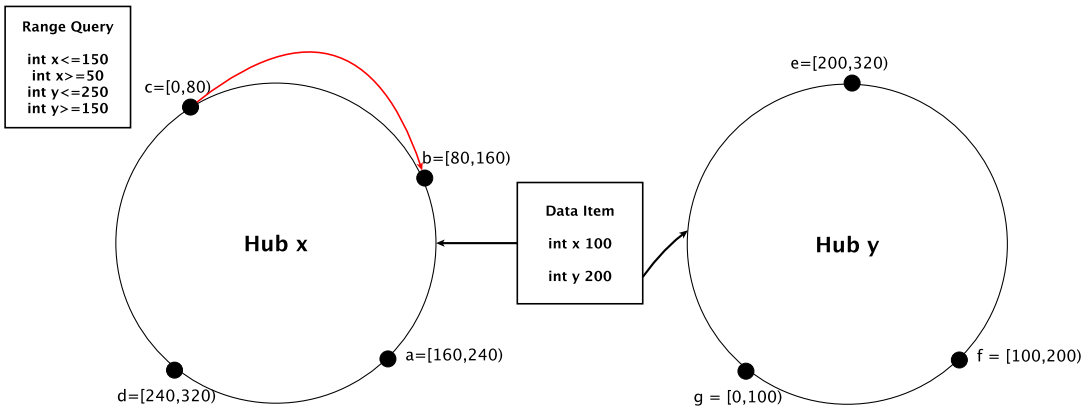


Figure 2.16: Example: Mercury query processing

M is reached. Figure 2.15 presents an example of a Chord ring defined in the domain $[0, 2^4 - 1]$. The range query over the interval $I = [3, 11]$ starts at the node n_4 and traverses the successor list until the node n_{12} is reached.

This solution presents two main drawbacks. Firstly, it induces high latency searches because of the sequential nature of the range queries that require $O(\log(N) + z)$ routing hops, where z is the number of nodes which maintain an interval. Secondly, *locality preserving hash functions* provide poor load balancing under skewed distributions.

2.4.2 Mercury

Mercury [BAS04] proposes a method to support range queries which is similar to MAAN [CFCS04]. It introduces a novel structure that partitions physical nodes into virtual rings called *routing hubs*. Every *routing hub* follows a circular overlay similar to Chord. The main difference is that the former maintains data locality (i.e, every node stores a range of values). Every node in a *routing hub* maintains links to its predecessor and successor within its own hub and a link to each of the others *hubs*.

Data insertions. A data item is sent to all hubs for which it has an associate attribute. It has a complexity of $O(d \times \log(N))$, where d is the number of attributes or dimensions.

Range queries. A range query starts at one single *hub*. The query processing starts at the lower bound and traverses the links of successors until the upper bound is reached.

Mercury also exhibits a poor load balancing under skew distributions. Furthermore, it introduces a high maintenance cost of because of its multiple ring structure.

2.4.3 Squid

Squid [SP08] is a Chord based overlay that supports multidimensional range queries. It proposes to use *Space Filling Curves* in order to map multidimensional data into a DHT identifier space, loosely preserving data locality. It uses the Hilbert Curve in order to represent a key k as a one dimensional integer in the domain $[0, 2^m - 1]$, where m is the

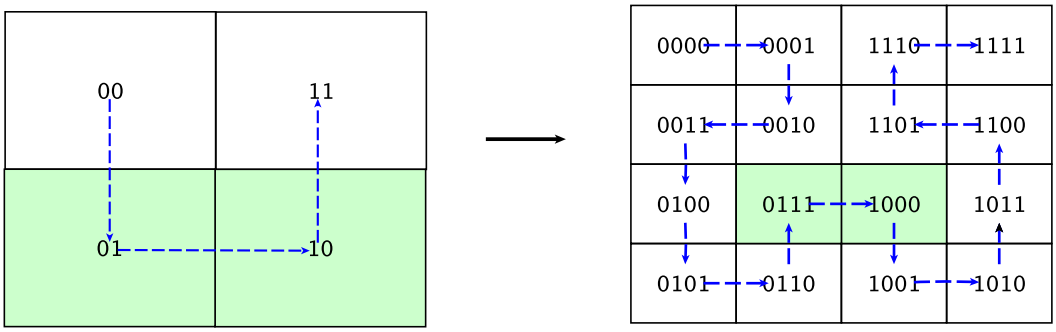


Figure 2.17: Example: Squid range query processing on the interval $I = [0111, 1000]$

number of bits used by the Hilbert Cuve that must be equal to the number of bits used to define the *nodeId* space. Data locality is achieved by directly mapping the Hilbert identifiers to Chord nodes.

Data insertions. An m-dimensional key k is mapped to its m-bit Hilbert value and then routed to the Chord node whose *nodeId* is successor of k. This is achieved in $O(\log(N))$ routing hops.

Range queries. Range query processing relies on the recursive hierarchical structure of the Hilbert Curve. It is resolved by multiple recursive DHT lookups using a prefix-based schema in order to reach all the Hilbert segments that cover the interval. At each search step, longer prefixes of candidate segments are looked up. This process continues recursively until all segments are reached. Figure 2.17 depicts an example of a range query on the interval $I = [0111, 1000]$. First the query is routed segments 01 and 10 which recursively forward the query to the target segments 0111 and 1000. This process distributes the computing of Hilbert segments among DHT nodes. However, the use of the DHT lookup interface in a recursive top down process imposes a high latency and message cost.

The main drawback of Squid is a poor load balancing under skewed distributions and a high message and latency cost of range queries.

2.4.4 Saturn

Saturn [PNT12] addresses the problem of load balancing by using order preserving functions to support range queries. Specifically, it addresses the problem of replica placement under skewed distributions. It introduces a novel multiple ring, order-preserving architecture implemented on top of an order-preserving DHT overlay such as MAAN or Mercury. The main idea is to detect overloaded nodes and randomly distribute their load using a novel order preserving *multiring hash function* $mrfh(k, \delta)$. This function returns a rotated identifier based on the replication degree δ. Every node maintains a counter α_n of the number of accesses per node. If $\alpha_n > \alpha_{max}$ the node is *overloaded*. An overloaded node must use the *multiring hash function* in order to replicate its data on a new ring that has a rotated *nodeId* space. This strategy distributed the load to different areas of

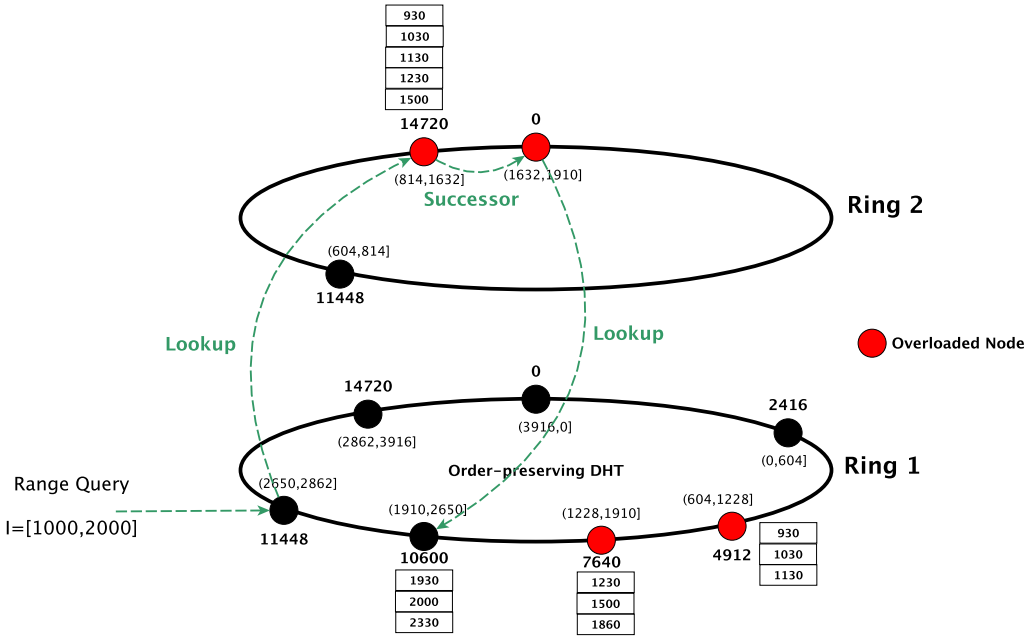


Figure 2.18: Example of Saturn architecture

the DHT. The maximum number of replicas δ_{max} per data item is an application defined parameter.

Figure 2.18 presents an example of the Saturn architecture. In this example, nodes 4,912 and 7,640 are overloaded and therefore they are replicated at nodes 11,448, 14,720, and 0.

Data insertions. A key k is stored at the node n whose $nodeId$ is successor of k. The key k will eventually be replicated if n is overloaded.

Range queries. A range query over the interval $I = [L, M]$ is processed as follows. First, the client node retrieves the replication degree δ of the node in the first ring that stores the lower bound L. Then, it randomly selects a ring $[1, \delta_{max}]$ and submits the query. Finally, the query follows the successor list of nodes until the upper bound M is reached. If some of the internal nodes were not replicated, the query is randomly forwarded to a lower level ring.

Saturn improves load balancing when using order preserving functions over DHTs. Multidimensional range queries can be supported using SFCs such as the Hilbert Curve. However, the sequential search introduces a high overhead in terms of *false positives*.

2.5 Non-DHT solutions

Non-DHT solutions use other distributed data structures such as Skip Lists [Pug90], Voronoi Tesselations [OBSC09], and Trees in order to create data locality and support range queries. They do not rely on DHTs as a basic building block. This section present

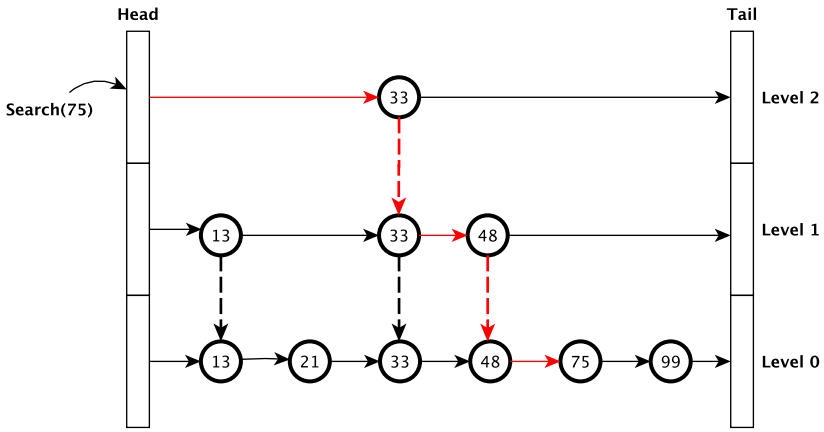


Figure 2.19: A Skip List with $n = 6$ nodes and $\lceil \log(n) \rceil = 3$ levels

a brief summary of these solutions as well as their main strengths and drawbacks.

2.5.1 Skip-List based solutions

A skip list [Pug90] is an alternative data structure to traditional B-trees that supports range queries. The main difference with B-Trees is that skip lists do not need to be load balanced. Skip lists combine a probabilistic model and a randomised data structure organised as a tower of increasingly sparse linked lists. Linked lists are organised into l_{max} levels and nodes appear at a level i with some fixed probability p that is inversely proportional to i. A data item is replicated in $1/(1-p)$ lists in average. The linked list at level $i = 0$ contains all the data items, then the probability decreases following a simple pattern such as $p = \frac{1}{2^i}$. Similarly to trees, insertions and range queries follow a top-down traversal from the higher l_{max} level to a lower level. Figure 2.19 presents an example of a skip list that indexes $n = 6$ data items. For instance, the search for key $k = 75$ starts at the higher level and goes down through node links until it is reached at level 0. This structure provides an expected search, insertion, and deletion complexity of $O(\log(n))$.

SkipNet and SkipGraph. SkipNet [HJS+03] is a distributed generalisation of a skip list [Pug90] in a P2P environment. There are two main differences with skip lists: (i) nodes at every level form a double linked ring instead of a linked list, and (ii) there may be many lists at a given level i. Furthermore, SkipNet decouples the node identifier space from attribute values. It defines both numerical random *nodeIds* and string name identifiers called *nameIds* and provides independent routing algorithms.

The logical structure is a multi ring architecture. Every level is represented as a ring that maintains nodes ordered by *nameId*. Every level i is split into two rings at level $i+1$ composed of randomly selected nodes. In order to distribute nodes on a ring, a prefix naming strategy is adopted. Every ring maintains a binary prefix that determines node membership at level i. For instance, the ring at level $l = 0$ is named the root ring and it splits into two rings named 0 and 1. Thus, a node with *nodeId* 11 will belong to ring

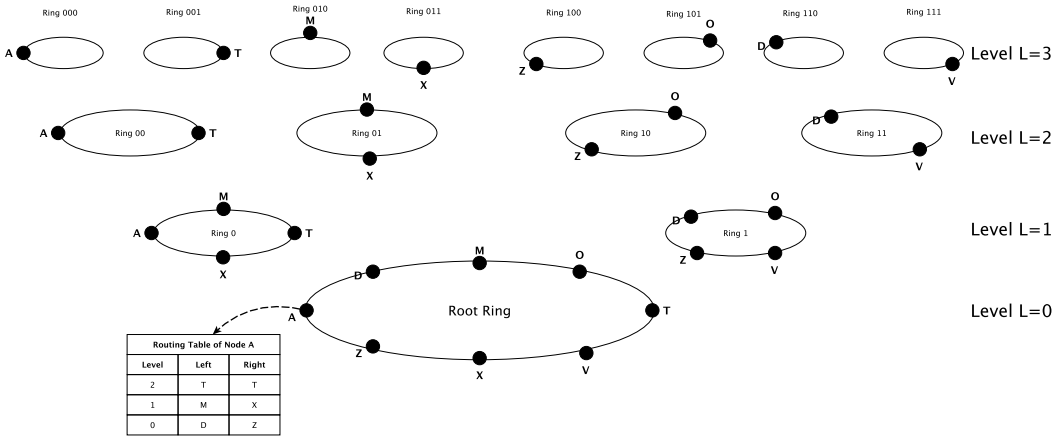


Figure 2.20: Example: SkipNet Logical Architecture

1. Figure 2.20 presents an example of the logical SkipNet architecture. The root ring is composed of all nodes that are distributed into a multi ring structure.

Every node maintains a routing table composed of $2\log(N)$ pointers. Every entry of the table maintains two links nodes of a level i that are at a distance of 2^i nodes in the name identifier space. That is, every link at level i traverses exactly 2^i nodes. This strategy is similar to the one used by Chord, the main difference being that SkipNet considers the name identifier space for its distance metric instead of the numerical $nodeId$ space. This table routing algorithm presents a message complexity of $O(\log(N))$ routing hops.

Routing algorithm. SkipNet provides both routing fin the $nodeId$ space and the $nameId$ space. The routing algorithm in the $nodeId$ space arrives to the node whose $nodeId$ is closest to the target identifier. It uses a prefix based strategy that works as follows. It starts in the root ring (level 0) until a node that share the first digit is reached. At this point, the routing operation forwards the message to the first level until a node that shares the first two digits is reached. This procedure continues until the node that maintains the closest $nodeId$ is reached.

The routing procedure in the $nameId$ space finishes when the message arrives at a node whose $nameId$ is closest to the destination. This procedure follows the same top-down strategy as the Skip List search presented above. At every step, the message is routed along the highest-level pointer that does not exceed the destination value. Both algorithms reaches their destination in $O(\log(N))$ routing hops with a high probability [HJS+03].

SkipNet was not conceived to perform range queries. However, it does support them via its skip list based structure. Given a data interval $I = [L, M]$ the range query starts with the routing algorithm presented above in order to locate the lower bound L at the root ring. Then the double linked list is used to reach the upper bound M.

Fault tolerance is achieved through replication in a set of nodes called the *leaf set*. This set is similar to the fault tolerance mechanism introduced in Pastry [RD01], and contains L closest nodes in the $nameId$ space.

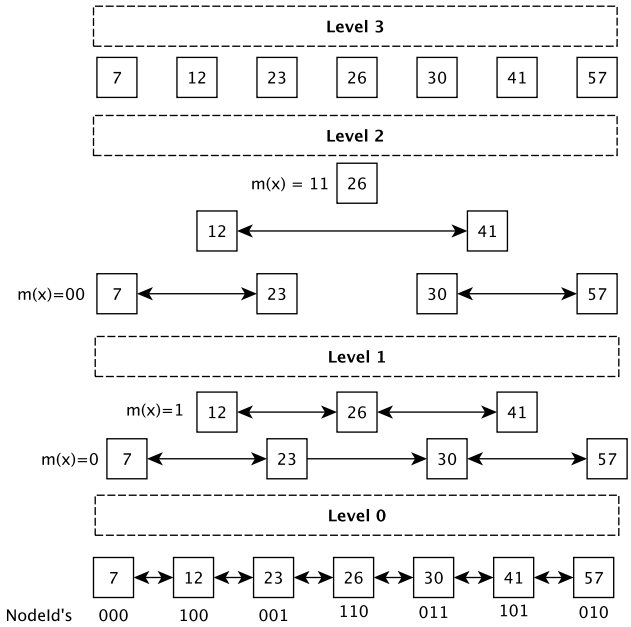


Figure 2.21: Example of a skip graph composed of 7 nodes and 4 levels

Skip Graph [AS07] is similar to SkipNet [HJS+03] in that both solutions generalise skip lists [Pug90] in a P2P environment. The main differences come from the data structure and how they build data locality. Skip Graph organises the skip list into hierarchical double linked lists instead of sorted rings. Furthermore, a Skip Graph node represents a *resource* instead of a computer name. The nodes that compose a list are defined by a *membership vector* $m(x)$ that is defined as a prefix of length i, where i is the level of the list. Figure 2.21 presents an example of a skip graph composed of 7 nodes and 4 levels. For instance, the list of level $i = 1$ is composed of nodes that have a $nodeId$ starting by the prefix of the membership vector $m(x) = 0$ and $m(x) = 1$. This strategy sparse nodes into $O(\log(N))$ lists.

Search and range queries are performed in a way similar to SkipNet. The search process starts at the highest level of the node issuing the query. At every step it traverses the highest list without passing the key. If the search cannot go further in the list, it goes down one level until in the worst case it reaches level $i = 0$. Range queries start locating the lower bound L at the list of level $i = 0$ and then it traverses the double linked list until the upper bound M is reached.

There are two main drawbacks when using Skip Graph to index a multidimensional space: (i) it uses a linear linked list that introduces a variable amount of false positives, and (ii) it induces high latency searches because it assumes one key per node.

ZNet. ZNet [SOTZ05] combines *z-ordering* [Mor66] and *Skip Graph* [AS07] in order to provide a multidimensional index. It improves Skip Graph by introducing a quadtree partition strategy in order to reduce the latency of range queries. Every subspace of the quadtree decomposition is mapped along a one dimensional position in the *z-ordering*

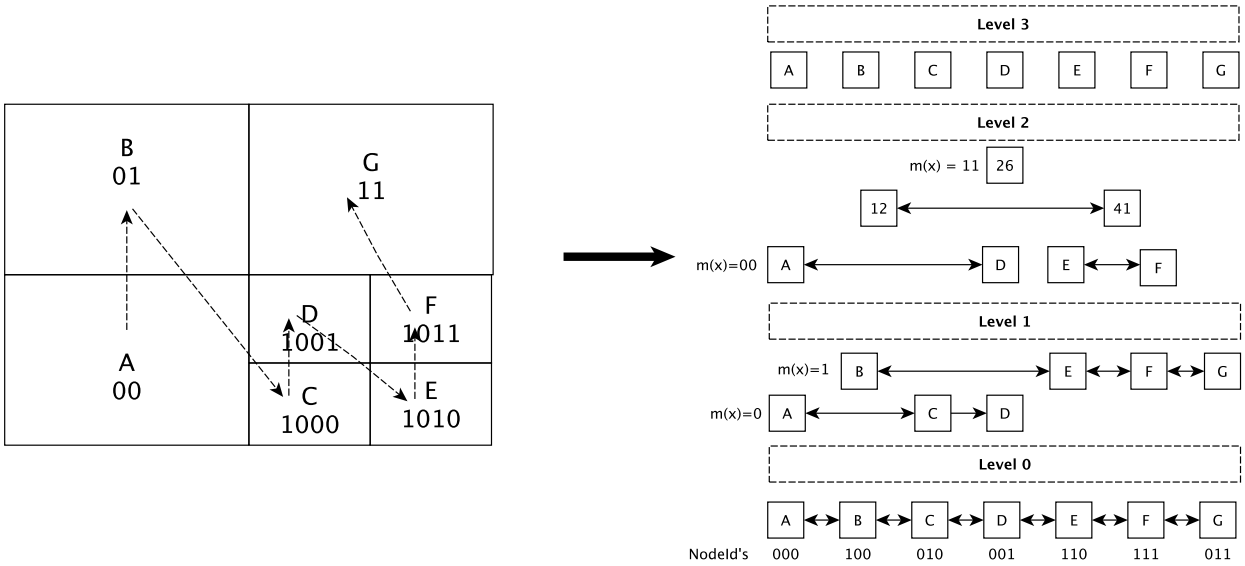


Figure 2.22: Example: ZNet overview

space filling curve. These subspaces are organised in a Skip Graph structure. Figure 2.22 presents an example of the ZNet structure. Every node maintains a routing table that contains the left and right neighbour for every skip list i. The size of the list is at most $\lceil \log(N) \rceil$ entries.

The lookup algorithm receives a key k as input and returns the node that manages the area containing k. First, the client node maps k to its *z-order* value. Then, the client node routes the message to the node that maintains the closest subspace. For instance, the lookup procedure from the node A (00) to the key $k = 1011$ (node F) starts by locating the neighbours of A that share the longest common prefix with k (nodes C or D). If A chooses D, it forwards the query to E, which is closer to F. Finally, E forwards the query to F. This procedure takes $O(\log(N))$ routing hops. Range queries are forwarded to the node with the shorter prefix. Then the query is recursively forwarded until all nodes that cover the interval are reached.

