ZHT+: A Graph Database On ZHT

TongLin Li, Chaoqi Ma, Jiabao Li, Ioan Raicu
tli13@hawk.iit.edu, cma17@hawk.it.edu, jli146@hawk.iit.edu, iraicu@cs.iit.edu
Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA

***Abstract*— Although the traditional relational database has been used and dominated for many year, the limitation of it has appeared with the huge number of connected data which is generated by today's Internet, Web2.0 and social networks. The query operation will be tremendously slow to query by the traditional relational database. The state-of-the-art graph database is usually Master/Slave architecture thus these systems cannot achieve high scalability and they will become harder and harder to handle the huge number of connected data.**

In this paper we have design and implement a graph database using ZHT as a block. This graph database is also following the principle of Bulk Synchronous Parallel(BSP) model. We have overcome several problems of using ZHT as basic component in BSP model. We will give the information of our design and implementation in the following part. At last we will also give a comprehensive performance evaluation.

Keywords—ZHT; Graph Database; Graph Processing System;

I. INTRODUCTION

A graph database is a database that uses graph structures with nodes, edges, and properties to represent and store data. An example of a graph database is Neo4j. By definition, a graph database is any storage system that provides index-free adjacency. This means that every element contains a direct pointer to its adjacent element and no index lookups are necessary. General graph databases that can store any graph are distinct from specialized graph databases such as triplestores and network databases.

The traditional relational database has been used and dominated for many years,and it also works well for a long time. However, the Internet, Web2.0 and social networks will produce a huge number of data, especially the highly connected data which was tremendously slow to query by the traditional relational database. This is one problem that traditional relational database can not solve. A solution of this problem is to replace the traditional SQL semantic with a graph-centric model, thus it will be much easier for programmer to navigate these highly connected data. Graph databases are almost the best way to structure and query connected data.

Today's graph databases such as Neo4j are usually Master/ Slave architecture, the master server is easy to be the bottleneck of the whole system. Some paper also shows the other graph database such as Graphlab, the scalability of them is very poor.

ZHT is a zero-hop distributed hash table, which has been used for the high-end computing systems. It can be a building block for many de-central systems, such as graph database. We aim to design and implement a graph database using ZHT as a block and by using ZHT as a block, we can achieve high scalability, low latency and high performance.

The contributions of this paper are as follows:

- Design and implementation of ZHT+, a BSP model graph database on ZHT.
- Overcome several problem such as: Reduce both the communication times and message sizes between each nodes. Data-Locality despite ZHT-server is a separate system and doesn't provide any information about each vertexes' physical location.The bottleneck of using master-slave architecture to control supersteps.
- Benchmarks up to 16-core scales and compares with a state-of-the-art graph database system: graphlab.

II. ZHT+ DESIGN AND IMPLEMENTATION

There are so many models about Graph processing system. As we want to achieve high scalability and make ZHT+ as a de-central distributed system, so we choose to follow the principle of Bulk Synchronous Parallel(BSP) model. The process of our Graph processing system will be divided into many supersteps, and each supersteps will be divided into two sub-processes: sending messages and handle messages. And the sending messages part will also be into three sub-processes: generating messages, pre-handle messages, combining and sending messages to each nodes. We will talk about all of these processes in the following of this section.

ZHT is a zero-hop distributed hash table that can achieve high scalability, load balance and very good fault tolerance. Thus we will use ZHT as the basic components of our Graph processing system. We use ZHT to support random access, remove, update and create vertices and edges. We also use the key-value entry in ZHT to handle the communication messages between each nodes. We will also talk about more detail about this in the following of this section.

A. The BSP Model

In this part we will talk about a little information about the BSP Model in order to help reader understand the design and implementation of our Graph processing system.

The BSP model is a vertex-central model, the most important part is vertex. Each vertexes will have a statue to indicate whether it is active or inactive. In the beginning of the

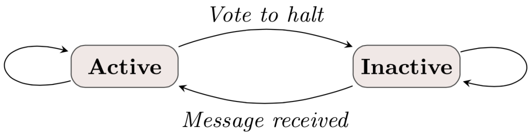


Figure 1: Vertex State Machine

process, every vertex is in the active state; all active vertices participate in the computation. A vertex will be inactive by voting to halt itself. This means that the vertex has no further work to do unless it receive an external message. If reactivated by a message, a vertex must explicitly deactivate itself again. This simple state machine is illustrated in Figure 1[14].

