Virtual and Consistent Hyperbolic Tree: A New Structure for Distributed Database Management

Telesphore Tiendrebeogo, Damien Magoni

Virtual and Consistent Hyperbolic Tree: A New Structure for Distributed Database Management

Telesphore Tiendrebeogo[1] and Damien Magoni[2]

[1] Polytechnic University of Bobo-Dioulasso
Bobo-Dioulasso, Burkina Faso
tetiendreb@gmail.com
[2] University of Bordeaux – LaBRI
Talence, France
magoni@labri.fr

Abstract. This paper describes a new scalable, reliable and consistent structure for implementing a distributed database system. This structure is based on the hyperbolic geometry and we call it a Virtual and Consistent Hyperbolic tree (VCH-tree). This structure is intended for supporting queries over possibly large spatial datasets distributed over interconnected servers. The VCH-tree is comparable to the well-known R-tree structure, but it uses the hyperbolic geometry properties of the Poincaré disk model. It uses a distributed balanced Q-degree spatial tree that scales with data objects' insertions into potentially any number of storage servers through virtual hyperbolic coordinates. A user application manipulates the structure from a client node. The client can connect to the system through one of the database servers that is already in the VCH-tree. Messages are then routed towards the proper server by a greedy algorithm which is using the virtual hyperbolic coordinates attributed to each server. We have performed simulations to assess the efficiency and reliability of the VCH-tree. Results show that our VCH-tree is consistent and has expected performances given the scalability and flexibility it provides to distributed database applications.

1 Introduction

We aim at indexing large data sets of spatial objects, each uniquely identified by an object identifier (OID) and stored in the VCH-tree with a given address. We define a scalable and reliable index that generalizes R-tree structure for distributed data structure [16] that we call VCH-tree. A VCH-tree permits redundancy of object references, like R-tree [5] or R*-tree [9]. The fundamental principle of our system is to map a large database object identifier space onto a set of servers in a deterministic and distributed way. Roughly, given an object key, the system is able to obtain the location of several database servers where are stored the corresponding values. In order to route the queries, each database server usually maintains the status of its connections to all other databases servers which increases drastically the number of messages exchanged, and this

may constitute a severe scaling limitation. The same applies to the number of routing hops that must not grow too fast with the number of database servers in the system [7]. Moreover, most of distributed database systems suffer from the lack of flexibility in storage queries (i.e., values' placement) involving consequently a heavy lookup traffic load on the lookup paths of the underlying nodes. We argue that we can address and overcome simultaneously all the aforementioned issues by maintaining a good trade-off between robustness, efficiency and system complexity. To this end, we promote an indexing system for distributed database relying on hyperbolic geometry [1]. In this paper we make the following contributions:

- We provide a new architecture for indexation in the distributed database system without any constraints. The database servers can connect arbitrarily to each other, the data objects can be inserted, updated or deleted without the cost of maintaining any global knowledge of the servers' topology. Our approach is based on the groundbreaking work of Kleinberg [11], that we have enhanced in order to manage a dynamic topology which is able to grow and shrink over time.
- We propose a specific key based greedy routing system for storing the mapping of database object identifiers to the servers' addresses in the hyperbolic plane in a flexible and efficient way. Values are stored in order to avoid overloading a particular zone of the distributed system. Furthermore, storage and retrieving queries can be solved within $O(logN)$ hops.
- To improve database object availability and access performance, our system which is akin to a Distributed Hash Table (DHT) system embeds a redundancy and caching mechanism that can be adjusted to obtain a good trade off between reliability and storage consumption.
- We have carried out simulations to demonstrate the interest of our solution in terms of feasibility to build a distributed database system over a topology based on the hyperbolic plane.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the related previous work. Section 3 highlights some properties of the hyperbolic plane when represented by the Poincaré disk model. Section 4 defines the local addressing and greedy routing algorithms of our distributed VCH-tree system. Section 5 defines the binding algorithm of our VCH-tree system. Section 6 presents the results of our practicability evaluation obtained by simulations and we conclude in Section 7.