MURK and SCRAP. MURK [GYGM04] uses a kd-tree [Sam06] data partitioning strategy that splits one dimension at a time following a defined order. When a new node joins the overlay, the space is partitioned into two equal load areas. This strategy is similar to the one proposed by CAN. The main difference is that MURK does not split the space into equal areas. Instead, it splits the space based on the load in order to provide load balancing.

Every MURK node maintains links to its neighbors in the space. It allow nodes to follow a *greedy routing* strategy that routes a message to the node that minimises the distance to the target space. Altough this structure is similar to CAN, the authors remark that this baseline structure has two shortcomings: (i) Non-uniformity: The number of grid neighbors is not uniform because the space is not equally split. (ii) Inefficiency:

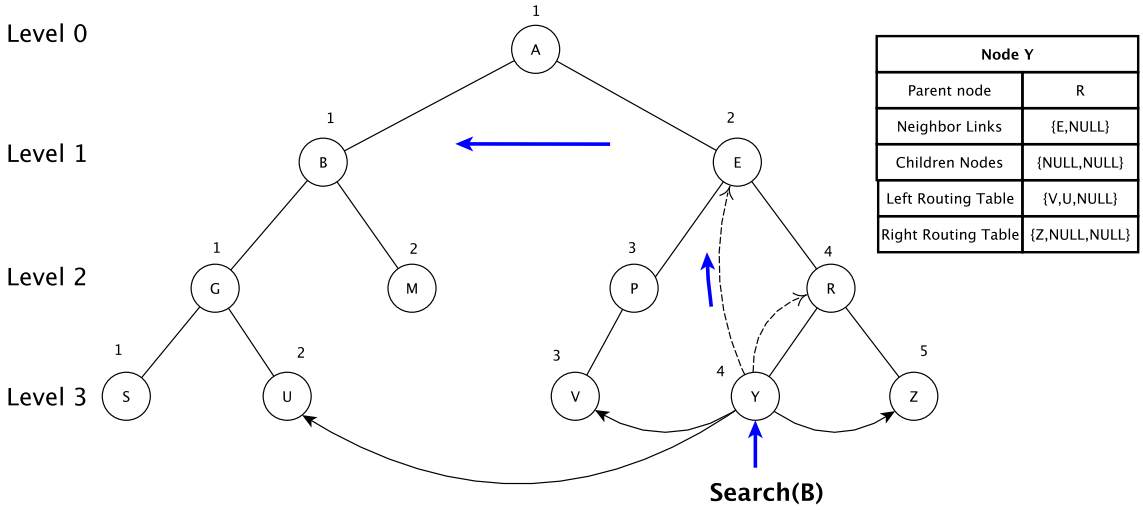


Figure 2.23: Example: BATON network

The neighbor links does not promote efficient routing, especially when the number of dimensions is low. In order to address the second problem, they propose to apply Skip Graphs [AS07] in order to allow nodes to connect with farther nodes in the $nodeId$ space. In order to choose such nodes they propose two different approaches: (i) Random: Every node maintains a random set of links. (ii) *space-filling Skip graph*: This strategy combines SFCs and Skip Graph in order to create a one dimensional linear order among partitions.

SCRAP [Sam06] is very similar to ZNET [SOTZ05], both of them propose to combine SFCs and Skip Graph [AS07]. The only difference is that ZNET explicitly uses the *z-order curve*.

2.5.2 Tree-based solutions

BATON [JOV05] is structured as a *balanced binary tree* overlay network. Every node is associated to a level L (starting from $L = 0$ at the root node) and a number from 1 to 2^L. The tree level and number combine into a unique identifier for each node in the tree. Both *internal nodes* and *leaf nodes* are allowed to store data. The internal structure of a BATON node X is composed of a set of links to its parent node, children nodes, and adjacent nodes (left and right). The left and right adjacent nodes are determined following an *in-order* traversal. Additionally, every node maintains the data range it covers and two routing tables that link to nodes to the left and right in the same level of the tree . These tables are similar to the *finger table* proposed by Chord. That is, they contain nodes which are 2^i nodes away to the left and right ($i = 1, 2, 3$).

When a new node X joins the network it must send a *JOIN* message to an existing node A. Node A can accept X as a child node if A has less than two children nodes and if its left and right routing tables are full. Otherwise, node A must route the query to its parent node if one of its two adjacent tables are not full or to an adjacent node if both

tables are full. This procedure continues until a node that fulfils the condition is reached. When a node accepts X as its child, it notifies all its neighbours nodes in its routing tables and updates its adjacent links.

Node departures may impact the tree balance. Only a *leaf node* that has no neighbours with children nodes in its routing tables can leave the network without affecting the tree balance. In other cases, a leaving node must find a replacement for itself: a node whose absence does not impact the tree balance. Both join and departure procedures cost $O(\log(N))$ messages.

Figure 2.23 presents an example of the BATON network. For node Y, all the links are indicated. The lookup algorithm uses these links in order to reach the destination node. For instance, a search for peer B starting at peer Y, is first forwarded to node E via its left adjacent link, and then to peer B using the left routing table of E. This procedure requires $O(\log(N))$ routing hops.

Range queries follow a similar algorithm. The main difference is that instead of looking for the node that contains the data range, it looks for a node that manages an interval which intersects the target interval $I = [L, M]$. Once the intersecting interval is reached, the query is forwarded until all of interval I is covered. The complexity of this operation is $O(\log(N) + z)$ sequential messages, where z is the number of nodes that covers the range query interval.

The main drawback of BATON is its poor load balancing under skewed distributions (such as location-temporal data). In order to overcome this issue, BATON migrates data among nodes in order to improve load balancing. However, this process introduces an additional overhead in terms of bandwidth and network maintenance.

VBI-Tree Virtual Binary Index (VBI-Tree) [JOV+06] is an extension of BATON [JOV05] to support multidimensional range queries. Its overlay network is based on a balanced binary tree. Upon this layer it defines abstract methods for multidimensional indexing that are extensible to several multi-dimensional indexing structures such as R-Tree [Gut84], X-Tree [DP78], SS-Tree [WJ96], and M-Tree [CPRZ97].

VBI-Tree defines two types of logical nodes: *Internal nodes* and *leaf nodes*. *Internal nodes* are routing nodes that only store links to other tree nodes. *Leaf nodes* maintain routing information and actually store data. There are three main differences with BATON. (i) Every VBI-Tree physical node is associated to one *leaf node* and one *internal node*; (ii) Routing tables in VBI-Tree do not need to keep information about ranges covered by neighbour nodes. Instead, each *internal node* maintains an *upsite table* with information about ranges covered by each of its ancestors; (iii) Each VBI-Tree node needs to keep information about heights of sub-trees rooted at its children nodes.

A new node must follow a two step process. The first step consists in determining the position of the node in the tree, in almost the same way as BATON does. The only difference is that only internal nodes are considered. The second step consists in locating an *internal node* that has both its routing tables full (left and right) and has at most one *internal node* as a child node. When this node is located, it accepts the new joining node as a child node and two new *leaf nodes* are created.

The node departure mechanism is similar to the one proposed by BATON. If a leaf node wants to leave the network and there is no internal node in its routing table with a routing

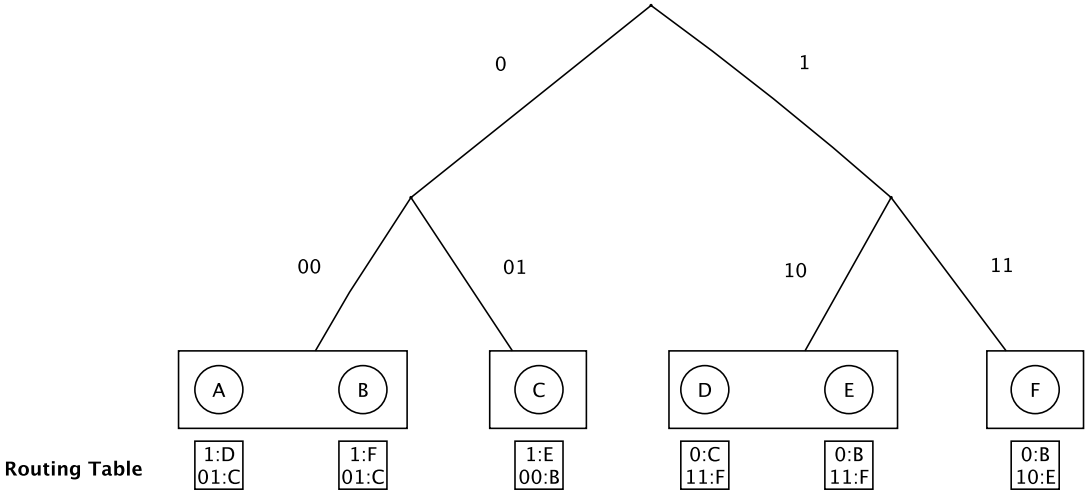


Figure 2.24: P-Grid overlay

child, the node can leave the network without affecting the tree balance. Otherwise, it needs to find a replacement *leaf node*. Both join and departure procedures cost $O(\log(N))$ messages.

The index construction on the tree overlay consists in assigning a region of the attribute space to every *leaf node*. Every internal node gets assigned a region that covers all regions maintained by its children nodes. This approach allows to implement several tree indexing structures. The main difference relies on how the split algorithm divides the data set.

The range query algorithm over the data interval $I = [L, M]$ follows a recursive and parallel process that arrives to all nodes that intersect the query interval. This procedure requires $O(\log(N) + z)$ messages. Similarly to BATON, the main drawback of VBI-Tree is load balancing under skewed data distributions.

P-Grid P-Grid [ACMD+03] is a binary prefix tree overlay network similar to PHT in which every node maintains a *leaf node* of the tree. The main difference with PHT is that P-Tree does not rely on a DHT overlay. Every node maintains a label $l = \{0, 1\}^D$ (i.e, a binary string of length D) that represents the position of the node in the trie. A key k is stored in a node that has a label prefix of k. In order to route messages, every node maintains a *routing table* which contains one entry for every level of the trie. Each entry is chosen as a node that does not share a common prefix at level i. Additionally, every node maintains adjacent links to connect with neighbour nodes. In order to provide fault tolerance, a given prefix can be assigned to multiples nodes. Figure 2.24 depicts an example of a P-Grid overlay composed of 4 identifiers and 6 nodes. The routing table of node A is composed of the node D at level 0 and node C at level 1. Two identifiers (00 and 10) span over two nodes in order to provide fault tolerance.

The lookup algorithm receives a key k and returns the node whose label is prefix of k. At every step, this algorithm uses the routing tables to find a closer node until the target node is reached. It has a message complexity of $O(\log(N))$, where N is the number of

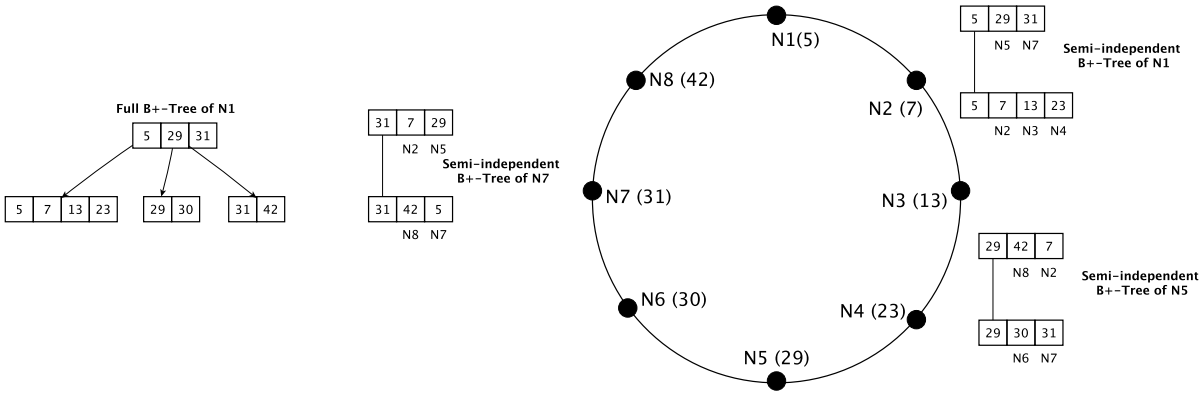


Figure 2.25: Example of a P-tree structure from the perspective of node N1

nodes in the network.

Range queries over the data interval $I = [L, M]$ start by locating the node that stores the lower bound L and then use the adjacent nodes to sequentially forward the query until the upper bound M is reached.

The main drawbacks of P-Grid are load balancing under skewed distributions, and the high range query latency due to the sequential nature of the range query processing algorithm.

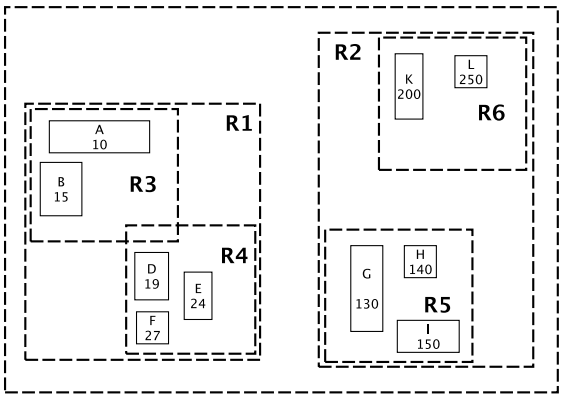
P-Tree [CLGS04] is a virtual balanced B+-Tree [Com79] built using Chord [SMK+01] as baseline. A B+-Tree of order d is a balanced search tree in which the root node of the tree has between 2 and $2d$ entries. All non root nodes have between d and $2d$ entries. The key idea of P-Tree is to store data as Chord does and build *semi-independent* B+-Trees at every node in order to index data. At every node, a *semi-independent* B+-Tree comprises the local data and links to the *left-most root-to-leaf* path of the tree.

Figure 2.25 depicts an example of P-Tree. The data set $\{5, 7, 13, 23, 29, 30, 31, 42\}$ is stored in a Chord ring composed of nodes $N1$ to $N8$. For the sake of clarity, only the structures related to node $N1$ are represented. The full B+-Tree of $N1$ is distributed using links to other nodes that maintain links to the rest of the tree structure.

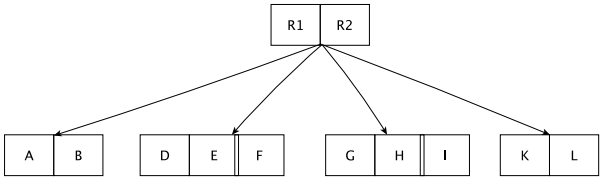
Range query processing over the data interval $I = [L, M]$ is similar to the range query processing provided by a B+-Tree. The main difference is that when a query traverses from a level to the next level, it actually jumps from a node to an other via the links provided by the structure. First, the query reaches the lowest tree level that stores the lower bound L. Then, it sequentially traverses the Chord links until the upper bound M is reached. For instance, if node $N1$ sends a range query over the interval $I = [29, 31]$ it forwards the query through the node $N5$ that stores the lower bound 29 and then $N5$ forwards the query to its successors until $N7$ that stores 31 is reached. The latency of this operation is $O(\log(N) + z)$, where z is the number of nodes that cover the interval.

P-tree has the following main drawbacks. (i) It assumes that one node stores a single data item which is not realistic. (ii) The sequential nature of the range query algorithm provides high latencies. (iii) Inconsistent views that nodes might have on the global

B+-Tree adds an additional overhead.



(a) Example: Hilbert R-Tree data rectangles



(b) Example: Hilbert R-Tree

Figure 2.26: DHR-Tree indexing strategy

DHR-Tree Distributed Hilbert R-Tree (DHR-Tree) [WS06] is a multidimensional data structure that provides insertions and multidimensional range queries. It combines linearisation and the R-Tree [Gut84] spatial indexing structure. An R-tree is a balanced data structure similar to B-Trees, that can index geometrical and multidimensional point data. The main difference with traditional B-Trees is that R-trees index objects that can have a rectangular shape with no linear order defined on them. Its structure is hierarchical with *internal nodes* storing pointers to the *minimum bounding rectangle* (MBR) that encloses all the keys in its subtree. A node stores between m and M data items ($1 < m \leq M$). The only exception is the root node, which may hold between 1 and M entries.

A Hilbert R-Tree [Sam06] combines the overlapping region technique of R-Trees and the Hilbert space-filling curve. It first stores the Hilbert values of the data rectangle centroid in a B+-Tree, then enhances each interior B+-tree node by the MBR of the sub-tree. This strategy reduces the dimensionality of the data to one dimension and therefore simplifies insertions and range queries.

Figure 2.26 presents an example of a Hilbert R-Tree that indexes rectangles in a two dimensional space. Every rectangle is mapped to a one dimensional identifier using the Hilbert Curve. These rectangles are indexed in a B+-Tree using its Hilbert identifiers as key.

The distributed implementation of the B+-Tree is inspired by P-Tree [CLGS04]. It enhances the P-Tree structure by adding the MBR information to the node routing tables. Similarly to P-Tree, every node maintains a local view of the *left-most root-to-leaf* path of its local tree. Lookups and range queries are performed similarly to P-Tree. A range query over the interval $I = [L, M]$ reaches the lower bound L and then sequentially traverses node links until the upper bound is reached. This solution inherits all the drawbacks of P-Tree presented above.

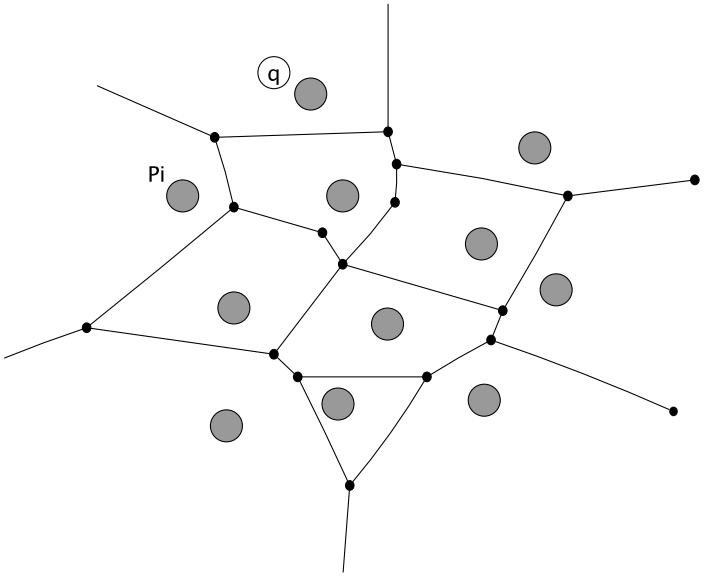


Figure 2.27: Voronoi diagram respresentation

2.5.3 Voroni-Based Solutions

Voronoi-based solutions [BKMR07, FRA+16] apply Voronoi diagrams [OBSC09] to a distributed setting in order to support range queries.

A Voronoi diagram of a set of distinct points $P = \{p_1, p_2, ..., p_n\}$ is the subdivision of the plane into n distincts cells or regions $R = \{R_{p_1}, R_{p_2}, ..., R_{p_n}\}$. A given point q lies in the cell corresponding to the point $p_i \in P$ if $Distance(q, p_i) < Distance(q, p_j)$ $\forall p_i \in P$ $i \neq j$. $Distance(q, p_i)$ is the euclidean distance between points q and p_i. A *Voronoi vertex* is defined as the centre of an empty circle that passes over 3 or more points. A *Voronoi edge* is the boundary between two cells. Figure 2.27 presents a representation of a Voronoi diagram composed of 11 points. The number of vertices and edges have linear complexity $O(n)$ with respect to the number of points n.

VoroNet. VoroNet [BKMR07] is a scalable object network based on the idea of Voronoi diagrams. VoroNet does not rely on the use of hash functions to provide load balancing or to generate the space of identifiers. Instead, every physical node is responsible only for the objects it has published in the overlay. It is different from previous approaches in the sense that physical nodes represent application objects instead of virtual identifiers. It supports searches on large collections of data.

Objects are organised in an attribute space according to a Voronoi diagram. The overlay space is a *d-dimensional torus* similar to the one proposed in CAN [RFH+01]. The main difference is that CAN organises node identifiers instead of object identifiers. Thus, attributes in a 2-dimensional space are organised in the domain $[0, 1] \times [0, 1]$. Objects with similar characteristics are neighbours in the object to object network, providing natural support for range queries.

Every object maintains a *routing table* composed of the IP address of the node that stores the object and its coordinates in the *d-torus*. This table comprises three types of nodes: (i) *Voronoi Neighbours* are objects whose region shares a common Voronoi *edge*; (ii) *Long range neighbours* are remote nodes used for routing purposes; and (iii) *Close neighbours* are a small set of nodes within a very short distance of the object, used to ensure routing properties. These links represent the local view every object has of the system.

The routing algorithm uses a greedy and fully deterministic strategy that routes a message in $O(\log^2(|O|_{target}))$ messages, where $|O|_{target}$ is an application defined parameter that represents the number of objects for which the system is optimised.

The main advantage of VoroNet is that nodes store $O(1)$ links instead of $O(\log(N))$ as in most solutions. However, its main drawback is that induces higher latency routing compared to existing solutions that provide a routing interface in $O(\log(N))$ routing hops. The number of nodes N is expected to be lower than the number of expected objects $|O|_{target}$.

Other solutions such as SWAM-V [BKS04] and VoRaQue [ARBB08] follow a similar approach. The main drawback of these solutions is the *curse of dimensionality* [Sam06] due to the high average number of neighbours of a peer when the number of dimensions increases [BKMR07].

Hivory. Hierarchical Voronoi Range Query (Hivory) [FRA+16] extends VoroNet [BKMR07] in order to overcome the *curse of dimensionality*. Its structure is a hierarchical Voronoi overlay that exploits object clustering in order to define a hierarchy of Voronoi Tessellations. This hierarchy follows a *tree* structure of recursive levels of Voronoi cells. Thus, every cluster C at tree level l that exceeds a threshold of T objects defines a new and different Voronoi Tessellation at the next level $l+1$. The main drawback of this clustering strategy is that it introduces a high number of connections between neighbour objects that reduce the advantages of Voronoi Tessellations. In order to overcome this issue, Hivory introduces a *super-peer* (SP) election mechanism that reduces the number of nodes paired to the same cell. This mechanism selects $\lfloor \log(|C|) \rfloor$ nodes, for every Voronoi cell: these act as forwarder nodes between different levels of the hierarchy. Their range query processing procedure follows a top down strategy. The query is forwarded to the next level of the hierarchy until all objects that match the query predicate are reached. Hivory inherits the advantages and drawbacks of Voronet [BKMR07].