Figure 2[14] uses a simple example to explain the graph computing process of BSP model: given a strongly connected graph where each vertex contains a value, it propagates the largest value to every vertex. In each superstep, any vertex that has learned a larger value from its messages sends it to all its neighbors. Superstep 0 : every vertex is in the active state, Superstep 1 : all the vertices send their values to their neighbors, then the second and third one will be inactive. Superstep 2 : only first one and forth one send message to their neighbors, then the first and forth one will be inactive and third one become active again. Superstep 3 : only third one send message to its neighbors. When no further vertices change in a superstep, the algorithm terminates.

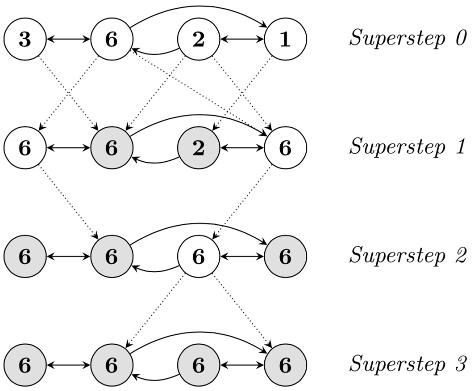


Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.

B. The Overview of ZHT+

This part is mainly about the architecture of ZHT+ and how ZHT+ applies the BSP model.

ZHT+ is consisted of three components, master node, worker node and storage server.

Since we may have billions of vertexes, we cannot simply store all the vertexes in one node. We need a storage server to storage manage and distribute all the vertexes into several nodes. The worker can only communicate with this storage server instead of handle vertexes directly. We use ZHT as the storage server of our system. ZHT will handle the load-balance fault tolerance and persistence of all the vertexes. We don't need to do any extra work about storage server part, it's all ZHT's job. In order to achieve data-locality, we will deploy ZHT and worker node in the same machine.

The worker node is the calculation unit of our system. It first load all the information of vertexes from ZHT server, then it will do some calculation depended on specific algorithm. As the BSP model shows, each vertex need to communicate with each other. If two vertexes are not in the same node, we need to handle the communication of work node. We use some special key-value entries in ZHT as message transfer station, so each worker node don't need to communicate with each directly, they only need to communicate with ZHT.

Just as we talked before, the processes of BSP model is consisted of several supersteps, since we cannot achieve a perfect load-balance, some worker nodes may still calculate its own vertexes while some other worker nodes have already finish their current job. The main job of master node is controlling the superstep, only when all worker nodes have already completed their current superstep master node will notify them to do the next superstep.

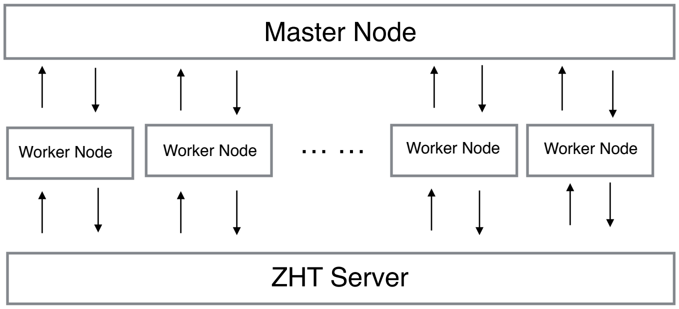
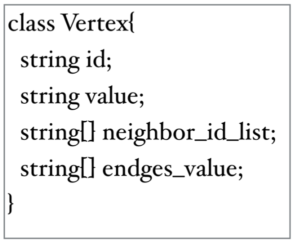


Figure 3: The ZHT+ Architecture

B.1 The Basic Data Structure

We follow the principle of the vertex-central BSP model, so we only have Vertex type in our system. The information of edges are stored in vertexes. Figure 3 shows the basic structure of vertex, however, the actual code and implementation of class Vertex is generated by Google protobuf because all the data of vertexes can be serialized by Google protobuf, thus we can store Vertexes directly in the ZHT server. Then We can simply use ZHT to support random access, remove, update and create the vertices and edges and don't need to do any extra work to handle the distribution of vertexes. It all totally depends on ZHT-server. We can simplify our design and implementation. It may be still a little hard for user to use our Graph processing system directly, so we provide a lot of APIs for user, thus they don't need to handle this complex structure directly.



```
class Vertex{
  string id;
  string value;
  string[] neighbor_id_list;
  string[] endges_value;
}
```

Figure 4: The Vertex Structure

Problem: This data structure works well when the graph is very balance (the edges of each vertexes are in a small range). However, in some situation, like facebook, a famous people may have millions of fans, and on the other hand an ordinary people may only have a little fans. If we treated these two vertexes the same way, the load-balance of worker node may be very poor.