2 Related Work

Until recently, most of the spatial indexing design efforts have been devoted to centralized systems [4] although, for non-spatial data, research devoted to an efficient distribution of large data sets is well-established [2] [3]. Many Scalable Distributed Data Structure (SDDS) schemes are hash-based, e.g., variants of LH* [8], or use a Distributed Hash Table (DHT) [2][13]. Some SDDSs are range

partitioned, from RP* based systems [10] up to BATON [4] most recently. There were also proposals for k-d partitioning, e.g., k-RP [14] using distributed kd-trees for data points, or hQT* [10] using quad-trees for the same purpose. Hambrusch and Khokhar [6] present a distributed data structure based on orthogonal bisection trees (2-d KD trees). Kriakov et al. [12] describe an adaptive index method which offers dynamic load balancing of servers and distributed collaboration. The structure requires a coordinator which maintains the load of each server.

3 Hyperbolic Geometry

The model that we use in our system to represent the hyperbolic plane is called the Poincaré disk model. In the Poincaré disk model, the hyperbolic plane is represented by the open unit disk of radius 1 centered at the origin. In this specific model:

- Points are represented by points within this open unit disk.
- Lines are represented by arcs of circles intersecting the disk and meeting its boundaries at right angles.

In this model, we refer to points by using complex coordinates.

An important property is that we can tile the hyperbolic plane with polygons of any sizes, called p-gons. Each tessellation is represented by a notation of the form $\{p, q\}$ where each polygon has p sides with q of them at each vertex. There exists a hyperbolic tessellation $\{p, q\}$ for every couple $\{p, q\}$ obeying $(p-2)*(q-2) > 4$. In a tiling, p is the number of sides of the polygons of the *primal* (the black edges and green vertices in figure 1) and q is the number of sides of the polygons of the *dual* (the red triangles in figure 1). Our purpose is to partition the plane and address each node uniquely. We set p to infinity, thus transforming the primal into a regular tree of degree q. The dual is then tessellated with an infinite number of q-gons. This particular tiling splits the hyperbolic plane in distinct spaces and constructs an embedded tree that we use to assign unique addresses to the nodes. An example of such a hyperbolic tree with $q = 3$ is shown in figure 1.

In the Poincaré disk model, the distances between any two points z and w are given by curves minimizing the distance between these two points and are called geodesics of the hyperbolic plane. To compute the length of a geodesic between two points z and w and thus obtain their hyperbolic distance $d_{\mathbb{H}}$, we use the Poincaré metric which is an isometric invariant given by the formula:

$$d_{\mathbb{H}}(z, w) = arg cosh(1 + 2 \times \frac{|z - w|^2}{(1 - |z|^2)(1 - |w|^2)}) \quad (1)$$

This formula is used by the greedy routing algorithm shown in the next section.