2.6 Discussion

This section presents a discussion about the categories of solutions presented in the previous sections. These solutions are compared based on the following criteria.

- **Partition Strategy.** The index partition strategy (i.e, static or dynamic).
- **Dimensions.** The number of dimensions the solution targets.
- **Data Structure.** The data structure the architecture implements.

Variable	Description
N	Number of DHT nodes
D	Number of bits per key
z	Number often nodes that resolve a range query
B	Maximum number of data items stored per node
l	number of tree levels
p, r	LBM number of replicas (r) and partitions (p)
C	Voronoi cluster size

Table 2.1: Variables and description

- **Message Traffic Cost.** The message cost of insertions and range queries.
- **Split Bandwidth Cost.** The cost of the split index maintenance operation.
- **Load Balancing.** How the system reaches load balancing of insertions and range queries.
- **Scalability.** The ability of the system to add users and resources without loss of performance or increase in administrative complexity. This property depends on several factors such as load balancing and message cost.

Tables 2.2, 2.3, and 2.4 present a summary of every solution with respect to the criteria presented above.

2.6.1 Over-DHT solutions

Most of these solutions choose to implement a *prefix tree* data structure instead of a tree. Prefix-based partitioning differs from key-based partitioning used in trees in the way it partitions the domain: it divides every coordinate domain into two parts independently of the input data set. Therefore the shape of the tree that results from prefix partitioning follows the data distribution. Conversely, key partitioning divides the space relatively to the input data set in order to create a more balanced index structure.

Choosing a prefix partition strategy over a DHT has several advantages over key partitioning. First, it creates a global implicit knowledge about the location of keys on the prefix tree; this improves load balancing for insertions and range queries. Since a key is stored on a *leaf node* whose label is prefix to a given key, a client node can directly access any node in the index structure without necessarily going through the root node. Conversely, an index structure which uses key partitioning lacks this knowledge and therefore must route all insertions and queries via the root node. This creates a potential bottleneck which is unacceptable for applications that generate data at a high input rate concurrently with range queries.

The implicit knowledge also improves fault tolerance. The failure of an internal node only has little impact on the other nodes, whereas a failure on an index using a tree impacts all the subtree.

PHT [RRHS04] presents the best properties based on the criteria defined above. Compared to LIGHT [TZX10], PHT generates a lower amount of message traffic for performing range queries. Compared to DRING [HRA+11] it provides a lower latency for range queries. Compared to ECHO [HASB16], PHT exhibits similar perfomance via binary search, but without the cost of using a distributed cache. Finally, RST [GS04] provides low latency range query support, but at a high cost in terms of index maintenance and message traffic due to data insertions.

Therefore, PHT provides the best solution for range queries at large scale inheriting the properties of DHTs. However, the deployment of this solution in real world systems is limited due to the following issues.

- **Query latency**. Insertions and range queries are built upon the binary lookup algorithm that provides the best trade off between latency and message traffic. However, this algorithm is inherently sequential and provides a high lookup latency. For instance, if the latency among two nodes is 100 ms, then a PHT of $D = 96$ bits built on top of a DHT of $N = 10^6$ nodes could reach a lookup latency of around 3 seconds. Thus, the latency of range queries could be even worse because, in some particular cases, they rely on the lookup algorithm as a starting point.

- **Message traffic**.Most of these class of solutions generate a high message traffic. Reducing both message traffic and latency is a difficult task and is still an unresolved issue in these class of solutions. For instance, DRING [HRA+11] reduces the message traffic of the parallel range query algorithm presented in PHT [RRHS04]. However, it increases latency because it introduces a sequential search.

2.6.2 DHT-dependent solutions

DHT-dependent solutions aim to adapt the DHT structure by means of order preserving functions. MAAN [CFCS04] and Mercury [BAS04] provide sequential range query support through *locality preserving hash functions*. However, they sacrifice load balancing and induce high range query latency. Squid [SP08] relies on SFCs [Sam06] and adds a hierarchical structure that uses the DHT lookup interface in order to provide parallel range query support. However, it still incurs poor load balancing and a high message traffic due to the use of the DHT lookup interface to forward queries instead of direct links between nodes. Saturn [PNT12] tackles the load balancing problem by using a multi ring architecture, where every ring has a rotated $nodeId$ space. However, it introduces a high maintenance cost with its multiple ring architecture. The main drawbacks of these solutions can be summarised as follows.

- **Query latency**. Most of these solutions use sequential range search which induces a high latency. Squid [SP08] aims to resolve this issue by creating a logical tree to efficiently prune the space. However, it introduces a high cost in terms of message traffic.

- **Load balancing**. The use of order preserving functions works well in terms of load balancing when the input distribution is uniform. However, this does not hold for

skewed distributions. Saturn [PNT12] aims to resolve this issue with a multi ring architecture, but it introduces a high maintenance cost.

2.6.3 Non-DHT solutions

Non-DHT solutions do not rely on the use of DHTs in order to create data locality. Instead they use data structures such as Skip Lists [Pug90], Voronoi Tesselations [OBSC09], and Trees.

Skip-List based solutions provide a lookup interface that can reach a data item in $O(log(N))$ routing hops. They can be used to index one-dimensional data such as Skip-Graph [AS07] and they also can be extended to index multi dimensional data using SFCs [Sam06] such as ZNet [SOTZ05]. The main drawback of this class of solutions is that Skip Graphs assume that each node stores one key: This is unrealistic and introduces a very high search latency and message cost. Furthermore, they exhibit a poor load balancing under skewed distributions.

Tree-based solutions follow both *trees* and *prefix trees* based network topologies. For instance, VBI-Tree [JOV+06] builds a balanced binary tree of nodes and P-Grid [ACMD+03] follows a prefix tree overlay. The main advantage of these solutions with respect to over-DHT solutions is that they reduce the maintenance cost of using a DHT as a building block. However, the main drawback of most of these solutions their poor load balancing under skewed distributions as they organise nodes independently of the data distribution. Furthermore, they incur a high latency for range queries.

Voronoi-Based solutions rely on the use of Voronoi Tessellations [OBSC09] in order to create data locality. VoroNet [BKMR07] was the first proposal to apply this data structure in a P2P environment. Then Hivory [FRA+16] extends this idea to a multi dimensional space. The main advantage of these solutions is a low maintenance cost: only $O(1)$ neighbour links instead of $O(\log(N))$. However, they present a higher lookup message cost and latency $O(\log^2(|O|_{target}))$, where $|O|_{target}$ is the number of objects in the system.

2.7 Summary

This chapter presents current solutions which build data locality on top of P2P networks in order to support range queries. Most of the current solutions do not target location-temporal range queries. However, they can be extended by using *linearisation* techniques such as *Space Filling Curves* [Sam06]. The analysis of such solutions focused on the desired properties of LBSNs such as scalability, load balance, low latency, and low message traffic. However, none of these solutions provide all these properties. DHT-dependent solutions and non-DHT solutions exhibit poor load balance under skewed distributions and high latency for range queries.

Over-DHT solutions such as PHT [RRHS04] provide scalability and good load balancing properties. However, it induces high latency for insertions and range queries over internet connected nodes. Insertions present high latency due to the inherent sequential

routing of binary search. Other solutions such as DST [ZSLS06] reduces message traffic and latency, but it provides poor load balancing. RST [GS04] aims to improve the load balancing, but it introduces a high index maintenance cost. Dring [HRA+11] reduces the message traffic of PHT [RRHS04], but it introduces a high range query latency due to sequential range search. ECHO [HASB16] proposes a distributed cache solution to decrease latency and message traffic in PHT. However, it presents a similar message traffic and latency to the binary search introduced in PHT. This is because it performs a linear search that highly relies on the DHT lookup interface.

Therefore, achieving scalability, load balance, low latency, and low message traffic is a difficult task. It remains an unresolved issue when building a large scale system for location-temporal range queries.

	PHT	**LIGHT**	**DRING**	**ECHO**	**RST**
Partition Strategy	Dynamic	Dynamic	Dynamic	Dynamic	Static
Dimensions	Multi	Multi	Multi	Multi	Multi
Data Structure	Trie	Trie	Trie	Trie	Tree
Lookup Message Cost	$O(\log(D) \times \log(N))$	$O(\log(D/2) \times \log(N))$	$O(\log(D) \times \log(N))$	$O(D \times \log(N))$	$O(l \times \log(N))$
Range Query Message Cost	$O(\log(N) + z)$	$O(z \times \log(N))$	$O(\log(N) + z)$	$O(\log(N) + z)$	$O(p \times r \times z \times O(\log(N))$
Split Bandwith Cost	$O(B)$	$O(B)$	$O(B)$	$O(B)$	$O(p \times r \times B)$
Load Balancing	Good	Good	Good	Good	Good
Scalability	Good	Good	Fair	Good	Good

Table 2.2: Comparision of over-DHT solutions

	MAAN	**Mercury**	**Squid**	**Saturn**
Partition Strategy	Static	Static	Static	Static
Dimensions	Multi	Multi	Multi	Multi
Data Structure	Locality Hash	Locality Hash	SFCs	Locality Hash
Lookup Message Cost	$O(log(N))$	$O(d \times \log(N))$	$O(\log(N))$	$O(\log(N))$
Range Query Message Cost	$O(\log(N) + z)$	$O(\log(N) + z)$	$O(m \times \log(N))$	$O(\log(N) + z)$
Split Bandwith Cost	None	None	None	None
Load Balancing	Poor	Poor	Fair	Fair
Scalability	Poor	Poor	Fair	Fair

Table 2.3: Comparision of DHT-dependent solutions

	SkipGraph	**ZNet**	**BATON**	**P-Grid**	**P-Tree**	**Hivory**
Partition Strategy	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic
Dimensions	Single	Multi	Single	Single	Single	Multi
Data Structure	Skip List	SkipGraph/SFCs	AVL-Tree	Trie	B-Tree	Tree/Voronoi
Lookup #Message Cost	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$	$O(\log^2(\lvert O\rvert_{target}))$
Range Query #Message Cost	$O(\log(N)+z)$	$O(\log(N)+\log(z))$	$O(\log(N)+z)$	$O(\log(N)+z)$	$O(\log(N)+z)$	$O(\log^2(\lvert O\rvert_{target})+z)$
Split Bandwith Cost	None	$O(B)$	$O(B)$	$O(B)$	$O(B)$	$O(\lvert C\rvert)$
Load Balancing	Poor	Poor	Fair	Fair	Fair	Fair
Scalability	Fair	Fair	Fair	Good	Good	Poor

Table 2.4: Comparison of main non-DHT solutions

Chapter 3

Big-LHT: An index for n-recent geolocalized queries

Contents

3.1 Introduction

Finding the most recent data uploaded inside geographic zones of interest, is a useful type of location-temporal range query for Location Based Social Networks. For instance, people usually share comments and ratings about places on social networks. They are interested in reviewing the latest information about places which are inside a geographic

area. We refer to this type of location-temporal range query as *n-recent location-temporal zone query*.

Depending on the application scenario, a geographic zone can be as small as a (latitude,longitude) coordinate or as big as required. For instance, for some applications, geographic zones could represent Points of Interest (POIs) such as Restaurants or Train Stations. A POI is located by a single latitude and longitude coordinate. On the other hand, other applications could want to represent bigger zones in order to have a more global view of the data.

One of the main challenges of building such service is to provide an architecture which scales with the load of concurrent insertions and queries. Unfortunately, current solutions reviewed in chapter 2, are not adapted for this kind of query due to the following issues.

- They do not provide direct access to the most recent uploaded data.
- They do not target applications that aim to perform data partitioning by geographic zones of interest.

A naive solution could be to implement a large scale index based only on location. This approach has the downside that all the data generated inside the target location range must be filtered and sorted by upload time in order to find the latest uploaded items. It induces an extra processing overhead that impacts negatively the query response time.

This chapter presents Big Location Hash Tree (Big-LHT), a scalable architecture built on top of a fully scalable and decentralised overlay such as Pastry [RD01] or Chord [SMK+01] because DHTs provide a strong basis for scalable solutions as opposed to centralised server architectures.

Big-LHT implements a novel data structure that provides a primary index based on location and a secondary index based on upload time. The primary index partitions the data into geographic zones, whose size depends on the application. The minimum granularity of a query is at *zone level*. This means that queries which target a smaller area than a single zone will cover all the zone. The secondary index organises the data by upload time for fast retrieval of the latest uploaded data.

A theoretical evaluation shows that the protocols provided by Big-LHT scale with the number of DHT nodes N. Then, a practical evaluation assesses the impact of choosing a geographic zone size on load balancing.

This chapter is organised as follows. Section 3.2 introduces Big-LHT, its design, and main operations. Section 3.3 presents a theoretical and practical assessment of this solution in terms of scalability and load balance. Then, Section 3.4 presents a discussion of the main advantages and limitations of Big-LHT. Finally, we conclude in Section 3.5.

3.2 Design

This section details the type of query that Big-LHT targets and its internal data structure and protocols.

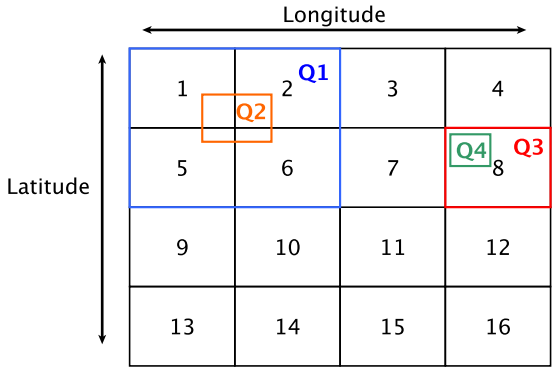


Figure 3.1: *n-recent location-temporal zone query* example

3.2.1 Query definition

A *n-recent location-temporal zone query* can be formulated as follows. Given a partition of the latitude and longitude domain into geographic zones, retrieve the latest n elements, *per geographic zone*, covered by the following location interval.

$$\begin{array}{rcll} \Delta Lat & = & [lat1, lat_2], & lat_1, lat_2 \in [-90, 90] \\ \Delta Lon & = & [lon_1, lon_2], & lon_1, lon_2 \in [-180, 180] \end{array} \tag{3.1}$$

ΔLat and ΔLon represent the latitude and longitude range respectively.

Figure 3.1 presents an example of different queries. In this example, the whole earth is divided into 16 geographic zones. The query *Q1* asks for the n most recent data items, per zone, generated in zones $\{1, 2, 5, 6\}$. It receives at most $4n$ data items (at most n data items per zone). *Q2* is equivalent to $Q1$ due to the maximum granularity of a query is at zone level. *Q3* and *Q4* receive the same result, the most recent n data items generated in zone 8.

In order to support such queries at scale, it is required a distributed data structure able to partition the input data set into geographic zones of predefined size. The size of the zone depends on the applications needs. This data structure must provide direct access to the most recent data items. The next subsection presents the data structure Big-LHT implements in order to support *n-recent location-temporal zone queries*.

3.2.2 Indexing structure

The protocol that Big-LHT follows is organised in two steps: *mapping* and *indexing*. The *mapping* step uses insertion requests in order to dynamically map every (latitude, longitude) tuple to geographic zones represented as a p-bit word Z_p. These zones are managed by DHT nodes called *zone managers*. The value of p is an application parameter which represents the size of a single geographic zone.

The *indexing* step dynamically organises zone managers as part of a trie-like distributed data structure similar to PHT [RRHS04], called Location Hash Tree (LHT).

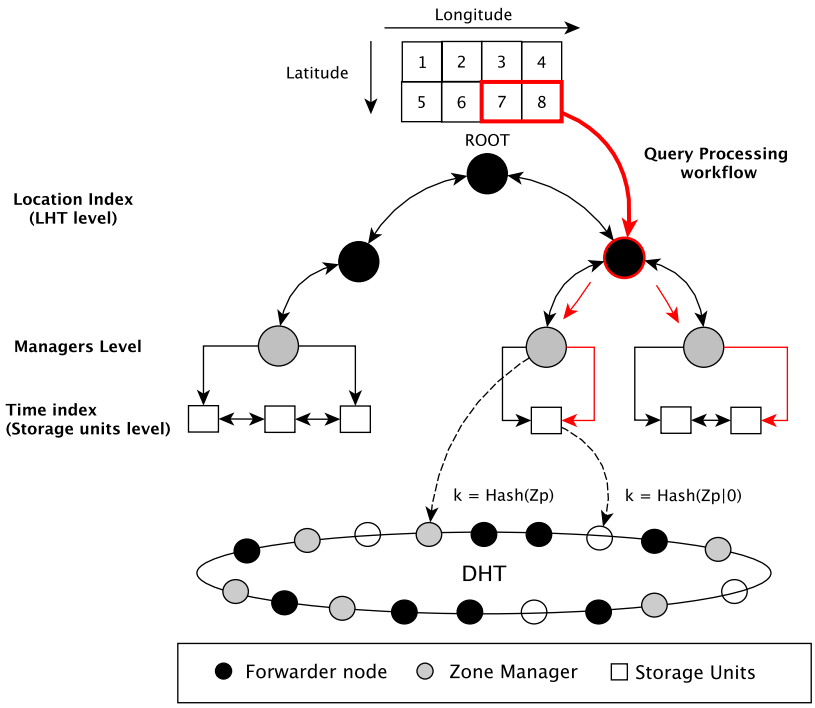


Figure 3.2: Architectural overview of Big-LHT

LHT is composed of DHT nodes called *forwarder nodes* which filter the location space in order to locate all zone managers inside a given location range.

In order to provide direct access to the most recent uploaded data items, every manager maintains a distributed double linked list, which allows to scale the storage to multiple DHT nodes. The double linked list is composed of nodes called *storage units* that store data sorted by upload time in order to provide direct access to the most recent data items, similarly to a temporal log.

Figure 3.2 presents an example of the overall architecture of Big-LHT and its protocol as detailed in the next sections. In this example, a query represented in red asks for the latest n objects generated in geographic zones 7 and 8. This query is routed to a *forwarder* node that filters the location space and forwards the query to the target zone managers. Then, zone managers send the query to the last storage units which contains the most recent data items. Finally, the query traverses every double linked list until the latest n elements uploaded per geographic zone are reached.

Figure 3.3 depicts the distributed storage data structure. 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.

A storage unit maintains a state which can be either *frozen* or *live* and references to its immediate predecessor and successor nodes. Only live nodes store new incoming data. Similarly to managers nodes, storage units maintain a label $l = Z_p|i$, where i represents

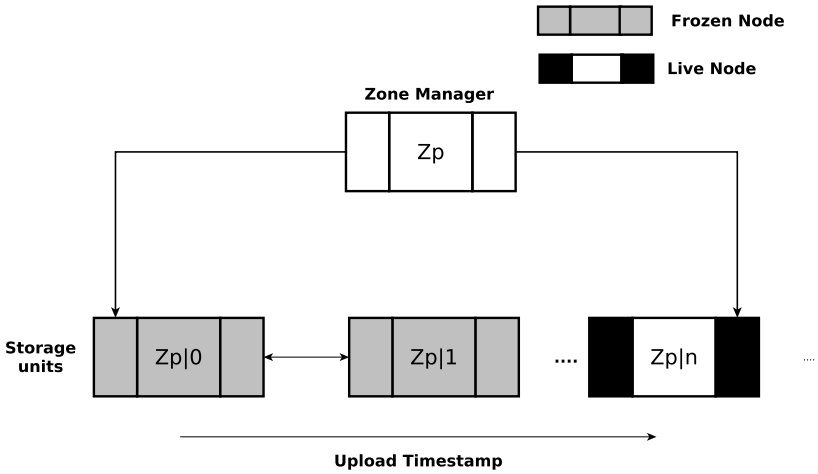


Figure 3.3: Management of storage units.

the identifier of the storage unit ($i \geq 0$).

In the next, our approach assumes that all DHT nodes support the Network Time Protocol, and thus maintain their local time within a small deviation from the Coordinated Universal Time (UTC).

3.2.3 Mapping

This step partitions and distributes every data item by latitude and longitude coordinates to a DHT node that manages a geographic zone Z_p. The actual value of Z_p is a $p-bit$ word which is computed from the latitude and longitude coordinates. The latitude coordinate contributes with $\lfloor p/2 \rfloor$ bits and the longitude contributes with $\lceil p/2 \rceil$ bits, p is an application parameter that represents the size of every zone.

The identifier of every geographic zone Z_p is computed using a prefix based partition strategy which is performed as follows. The domain $[a, b]$ of every latitude and longitude 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 sub-domain. This process continues recursively until $p/2$ bits are set. For the latitude coordinate, the domain is $[-90, 90]$ and for the longitude is $[-180, 180]$. For instance, the first bit of latitude -45 is 0 because it belongs to the first left sub-domain [-90,0]. This mapping function reaches its lower bound at latitude value $lat = -90$ represented as the identifier $\{0\}^{\lfloor p/2 \rfloor}$, and its upper bound at $lat = 90$ represented as the identifier $\{1\}^{\lceil p/2 \rceil}$. Longitudes incur the same process.

Algorithm 1 presents a pseudocode of this procedure. It encodes a given coordinate value (latitude or longitude) into a binary string of size max given as parameter [1]. First, it initialises the domain bounds according to the type of coordinate (lines 8 to 13). Then, this domain is partitioned into halves max times (lines 14 to 21). If the input value is on

[1] $max = \lfloor p/2 \rfloor$ for latitude + $\lceil p/2 \rceil$ for longitude

```
1  Input:
2  value // coordinate value
3  type // type of coordinate (latitude or longitude)
4  max // Number of bits used to partition the value domain
5  Output:
6  bString: // Binary string of length p
7  Body :
8  if type is latitude then
9      lowerBound = −90
10     upperBound = 90
11 else
12     lowerBound = −180
13     upperBound = 180
14 while bString.length() ≤ max do
15     middleBound = (lowerBound + upperBound)/2
16     if value ≤ middleBound then
17         bString.append(0)
18         upperBound = middleBound
19     else
20         bString.append(1)
21         lowerBound = middleBound
22 return bString
```

Algorithm 1: *partition* procedure.

```
1  Input:
2  lat, lon // latitude and longitude values
3  p // Number of bits used to represent Z_p (length of Z_p)
4  Output:
5  Z_p // Binary string of length p
6  Body :
7  latBin = partition(lat, latitude, ⌈p/2⌉)
8  lonBin = partition(lon, longitude, ⌊p/2⌋)
9  /* z-order interleaves the bits of latBin and lonBin.          */
10 Z_p = z-order(latBin, lonBin)
11 return Z_p
```

Algorithm 2: *getManagerId* procedure.

the left half (lines 16 to 18), a bit with value 0 is appended to the result, otherwise the bit is set to 1. This procedure finishes when all the *max* bits have been set.