Proposed Solution: We can treated these vertexes which contains a huge number of edges as a sub-graph. The neighbor_id_list it contains will be the virtual vertexes that contain a subset of its real neighhor_id_list. This mechanism is a little like the inode of file system.

B.2 The Main Problems Of The Graph Computing Process in Our System

There are mainly three problems we need to solve in the graph computing process part.

The general steps of our graph computing process are followed the principle of BSP model, however, because of the huge number of vertexes data, we can not store them all in one node, otherwise the process will become very slow, so we distribute them into many nodes. Obviously, when one vertex send a message to another vertex which is not in the same node, we cannot just simply send a message from one node to another, or there will be a large amount of communications between each nodes.Thus, the first problem we need to solve is how to reduce both the communication times and message sizes between each nodes.

There is also another problem we need to solve by using ZHT as basic component to manage vertexes. Since all the vertexes are stored in the ZHT server directly and the ZHT's API is very simple(*only look(), insert(), remove() and append()*). It doesn't provide any information about where the key-value entry stores physically. When we start a graph computing process, how can each node know which vertex it need to handle(ZHT is a separate system, and again it doesn't provide any information about the vertexes' location). Moreover, we also store the message list that one node need to handle in the ZHT server directly, of course, it's not wise that the message list one node need to handle is stored in the other nodes.(as we said before ZHT is a separate system, if you simply use an arbitrary key, this message list will have a very high probability stores in the other node. It totally depends on the behavior of ZHT.)

The last problem is that if one superstep is not over, we cannot begin the next superstep. Thus, we use one node as master to control the other work nodes. Since the master node need to communicate with all the other node, it may become the bottleneck of our system.

The next part of this section, we will talk about the process of our system step by step and show how we solve these three main problems:

- **Reduce both the communication times and message sizes between each nodes.**
- **Data-Locality despite ZHT-server is a separate system and doesn't provide any information about each vertexes' physical location.**
- **The bottleneck of using master-slave architecture to control supersteps.**

B.3 The Communication Between Worker Node and ZHT

Our system is totally built on the ZHT-Server, and we will store all the vertexes in the ZHT-Server directly. ZHT-Server is also a separate system just like a NoSQL database. ZHT+ will communicate with ZHT-Server by using its API. ZHT-Server doesn't provide any information about where each node are actually stored, just as showing in Figure 4, for each node(1, 2, 3), they all only know that there are 7 vertexes stored in the ZHT-server, and they don't know which node is stored in themselves,if you only use the ZHT-API to get information. for example, node 1 doesn't know vertexes(1, 2,3) are stored in the same machine.

Although the entry which stores in the ZHT-Server doesn't not contain the information about location, We are so lucky that ZHT is a open source project, thus we can use the same hash function in the ZHT to transfer each vertex's ID to the location information. So in order to get data-locality, when user insert a vertex, we add an extra step before insert this vertex into ZHT-Server. We first use the hash function to get the information where this vertex is stored. Then we will stored these information in a specific key-value entry of ZHT-Server for each node. Each node will have one specific key-value entry. These specific key-value entries's key is also generated by using the same hash function with ZHT, so information will be guaranteed stored in the same machine. So each node will know its local vertexes list simple do a look() operation with ZHT-Server.

B.4 The Loading Process of Graph Computing

Our system use a master node to invoke all the worker nodes that need to take part in the computing process. After the worker nodes are invoked, they will load the local vertexes list from ZHT-Server directly. Then depending on the specific algorithm, the vertexes will add the vertexes they need to the active vertex list. Typically, they usually add one vertex or all the local vertexes in the active vertex list. After all the nodes have already finish their job, all of them will send a message to master and said their parts of SuperStep 0 was already completed. Util master have already received all the message from worker nodes, it will send another message to tell all the worker that they can begin the first half part of SuperStep 1.