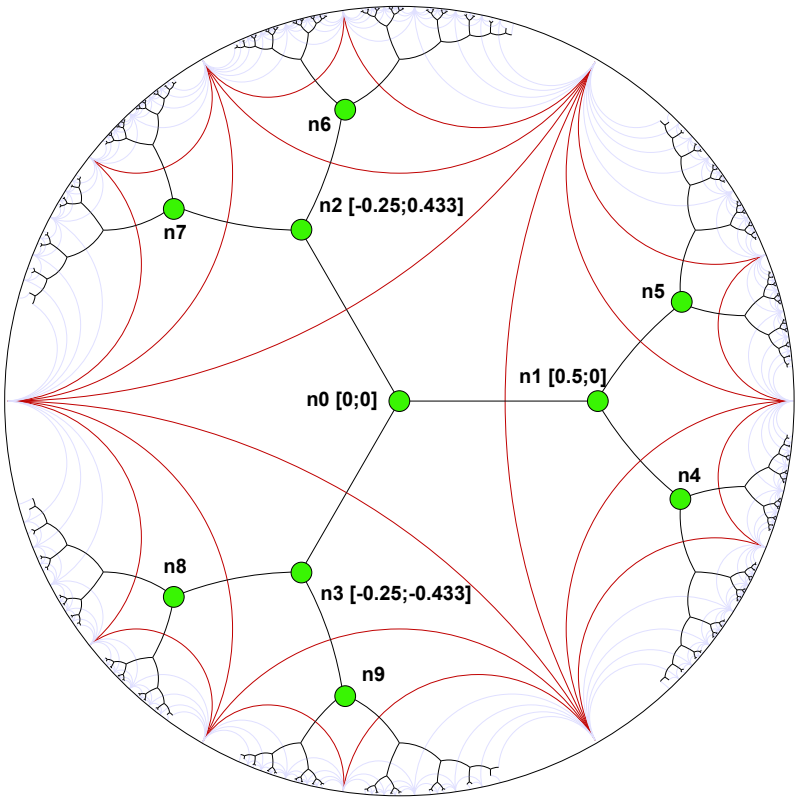

Fig. 1. 3-regular tree in the hyperbolic plane

4 Topology of the Servers

We now explain in this section how we create the hyperbolic addressing tree for database servers interconnections and how queries can be routed in our distributed database system. The first step in the creation of a VCH-tree of servers nodes is to start the first database server and to choose the degree of the addressing tree.

We recall that the hyperbolic coordinates (i.e., a complex number) of a server node of the addressing tree are used as the address of the corresponding database server in the distributed data base system. A server node of the tree can give the addresses corresponding to its children in the VCH-tree. The degree determines how many addresses each database server will be able to give for news nodes servers connections. The degree of the VCH-tree is defined at the beginning for all the lifetime of the distributed database system. The distributed database system is then built incrementally, with each new data server joining one or

more existing data servers. Over time, the data servers will leave the overlay until there is no server left which is the end of the distributed database system. So, for every data object that must be stored in the system, an OID is associated with him and map then in key-value pair. The key will allow to determine in which data servers the object will be stored (as explained in the following section). Furthermore when a data object is deleted, the system must be able to update this operation in all the system by forwarding query. This method is scalable because unlike Kleinberg [11], we do not have to make a two-pass algorithm over the whole distributed system to find its highest degree. Also in our system, a server can connect to any other server at any time in order to obtain an address.

The first step is thus to define the degree of the tree because it allows building the *dual*, namely the regular $q - gon$. We nail the root of the tree at the origin of the *primal* and we begin the tiling at the origin of the disk in function of q. Each splitting of the space in order to create disjoint subspaces is ensured once the half spaces are tangent ; hence the *primal* is an infinite q-regular tree. We use the theoretical infinite q-regular tree to construct the greedy embedding of our q-regular tree. So, the regular degree of the tree is the number of sides of the polygon used to build the *dual* (see figure 1). In other words, the space is allocated for q child database servers. Each database server repeats the computation for its own half space. In half space, the space is again allocated for $q-1$ children. Each child can distribute its addresses in its half space. Algorithm 1 shows how to compute the addresses that can be given to the children of a database server. The first database server takes the hyperbolic address (0;0) and is the root of the tree. The root can assign q addresses.

Algorithm 1 Calculating the Coordinates of a Server's Children.

```
1: procedure CalcChildrenCoords(server, q)
2:     step ← argcosh(1/sin(π/q))
3:     angle ← 2π/q
4:     childCoords ← server.Coords
5:     for i ← 1, q do
6:         ChildCoords.rotationLeft(angle)
7:         ChildCoords.translation(step)
8:         ChildCoords.rotationRight(π)
9:         if ChildCoords ≠ server.ParentCoords then
10:            StoreChildCoords(ChildCoords)
11:        end if
12:    end for
13: end procedure
```

This distributed algorithm ensures that the database servers are contained in distinct spaces and have unique coordinates. All the steps of the presented algorithm are suitable for distributed and asynchronous computation. This algorithm allows the assignment of addresses as coordinates in dynamic topologies. As the global knowledge of the distributed database system is not necessary, a

new server can obtain coordinates simply by asking an existing server to be its parent and to give it an address for itself. If the asked server has already given all its addresses, the new server must ask an address to another existing database server. When a new server obtains an address, it computes the addresses (i.e., hyperbolic coordinates) of its future children(The new database servers which shall connect to the distributed database system). The addressing VCH-tree is thus incrementally built at the same time than the distributed database system.