Figure 3.4 presents an example of the mapping strategy for latitude value 24.550558. This partitioning scheme produces two bit words of size $\lfloor p/2 \rfloor$ and $\lceil p/2 \rceil$ that represent the longitude and latitude coordinates respectively.

In order to create the identifier of a given zone Z_p of length p, all two bit words are merged using the *z-ordering* [Sam06] *Space-Filling Curve* (SFC). This SFC interleaves

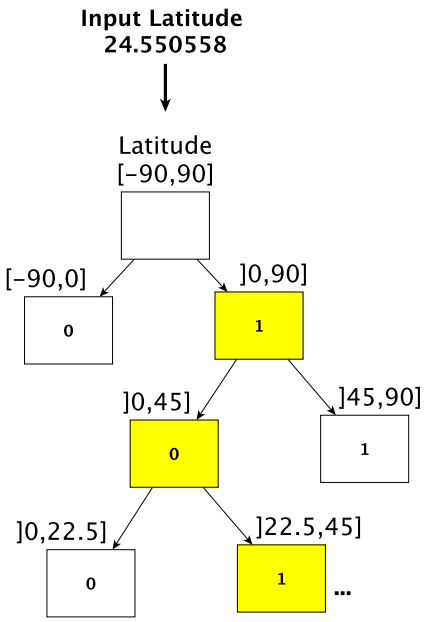


Figure 3.4: Mapping example.

the bits of every coordinate. For instance, if $latitude = 000$, and $longitude = 111$ then $z(longitude, latitude) = 10\ 10\ 10$. Algorithm 2 presents a pseudocode of this procedure. It uses the partition pseudocode presented above in order to generate two binary strings (lines 7 to 8). Then, it uses *z-ordering* in order to generate a p-bit word, which represents a zone manager identifier Z_p (line 10). The result of this procedure is a data set partition into at most 2^p geographic zones.

These p-bit words used to represent Z_p present two properties.

1. **Recursive prefix domain partition**. The mapping function presented above recursively partitions every latitude and longitude coordinates into two areas represented by prefixes. For instance, the geographic zone $Z_p = 00$ represents all the latitude and longitude tuples within the interval latitude $([-90, 0])$, longitude $([-180, 0])$. The next geographic zone $Z_p = 0000$ represents the domain latitude $([-90, -45])$, longitude $([-180, -90])$, and so on.

2. **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) belong to the interval covered by this prefix.

3.2.4 LHT maintenance

This section details how manager nodes dynamically join the LHT structure. In order to perform this task, every manager node exploits the recursive definition of its identifier

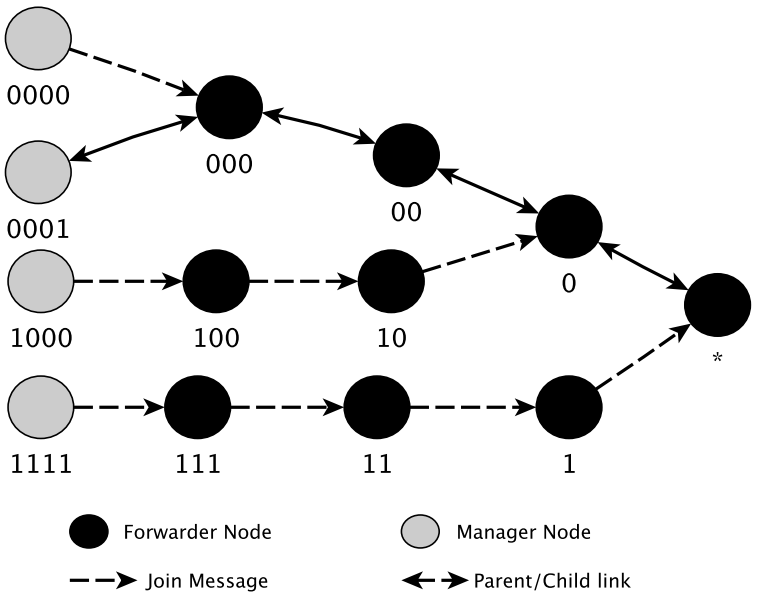


Figure 3.5: Example of an index with LHT

Z_p and joins LHT as follows. A new manager M with label Z_p, deletes the last bit from its label (l_{-1}), and routes a *JOIN* message to the DHT node whose nodeId is closest to $K = SHA(l_{-1})$. This message contains the IP and label l of the joining node. A DHT node which receives this message is called *forwarder* of the joining node.

Upon reception of a *JOIN* message there are two possible cases:

1. The node that receives this message belongs to LHT. In this case, the forwarder node adds the joining node as its child and drops the JOIN message.

2. The DHT node that receives this message does not belong to LHT. In this case, the new node sets itself as forwarder node of M, adds it as a child node, stores l_{-1} as its label, and recursively repeats the process forwarding the JOIN message to the DHT node whose nodeId is closest to the next key ($K = SHA(l_{-2})$). This process continues recursively until either the JOIN message reaches an existing forwarder node or the root node represented as $l = *$.

Every *forwarder* maintains a *children table* which contains two entries: the *label* Z_p of every child and its direct IP address. Upon the reception of a JOIN message, a *forwarder node* sends back an ACK message to the joining node with its own address in order to provide bidirectional links among LHT nodes.

Figure 3.5 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 case, the manager node routes the *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

```
1  Input:
2  join // Join request
3  Z_p // manager's label of length p
4  Body :
5  mylabel = Z_p
6  /* myLabel_-1 deletes the last character (bit) from myLabel                */
7  targetLabel = myLabel_-1
8  if isEmpty(targetLabel) then
9      /* send the request to the root node of LHT with label l=*             */
10     route(<join,myLabel,myIP>,SHA(*))
11 else
12     route(<join,myLabel,myIP>,SHA(targetLabel))
```

Algorithm 3: LHT join request (manager node side).

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 1111 joins LHT, its insertion generates $p = 4$ recursive join messages until the root node labeled $l = *$ is reached.

Algorithm 3 presents a pseudocode of this procedure from the manager node side. This node creates a *JOIN* request message, then deletes the last bit to its label ($myLabel_{-1}$), and routes a *JOIN* message, using the DHT lookup interface, to the DHT node whose nodeId is closest to the key $k = SHA(myLabel_{-1})$ (lines 8 to 12).

Algorithm 4 presents the procedure that performs a forwarder node in order to process this request. It receives the *JOIN* request, adds the new child node into its children table, and sends back one ACK to the joining node (lines 6 to 11). Finally, if this node is a new forwarder node, it recursively forwards up the message in order to join itself to the LHT structure (lines 12 to 17).

LHT follows a *prefix tree* (trie) indexing structure similar to PHT [RRHS04]. Both solutions follow a prefix tree strategy to index data, but they are different in two key aspects:

1. LHT grows along a bottom-up data flow.

2. LHT defines a grid where every leaf node (manager) holds rectangular geographic zones, and introduces a time index with horizontal splits

3.2.5 Managers maintenance

Zone managers are dynamically assigned to DHT nodes using insertion requests as follows. An insertion request is routed to the DHT node whose *nodeId* is closest to the key $K = SHA(Z_p)$. The DHT node that receives this request, set itself as manager of Z_p storing a label $l = Z_p$ that identifies its assigned zone.

In order to distribute storage resources and to improve location-temporal data access, every manager maintains a distributed double linked list storage data structure, that

```
1  Input:
2  join // join request
3  join.label // Label of the joining node
4  join.IP // IP address of the joining node
5  Body :
6  addChild(join.label,join.IP)
7  if isEmpty(join.label_-1) then
8      myLabel = *
9  else
10     myLabel = join.label_-1
11 send(ACK, join.IP)
12 if node is not root AND node is not forwarder for join.label then
13     myState = forwarder
14     if isEmpty(join.label_-1) then
15         route(<join,myLabel,myIP>,SHA(*))
16     else
17         route(join,SHA(<join,myLabel,myIP>,SHA(myLabel_-1)))
```

Algorithm 4: LHT join message reception and processing (LHT side).

stores data sorted by upload time. This structure is similar to a temporal log because it appends data as it arrives. It is composed of DHT nodes that act as *storage units*.

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

The *split* index maintenance works as follows. First, the current *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 $l_{new} = (Z_p|i+1)$, where $i+1$ is the identifier of the new live storage unit. 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* index maintenance operation occurs when two consecutive *storage units* sum less than B data items. Every Δt, each storage units sends a *COUNT_DATA* message to its predecessor and successor node. This message contains the number of data items stored in the node. When the merge condition is reached, the node that has the lower number of data items transfers its data to its consecutive node. In order to update the linked list structure, the leaving node sends an ALERT_MERGE message to its own predecessor and successor. If this node is the current live node or the first storage unit, it also alerts the zone manager node.

```
1  // Task1: Client side: Compute the zone identifier Z_p and route an
       insertion request to Z_p
2  Input:
3  lat,lon // latitude and longitude values
4  p // length of Z_p
5  ins // Insertion request
6  Body :
7  /* Extracts a binary string of length p using z-ordering (see algorithm 2)
       */
8  Z_P = getManagerId(lat, lon, p)
9  route(<ins,Z_p>,SHA(Z_p))
10 // Task 2: Manager side: Process an incoming insertion request from a
       client node. This process possibly creates a new manager node. This
       is an asynchronous procedure
11 Input:
12 ins // Insertion request
13 Z_p zone identifier
14 Body :
15 if isManager(Z_p) then
16     /* The node is already manager of Z_p. It forwards the request to the
           live storage unit                                              */
17     sUnitLabel = getLiveLabel()
18     sUnitIP = getLiveIP()
19     send(<ins,sUnitLabel,insertion=true>,sUnitIP))
20 else
21     /* The node is not manager of Z_p. It must set up its state, to forward
           the request to first new storage unit                          */
22     set(state=manager,myLabel=Z_p)
23     sUnitLabel = Z_p|0
24     route(<ins,sUnitLabel,create=true>,K=SHA(sUnitLabel))
25     /* The new manager joins LHT, see algorithm 3                     */
26     join(Z_p)
```

Algorithm 5: Insertion request and processing.

3.2.6 Query processing

- **Data insertions and deletions**. Algorithm 5 presents the insertion procedure. The client node computes the idenfier Z_p of the zone that covers the latitude and longitude coordinate using the $getManagerId$ procedure presented in Algorithm 2. Then, it routes the insertion request using the DHT lookup interface (lines 8 to 9).

  An insertion request that arrives to an existing manager node labeled Z_p is forwarded to the current live storage unit (lines 15 to 19). On the other hand, an insertion request which triggers the creation of a new manager is routed to the DHT node whose nodeId is closest to the key $K = SHA(Z_p|0)$. This node sets itself as the first storage unit of the zone manager Z_p. Finally, a new zone manager node uses

```
1  Input:
2  query // Query request
3  n // Number of data items
4  lat1,lat2 // Latitude lower and upper bound
5  lon1,lon2 // Longitude lower and upper bound
6  Body :
7  /* Get the lower and upper bound of manager identifiers                    */
8  Z_{P_low} = getManagerId(lat1, lon1, p)
9  Z_{P_high} = getManagerId(lat2, lon2, p)
10 /* Extract the common prefix label between Z_{P_low} and Z_{P_high}        */
11 prefix = commonPrefix(Z_{P_low}, Z_{P_high})
12 route(<query,lat1,lat2,lon1,lon2,n,prefix>,SHA(prefix))
```

Algorithm 6: n-Recent location-temporal zone query processing (client side).

```
1  Input:
2  query // Query request
3  prefix // node label
4  n // Number of data items
5  lat1,lat2 // Latitude lower and upper bound
6  lon1,lon2 // Longitude lower and upper bound
7  Body :
8  if getState(prefix) is forwarder node then
9      for every children node in children table do
10         /* Checks if the location range of the query intersects the latitude
              and longitude bounds covered by the label of the children node */
11         if intersection(children.label,lat1,lat2,lon1,lon2) is not empty then
12             prefix = children.label
13             /* forward the request to the child node                       */
14             send(<query,prefix,n,lat1,lat2,lon1,lon2>,children.IP)
15 if getState(prefix) is manager node then
16     /* Forward the request to the live node                                */
17     send(<query,lat1,lat2,lon1,lon2,n>,getLiveId())
```

Algorithm 7: n-Recent location-temporal zone query message reception and processing (Big-LHT side).

the join procedure presented in algorithm 3 in order to join LHT (lines 20 to 26).

Deleting an object o successfully uploaded at a time t, works as follows. First, the client node routes the deletion request to the DHT node manager of the zone. Then, the request traverses the double linked list until the *storage unit* which stores o is reached.

- **n-Recent location-temporal zone queries**. This query retrieves the latest n uploaded elements per zone of interest covered by a given input location interval. The query processing algorithm combines parallel and sequential data access and

works as follows.

Algorithm 6 presents the procedure performed by the DHT node which sends the query. It computes the identifiers of lower and upper bound zone managers ($Z_{p_{low}}$ and $Z_{p_{high}}$), using the $getManagerId$ procedure provided in Algorithm 2 (lines 8 to 9). Then, it extracts the common prefix of these identifiers and routes the query to the DHT node whose $nodeId$ is closest to the key $k = SHA(prefix)$ (line 12).

Algorithm 7 presents the procedure performed in order to process a query message. Depending on the state of the node which receives the query, there are two cases.

1. The node is a *forwarder* node (lines 8 to 14). In this case, it recursively forwards the query onwards to its children nodes whose label covers the location range.
2. The node is a *manager* node (lines 15 to 17). In this case, the forwarder node forwards the request to the live storage unit. Then, the query traverses the double linked list, from the most recent items to the least recent ones, until n objects that accomplish the range are reached. For the sake of simplicity, we left out the traversal through the double linked list of storage units in the pseudocode.

At the end of this procedure, the client node receives at most n objects per zone manager.

3.3 Evaluation

This section presents theoretical and experimental assessments of Big-LHT both for data insertions and n-recent location-temporal zone query. Our theoretical evaluation measures the message complexity for every operation of Big-LHT. Our experimental evaluation uses the Yahoo! public dataset [TSF+16] which contains millions of geotagged multimedia files (photos and videos) to assess the impact of Big-LHT parameter settings on system performance.

3.3.1 Theoretical evaluation

Insertion and deletion cost. Let N be the number of nodes in the DHT. The insertion of a data item directly routes, through the DHT routing interface, to the manager of the zone Z_p the data belongs to. This manager directly forwards the query to the *live* storage unit. Equation 3.2 presents the message complexity of this operation.

$$C_{insertion}(N) = O(\log(N)) \tag{3.2}$$

Once a storage unit reaches its maximum storage capacity B, the next insertion request (i.e, the insertion at position $B + 1$) is used to dynamically create a new storage unit, which adds a cost of $O(\log(N))$ routing hops.

A delete operation is routed to the manager of the zone and goes through the double linked list until the target data item is reached. Let s be the number of storage units of a single manager node. Equation 3.3 presents the message complexity of a delete operation.

$$C_{deletion} = O(\log(N) + s) \tag{3.3}$$

Delete operations cost more than insertions. In practice, applications that generate massive geotagged data sets incur far less deletions than insertions.

Storage index maintenance cost. The *split* index maintenance operation performed by a storage unit consists in forwarding an insertion request, through the DHT routing interface, 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 3.4 gives the message complexity of a storage split index maintenance operation.

$$C_{storage-split} = O(\log(N)) \tag{3.4}$$

Let B be the maximum number of items stored on a single node. The *merge* index maintenance moves at most B data items between two consecutive storage units. It involves the emission of four direct ACK messages. Equation 3.5 gives the message complexity of a merge operation.

$$C_{storage-merge} = O(B) \tag{3.5}$$

LHT index maintenance cost. Let p be the size of the prefix used to define geographic zones; p is a constant parameter of the system. When a new *manager* joins LHT it sends a *JOIN* message which gets forwarded recursively, possibly as far as the root node. Equation 3.6 computes the index maintenance cost.

$$C_{LHT-index} = O(\log(N)) \tag{3.6}$$

n-Recent location-temporal zone query cost. Let m be the number of manager nodes which target a given location-temporal range query. Let s be the maximum number of storage units covered per manager node. Equation 3.7 computes the message complexity for a given location-temporal range query.

$$C_{range-query} = O(m \times s + \log(N) + 2^m) \tag{3.7}$$

3.3.2 Experimental evaluation

The main goal of the experimental evaluation is to assess the impact of the geographic zone size of Z_p, on load balancing and its impact on the system performance.

We implemented a prototype of our architecture on top of FreePastry, an open-source implementation of Pastry [RD01]. 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 .

Length of zone (p)	Zone area size	Generated manager nodes/ Maximum number of zones
5	4975.5 km × 5009.4 km	32/32
10	621.9 km × 1252.3 km	531/1024
15	155.4 km × 156.5 km	5,189/32,768
25	4.85 km × 4.89 km	127,167/33,554,432

Table 3.1: Impact of p on zone coverage of geolocated data.

Every experiment indexes 1,000,000 geotagged multimedia files (photos and videos) from the Yahoo! Creative Commons dataset (YFCC100m)[TSF+16] in a DHT comprising $N = 100$ nodes. Every DHT node runs as a process on a single machine. Every storage unit has a capacity of $B = 1,000$ data items. That is, there is a maximum of $1,000$ storage units distributed on 100 DHT nodes.

Zone coverage. This experiment assesses the impact of the data distribution and the zone size p, in the number of generated manager nodes respect to the maximum number of zones required to cover the entire map. The value of p is an application parameter that represents the length of the p-bit word Z_p. A small value of p generates manager nodes that cover bigger areas than a larger value of p.

In order to conduct these experiments we indexed $1,000,000$ geotagged data items. Table 3.1 presents the results. It gives the area size and the number of zones generated for different values of p [2].

We noticed that when the value of p increases (i.e, the size of the zone covered by a manager node decreases), the number of generated managers respect to the maximum number of zones required to cover the entire map decreases fast. For instance, with $p = 10$ the number of generated manager nodes is 51.8% of the total number of zones (531 of 1024). Increasing the value of p to $p = 25$ decreases this value to only 0.3% of the total number of zones. This is a benefit of our approach towards spatial skewness of geotagged data: Big-LHT does not allocate *managers* for zones that contain no data.

Storage data distribution. This experiment analyses the distribution of data items on DHT nodes. It is measured as the the percentage of data items stored per DHT node respect to the total of items (1,000,000). We compare our results with the ideal case using the configuration when every node (out of $N = 100$) reaches exactly 1% of the total amount of data items. According to this system configuration, a single DHT node can maintain several storage units, and therefore it can store more than B data items.

Figure 3.6 presents the storage distribution for different values of p; s is the standard deviation for the number of data items on every node. Increasing the value of p from 5 to 25 only increases the standard deviation s from 0.006 to 0.007. These results suggest

[2]These values represent the biggest zone sizes at the equator where one degree of latitude is 110.567 km and one degree of longitude is 111.320 km

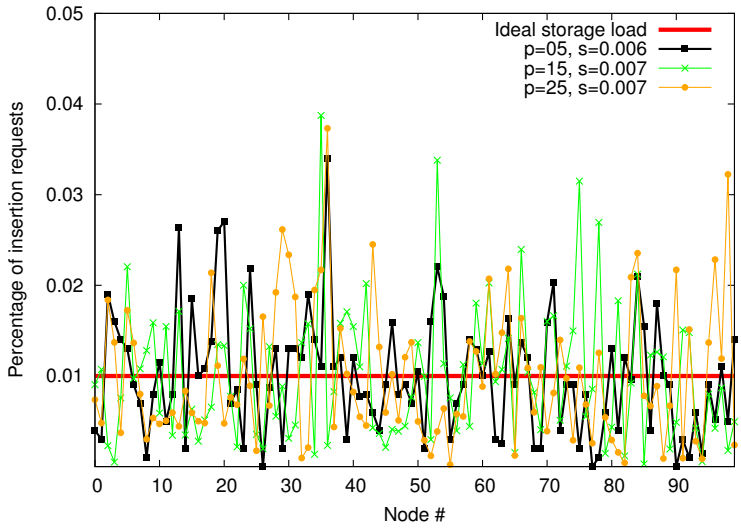


Figure 3.6: Storage data distribution

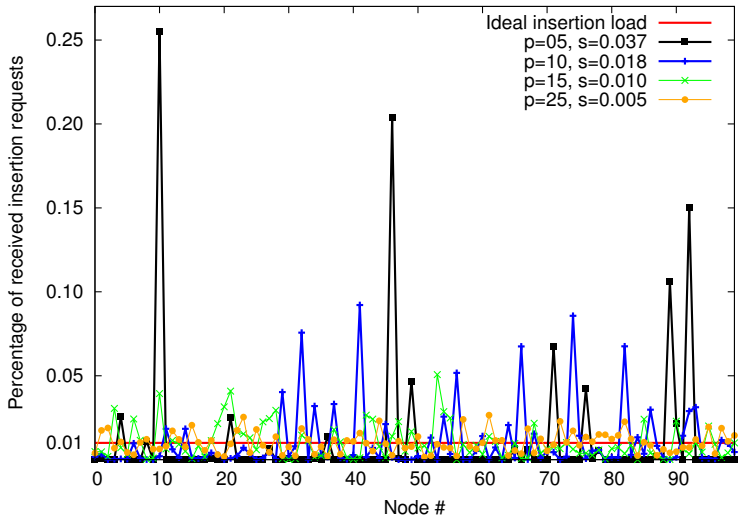


Figure 3.7: Insertion load distribution on *managers* nodes

that p bears little impact on the data distribution. This is because *storage units* are distributed uniformly among DHT nodes.

Insertion load distribution. This experiment assesses the impact of the zone size on the insertion load distribution, measured as the percentage of insertions requests per *manager* node. Figure 3.7 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.