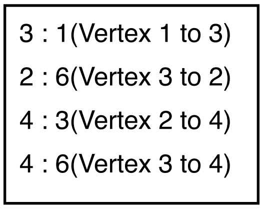
Problem: This part is not as simple as it looked, actually it will cost a lot time to load all the information of vertexes into memory, sometimes even more than the computing time.

Proposed Solution: Maybe, we can do some extra work when we load these information, for example, we can group some small vertexes into a big virtual vertex in order to reduce the size of vertexes.

B.5 The first step of one SuperStep: Sending Message

We have divided one SuperStep into two sub-steps, the first sub-step is Sending Message. This sub-step is also divided into three sub-process. We will talk about them one by one in this section.

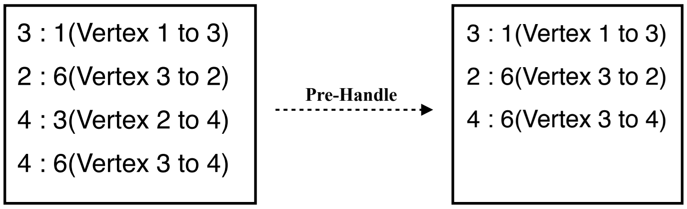
We first traverse the activeVertex list in each node, and then we can get all the information of the vertexes directly from ZHT-Server by using lookup() method. Then we traverse the activeVertex list in each node and generate the messages (target_id : value) that each vertex will send depending on specific algorithm. If a graph contains 2 millions edges, we will have 2 millions messages in PageRank algorithm each step. Obviously, we can not send them immediately. So we just store them in the data structure in each node. The example messages node1 will send are shown in Figure 5.



3 : 1(Vertex 1 to 3)
2 : 6(Vertex 3 to 2)
4 : 3(Vertex 2 to 4)
4 : 6(Vertex 3 to 4)

Figure 5: Node1's Sending Messages

Like I talk before, a graph contain 2 millions edges will generate 2 millions messages. The size of message is very large, in order to solve this problem, we provide a simple and novel approach. After all the message is generated in each node , we will traverse the sendingMessage list in each node and pre-handle these messages depend on specific algorithm. the pre-handle process is almost the same with the handle message process. This approach is shown in Figure 6. We can see that there are two messages sending to Vertex 4 which is located in another node. However, after compare the value of these two message only the last message is useful, thus we simply discard the first one. By using this approach we can reduce the size of messages.



3 : 1(Vertex 1 to 3)
2 : 6(Vertex 3 to 2)
4 : 3(Vertex 2 to 4)
4 : 6(Vertex 3 to 4)

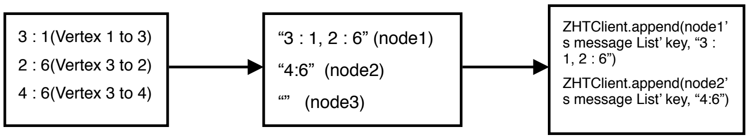
3 : 1(Vertex 1 to 3)
2 : 6(Vertex 3 to 2)
4 : 6(Vertex 3 to 4)

Figure 6: Pre-Handle Message

Problem: the efficiency of this message reduce approach is highly depends on the type of workload. If the size of edges is far more than the size size of vertexes, this approach works pretty well, it can reduce the size of messages to one tenth. However, in some situation, if the size of edges is similar with the size of vertexes or even less than the size of vertexes. Our approach works very poor.

Proposed Solution: as I talked before, we can group many vertexes into a big virtual vertex, thus we can reduce both the size of vertexes and the size of messages.

The next step we will combine all of these large number of messages into one big message. Since we need to send message to several nodes, we need to know where is the location of target vertex. This information is also can be generated by using the same hash function with ZHT. Then we can send to each node one big message instead of a huge number of little message. The implementation of sending message is very simple, we benefit a lot from using ZHT. We just need to use the append API of ZHT. This process is shown in Figure 7. After all these part have down, each node will send a message to master. And then master will rely a message to them. They can begin the next step.