When a new database server has connected to database servers already inside the distributed database system and has obtained an address from one of those database servers, it can start sending requests to store or lookup database object in the distributed database system. The routing process is done on each database server on the path (starting from the sender) by using Algorithm 2, a greedy algorithm based on the hyperbolic distances between the servers. When a query is received by a database server, the database server computes the distance from each of its neighbors to the destination and forwards the query to its neighbor which is the closest to the destination (destination database server computing is given in Section 5).

Algorithm 2 Routing a Query in the Distributed Database System.

```
1: function getNextHop(server, query) return server
2:     w = query.destinationServerCoords
3:     m = server.Coords
4:     d_min = argcosh(1 + 2 |m−w|^2 / ((1−|m|^2)(1−|w|^2)))
5:     p_min = server
6:     for all neighbor ∈ server.Neighbors do
7:         n = neighbor.Coords
8:         d = argcosh(1 + 2 |n−w|^2 / ((1−|n|^2)(1−|w|^2)))
9:         if d < d_min then
10:            d_min = d
11:            p_min = neighbor
12:        end if
13:    end for
14:    return p_min
15: end function
```

In a real network environment, link and server failures are expected to happen often. If the addressing VCH-tree is broken by the failure of a database server or link, we flush the addresses attributed to the servers beyond the failed server or link and reassign new addresses to those servers (some servers may have first to reconnect to other servers in order to restore connectivity). But this solution is not developed in this paper.

5 Storage and Retrieval of Data Objects

In this section we explain how our distributed database system computes the destination database servers addresses for storing and retrieving queries. Indeed, the first server contacted by a client (prime server) for sending a query in the system consider the latter as a data object that can be stored or looked up. Thus this server generates an OID associated to the data object and the latter is mapped onto hyperbolic addresses corresponding to destination database servers' addresses in the VCH-tree.

On startup, each new client query is associated with the data object with OID corresponding to the name of the query and that identifies the query it runs on. This name will be kept by data object during all the lifetime of the distributed database system.

When the prime database server computes some specific addresses of database servers, when it is about a storage query, it stores the name (OID) and value of query in these specific addresses of distributed database servers, thus the data object in the DHT, when it is about a retrieving query, it contacts database servers which addresses has been computed. In our distributed system, the name is used as a key by a mathematical transformation. If the same name is already stored in the distributed database system, an error message is sent back to the prime server (Server by whom the client is directly bound) in order to generate another name. Thus the distributed database system structure itself ensures that OIDs are unique.

A (OID, value) pair, with the name acting as a key by mapping is called a *binding*. Figure 2 shows how and where a given binding is stored in the distributed database system. A binder is any database server that stores these pairs. The depth of a server in the addressing VCH-tree is defined as the number of parent servers to go through for reaching the root of the VCH-tree (including the root itself). When the distributed database system is created, a maximum depth for the potential binders is chosen. This value is defined as the *binding VCH-tree depth*. To ensure a load balancing of the system, the depth p is chosen such p minimize the inequality 2 :

$$p \times (\frac{(1-(p-1)^d)}{2-p} + 1 \geq N \tag{2}$$

with p the depth, d the degree and N the number of database servers.

When a new database server joins the distributed database system by connecting to other servers, it obtains an address from one of these servers. Next, the server stores its own binding in the system. So, during his life, each database server tries to join others by sending a join query. Each server cannot accept that a limited number of join queries independently of the degree of the VCH-tree. The new connections serve as shortcuts during the phases of storage and retrieving of data objects. We call these connections, shortcut links as indicated in Figure 2.