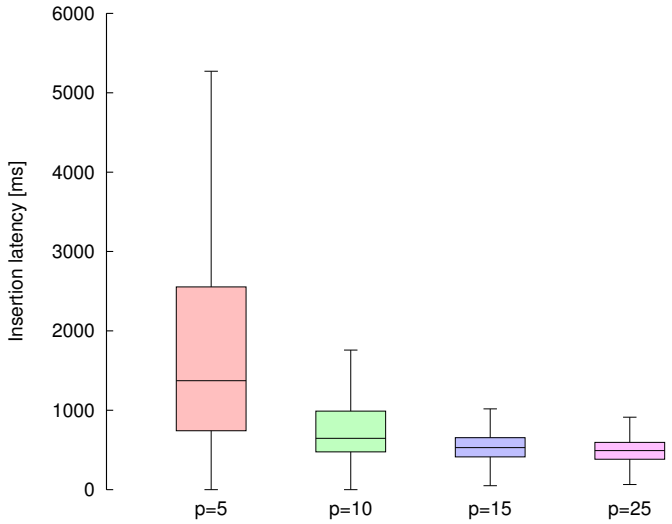


Figure 3.8: Impact of Z_p on insertion latency

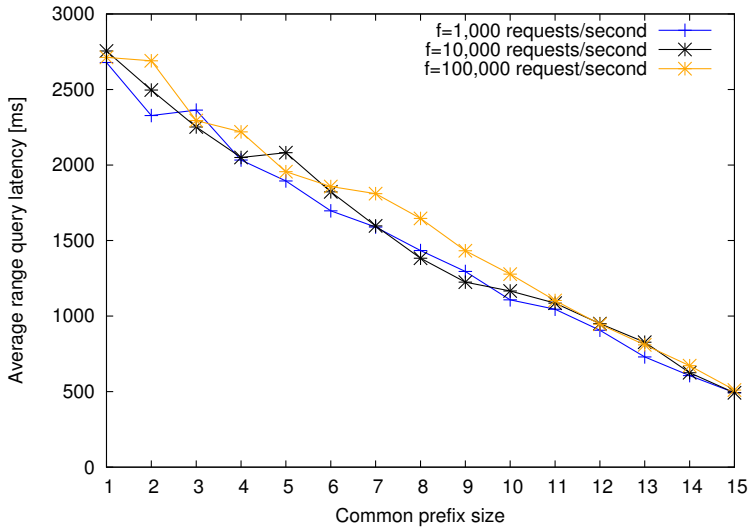


Figure 3.9: Average range query latency for $p = 15$

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%.

Impact of load on insertion latency. This experiment stresses the system with a high insertion load in order to evaluate the impact of choosing a given zone size p on insertion latency. As all the DHT nodes run on a single machine, we are interested in measuring the differences on insertion latency instead of its absolute value. The insertion latency is measured as the time elapsed between the emission of a request and the reception of an insertion ACK from the responding *live storage unit*.

In order to achieve this task, we generated a workload composed of $1,000,000$ insertions of geotagged items extracted from the YFCC100m [TSF+16] dataset at a rate of $100,000$ insertions per second uniformly distributed among DHT nodes ($1,000$ insertions/second per DHT node).

Figure 3.8 shows how insertion latencies are affected by zone sizes. 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 $p = 25$ drastically reduces the insertion latency to a maximum value of about 1 second with a median of about 500 milliseconds. This is because a higher value of p improves load balance (i.e, insertions are distributed among a higher number of DHT nodes).

Impact of location range on query latency. The goal of this experiment is to assess the impact of the location range size on range query latency. The metric used is the time elapsed between the client node sends the request and the time the query is received on the current *live* storage unit.

Every query asks for the *live* storage unit of a given location range: it reads an input (latitude, longitude) tuple 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 location space For instance, a value $cp = 00$ generates a query which covers half of the location domain (i.e, $\Delta Lat = [-90, 0]$, $\Delta Lon = [-180, -90]$). It enters the tree at a high level and then goes down in parallel until it reaches all *live* storage units. On the other side, a value $cp = Z_p$ generates a range query which asks for data inside a single zone manager.

In this evaluation we set the manager prefix size to $p = 15$, which produces the best trade-off between insertion latency and storage data distribution according to our previous results, and index $1,000,000$ geotagged items extracted from the YFCC100m [TSF+16] dataset.

We then measure the average range query latency with different input query workloads: from $f = 1,000$ range queries per second to $f = 100,000$ queries per second.

Figure 3.9 presents the average range query latency of Big-LHT in this experiment. Queries that cover the lower values of cp present the higher latency. This is because they arrive to the lower levels of LHT and they must traverse a higher number of nodes in order to reach the current live storage unit.

The average range query latency evolves linearly with respect to the common prefix size cp, this is because the parallel sweeps down the tree. This is an advantage of the parallel part of the range query algorithm implemented on Big-LHT.

In terms of input load, for this system configuration, Big-LHT exhibits a good resistance to the input query load and the query distribution. A variation between $1,000$ and $100,000$ requests per seconds shows a small impact in the average query latency.

3.4 Discussion

This section presents a discussion about the main results obtained during the evaluation of Big-LHT.

3.4.1 Advantages

- **Low index maintenance cost**. One of the main advantages of Big-LHT is its low index maintenance in terms of number of messages and latency in both storage units and the LHT indexing structure. This is because this solution relies on a static data partition defined by the application parameter p. The main consequence of this partition strategy is that insertions are performed using $O(\log(N))$ messages.

  At a storage unit level, the *split* operation only requires $O(\log(N))$ messages in order to create a new storage unit node. This is an improvement compared to most solutions that rely on dynamic data partitioning such as PHT [RRHS04]. These solutions present a high index maintenance cost in terms of number of messages and latency under skew distributions such as location-temporal data due to the high frequency of *split* operations in the tree structure. The *merge* index maintenance operation is more expensive. It involves moving at most B data items. However, applications that generate location-temporal data perform far more insertions than deletions.

  At the index structure level (LHT) there is an index maintenance cost only when a manager node joins the indexing structure. The upper bound of this cost is p DHT routing messages and happens only when the first manager node joins LHT. As the value of p is a constant system parameter, this process generates $O(\log(N))$ messages.

- **Storage data distribution**. In terms of storage data distribution, Big-LHT provides a good load balancing that slightly depends on the zone size Z_p. The evaluation shows that no node stores more than 4% of the total number of items for different values of p. It also shows that this upper bound decreases as the zone size decreases. This is because the linked list structure relies on the use of a hash function (SHA-1) that provides uniform distribution of storage units on DHT nodes.

- **Scalability and high availability**.

  Most of the operations provided in Big-LHT scale with the number of DHT nodes as they are performed in $O(\log(N))$ messages. Thus, Big-LHT scales with the total number of DHT nodes N and it does not depends on the zone size.

In a distributed environment such as a DHT, nodes can freely join or leave the network, or fail due to hardware problems. Big-LHT can use replication at the DHT level in order to increase the availability of indexed objects. Replicas are usually stored on numerically close nodes in the DHT to ensure that the overlay can resist to network partitions (in the leafset for Pastry or in the list of successors for Chord). Managing replicas induces a higher cost for the maintenance operations. On the other hand, it provides high availability.

3.4.2 Limitations

- **Query Load balance**.

  The load balance of Big-LHT depends on the application scenario. Applications that need to define smaller geographic zones such as POIs present a better insertion and query load distribution than applications that need to define bigger geographic zones. This is because smaller zones distribute the input load among a higher number of DHT nodes compared to bigger zones. One way to mitigate this issue is to replicate both storage units and manager nodes using the replication service provided by most of DHT implementations. For instance, the *leaf set* of Pastry can be used to maintain replicas of managers and storage units. Due to the properties of the routing algorithm, all the requests necessarily pass through one node in the *leaf set* before to arrive to the target node. These nodes can be used in order to improve load balance.

  Another limitation of applications that need to define big geographic zones is query granularity. Big-LHT supports queries at zone level. This means that allowing queries that target a smaller space than a single zone can have a negative impact in terms of both query latency and processing overhead.

3.5 Conclusion

This chapter presents Big Location Hash Tree (Big-LHT), a scalable architecture built on top of a DHT which indexes and retrieves the most the latest n elements per geographic zone uploaded inside a location interval. Big-LHT implements a novel data structure that provides a primary index based on location and a secondary index based on upload time. The primary index partitions the data into geographic zones which size depends of the application. The secondary index organises the data by upload time for fast retrieval of the latest uploaded data.

A theoretical evaluation shows that the protocols provided by Big-LHT scale with the number of DHT nodes N. Then, a practical evaluation assess the impact of choosing a geographic zone size on load balancing. The main advantage of this solution is its low maintenance cost and low insertion cost. Big-LHT exhibits the best input load balance for applications that need to define small geographic zones such as POIs. However, big zones can have a negative impact on load balance. This issue can be mitigated using the replication layer provided by most of the DHT implementations.

Chapter 4

GeoTrie: An index for location-temporal range queries

Contents

4.1 Introduction

Allowing people to perform *location-temporal range queries* over massive geotagged datasets, can help them to share and explore data generated inside a given geographic area and time interval.

For instance, people who attended a concert at the Eiffel Tower in Paris 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 might want to acquire more feedback about the overall concert experience by analysing pictures, comments and tags.

The previous solution, Big-LHT, is optimised for supporting *n-recent location-temporal zone queries*. However, it reaches its limits for supporting *location-temporal range queries* due to the following issues.

- *Dynamism.* Big-LHT performs static data partitioning based on predefined geographic zones. This strategy works well for *n-recent location-temporal zone queries*. However, as it was discussed in chapter 3, it limits load balance.

- *Time dimension.* The Big-LHT data structure is optimised for applications that index data following a temporal order. However, Big-LHT is ill-suited for applications that need to index data by generation time (i.e., The time at which the data has been created).

Over-DHT solutions [RRHS04, CRR+05, TZX10, HRA+11, HASB16] can support multidimensional indexing using Space Filling Curves (SFCs) [Sam06]. They provide scalability and load balance. However, they do not target *location-temporal range queries* and they present high latency and high message traffic for range queries.

Therefore, a key challenge is to provide an index which supports scalable *location-temporal range query* processing with low latency and message traffic compared to current solutions.

This chapter presents GeoTrie, a highly scalable architecture that supports location-temporal range queries built on top of a DHT such as Pastry [RD01] or Chord [SMK+01]. 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.

A theoretical evaluation assesses the scalability of GeoTrie in terms of the message complexity for insertions and range queries. Then, an experimental evaluation studies the performance of data insertions and location-temporal range queries in terms of query latency, load balance, and message traffic.

We chose to conduct our experimental evaluation with a real dataset: the Yahoo Flickr Creative Commons (YFCC100m) dataset [TSF+16] released by Yahoo! It comprises 48,469,177 geotagged multimedia files (photos and videos). We study the impact of insertions and location-temporal range queries on latency, load balance and message traffic. These metrics has direct impact on the user experience with the application. We compared the performance of GeoTrie with its closest solution PHT [RRHS04].

This chapter is organised as follows. Section 4.2 introduces the design of GeoTrie and its main protocols and operations. Then, Section 4.3 presents a performance evaluation of GeoTrie. A discussion of the main results is provided in Section 4.4. Finally, Section 4.5 presents the conclusion of this chapter.

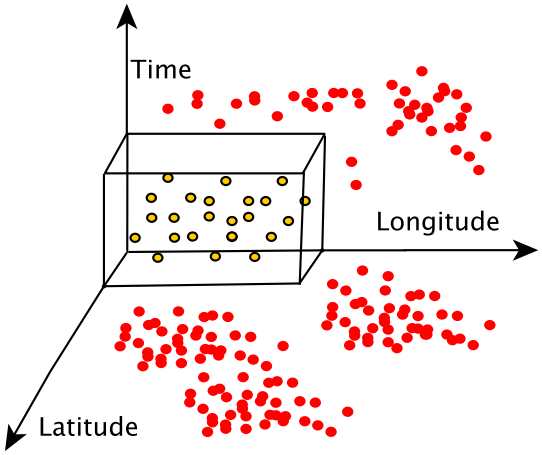


Figure 4.1: Location-temporal range query representation

4.2 Design

This section presents GeoTrie, a highly scalable architecture that supports location-temporal range queries.

4.2.1 Query definition

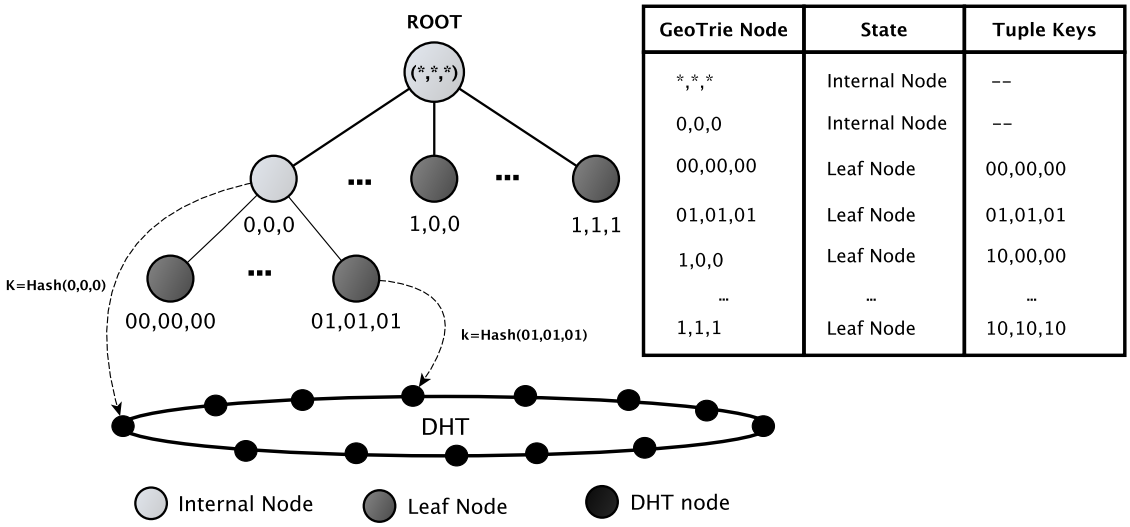
A location-temporal range query asks for all the objects inside of a three dimensional range as presented in equation 4.1.

$$\begin{array}{rcll} \Delta Lat & = & [lat_1, lat_2], & lat_1, lat_2 \in [-90, 90] \\ \Delta Lon & = & [lon_1, lon_2], & lon_1, lon_2 \in [-180, 180] \\ \Delta t & = & [t_i, t_f], & t_i, t_f \in [DateTimeStart, DateTimeEnd] \end{array} \tag{4.1}$$

ΔLat represents the *latitude* starting and ending point ($[lat1, lat_2]$), ΔLon represents the *longitude* starting and ending point ($[lon_1, lon_2]$), and Δt represents a time interval. This query can be represented as ilustrated in figure 4.1. Yellow points represent data items that fulfil the location-temporal query predicate, red points represent data outside of the interval.

4.2.2 Indexing structure

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 $D = 32$-bit word. The *indexing* step inserts every tuple key T_k in a fully distributed prefix octree which allows efficient location-temporal range queries.



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
1,0,0	Leaf Node	10,00,00
...	...	...
1,1,1	Leaf Node	10,10,10

Figure 4.2: GeoTrie Example: Distributed data structure for keys of D=2 bits size

The GeoTrie structure indexes every tuple key T_k into a distributed prefix octree 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. 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 state and dispatching its load onto eight new *leaf nodes* created via recursive prefix domain partitioning.

Figure 4.2 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 [RD01] Geotrie can use the *leaf set* in order to replicate and maintain the state of every Geotrie node.

4.2.3 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. Algorithm 8 presents a procedure to compute T_k. It partitions every latitude and longitude coordinate using the *partition* procedure intro-

1 **Input**:
2 lat,lon // latitude and longitude values
3 t // Unix Epoch timestamp $\in [0, 2^{32} - 1]$
4 **Output**:
5 $T_k = (T_{lat}, T_{lon}, T_t)$ // tuple key
6 **Body** :
7 /* To perform partition procedure (see algorithm 1) to encode every coordinate to a p-bit word */
8 T_{lat} = **partition**$(lat, latitude, 32)$
9 T_{lon} = **partition**$(lon, longitude, 32)$
10 T_t = binaryString(t,32)
11 $T_k = (T_{lat}, T_{lon}, T_t)$
12 return T_k

Algorithm 8: *getTupleKey* procedure

duced in algorithm 1 (lines 8 to 9). Then, it computes the a 32-bit word representation of the Unix Epoch and returns T_k (lines 10 to 12). Finally, it maps 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., a 32-bit binary word per coordinate).

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[1], and ensures a one second time precision. This procedure is similar to the mapping performed in Big-LHT. The main differences are that it includes the time dimension and does not rely on the use of *space filling curves* such as *z-ordering*. This mapping strategy inherits the following properties.

- **Recursive prefix domain partition**. The mapping function presented above recursively partitions every latitude, longitude, and timestamp domain 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.

4.2.4 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

[1]These values are computed considering that at the equator, one degree of latitude and longitude are 110.567 km and 111.320 km respectively.

```
1  // Task1: Client node sends a lookup request
2  Input:
3  T_k // Tuple key (T_lat, T_lon, T_t)
4  Output:
5  < label, IP > // identifiers of the target leaf node
6  Body :
7  lower = 0
8  higher = D
9  while lower ≤ higher do
10     middle = (lower + higher)/2
11     // Extract the prefix of size middle of every coordinate of T_k and
          route the message
12     targetLabel = T_middle(T_k)
13     route(<lookupReq,targetLabel>,SHA(targetLabel))
14     // Task2: Client node receives a reply from a lookup request
          (asynchronous process)
15     reply = receive()
16     if reply.state is a leaf node then
17        return < reply.getLabel(), reply.getIP() >
18     else
19        if reply.state is an internal node then
20           lower = middle + 1
21        else
22           // reply.state is external node
23           higher = middle − 1
```

Algorithm 9: *binaryLookup* procedure

leaf node scales out its load by creating eight new children leaf 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 the longest 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*.

The *merge* operation is the opposite of 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 B objects. When this happens the *internal parent 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.

```
1 Input:
2 T_k // tuple key
3 ins // insertion request
4 Body :
5 < label, IP > = binaryLookup(T_k);
6 // route the message to the target leaf node
7 route(ins,SHA(label),hint=IP);
```

Algorithm 10: Insertion procedure

4.2.5 Query Processing

This section presents the algorithms GeoTrie provides in order to support data insertions/deletions and location-temporal range queries.

Lookup interface. Given a tuple key $T_k = (T_{lat}, T_{lon}, T_t)$ the lookup interface provided by GeoTrie allows any node to locate the *leaf node* whose label is prefix of T_k. GeoTrie provides two algorithms in order to perform this task.

The lookup algorithm performs a *binary search* over different possible prefixes of T_k until a node with state *leaf node* is reached. This algorithm is similar to the binary search presented in PHT [RRHS04]. It relies on the DHT routing interface in order to route messages.

Algorithm 9 presents a pseudocode of this procedure. For every tuple key T_k, there are exactly D possible prefix labels plus the root node, and only one of them can designate a *leaf node*.

Initially a lookup message is routed to the DHT node whose identifier is closest to the result of applying a hash function to a label prefix of T_k of size $D/2$ (i.e., the middle of the space of possible candidates) (lines 10 to 13). 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 (lines 21 to 23). Instead, if the target node is an *internal node* it means that the *leaf node* keeps a longer prefix (lines 19 to 20). 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 (lines 16 to 17). This binary search procedure benefits from the prefix property of GeoTrie which allows queries to start at any node, and thus prevents the root node from becoming a bottleneck.

Insertions and deletions. Data insertions rely on the lookup interface presented above. Storing 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. Algorithm 10 presents a pseudocode of the insertion operation. Using the *binaryLookup* procedure, it retrieves the label and the IP address of the target *leaf node*. Then, it routes the message using the IP address as a hint (i.e., the message is sent directly if the target node is alive).

```
1  Input:
2  query // range query request
3  target // Node label
4  < lat_1, lat_2 > // latitude interval
5  < lon_1, lon_2 > // longitude interval
6  < t_1, t_2 > // time interval
7  Body :
8  if localState(target) is leaf node then
9      // process the request.  Send back to the client all the data items
          that covers the interval
10     send(<data,LocalFilter(lat_1, lat_2, lon_1, lon_2, t_1, t_2)>,query.clientIP)
11 else
12     if localState(target) is internal node then
13         for every children node do
14             /* Checks if the location-temporal range of the query intersects
                  the interval covered by the label of the children node      */
15             if intersection(children.label,lat_1,lat_2,lon_1,lon_2,t_1,t_2) is not empty then
16                 target = children.label;
17                 send(<query,target,lat_1, lat_2, lon_1, lon_2, t_1, t_2>,children.IP);
```

Algorithm 11: Location-temporal range query

The delete operation is similar to the insertion procedure. It uses the lookup interface in order to find the leaf node which label is prefix of T_k. Then, it sends a delete request over this node.

Location-temporal range query processing. A location-temporal range query is resolved as follows. First, the sender node uses the *getTupleKey* procedure presented in Algorithm 8 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 4.2.

$$\begin{array}{rcrr} \Delta Lat_b & = & [lat_1^{32}, lat_2^{32}], & lat_1^{32}, lat_2^{32} \in \{0,1\}^{32} \\ \Delta Lon_b & = & [lon_1^{32}, lon_2^{32}], & lon_1^{32}, lon_2^{32} \in \{0,1\}^{32} \\ \Delta t_b & = & [t_1^{32}, t_2^{32}], & t_1^{32}, t_2^{32} \in \{0,1\}^{32} \end{array} \tag{4.2}$$

Then, it computes the greatest common prefix for every coordinate constraint and it forwards the query to the node which covers this label. For instance, a query Q which combines constraints $\Delta Lat_b = [00..., 01...]$, $\Delta Lon_b = [10..., 11...]$, and $\Delta t_b = [110..., 111...]$ has a greatest common prefix of every coordinate $(0, 1, 11)$. This common prefix label has a time coordinate that 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 the data inside the interval of Q. If there is no common prefix, the query must start from the root node labeled $l = (*, *, *)$. Algorithm 11 presents the procedure performed by a GeoTrie node which receives a range query request.