3 : 1(Vertex 1 to 3)
2 : 6(Vertex 3 to 2)
4 : 6(Vertex 3 to 4)

"3 : 1, 2 : 6" (node1)
"4:6" (node2)
""   (node3)

ZHTClient.append(node1's message List' key, "3 : 1, 2 : 6")
ZHTClient.append(node2's message List' key, "4:6")

Figure 7: The Process of Combine and Sending Messages

B.6 The last step of one SuperStep: Handling Message

After a worker received a message about Handling Message from master, the worker node will begin to handle message. Each node will load the message information directly from ZHT-Server.Then the nodes calculate the information from the messages. They will update the value of vertexes and add the active vertexes into its activeVertexes List. The key point of this section is that all the processes are locally there is no network communication even include ZHT-Server. Although the messages are stored in the ZHT Server, by using the same hash function, we can generate some special key. All the messages are guaranteed to store in the same node that will handle them.

B.7 The Control of SuperStep

Just as previous parts said, if one superstep is not end, we can not process the next. Ideally, since all the nodes are the same and we can achieve the perfect load balance, thus the computation time of each node will almost the same. However, in fact we cannot achieve the perfect load-balance, some nodes may handle more vertexes and some may handle less. Thus, some node may end, but some other may still calculate, so we need an approach to control superstep. Only all the node finish current work, we will process the next superstep.

The simplest approach to solve this problem is using master -slave architecture. In each SuperStep, master will send a message to all the workers and when the worker have completed its current job, it will reply a message to master. After master have already receive all the messages from all the workers. It will send another message to tall them to process the next step.

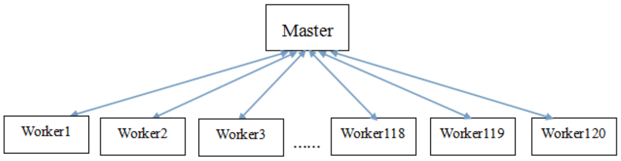


Figure 8: A Simple Master-Slave Approach

However, Since the master node need to communicate with all the other node, if we have thousands of nodes, it may become the bottleneck of our system. We need another approach to solve this problem.

Inspired by the broadcast approach, we provide our own broadcast method to solve this problem.We divide all the nodes into 5 cycles(user can define the size of each cycle by themselves). The message will spread through each cycle, just like a broadcast. This is also shown in Figure. Thus We don't add any more communication, and there is also no bottleneck anymore. If one node is failed, its upper worker node can not communicate with it, then its upper worker node will send a message to notify master node. After master received this message, it will notify all the worker nodes in this particular cycle and restart them from the beginning of this superstep again.

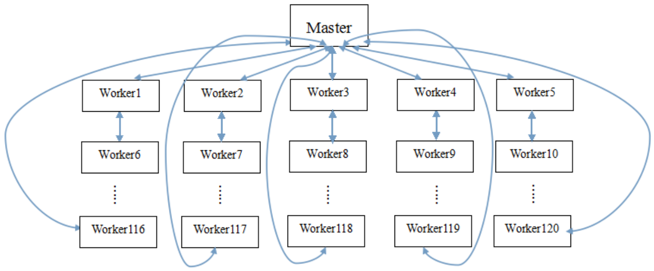


Figure 9: The Broadcast Method

C. *Load balance*

As we talked before, we stored the vertexes directly in the ZHT-server. When a node begins to work, it will handle the vertexes that are stored in the same node. Thus the load balance of our system is highly depended on ZHT-server, we don't need to do any extra work. Our system can achieve the same load balance as ZHT.

D. *Fault Tolerance*

This part also benefit a lot from using ZHT as basic component. As we talked before, we store all the message list, active vertex list and any information we need in Graph Computing Process in the ZHT server. These information can also be treated as checkpoints in each superstep. If one node is failed, we can simply read these informations from ZHT and restart this superstep again. We don't need to restart form the first step. It's very powerful when the data size is very big, if you restart all the step, it will waste a lot of time.

E. *Persistence*

We also don't need to do any extra work, all the information and vertexes (edge is an attribute of vertex) are stored in ZHT-server directly. This is also another big benefit we can get from ZHT.

F. *Graph Algorithms with ZHT+ Framework*

We have implemented three important graph algorithms by using our ZHT+ framework. We want to use them as an example of how to use our framework. Users can also write their own graph algorithms with ZHT+ framework. We will talk about the detail of each algorithm in this section.

Single Source Shortest Path Algorithm:The system will initialize all the value of vertexes to Infinite, and we let all the vertexes to be inactive. Then the system will send a message which the value is 0 to the source vertex. The source vertex will update its value to 0 and become active. It will send message (vertex's value plus edge's value) to all its neighbors and awake them; the neighbors will send messages to all their neighbors. If no update is done, the vertex will become inactive again. The system will keep doing this util no vertex is active.