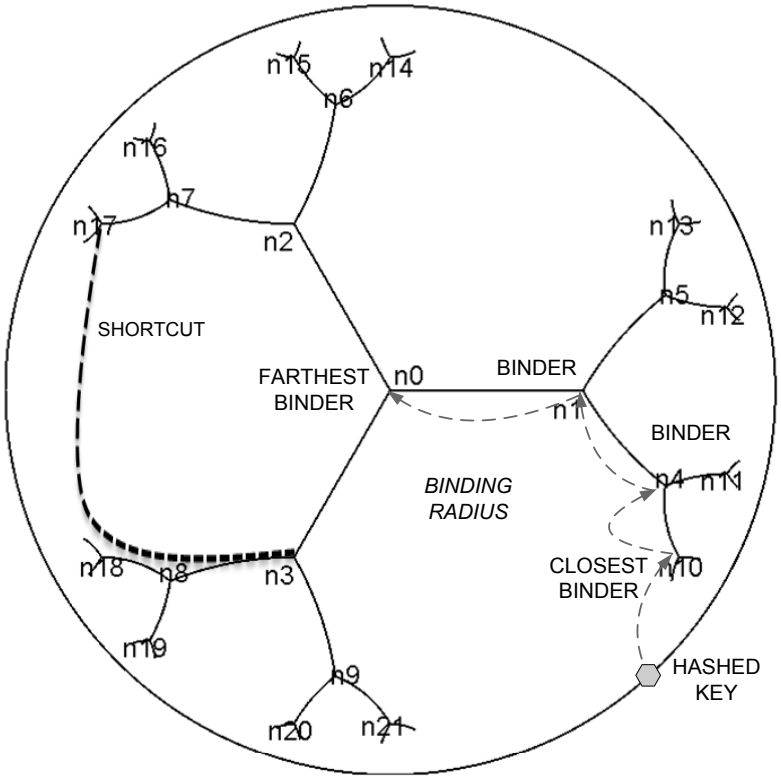

Fig. 2. Storage in the VCH-tree

5.1 Storage Query Processing

When a client wants to send a storage query (i.e., insertion), the first server with whom it is connected consider a query as an object (thus generating an OID) and creates a key by hashing its name with the SHA-512 algorithm. It divides the 512-bit key into 16 equally sized 32-bit subkeys (for redundant storage). The server selects the first subkey and maps it to an angle by a linear transformation.

The angle is given by:

$$\alpha = 2\pi \times \frac{\texttt{32-bit subkey}}{\texttt{0xFFFFFFFF}} \tag{3}$$

The database server then computes a virtual point v on the unit circle by using this angle:

$$v(x, y) \text{ with } \begin{cases} x = cos(\alpha) \\ y = sin(\alpha) \end{cases} \tag{4}$$

Next the database server determines the coordinates of the closest binder to the computed virtual point above by using the given *binding tree depth*.

In the figure we set the *binding VCH-tree depth* to three to avoid cluttering the figure. It's important to note that this closest binder may not really exist if no database server is currently owning this address. The database server then sends a storage query to this closest database server. This query is routed inside the distributed database system by using the greedy algorithm of Section 4. If the query fails because the binder does not exist or because of database server/link failures, it is redirected to the next closest binder which is the father of the computed binder.

The path from the computed closest binder to the farthest binder is defined as the binding radius. This process ensures that the queries are always stored first in the binders closer to the unit circle and last in the binders closer to the disk center. However to avoid overloading the farthest binder particularly and to keep a load balancing, we limit the number of storages S like follow:

$$S \leq \lfloor \frac{1}{2} \times \frac{log(N)}{log(q)} \rfloor \tag{5}$$

with N equal to number of distributed database servers, q to degree of VCH-tree.

Furthermore the previous solution, any binder will be able to set a maximum number of stored queries and any new database server to store will be refused and the query redirected as above. Besides, to provide redundancy and so ensure the availability and reduce the latency period in the lookup process, the database server does the storage process described above for each of the other 15 sub keys. Thus 16 different binding radii will be used at the most and this will improve the even distribution of the pairs (key-value).

In addition to this and still for redundancy purposes, a pair key-value of the data object may be stored in more than one database server of the binding radius. A binder could store a data object and still redirect its query for storage it in other ancestor binders. The number of stored copies of a key-value pair along the binding radius may be an arbitrary value set at the distributed system creation. Similarly the division of the key in 16 sub keys is arbitrary and could be increased or reduced depending on the redundancy needed. To conclude we can define two redundancy mechanisms for storage copies of a given binding:

1. We can use more than one binding radius by creating several uniformly distributed subkeys.
2. We can store the data object key-value pair in more than one binder in the same binding radius.

These mechanisms enable our distributed database system to cope with a non-uniform growth of the database servers and they ensure that a data object will be stored in a redundant way that will maximize the success rate of its retrieval. The numbers of sub keys and the numbers of copies in a radius are parameters that can be set at the creation of the distributed database system. Increasing them leads to a tradeoff between improved reliability and lost storage space in binders. Besides our solution has the property of consistent hashing:

if one database server fails, only its keys are lost but the other binders are not impacted and the whole system remains coherent. Algorithm 3 illustrates the previous mechanism.