Depending on the state s of the node which receives the query, we identify three cases.

```
1  // Task1: Client node sends a location-temporal range query request
2  Input:
3  lookupReq // lookup request
4  T_k // Tuple key (T_lat, T_lon, T_t)
5  Body :
6  IP = null
7  targetLabel = *,*,*
8  if cache is not empty then
9      for i = T_k.length() to 0 do
10         /* extracts a T_k prefix of length i                    */
11         targetLabel = substring(T_k, 0, i)
12         if i == 0 then
13             /* root node label                                  */
14             targetLabel = *,*,*
15         if cache.get(targetLabel) is not null then
16             IP = cache.get(targetLabel)
17             break
18 /* if hint is not null, it uses the IP address in order to send directly
      the message                                                  */
19 if IP is not null then
20     send(<lookupReq,T_k,targetLabel,∅,∅>,IP)
21 else
22     route(<lookupReq,T_k,targetLabel,∅,∅>,SHA(targetLabel))
23 // Task2: Client node receives the ACK (asynchronous task)
24 reply = receive()
25 /* update cache entries following a cache replacement policy
      [LRU,FIFO,LFU] (see algorithm 14)                            */
26 updateCache(reply.addList,reply.delList,Replacement)
```

Algorithm 12: *cacheSearch* procedure

1. The node is a *leaf node* (lines 8 to 10). In this case, this node covers all the required location-temporal range and returns the objects that satisfy the query.

2. The node is an *internal node* (lines 12 to 17). In this case, it checks if the location-temporal range of the query intersects the interval covered by the label of the children node. Then, it forwards the request to these nodes.

4.2.6 Caching optimisation

This section presents an alternative lookup algorithm which reduces both the number of messages and lookup latency compared to the binary lookup algorithm presented above.

The cache based lookup algorithm uses a local cache in order to find the closest GeoTrie node to the target leaf nodes. Then, it performs a *linear search*, using the GeoTrie node

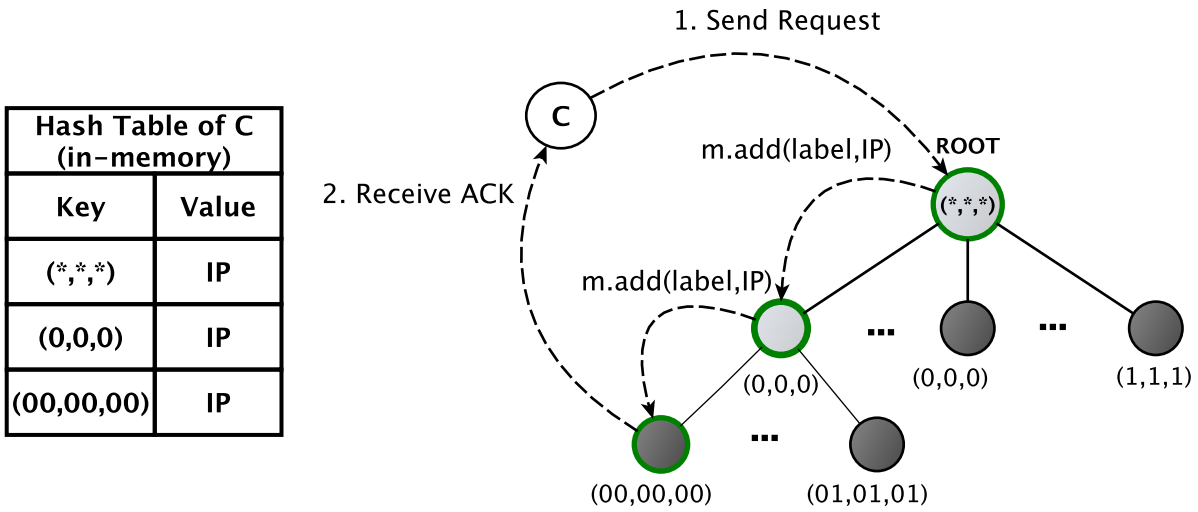


Figure 4.3: Cache based lookup example of the key $T_k = (00, 00, 00)$

links, until a *leaf node* whose label is prefix of T_k is reached. The data structure used in the cache is an in-memory hash table. It uses a GeoTrie node label as a *key* and an IP address as *value*.

Algorithm 12 presents the procedure from the client node side. First, the client node uses its cache in order to find an entry whose label is prefix of T_k (lines 8 to 16). After a *cache hit* (lines 15 to 17) the client node routes the lookup message using the IP address as a hint which tries first to send the message directly. If there is no such entry, the lookup message is routed to the root node (lines 19 to 22). Finally, it receives an ACK from the target leaf node (lines 24 to 26). This ACK contains the identifiers (label and IP) of the GeoTrie nodes that received the lookup message.

Algorithm 13 presents the processing of the lookup message. According to the state of the node which receives this message there are two cases.

1. The node is a GeoTrie node (internal or leaf) (lines 9 to 17). In this case, the GeoTrie node adds its data (label and IP) to a list called *add list* included in the lookup message. Then, it forwards the query down to the one node of its eight children nodes whose label is prefix to T_k. This process continues until the target leaf node replies back to the client.

2. The node is an external node (lines 19 to 24). In this case, the client node sends the lookup message to an external node that previously was a leaf node. This case happens when *merge* index maintenance operations have been carried out over leaf nodes that are stored in the cache.

   An external node which receives a lookup message adds itself to a *delete list* and forwards up the message to a node that has a shorter prefix until the target leaf node replies back to the client.

```
1  Input:
2  lookupReq // lookup request
3  T_k // Tuple key (T_lat, T_lon, T_t)
4  addList // list composed of <label,IP> tuples
5  delList // list composed of <label,IP> tuples
6  targetLabel // label of the target GeoTrie node
7  Body :
8  /* Target leaf node replies back to the client                    */
9  if localState(targetLabel) is leaf node then
10     addList.add(myLabel,myIP)
11     send(<ACK,addList,delList>,lookupReq.clientIP())
12 /* Lookup request arrives to an internal node, forward the request to the
      child node that shares the next prefix with T_k                */
13 if localState(targetLabel) is internal node then
14     addList.add(myLabel,myIP)
15     /* compute the next label                                      */
16     nextLabel = substring(T_k,0,targetLabel.length()+1)
17     send(<lookupReq,T_k,addList,delList,nextLabel>,getChildIp(nextLabel))
18 /* Lookup request arrives to an external node                     */
19 if state(targetLabel) is external node then
20     /* add the node information to the delete list                 */
21     delList.add(myLabel,myIP)
22     /* route the request one level up of GeoTrie                   */
23     nextLabel = substring(T_k,0,targetLabel.length()-1)
24     route(<lookupReq,T_k,addList,delList,nextLabel>,SHA(nextLabel))
```

Algorithm 13: Processing of a cache based lookup message (GeoTrie side)

Algorithm 14 presents the procedure that runs a node when a client node receives a reply ACK from a lookup operation (task 2 in algorithm 12). It first uses the deletion list (*delList*) in order to delete all the non up-to-date entries (lines 8 to 9). Then, it uses the add list (*addList*) in order to insert or update cache entries (lines 11 to 17). In order to limit the size of the cache, one entry replacement strategy such as First in First Out (FIFO), Least Frequently Used (LFU), and Least Recently Used (LRU) can be used (line 16).

Figure 4.3 presents an example of a lookup operation of the key $T_k = (00, 00, 00)$. As the cache is initially empty, the client node forwards the query to the root node labeled $l = (*, *, *)$. Upon its reception, the root node attaches its data to the message and forwards it to its child labeled $l = (0, 0, 0)$. The child node repeats this procedure until the leaf node labeled $l = (00, 00, 00)$ replies back to the client node. Thus, the client node fills its cache with the branch of GeoTrie whose label is prefix to $l = (00, 00, 00)$. Further searches can take advantage of this cache state. For instance, if the client node looks for the tuple key $T_k = (01, 01, 01)$, it will find the node labeled $l = (0, 0, 0)$ as a starting point of the query.

```
1  Input:
2  addList // List of tuples <label,IP> of nodes to add or update
3  delList // List of tuples <label,IP> of nodes to delete
4  Replacement // Cache replacement policy [FIFO,LRU,LFU]
5  Body :
6  /* delete entries in delList                                          */
7  for label in delList do
8      cache.delete(label);
9  /* add or update entries according to a replacement strategy          */
10 for <label,IP> in addList do
11     if cache.get(label) is null then
12         if cache is not full then
13             cache.put(label,IP);
14         else
15             cache.remove(replacement);
16             cache.put(label,IP);
```

Algorithm 14: *updateCache* procedure

4.3 Evaluation

This section presents a theoretical and an experimental assessment 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 performance of data insertions and location-temporal range queries in terms of latency, total message traffic, and load balance.

We compared the performance of GeoTrie with its closest solution PHT [RRHS04].

PHT can index multi dimensional data using Space Filling Curves (SFCs) (see section 2.3.1.1). We used *z-ordering* as SFC because it is less expensive to compute compared to *Hilbert curves* [Hil91] and it provides similar clustering properties using geo-tagged datasets [CRR+05]. We also implemented the binary lookup algorithm over PHT [RRHS04] because it presents the best tradeoff between message traffic and latency. Throughout this evaluation, we refer to this solution as PHT(binary). We implemented a client cache that stores leaf node references to improve the performance of PHT (binary). The node sends the request directly to the target leaf node upon a cache hit, otherwise it uses the binary lookup algorithm. We refer to this solution as PHT (cache) [CRR+05].

In GeoTrie, we implement both the *binary lookup* and the *cache based lookup* algorithm in order to support data insertions. We refer to these implementations as GeoTrie (binary) and GeoTrie (cache) respectively.

In order to perform location-temporal range queries we implemented the parallel range query algorithm in both GeoTrie and PHT. We refer to these implementations as PHT (parallel) and GeoTrie (parallel) respectively.

Figure 4.4 presents a diagram of the architecture the implementation follows. We implemented insertions and location-temporal range queries in both GeoTrie and PHT [RRHS04].

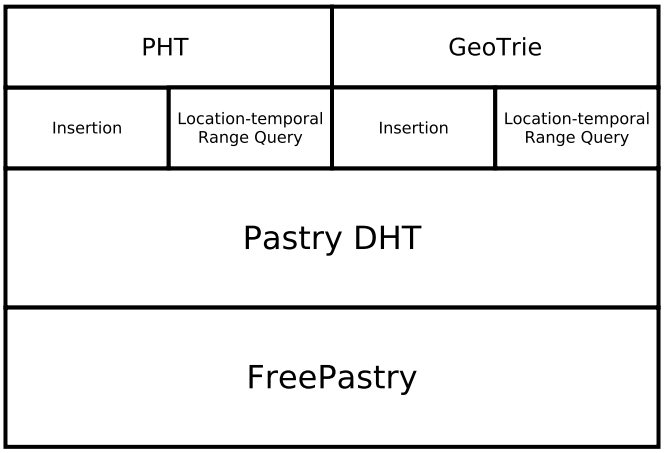


Figure 4.4: Implementation architecture

We chose to conduct our experimental evaluation with a real dataset: the Yahoo Flickr Creative Commons (YFCC100m) dataset [TSF+16] released by Yahoo! It comprises 48, 469, 177 geotagged multimedia files (photos and videos).

Solution	**Message Complexity**
PHT (binary)	$O(\log(D^*) \times \log(N))$
PHT (binary) + cache	$O(\log(D^*) \times \log(N))$
GeoTrie (binary)	$O(\log(D) \times \log(N))$
GeoTrie (cache)	$O(\log(N) + D)$

Table 4.1: Message complexity for insertions and deletions (worst case)

4.3.1 Theoretical evaluation

Insertion and deletion cost. Let D be the number of bits used by GeoTrie in order to represent every coordinate of the tuple key T_k. Let D^* the number of bits used to represent a three dimensional tuple key by PHT using *linearisation* [Sam06].

Table 4.1 presents the message complexity of insertions and deletions for both GeoTrie and PHT. Both operations rely on the lookup algorithm.

GeoTrie (binary) and PHT (binary) use the binary lookup algorithm. Both solutions spend a logarithmic number of DHT routing messages in order to reach a target leaf node.

The main difference is that PHT must use linearisation and therefore produces longer trie branches compared to GeoTrie (i.e., $D* > D$ for the same storage capacity B). As a consequence, PHT incurs a higher number of DHT lookups.

GeoTrie (cache) proposes a different algorithm in order to perform data insertions. It uses cached entries to start the query as close as possible to the target leaf node.

Furthermore, it uses the links among GeoTrie nodes to reduce the number of messages. It presents its worst case message complexity when the cache is empty. In this case, the client node must route a DHT message to the root node. The query traverses GeoTrie node links recursively until it reaches a leaf node. The worst case message complexity of this solution is $O(\log(N) + D)$.

PHT (binary) + cache aims to improve the performance of the binary lookup algorithm presented in PHT. Its worst case scenario occurs when there is a cache miss. In this case, it adds an extra cost of $O(\log(N))$ messages for each lookup request of the binary lookup algorithm. The performance of these two algorithms depends on both the state of the cache and the data distribution. We provide a performance evaluation using a real dataset in the next section.

Location-temporal range query cost. A location-temporal query first reaches the node which maintains the common prefix label of minimum common size, and then branches out in parallel down the octree until it reaches all *leaf nodes*. Equation 4.3 computes the number of messages for this operation in GeoTrie.

$$O(log(N) + 8^D) \tag{4.3}$$

Equation 4.4 presents the message complexity in PHT.

$$O(log(N) + 2^{D^*}) \tag{4.4}$$

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 and traverses the whole tree height in parallel. In terms of message complexity, GeoTrie (parallel) and PHT (parallel) incur the same message complexity.

Index maintenance cost. A *split* index maintenance operation distributes B data items to the new *leaf nodes*. A *merge* index maintenance operation moves up to $B - 1$ data items to a single new *leaf nodes*. Equation 4.5 presents the message complexity of these operations.

$$O(B) \tag{4.5}$$

PHT incurs the same index maintenance cost.

4.3.2 Experimental evaluation

Simulation environment and dataset. We implemented GeoTrie and PHT [RRHS04] on top of FreePastry [DEG+01], an open-source discrete event simulator for Pastry [RD01]. This simulator allows to set up a DHT on a single machine and to measure both latency (using fixed latencies among DHT nodes) and message traffic. However, it cannot simulate message loss and bandwidth [2].

[2] https://trac.freepastry.org/wiki/tut_simulator

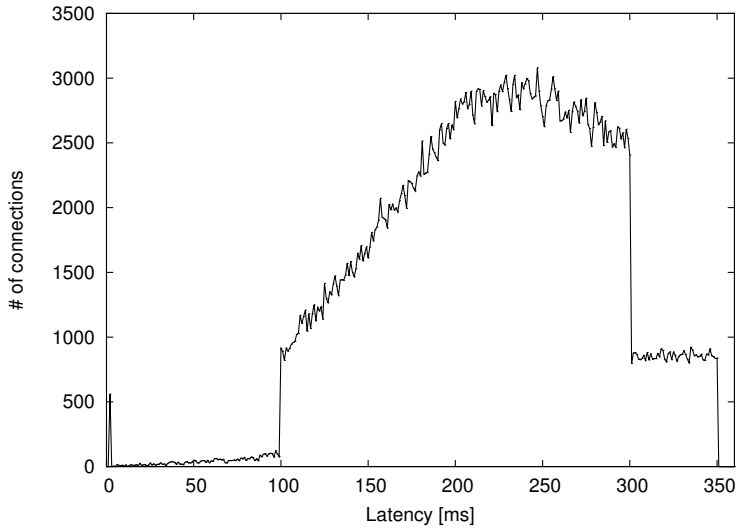


Figure 4.5: Distribution of latencies among DHT nodes

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

Table 4.2: Distribution of latencies among DHT nodes

We ran all experiments presented in this section on an Intel Xeon X5690 (12 cores, 24 threads) 3.47Ghz with 144GB RAM, debian kernel 3.0.0-1-amd64, and Java VM version 1.7.0_111.

In all of our experiments, we simulated $N = 1,000$ DHT nodes. We used a baseline latency file provided in the FreePastry distribution to generate latencies among DHT nodes. Figure 4.5 and Table 4.2 give the distribution of latencies among the DHT nodes during our experiments: most communications incur latencies between 178 and 269 [ms].

Table 4.3 presents the system parameters used in order to set up PHT and GeoTrie. That is, every leaf node stores at most $B = 10,000$ data items. Note that a single DHT node can manage more than one trie node.

Dataset and query distribution

In order to conduct this evaluation, we used the Yahoo Flickr Creative Commons 100 Million (YFCC100m) dataset [TSF+16]. This dataset contains $48,469,177$ geotagged data items (photos and videos). This dataset includes: username, latitude, longitude, timestamp, creation time, upload time, photo title, photo description, user tags, page

Variable	Value	Description
N	1,000	Number of DHT nodes
B	10,000	Number of items indexed per leaf node
D	$1 \leq D \leq 32$	Number of levels of GeoTrie
D^*	$1 \leq D* \leq 96$	Number of levels of PHT

Table 4.3: System parameters used in this evaluation

url, download url, and extension (photo or video). For further details about this dataset, please refer to [TSF+16].

In order to get insight about the data distribution of real world applications, we measured its distribution by latitude, longitude and time. Figure 4.6 presents this distribution: Figures 4.6a and 4.6b show that the input dataset is skewed spatially. It is mainly concentrated in the latitude range $[10, 70]$ and in longitude ranges $[-120, -70]$ and $[-20, 20]$, which corresponds to North America and Europe. Figure 4.6c presents the data distribution in terms of upload timestamp from 2004 to 2014: the rate of data generation follows cycles, and varies significantly over time.

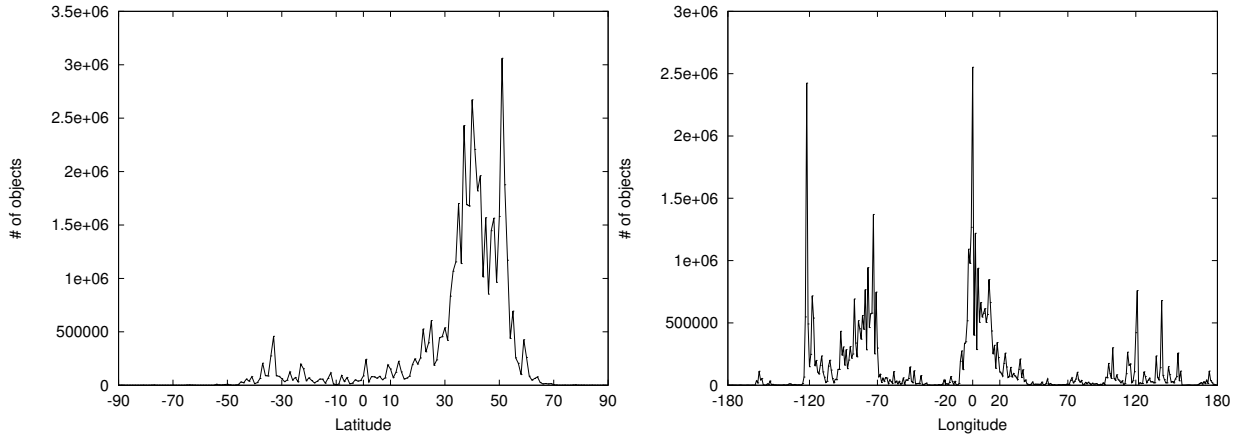
In order to assess the performance of GeoTrie under different range query spans, and due to the absence of query traces over this dataset, we generated six query sets (QS) which cover different location *bounding box* sizes and timespans as follows. First, we randomly picked $1,000$ t_k =(latitude,longitude,timestamp)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

Table 4.4 presents the size of M and H we chose for this evaluation and Table 4.5 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.

Metrics

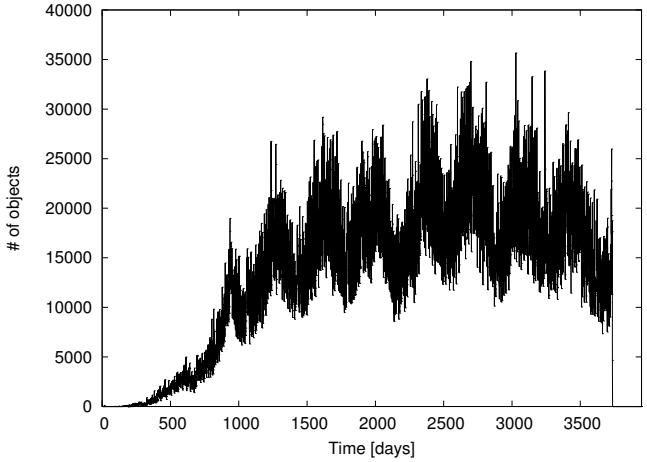
The metrics used in order to assess the performance of GeoTrie are the following.

- **Latency**. This is an important metrics from the perspective of the application client. It is the time elapsed between the issuance of a client request and the moment the client receives the last reply from a target node. On a large scale distributed



(a) Dataset distribution by latitude

(b) Dataset distribution by longitude



(c) Dataset distribution by time

Figure 4.6: Location-temporal input dataset distribution

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 4.4: Location-temporal range query sets

system such as a DHT, latencies among nodes vary from tens of milliseconds to several hundred milliseconds. Thus, an important fraction of the query response time is due to latency. We evaluated it for insertions and location-temporal range queries.

Distribution	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
Average	50	169	68	555	825	14,008

Table 4.5: Distribution of the number of data items that match every query set QS

- **Load balance**. This metric allows to measure the load distribution generated on DHT nodes for insertions and range queries. Our evaluation focuses on message traffic distribution and storage distribution.
- **Message traffic**. This metric allows to evaluate the global load of messages in the system.

Insertion Latency

In this experiment, we indexed all the 48,469,177 geotagged data items extracted from the YFCC100m dataset [TSF+16]. At every insertion, a DHT node is chosen randomly in order to perform the operation. Then, we measured the insertion latency as the time elapsed from the instant the client node starts the insertion process until it receives an acknowledge message of the insertion of the data item. This operation includes the lookup procedure, which returns the target *leaf node*, and a direct message exchange in order to perform the insertion operation.