PageRank Algorithm:The system initialize all the PageRank values of all the vertexes to 1/numberOfVertexes; Then, all the vertexes will send message (PageRankValue / numberOfNeighbors) to all their neighbors, then each vertexes will sum up the values it receives from messages; after that it will set its PageRank value to 0.15 / numberOfVertexes + 0.85 * sum. In practice, repeating this process 30 times, the statues of all the vertexes will be set inactive, and we will get the PageRank value we want.

Weakly Connected Components Algorithm:All vertices are initially active. Each vertex starts as its own component by setting its component ID to its vertex ID. When a vertex receives a smaller component ID, it updates its vertex value with the received ID and propagates that ID to its neighbours.

G. *Implementation*

ZHT+ has been under development for about 2 month. It is also implemented in C/C++ because we want to reuse some code in ZHT, and it has the same dependencies with ZHT. ZHT+ consists of around 3100 lines of code. The dependencies of ZHT+ are also NoVoHT and Google Protocol Buffer[22]. NoVoHT itself has no dependencies other than a modern gcc compiler.

III. EVALUATION

In this section, we describe the performance of ZHT+, including the three graph algorithms-PageRank Algorithm, Single Source Shortest Path Algorithm(SSSP), Weakly Connected Components Algorithm(WCC). Firstly we'll introduce the test beds and the benchmark metrics. Secondly the performance evaluation will be presented.

A. *Testbeds, Metrics, and Workloads*

We run experiments on setups of 2, 4, 8, and 16 machines. All machines are m3.2large Amazon EC2 spot instances, located in us-west-2c. Each instance has four virtual CPUs, equivalent to 2.5 GHz Xeon Family processors, and 30GB of memory.

The dataset we use in the evaluation are downloaded from SNAP (Stanford Network Analysis Project). It's an open source dataset from the real world. I think it's a good work-load to test our system.

The metrics measured and reported is the insert time, loading time, computation time, total time, workload distribution. These are all the most important metrics of a graph processing system.

Insert Time: the total time we insert all the nodes into ZHT+.
Loading Time: The time each worker node load all the vertexes into memory.
Computation Time: Total time from the beginning of superstep 0 to the last superstep
Total Time: Loading Time plus Computation Time, this is the whole time we run a graph computing process.
Workload Distribution: The vertexes each worker node need to handle.

By using the real-world dataset and various number of machines enable us to investigate the properties of our graph database.

Figure 12: PageRank Computation Time

B. *Insert Time*

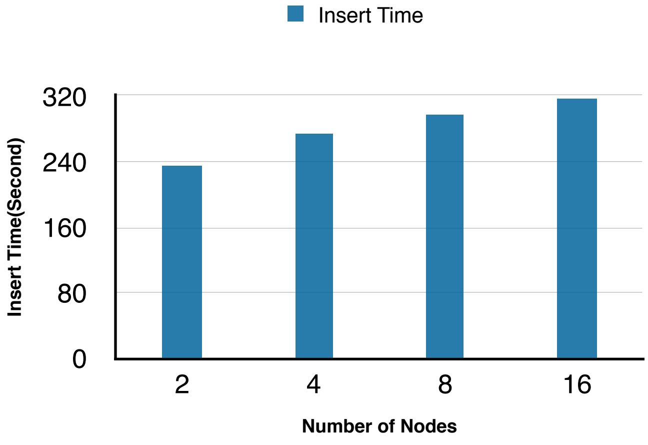


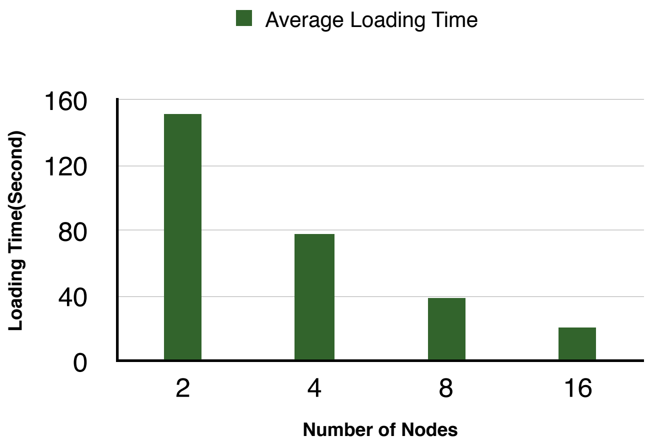
Figure 10: The Insert Time

We evaluated the insert time by inserting the same dataset(web-Google which contains 1 million vertexes and 5 million edges) into various scales of nodes size which is from 2 to 16. The result is showed in the figure 10.