Algorithm 3 Storage Algorithm.

```
1: function STORE(Query)
2:     OID ← Query.GetOID()
3:     Key ← Hash(OID)
4:     for all red ∈ R_Circular do
5:         depth ← P_Max
6:         i ← 1
7:         while i ≤ ⌊1/2 × log(N)/log(q)⌋ && depth ≥ 0 do
8:             SubKey[red][depth] ← ComputeSubkey(Key)[red][depth]
9:             TargetServerAddr[red][depth] ← ComputeAddr(SubKey[red][depth])
10:            TargetServer ← GetTarget(TargetServerAddr[red][depth])
11:            if route(Query, TargetServer) then
12:                i++
13:                put(OID, Query)
14:            end if
15:            depth − −
16:        end while
17:    end for
18: end function
```

5.2 Lookup Query Processing

Now, if the client wants to lookup a data object in the distributed database system, a prime server is contacted and generates an OID for the client query. Here again, the OID is mapped into a key by the SHA-512 algorithm, thus the 512 bits key is divided into 16 subkeys. Each subkey, by the process described in Section 5.1, will be transformed into an address that represents the address of the database server where the data object is stored. The latter is contacted by the prime database server for updating, deleting or retrieving the associated value. When the redundancy mechanism has been used to store the data object, the lookup repeats the latter process of lookup for any subkey, thus the operation will be performed on all database servers that contain the data object. Our distributed system ensures the coherence of data objects of the distributed database. This mechanism is illustrated by Algorithm 4.

6 Evaluation

We performed experiments for evaluating the behavior of a VCH-tree over large datasets. Furthermore, we consider that the system is static so there are no

Algorithm 4 Lookup and Update Algorithm.

```
1: function LOOKUP(Query) return Value
2:     QueryOID ← Target.GetQueryOID()
3:     Key ← Hash(QueryOID)
4:     for all red ∈ R_Circular do
5:         depth ← P_Max
6:         i ← 1
7:         while i ≤ ⌊1/2 × log(N)/log(q)⌋ && depth ≥ 0 do
8:             TargetServerAddr[red][depth] ← GetValue(Key)
9:             Value ← GetValue(TargetServerAddr[red][depth], QueryOID)
10:            if Value != null then
11:                if Query == delete then
12:                    delete(OID)
13:                end if
14:                if (Query == update) then
15:                    update(OID)
16:                end if
17:                if Query == select then
18:                    return Value
19:                end if
20:                i++
21:            end if
22:            depth−−
23:        end while
24:    end for
25: end function
```

join or leave of database servers during the simulation. We use the Peersim [15] simulator for running event-driven simulations. The study involved the following parameters of the VCH-tree:

- The number of database servers connected and used to store the data objects. Here we have considered 10000 database servers;
- Each run lasts 2 hours of simulated time;
- We try to store 6 millions of data objects in our distributed database system following an exponential distribution with a median equal to 10 minutes;
- The maximum capacity for each server is set to 6000 objects.

We studied the behavior of our structure for both data objects' storage and retrieval in the system. We are interested in observing the scalability of our system, the shape of the hyperbolic tree, the storage load balancing and the length of the paths of the queries.

6.1 Spatial Shape of the VCH-tree

Figure 3 shows an experimental distribution of points corresponding to the scatter plot of the distribution of the database servers in our system. We can see

that our VCH-tree is balanced. Indeed, we can notice by part and others around the unit circle which we have database servers. This has an almost uniform distribution around the root, which implies that our system builds a well-balanced tree that will more easily allow to reach a proper load balancing for the storage.

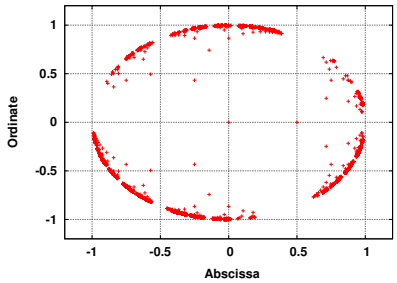

Fig. 3. Scatter plot of the spatial positions of the database servers.