Figure 4.7 and Table 4.6 present the overall distribution of insertion latencies. PHT (binary) exhibits the highest insertion latency. Most of the latencies are concentrated between 3.19 and 4.26 seconds, with an average of 3.65 seconds, and a maximum of 7.07 seconds.

GeoTrie (binary) slightly reduces the insertion latency relatively to PHT (binary). Most of the latencies are concentrated between 1.80 and 3.33 seconds. Its average insertion latency is 1.36 times faster than PHT (binary).

GeoTrie (cache) exhibits the lowest insertion latency. Compared to PHT (binary) + cache, it presents an average latency 2.47 times faster. Most of the latencies are concentrated between 0.58 and 0.86 seconds.

The insertion time depends of the size of the prefix label of the target *leaf node*. In the case of PHT (binary) and GeoTrie (binary), some insertions can directly arrive to the target *leaf node* through the binary lookup algorithm. Other insertions require more messages to locate the target *leaf node*. On the other hand, the protocol implemented in PHT (cache) is pessimistic. When the target *leaf node* is not in the cache, it falls back to binary lookup. The main advantage of GeoTrie (cache) compared to this algorithm is that it relies on the GeoTrie structure instead of a high use of the DHT lookup interface. This strategy allows to decrease both the number of messages and the insertion latency.

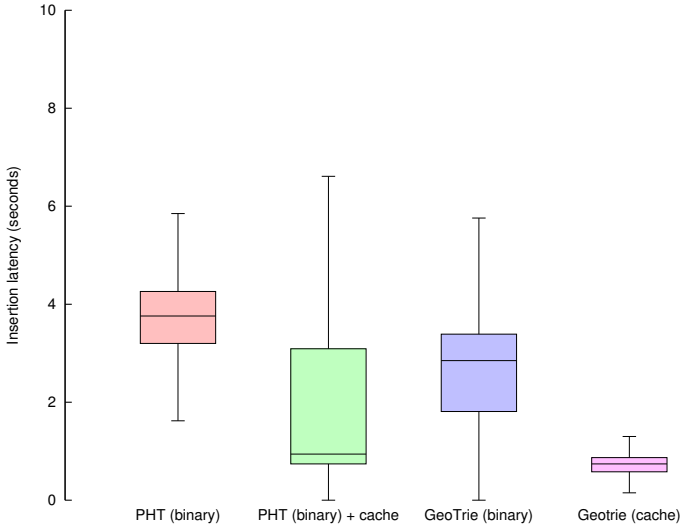


Figure 4.7: Insertion latency comparison

Distribution	**PHT (binary)**	**PHT (binary) + cache**	**GeoTrie (binary)**	**GeoTrie (cache)**
Quartile 1	3.19	0.74	1.80	0.58
Quartile 2	3.75	0.94	2.84	0.74
Quartile 3	4.26	3.09	3.33	0.86
Max	7.07	8.47	6.12	2.67
Average	3.65	1.78	2.67	0.72
Geotrie (Binary) average speedup	1.36			
Geotrie (Cache) average speedup	2.47			

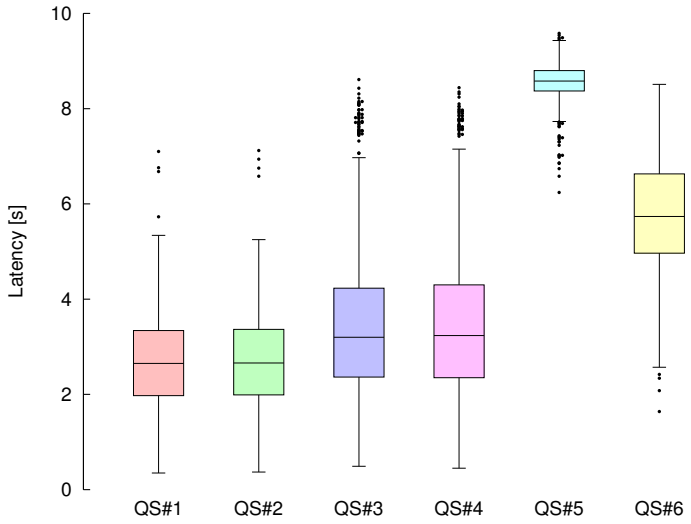
Table 4.6: Insertion latency distribution (seconds)

	PHT (Parallel)						**GeoTrie (Parallel)**					
	QS#1	**QS#2**	**QS#3**	**QS#4**	**QS#5**	**QS#6**	**QS#1**	**QS#2**	**QS#3**	**QS#4**	**QS#5**	**QS#6**
Min	0.35	0.37	0.49	0.45	6.24	1.64	0.21	0.09	0.01	0.15	2.47	1.34
Quartile 1	1.97	1.99	2.36	2.35	8.37	4.96	1.45	1.41	1.32	1.33	3.11	2.4
Quartile 2	2.65	2.66	3.20	3.23	8.58	5.73	1.76	1.77	1.62	1.64	3.24	2.65
Quartile 3	3.34	3.36	4.23	4.30	8.80	6.63	2.06	2.05	1.97	1.97	3.35	2.90
Max	7.10	7.12	8.61	10.4	12.3	13.0	3.11	3.07	3.76	3.99	4.52	4.67
Average	2.67	2.69	3.46	3.47	8.55	5.77	1.72	1.69	1.68	1.69	3.23	2.66

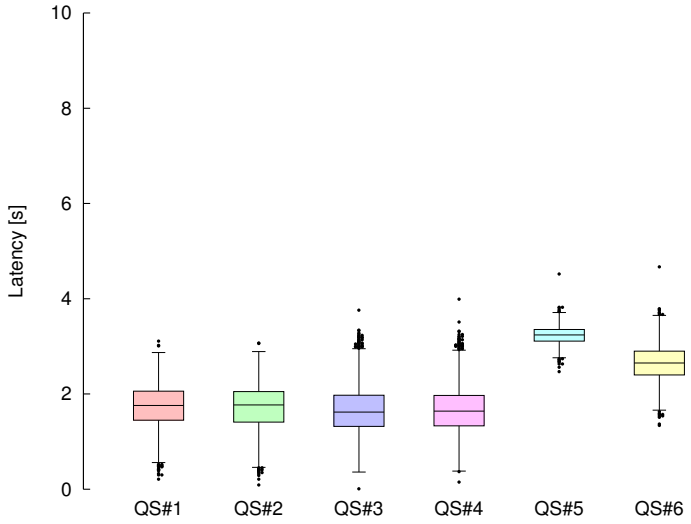
Table 4.7: Location-temporal range query latency distribution (seconds)

Location-temporal range query latency

This experiment compares the performance GeoTrie (parallel) and PHT (parallel) in terms of latency, under a workload composed of 6,000 queries distributed as presented in Table 4.4. Every query originates from a DHT node chosen at random. It consists in counting the number of items inside the location-temporal range. The query latency is computed as time elapsed from the instant the client node starts the query process until it receives the number of objects that are inside the location-temporal interval. This request does



(a) PHT (Parallel)



(b) GeoTrie(Parallel)

Figure 4.8: Location-temporal range query latency comparison

not take into account the time to transfer all the data items back to the client node.

Figure 4.8 and Table 4.7 present the query latency measured as the time elapsed between the emission of the query and the latest reception of results from all the *leaf nodes* that store data relevant to this query.

In GeoTrie, location-temporal range queries that span a shorter space present a lower latency than queries that span a larger space. For instance, GeoTrie presents an average latency that is 1.92 times lower for queries of *QS3* than for queries of *QS5* (1.68 and 3.23 seconds respectively). Indeed, we tailored queries in this experiment so that queries of *QS5* and *QS6* cannot avoid the root node. These queries represent an upper bound of latency because they always start by the root node. For instance, all the queries extracted

Solution	# Nodes (Internal/Leaf)	Total nodes
PHT	11,074/11,073	22,147
GeoTrie	3,630/25,411	29,041

Table 4.8: Number of nodes generated per solution

Solution	# Messages
PHT (binary)	1,238,811,455
PHT (binary) + cache	561,397,915
GeoTrie (binary)	881,449,302
Geotrie (cache)	109,638,622
GeoTrie (binary) traffic reduction (%)	28.8 %
PHT (binary) + cache	45.3%
Geotrie (cache) traffic reduction (%)	80.4%

Table 4.9: Total message traffic generated after the insertion of $48,469,177$ geotagged multimedia files

from *QS5* cover all the location domain which do not have a common prefix. In this case, the *greatest common prefix* is the root node. The same happens with all the queries of *QS6* that cover all the time domain.

For query sets *QS1* to *QS4*, we can observe a lower latency. This is because these queries, in most of the cases, start at a higher trie level and thus avoid the root node. It also allows to alleviate a potential bottleneck on the root node. This argument is also valid for PHT. This is because both solutions use a prefix tree domain partition strategy.

Compared to PHT (parallel), GeoTrie (parallel) gets a lower latency. For instance, GeoTrie (parallel) exhibits a query latency 1.55 times lower than PHT (parallel) for query set *QS1*. This difference increases when the search area increases. For instance, GeoTrie (parallel) presents a latency that is 2.05 times lower for *QS3* and 2.64 times lower for *QS5*.

In order to gain further insight about these results, we measured the number of *internal nodes* and *leaf nodes* in both GeoTrie and PHT. Table 4.8 presents the number of nodes in GeoTrie and PHT. For this dataset, the GeoTrie structure has 23.7% more nodes than PHT. In terms of type of node, PHT has 3.05 times more *internal nodes*. On the other hand, GeoTrie generates 2.29 times more *leaf nodes* than PHT. In conclusion, GeoTrie outperforms PHT in terms of range query latency because it presents a lower amount of internal nodes.

Insertion traffic

This experiment measures the message traffic generated after the insertion of $48,469,177$ data items extracted from the YFCC100m [TSF+16] dataset. It focuses on total message traffic and its distribution on DHT nodes. Every DHT node maintains a counter that

# DHT lookups	PHT (binary)(%)	PHT (binary) + cache(%)	GeoTrie (Binary)(%)
1	0.002	71.41	0.05
2	6.08	0	27.79
3	0.79	1.62	2.01
4	4.88	0.31	20.27
5	12.35	1.79	49.84
6	44.83	4.25	0.04
7	31.04	11.1	0
8	0	9.39	0

Table 4.10: Comparision of the Percentage of DHT lookups per insertion

measures the number of times it has received or forwarded a message related to an insertion operation. Table 4.9 summarises these results. PHT(binary) generates the highest amount of traffic with $1,238,811,455$ messages, whereas GeoTrie (binary) generates 28.8% less traffic.

Using a client cache drastically reduces the message traffic in both GeoTrie and PHT. PHT (binary) + cache reduces the message traffic generated in PHT (binary) by 45.3%. GeoTrie (cache) reduces the message traffic even more drastically: 80.4% less messages than PHT (binary) + cache.

In order to gain further insight about the message traffic cost, we measure and compare the number of DHT lookup messages generated in the insertion process (i.e., the number of times a node uses the DHT routing interface). Every DHT routing message has a cost of $O(\log(N))$ messages, and therefore the use of this interface by the lookup protocol has a direct impact on message traffic.

Table 4.10 presents the results. In these results we do not include GeoTrie + cache because its protocol relies mostly on the direct links between GeoTrie nodes than the use of the DHT routing interface. It uses the DHT routing interface not more than once, when there is a cache miss.

PHT (binary) generates either 6 or 7 DHT lookups (44.83% and 31.04% of the total number of lookups respectively). Very few insertions (0.002%) require only 1 DHT lookup. GeoTrie (binary) reduces the number of lookups required for data insertions. In most cases, it uses either 2 or 5 DHT lookups (49.84% and 27.79% of the total number of lookups respectively).

PHT (binary) + cache drastically reduces the number of DHT lookups relatively to PHT (binary). In this case, most insertions (71.41%) require only 1 DHT lookup. This is equivalent to the proportion of cache hits achieved by this solution. However, it performs a binary lookup if there is a cache miss or if the cache node is not a leaf node. In this case, a significant proportion of insertions produce either 7 or 8 DHT lookups (11.1% and 9.39% respectively).

Figure 4.9 presents the message traffic distribution on physical DHT nodes during data insertions. Table 4.11 presents the distribution of traffic per DHT node: the distribution of the number of messages handled per DHT node, and the percentage of the total traffic it represents.

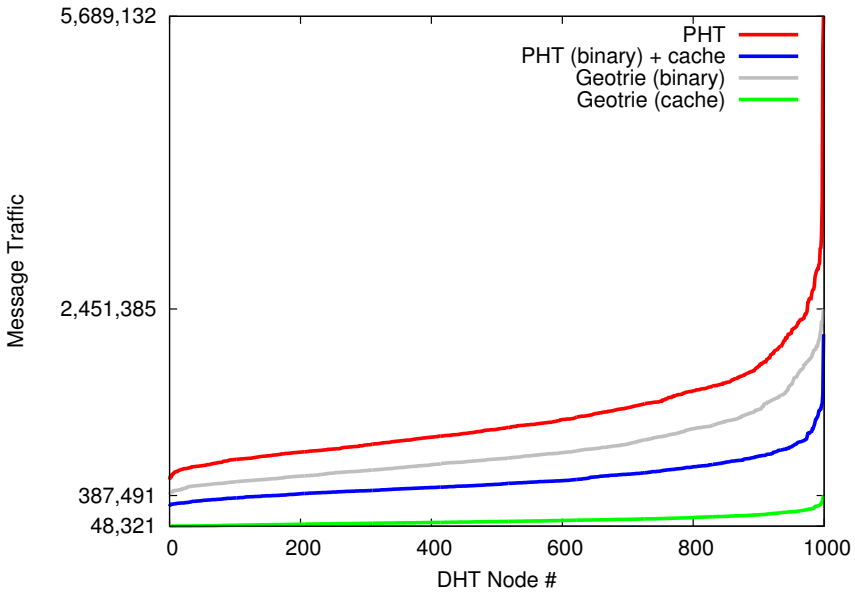


Figure 4.9: Message load distribution of insertion requests

Distribution	**PHT (binary)** Total/Percentage(%)	**PHT (binary) + cache** Total/Percentage(%)	**GeoTrie (Binary)** Total/Percentage(%)	**Geotrie (Cache)** Total/Percentage(%)
Min	562,689/0.04	265,516/0.04	394,559/0.04	48,321/0.04
Quartile 1	900,562/0.07	425,026/0.07	635,128/0.07	74,410/0.06
Quartile 2	1,118,710/0.09	510,624/0.09	791,298/0.08	99,232/0.09
Quartile 3	1,428,046/0.11	656,745/0.11	1,037,049/0.11	131,775/0.12
Max	5,689,132/0.45	2,172,621/0.38	2,451,385/0.27	387,491/0.35
Total	1,238,811,455/100	561,397,915/100	881,449,302/100	109,638,622/100

Table 4.11: Message traffic distribution of insertion requests per DHT node

GeoTrie and PHT incur similar load distributions. But the actual loads are significantly lower with GeoTrie: up to 5.12 times lower when comparing PHT (binary) + cache with GeoTrie (cache). Also, the maximum loads handled on a single node differ noticeably for each solution. The largest proportion of insertion message traffic load incurred by a single DHT node in PHT (binary) is 0.45% of the total traffic ($1,238,811,455$ messages) and 0.27% of the total traffic ($881,449,302$ messages) in GeoTrie (binary). In number of messages, the largest value handled by a single node of GeoTrie (binary) is actually half of that of PHT (binary).

Using a cache drastically reduces the number of messages, but it has a slightly impact in terms of the message traffic distribution: the most loaded node in PHT (binary) + cache incurs 0.38% of the total traffic ($561,397,915$ messages), against 0.35% of the total traffic ($109,638,622$ messages) for GeoTrie (cache). In conclusion, these results show that GeoTrie (cache) drastically reduces the message traffic on insertions with a minimal impact on message traffic load balance.

In order to measure the distribution of data items on physical DHT nodes and virtual

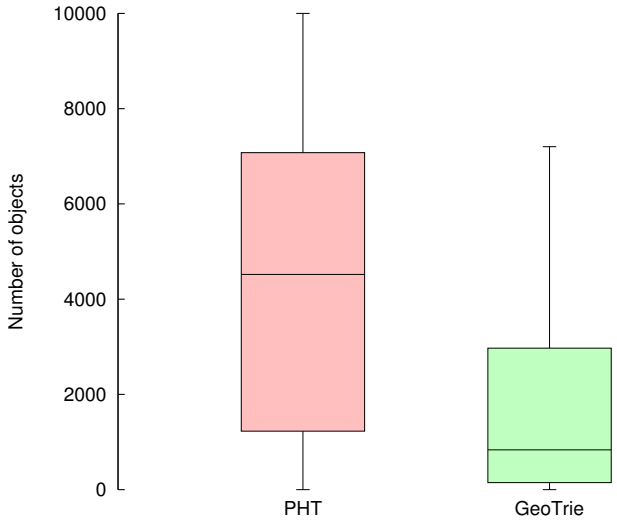


Figure 4.10: Distribution of the storage load among *leaf nodes*

Query Set (QS#)	**PHT (parallel)**	**GeoTrie (parallel)**	GeoTrie (parallel) reduction
1	14,665	10,377	29.23%
2	14,782	10,359	29.92%
3	37,419	21,498	42.54%
4	37,639	21,746	42.22%
5	3,690,055	2,533,308	31.34%
6	114,865	70,885	38.28 %
Total	**3,909,425**	**2,668,173**	**31.75%**

Table 4.12: Total message traffic generated from QS1 to QS6

tree nodes we measured their global distribution. Figure 4.10 presents a comparison of the distribution of data items in *leaf nodes*. The maximum storage capacity of every leaf node is $B = 10,000$ data items. After the insertion of all the dataset, most of *leaf nodes* of GeoTrie store between 148 and 2,971 data items. Instead, most of *leaf nodes* in PHT store between 1,227 and 7,077 data items. From this result, we conclude that, for this dataset distribution, GeoTrie produces a better storage load distribution over virtual *leaf nodes* than PHT.

Figure 4.11 presents the distribution of data items on DHT nodes after inserting the entire Yahoo! dataset. PHT and GeoTrie produce very similar storage load distributions on DHT nodes. This is because using this system configuration there is more virtual *leaf nodes* than physical DHT nodes.

Location-temporal range query traffic

This experiment measures the message traffic upon processing the query sets from *QS1* to *QS6*. It focuses on total message traffic and its distribution on DHT nodes. Table 4.12

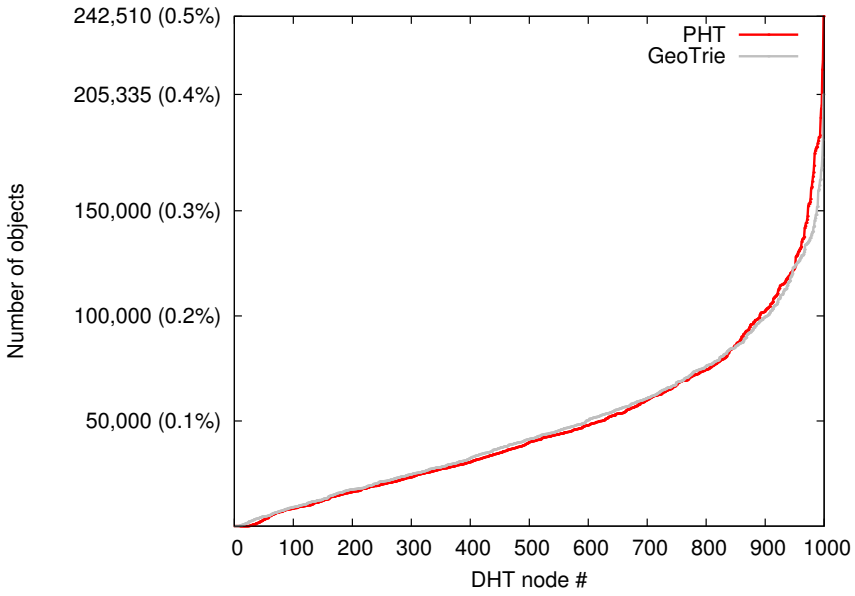


Figure 4.11: Storage load distribution

	PHT (Parallel)						GeoTrie (Parallel)					
Distribution	**QS#1**	**QS#2**	**QS#3**	**QS#4**	**QS#5**	**QS#6**	**QS#1**	**QS#2**	**QS#3**	**QS#4**	**QS#5**	**QS#6**
Min	1	1	1	1	636	3	1	1	1	1	796	4
Quartile 1	1	1	4	4	1,234	13	1	1	5	5	1693	15
Quartile 2	1	1	7	7	1,500	19	1	1	9	9	2023	23
Quartile 3	1	1	11	11	1,579	32	1	1	14	14	2161	38
Max	23	39	39	55	1,643	213	31	82	54	108	2227	284
Average	1.09	1.12	8.54	8.64	1396.19	28.46	1.10	1.17	10.56	10.81	1901.47	36.32

Table 4.13: Distribution of number of target leaf nodes per query set (QS)

presents the total message traffic for each query set. Overall GeoTrie (parallel) generates 31.76% less message traffic than PHT. It is worth noting that, since *QS5* covers the entire spatial domain, it induces far more traffic than any other query set, both on PHT and GeoTrie.

In order to get more insight about this result, we measured the distribution of the number of target leaf nodes per query set. Table 4.13 presents the results. Queries that target a large domain involve proportionally more *leaf nodes* in GeoTrie than in PHT. When the domain covered by the query increases, the difference between the number of *leaf nodes* involved in each approach also increases. For instance, the average number of *leaf nodes* that process query set *QS1* is 1.10 in GeoTrie and 1.09 in PHT. However, this difference increases for query set *QS5*: an average of 1,901.47 GeoTrie *leaf nodes* compared to an average of 1,396.19 PHT nodes.