As the result showing, with the increase of scales of nodes, the insert time is also becoming bigger and bigger. Although we inserted the same dataset into them, we use ZHT as our storage server and if we deployed ZHT on more nodes, the communication times between each nodes will also increased. However, the increase is not linear, we only add some extra network communication, so we think these overhead is affordable.

C. *Loading Time*

We evaluated the loading time by starting all the worker nodes and calculate the average loading time in different scales(form 2 to 16). The dataset is also the same(web-Google which contains 1 million vertexes and 5 million edges). The results is showed in the figure 11.



The increase of loading time is almost linear because each worker node only need to load its local vertexes, they don't need to communicate with a remote node and the load balance of ZHT is very good.

Although the loading time table looks pretty good, there still something else we should be careful. We cost so much time on loading process, in some situation(like 4 nodes) the loading time is even bigger than computing time. According to Han's[13] paper, other graph computing systems(for example Giraph, Graphlab) also has same problem. But the computing time of Graphlab is incredible, we believe that we can do some extra operation in loading step that can reduce the computation time.

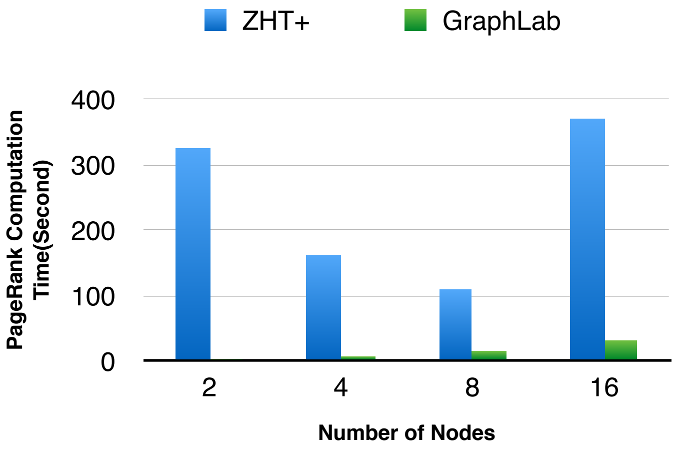
D. *PageRank Computation Time*

PageRank algorithm will use all the vertexes and edges in every superstep. Thus this is the best algorithm to test Data-Locality and load balance.

We evaluated PageRank Computation Time by comparing GraphLab with ZHT+. We tested them in the same testbed(Amazon EC2) and the same dataset(web-Google). The result of this experiment is shown in Figure 12.

The result of this experiment is very interesting. We will talk about it one by one. The first one is that we almost can not see the computation time of GraphLab. I first think there may be something wrong with my experiment. I checked my results with Han's[13] paper, they also get almost the same result with me. GraphLab's computation time is really really incredible. We believe that they may be doing some extra step in the loading part. And according to that paper our system is about 5 times slower than the other Graph database systems(Giraph and GPS) because the testbeds contains 8 CPUS in each nodes, and we only utilized one of them. In the future, if we can write our computing program concurrently,

Figure 11: The Average Loading Time



we can achieve the same performance with other system (Giraph and GPS).

The last thing we want to talk about in this experiment. Both ZHT+ and GraphLab will be slowest when we use 16 nodes to run this experiment. For GraphLab, the more nodes the less performance it can achieve. For ZHT+, when we have 8 nodes ,we will achieve the highest performance, and the increase is not linear. This is mainly because the workloads is too small. It only contains around 1 million vertexes and 5 million edges. Actually the workloads of graph computing system is usually very small, we can only use 32 nodes to satisfy almost every situation. Neo4j is also a good example of this opinion. It only support one node, but it is the most widely use graph database. Because in most situation one node is sufficient, however we believe in the future in some science computation program we may need to handle 1000 billions vertexes, so we still need a good distributed graph database.

E. PageRank Total Time

We add the loading time with the PageRank's computation time, then we can get the total running time of PageRank algorithm. The result is shown in Figure 13.