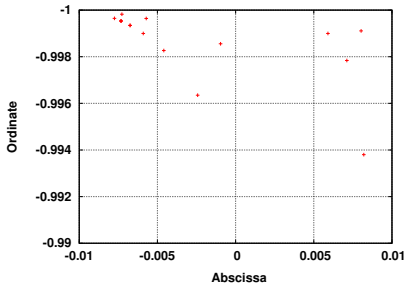

Fig. 4. Scatter plot of the positions of the servers in the neighborhood of the unit circle.

Figure 4 shows correspondingly Poincaré disk model that no address of database server belongs on the edge of the unit circle. Indeed, the addresses of database server were obtained by projection of the tree of the hyperbolic plane in a circle of the Euclidian plane of radius 1 and of center with coordinates (0; 0).

This result shows that our distributed database system can grow towards infinity in theory. In practice, other parameters such as real number precision do bring limitations.

6.2 Load Balancing in the VCH-tree

Figure 5 shows a plot of the average number of objects stored by the database servers over time. So this figure shows a regular growth of this number of data objects stored in function of time. Indeed, 293.27 data objects on average are stored by database server after 10 minutes vs 620.4 after 2 hours. It is interesting to notice that the standard deviation remains low, approximately at 10 % of the average. This indicates a low dispersal of the number of objects stored on the servers during the simulation.

Indeed, if we use our results to build the confidence interval, we can say that after 10 minutes of simulation, 68.2 % of the database servers store between 263.69 and 322.71 data objects and 95 % store between 234.18 and 352.22 against 68.2 % of the database servers which store between 560.18 and 681.58 data objects after 2 hours and 95 % of the database servers who store between 497.95

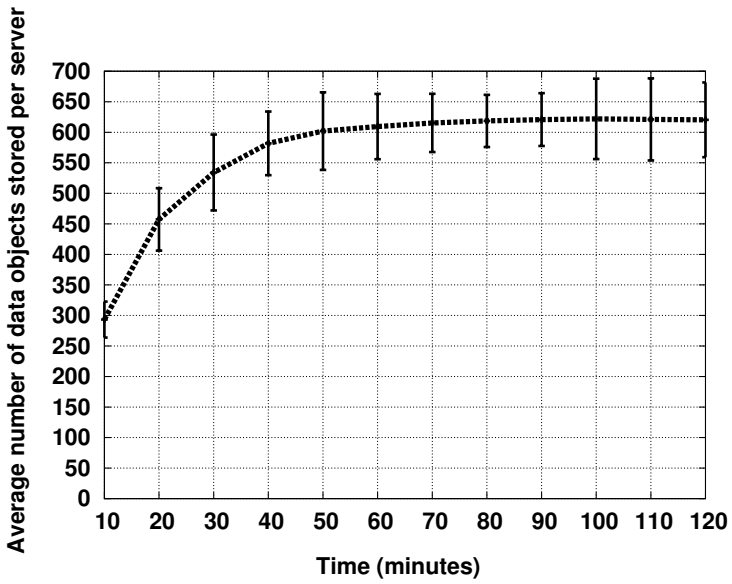

Fig. 5. Average load on the database servers over time

and 742.84 data objects after 2 hours. In view of these results, we can say that our system maintains a proper load balancing between database servers which ensures the stability of our distributed database system.

6.3 Storage and Retrieval Efficiency in the VCH-tree

Figure 6 and 7 show that during the simulation, queries in both cases can be answered within O(log N) where N is equal to the number of database servers in the system. As the standard deviation is very low (less than 5 % of the average for storage and retrieval), we did not represent it on the figures. In the worst case, queries need to travel less than 4 database servers in the system for either storing, or retrieving a data object. Besides, what is also interesting to note is that the plot decreases slowly to become stationary after around 100 minutes in both cases. It can be explained because during the simulation, the database servers create shortcuts as indicated in Section 5. These shortcuts allow to reach their target in fewer hops. The stationary situation is understandable by the fact that after a while, all the database servers reached their maximum number of shortcuts created and the most part of the queries is processed on average in less than 3.75 hops in both cases.