Figure 4.12 and Table 4.14 present the distribution of message traffic generated by location-temporal range queries on DHT nodes. PHT (parallel) and GeoTrie (parallel) produce very similar message traffic distributions, both in total load values proportionally. A small difference appears on the nodes that incur the larger loads. The largest proportion

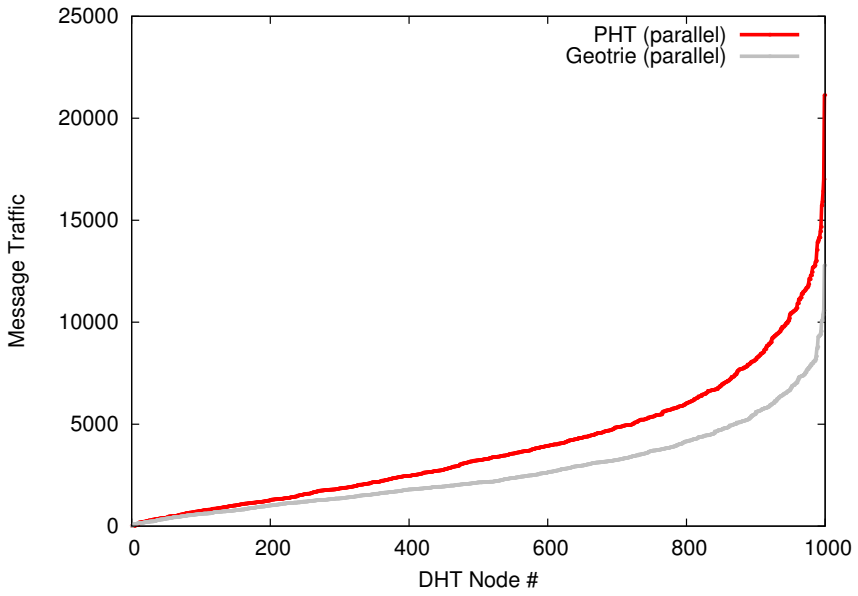


Figure 4.12: Total message load distribution of location-temporal range queries (*QS1* to *QS6*) on DHT nodes.

Distribution	**PHT (parallel)** Total/Percentage (%)	**GeoTrie (parallel)** Total/Percentage (%)
Min	$11/2 \times 10^{-4}$	$11/4 \times 10^{-4}$
Quartile 1	1,572/0.04	1,201/0.04
Quartile 2	3,229/0.08	2,143/0.08
Quartile 3	5,361/0.13	3,681/0.13
Max	21,136/0.54	12,790/0.47
Total	3,909,425/100	2,668,173/100

Table 4.14: Message load distribution of location-temporal range queries

of message load incurred by a single node is 0.54% for PHT against 0.47% for GeoTrie. The largest number of messages handled by a single PHT node is $21,136$ compared to $12,790$ for GeoTrie. Although the difference is small, GeoTrie produces a slightly better message traffic distribution for range queries.

Two main conclusions arise from these results. First, GeoTrie (parallel) exhibits a better distribution of location-temporal range queries over *leaf nodes*. Second, GeoTrie (parallel) presents a lower message traffic for location-temporal range queries because the GeoTrie structure reduces the number of *internal nodes* compared to PHT.

Cache performance

In order to assess the maximum size of the cache in GeoTrie, we do not limit the cache size on GeoTrie nodes and insert all $48,469,177$ geotagged multimedia files of the Yahoo!

Distribution	Cache Size [#Entries]	Cache Size [MB]
Min	12,663	2.21
Quartile 1	12,778	2.23
Quartile 2	12,813	2.24
Quartile 3	12,846	2.24
Max	12,953	2.26

Table 4.15: Cache size distribution

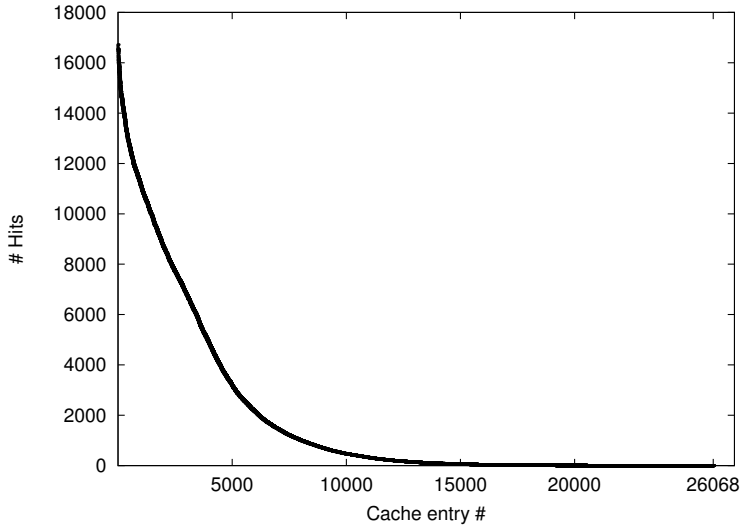


Figure 4.13: Total distribution of cache hits stored in N=1,000 DHT nodes

Distribution	Cache hits
Min	0
Quartile 1	5
Quartile 2	135
Quartile 3	1,742
Max	16,722
Total hits	48,468,083

Table 4.16: Cumulative distribution of cache hits stored in N=1,000 DHT nodes

dataset. Table 4.15 presents the cache size distribution among the DHT nodes. The whole insertion process produces a total number of 29,041 GeoTrie nodes; each node references between 12,663 nodes (43.6%) and 12,953 (44.6%) nodes in cache, which represents between 2.21 [MB] and 2.26 [MB] of main memory. This is a very low cost compared to the current capacity of main memories available on the market.

In order to study the usage of this cache, we indexed all the dataset. At every insertion

operation, a DHT node is chosen randomly. Then, we evaluated the distribution of cache hits by counting the total number of hits per cache entry. Figure 4.13 and Table 4.16 present our results. Our first observation is that a small proportion of GeoTrie nodes (6,517 nodes / 25%) concentrates almost all the cache hits (43,926,392 out of 48,468,083 / 90%). This is due to the skewness of the geotagged dataset. It means that a replacement algorithm such as LRU can be very effective with a maximum cache size set for 6,517 entries. In practice, this is an application parameter that depends on the amount of available main memory.

GeoTrie (cache) exploits the use of direct links (stored in cache and in the GeoTrie structure) in order to reduce the number of DHT lookups. A cache miss triggers a DHT lookup to reach the root node. Then, the query traverses GeoTrie links until it reaches the target leaf node.

Impact of churn

A P2P environment is a dynamic network where nodes can freely join or leave the network. The proportion of these nodes is called *churn* [YZZ+09], and it has a significant impact on cache. Either a cache entry becomes obsolete or a link between two GeoTrie nodes is no longer valid due to a node departure. In this case, the DHT lookup interface must be used with an extra cost of $O(\log(N))$ messages. To measure the impact of churn on GeoTrie (cache), we determine the probability that a link in the path to the target *leaf node* fails. We consider D = 32 and a churn rate. Let $w \in [1, 33]$ [3] be the number of GeoTrie (cache) nodes that a lookup request traverses. The probability of failure of exactly k links associated to cached references follows a binomial distribution given by equation 4.6.

$$P_{k_{cost}} = \binom{w}{k} \epsilon^k (1-\epsilon)^{w-k} \tag{4.6}$$

The expected number of link failures due to node departures is $E(w, \epsilon) = w \times \epsilon$. Table 4.17 presents the expected number of link failures relatively to the churn rate. The worst case occurs when a new node joins the network and performs an insertion request over a branch of GeoTrie of size 33. In this case, the cache is empty and the expected number of link failures is 1.65 under a churn rate of 5%. The consequence is an expected additional cost of $\lceil 1.65 \rceil = 2$ DHT lookups because of two obsolete links in this path.

4.4 Discussion

This section presents a discussion of the main advantages and limitations of GeoTrie.

4.4.1 Advantages

- **Low query latency**. This is an important metric from the user's perspective as it represents an important proportion of the time to complete an operation.

[3] There is a maximum of D=32 possible labels plus the root node

	# **Links** (w)					
Churn rate (ϵ)	4	8	12	16	24	33
0.025	0.1	0.2	0.3	0.4	0.6	0.825
0.05	0.2	0.4	0.6	0.8	1.2	1.65
0.075	0.3	0.6	0.9	1.2	1.8	2.475
0.1	0.4	0.8	1.2	1.6	2.4	3.3

Table 4.17: Impact of churn on GeoTrie (cache): Expected number of link failures relative to the churn rate

GeoTrie presents a solution that drastically reduces the latency of insertions and range queries associated with massive geotagged datasets.

In terms of insertion requests latency, GeoTrie (cache) performs better than PHT. GeoTrie (cache) can carry out lookups up to 2.47 times faster because it proposes an optimistic protocol which takes advantage of a client cache. Because of the inherent skewness of geotagged datasets, the required client cache size can be small compared to the current capacity of desktop computers.

In terms of location-temporal range query latency, GeoTrie (parallel) reduces up to 2.64 times the latency incurred by PHT (parallel). GeoTrie spawns less internal nodes, so the size of paths for range queries is smaller.

- **Load balance**. GeoTrie relies on the properties of *prefix trees* that create a global knowledge of the location of every node. This global knowledge allows to avoid a bottleneck on the root node upon insertions and range queries. For instance, location-temporal range queries that target smaller domains can be directly forwarded to lower trie levels, avoiding the root node and thus distributing the load.

  However, a single GeoTrie node can receive a high load of location-temporal range queries. In this case, GeoTrie can take advantage of the underlying replication mechanism implemented in the DHT layer to provide query load balance. For instance in Pastry [RD01], requests can be equally processed by replica nodes maintained in the *leaf set*. The number of these replicas can be dynamically adapted to the query load.

  In terms of storage, GeoTrie dynamically balances the load among DHT nodes. When a *leaf node* stores more than B data items, it distributes the load over eight new DHT nodes chosen as a result of applying consistent hashing to the node label. Thus, the GeoTrie structure is uniformly distributed among DHT nodes and the maximum amount of data on a single GeoTrie node is bounded by the system parameter B.

- **Scalability and high availability.** The message complexity of all GeoTrie operations (insertion, query, maintenance) depends partly on constant parameters: the number D of bits used in the key, and the maximum number B of objects a node

can store. Another significant parameter is the number N of nodes in the underlying DHT. The cost of GeoTrie operations does not depend on the size of the dataset. Therefore, GeoTrie scales with the number of nodes of the underlying DHT. This is particularly important when indexing continuous flows of data.

GeoTrie can also use replication at the DHT level in order to increase the availability of indexed objects. Replicas are usually stored on numerically close nodes in the DHT to ensure that the overlay can withstand network partitions (in the *leaf set* for Pastry or in the list of successors for Chord). Managing replicas induces a higher cost for maintenance operations. Nevertheless, it provides high availability.

4.4.2 Limitations

- **Query load balance.** The greatest common prefix strategy used to resolve range queries allow to distribute the load of queries on different GeoTrie levels and thus achieving load balance. However, queries that have an empty common prefix are always routed to the root node. Examples of these queries are queries from sets *QS5* and *QS6* that covers either all the location or the temporal domain. These types of queries can create a bottleneck on the root node. One strategy to alleviate this bottleneck is to use the cache information in order to start the query at a higher GeoTrie level.

4.5 Conclusion

This chapter presents an extensive performance evaluation of GeoTrie. 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 GeoTrie incurs lower latency and message traffic than PHT, the closest solution. In a configuration involving $1,000$ nodes, GeoTrie performs insertions up to 2.47 times faster than PHT on average, and generates up to 80.4% less message traffic. GeoTrie also carries out location-temporal range queries up to 2.64 faster on average.

GeoTrie produces a smooth distribution of the load of insertions and location-temporal range queries. This is an important property because it alleviates a potential bottleneck on the root node, and because range queries that avoid the root node present an average latency up to 1.9 times faster than queries which start at the root level.

Chapter 5

Conclusion and Future Work

Contents

This chapter presents the conclusion of the former research and it gives directions for future work.

5.1 Conclusion

This thesis presented two solutions for supporting scalable location-temporal queries for Location Based Social Network (LBSN) services. The main requirements of such services are scalability, load balance, low latency, and low message traffic when coping with insertions and range queries from millions of users.

Centralised relational databases cannot scale up to accommodate concurrent location-temporal queries from millions of users: their response time significantly increases when concurrent insertions and queries have to cover billions of objects. Cluster based solutions can scale out the number of servers. However, these solutions present a bottleneck in terms of output bandwidth, which limits their scalability and application scenarios. As bandwidth is currently an expensive resource, LBSN services are limited to applications that generate a lower network traffic. Moreover, the financial cost of maintaining a large number of servers is not negligible.

In order to overcome these issues, the approaches presented in this thesis follow the P2P computing paradigm using a DHT as building block since DHTs provide a highly scalable lookup interface that reaches a resource in $O(\log(N))$ routing hops, where N is the number of nodes. They also provide a replication mechanism that ensures high availability and fault tolerance. However, DHTs are ill-suited to support range queries because they rely on hash functions that destroy data locality. Even if, several solutions have been proposed in the literature, none of them meet all the requirements of LBSN services.

The first contribution, called Big-LHT, supports scalable *n-recent location-temporal zone queries*. It implements a primary index based on location and a secondary index based on upload time. This indexing strategy provides direct access to the most recent data items in any given geographic zone. It performs static data partition based on location and a pre-defined system parameter. Theoretical and practical evaluations show that Big-LHT presents a low index maintenance cost and a low insertion cost. It exhibits good insertion load balance for applications that need to define small geographic zones such as POIs. However, it deteriorates with the size of the geographic zone.

The second contribution, called GeoTrie, supports scalable *location-temporal range queries*. It implements a prefix octree which indexes data based on (latitude, longitude, timestamp) tuples. It also introduces a dynamic partition strategy that adapts to the data distribution, thus providing scalability and load balancing.

An extensive performance evaluation shows that GeoTrie outperforms PHT, its closest competing solution, in terms of message traffic and latency of insertions and range queries. This evaluation was conducted over a real world dataset composed of the data extracted from about 48 million multimedia items (pictures and videos).

GeoTrie presents the following characteristics:

- GeoTrie performs insertions up to 2.74 times faster on average than PHT. It also performs range queries up to 2.64 times faster on average.
- GeoTrie reduces the message traffic of insertions by up to 80.4% compared to PHT. It performs range queries using up to 31.74% fewer messages.
- All the operations offered by GeoTrie scale with the number of DHT nodes. The data partitioning strategy provides storage load balance and query load balance.
- Realistic churn rates (under 10%) have a minimal impact on GeoTrie protocols.

5.2 Future Work

Storing and analysing information extracted from massive geotagged datasets is currently a hot research topic [EM15, VGC+12, SGEY12]. This work represents a building block for storing and retrieving such data at scale. This thesis opens several perspectives, among which:

1. **Scalable location-temporal aggregation queries.** Aggregation operators such as *min*, *max*, *count*, and *topk* are useful to summarise data generated inside a geographical area in a given time interval. It is indeed interesting and useful to explore how to deploy such operators on a large scale.
2. **Location-aware publish/subscribe.** The traditional publish/subscribe model [EFGK03] allows to multicast messages within a given group. Location can be used as a new dimension in order to multicast messages to groups that are inside a given geographic area.

3. **Reliable indexing for structured P2P networks**. A P2P environment is exposed to *malicious nodes* that can change its behaviour (i,e., they can change the implementation of the application protocols) in order to attack the system. Existing approaches such as *Reputations systems* [BR07] [RKZF00] and *accountability systems* [HKD07] can be used in order to improve the global reliability of the system. However, specific attacks on indexing systems can introduce failures on insertions and range queries. For instance, a malicious node can reply with a different state to a lookup operation in order to introduce a failure into a single insertion operation.

Bibliography

[ACMD+03] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-grid: a self-organizing structured p2p system. *ACM SIGMOD Record*, 32(3):29–33, 2003.

[ARBB08] Michele Albano, Laura Ricci, Martina Baldanzi, and Ranieri Baraglia. Voraque: Range queries on voronoi overlays. In *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on*, pages 495–500. IEEE, 2008.

[AS07] James Aspnes and Gauri Shah. Skip graphs. *Acm transactions on algorithms (talg)*, 3(4):37, 2007.

[BAS04] Ashwin R Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. In *ACM SIGCOMM computer communication review*, volume 34, pages 353–366. ACM, 2004.

[BKMR07] Olivier Beaumont, Anne-Marie Kermarrec, Loris Marchal, and Étienne Rivière. Voronet: A scalable object network based on voronoi tessellations. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10. IEEE, 2007.

[BKS04] Farnoush Banaei-Kashani and Cyrus Shahabi. Swam: a family of access methods for similarity-search in peer-to-peer data networks. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 304–313. ACM, 2004.

[BR07] Xavier Bonnaire and Erika Rosas. A critical analysis of latest advances in building trusted p2p networks using reputation systems. In *International Conference on Web Information Systems Engineering*, pages 130–141. Springer, 2007.

[CDKR02] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.

[CFCS04] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. Maan: A multi-attribute addressable network for grid information services. *Journal of Grid Computing*, 2(1):3–14, 2004.

[CLGS04] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, pages 25–30. ACM, 2004.

[Com79] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[Cor09] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[CPRZ97] Paolo Ciaccia, Marco Patella, Fausto Rabitti, and Pavel Zezula. Indexing metric spaces with m-tree. In *SEBD*, volume 97, pages 67–86, 1997.

[CRR+05] Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph Hellerstein. A case study in building layered dht applications. *ACM SIGCOMM Computer Communication Review*, 35(4):97–108, 2005.

[DBVKOS00] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[DEG+01] P. 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.

[DP78] Alvin M Despain and David A Patterson. X-tree: A tree structured multi-processor computer architecture. In *Proceedings of the 5th annual symposium on Computer architecture*, pages 144–151. ACM, 1978.

[EFGK03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[EJ01] Donald Eastlake and Paul Jones. Us secure hash algorithm 1 (sha1), 2001.

[EM15] Ahmed Eldawy and Mohamed F Mokbel. The era of big spatial data. In *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on*, pages 42–49. IEEE, 2015.

[FRA+16] L Ferrucci, L Ricci, M Albano, R Baraglia, and M Mordacchini. Multidimensional range queries on hierarchical voronoi overlays. *Journal of Computer and System Sciences*, 2016.

[GS04] Jun Gao and Peter Steenkiste. An adaptive protocol for efficient support of range queries in dht-based systems. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 239–250. IEEE, 2004.

[Gut84] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[GYGM04] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, pages 19–24. ACM, 2004.

[HASB16] Nicolas Hidalgo, Luciana Arantes, Pierre Sens, and Xavier Bonnaire. Echo: Efficient complex query over dht overlays. *Journal of Parallel and Distributed Computing*, 88:31–45, 2016.

[Hil91] David Hilbert. Ueber die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

[HJS+03] Nicholas JA Harvey, Michael B Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. *networks*, 34:38, 2003.

[HKD07] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. In *ACM SIGOPS operating systems review*, volume 41, pages 175–188. ACM, 2007.

[HRA+11] Nicolas Hidalgo, Erika Rosas, Luciana Arantes, Olivier Marin, Pierre Sens, and Xavier Bonnaire. Dring: A layered scheme for range queries over dhts. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 29–34. IEEE, 2011.

[JOV05] 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.

[JOV+06] HV Jagadish, Beng Chin Ooi, Quang Hieu Vu, Rong Zhang, and Aoying Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 34–34. IEEE, 2006.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 84–95, New York, NY, USA, 2002. ACM.

[Mor66] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[Neu94] B. Clifford Neuman. Scale in distributed systems. In *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, 1994.

[OBSC09] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial tessellations: concepts and applications of Voronoi diagrams*, volume 501. John Wiley & Sons, 2009.

[Ora01] Andy Oram. *Peer-to-Peer: Harnessing the power of disruptive technologies*. " O'Reilly Media, Inc.", 2001.

[PNT12] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Saturn: range queries, load balancing and fault tolerance in dht data systems. *IEEE Transactions on Knowledge and Data Engineering*, 24(7):1313–1327, 2012.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[RD01] 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.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[RH13] Oliver Roick and Susanne Heuser. Location based social networks–definition, current state of the art and research agenda. *Transactions in GIS*, 17(5):763–784, 2013.

[Rip01] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *1st International Conference on Peer-to-Peer Computing (P2P 2001), Sweden*, pages 99–100. IEEE Computer Society, 2001.

[RKZF00] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.

[RRHS04] 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.

[Sam06] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[SAZ+02] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 73–86. ACM, 2002.

[SBR04] N. Sarshar, P. O. Boykin, and V. P. Roychowdhury. Percolation search in power law networks: making unstructured peer-to-peer networks scalable. In *Peer-to-Peer Computing, 2004. Proceedings. Proceedings. Fourth International Conference on*, pages 2–9, Aug 2004.

[SGEY12] Shashi Shekhar, Viswanath Gunturi, Michael R Evans, and KwangSoo Yang. Spatial big-data challenges intersecting mobility and cloud computing. In *Proceedings of the Eleventh ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pages 1–6. ACM, 2012.

[SMK+01] 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.

[SOTZ05] Yanfeng Shu, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. In *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, pages 173–180. IEEE, 2005.

[SP08] Cristina Schmidt and Manish Parashar. Squid: Enabling search in dht-based systems. *Journal of Parallel and Distributed Computing*, 68(7):962–975, 2008.

[SW04] Ralf Steinmetz and Klaus Wehrle. Peer-to-peer-networking &-computing. *Informatik-Spektrum*, 27(1):51–54, 2004.

[TSF+16] Bart Thomee, David A Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. Yfcc100m: The new data in multimedia research. *Communications of the ACM*, 59(2):64–73, 2016.

[TZX10] Yuzhe Tang, Shuigeng Zhou, and Jianliang Xu. Light: a query-efficient yet low-maintenance indexing scheme over dhts. *Knowledge and Data Engineering, IEEE Transactions on*, 22(1):59–75, 2010.

[VGC+12] 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, 2012.

[WJ96] David A White and Ramesh Jain. Similarity indexing with the ss-tree. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 516–523. IEEE, 1996.

[WS06] Xinfa Wei and Kaoru Sezaki. Dhr-trees: A distributed multidimensional indexing structure for p2p systems. In *2006 Fifth International Symposium on Parallel and Distributed Computing*, pages 281–290. IEEE, 2006.

[YGM02] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 5–14, 2002.

[YZZ+09] Dong Yang, Yu-xiang Zhang, Hong-ke Zhang, Tin-Yu Wu, and Han-Chieh Chao. Multi-factors oriented study of p2p churn. *International Journal of Communication Systems*, 22(9):1089–1103, 2009.

[ZSLS06] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over DHT. In *IPTPS, 5th International workshop on Peer-To-Peer Systems*, 2006.