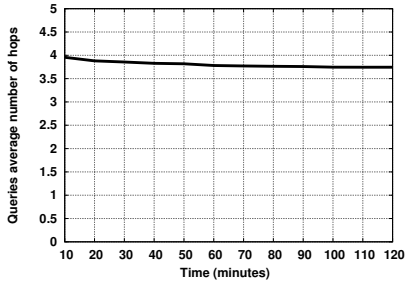

Fig. 6. Path length of storage queries

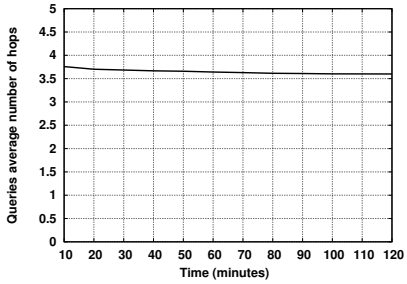

Fig. 7. Path length of retrieval queries

7 Conclusion

In this paper, we have presented a new structure called VCH-tree. This hyperbolic tree presents some properties that allow us to propose a consistent system of distributed database servers using virtual addresses made from hyperbolic coordinates. We have evaluated the performances of our system by simulation. We have shown that our system is scalable in terms of the number of database servers that can be interconnected as well as in terms of the number of hops to route the queries. We have also shown that the placement of the different database servers allows us to keep a well-balanced tree. Furthermore, we have shown that our system maintains a load balancing for the storage of data objects. For future work, we plan to study our solution comparatively to the other ones described in the state of the art in order to assess its benefits relatively to those existing solutions.

References

[1] Anderson, J.W.: Hyperbolic geometry; 2nd ed. Springer undergraduate mathematics series, Springer, Berlin (2005)

[2] Crainiceanu, A., Linga, P., Gehrke, J., Shanmugasundaram, J.: 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. pp. 25–30. WebDB '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1017074.1017082

[3] Devine, R.: Design and implementation of ddh: A distributed dynamic hashing algorithm. In: 4th International Conference on Foundations of Data Organization and Algorithms (FODO. pp. 101–114 (1993)

[4] Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. 30(2), 170–231 (Jun 1998), http://doi.acm.org/10.1145/280277.280279

[5] Guttman, A.: R-trees: A dynamic index structure for spatial searching. SIGMOD Rec. 14(2), 47–57 (Jun 1984), http://doi.acm.org/10.1145/971697.602266

[6] Hambrusch, S.E., Khokhar, A.A.: Maintaining spatial data sets in distributed-memory machines. pp. 702–707. IEEE Computer Society (1997)
[7] Idowu, S.A., Maitanmi, S.O.: Transactions- distributed database systems: Issues and challenges. IJACSCE. 2(1) (Mar 2014), http://www.sciencepublication.org/ijacsce/documents/cscevol2/5.pdf
[8] Jajodia, S., Litwin, W., Schwarz, T.J.E.: Lh*re: A scalable distributed data structure with recoverable encryption. pp. 354–361. IEEE (2010)
[9] Jansson, J., Sung, W.K.: Constructing the r* consensus tree of two trees in subcubic time. vol. 6346, pp. 573–584. Springer (2010)
[10] Karlsson, J.S.: hqt*: A scalable distributed data structure for high-performance spatial accesses. pp. 37–46 (1998)
[11] Kleinberg, R.: Geographic routing using hyperbolic space. In: INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE. pp. 1902–1909 (May 2007)
[12] Kriakov, V., Delis, A., Kollios, G.: Management of highly dynamic multidimensional data in a cluster of workstations. Lecture Notes in Computer Science, vol. 2992, pp. 748–764. Springer (2004)
[13] Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (Apr 2010), http://doi.acm.org/10.1145/1773912.1773922
[14] Litwin, W., Neimat, M.A.: k-rp*s: A scalable distributed data structure for high-performance multi-attribute access. pp. 120–131. IEEE Computer Society (1996)
[15] Montresor, A., Jelasity, M.: PeerSim: A scalable P2P simulator. In: Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09). pp. 99–100. Seattle, WA (2009)
[16] Silberschatz, A., Korth, H., Sudarshan, S.: Database Systems Concepts. McGraw-Hill, Inc., New York, NY, USA, 5 edn. (2006)