This table is a good example to show that the loading time have a huge effect to the total running time. In some situation(2 nodes scale), it may occupy half of the total time. Even in the other situation loading step is also costed a lot of time.

F. SSSP & WCC Computation Time

Single source shortest path(SSSP) is the simplest algorithm that tests how well a system handles dynamically changing communication.Weakly Connected Components Algorithm(WCC) unlikes SSSP, all vertices are initially active and unlikes PageRank,some vertices can halt before others.

We also evaluated these two algorithms by using the same dataset and up to 16 nodes. The results of these two algorithms are shown in Figure 14 and Figure 15.

The results are also very funny, we use more nodes and we get slower performance. There are mainly two reasons can explain these. First the workloads is very small, the largest sssp is only 16, most of our time is spent on network communication. Second, we random choose one vertex, maybe in this specific case, we only need a little step that can calculate the sssp, but it's also impractical to calculate all the vertexes' shortest path. The WCC algorithm is also the same reason, the workload is too small. It's really hard to generate a

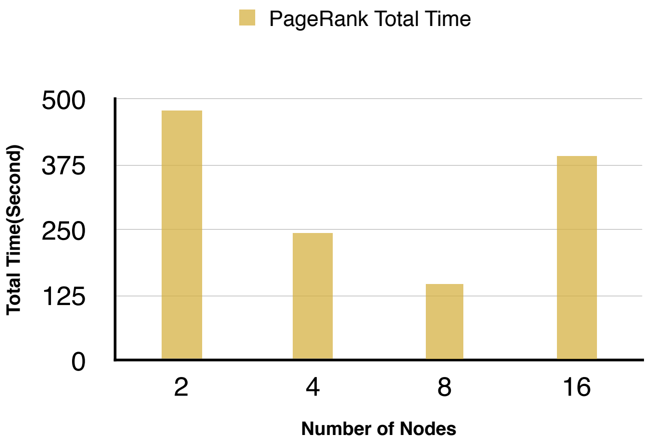


Figure 13: PageRank Total Time

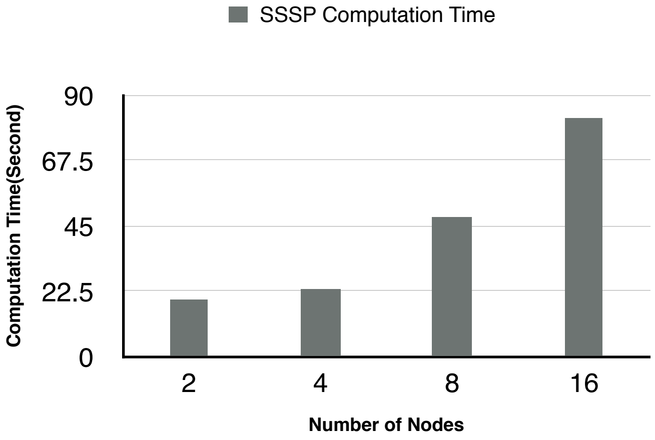


Figure 14: SSSP Computation Time

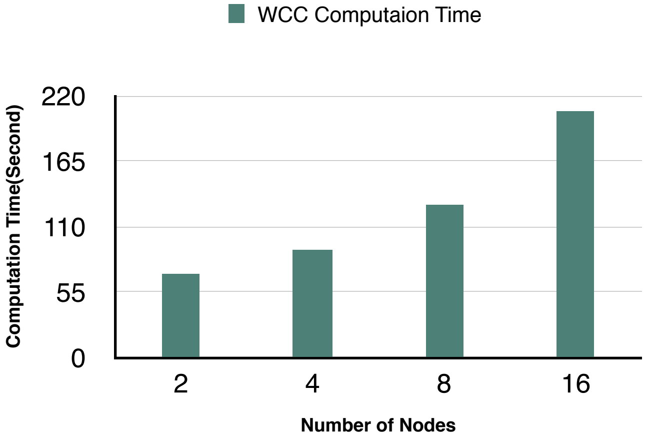


Figure 15: WCC Computation Time

large enough workload in graph computing system.

G. Workload Distribution

This section we will talk about the workload distribution, we first calculate the average vertexes size that each node need to handle. Then we will give the scope of vertexes size(The purple line in Figure 16). The result is shown in Figure 16.

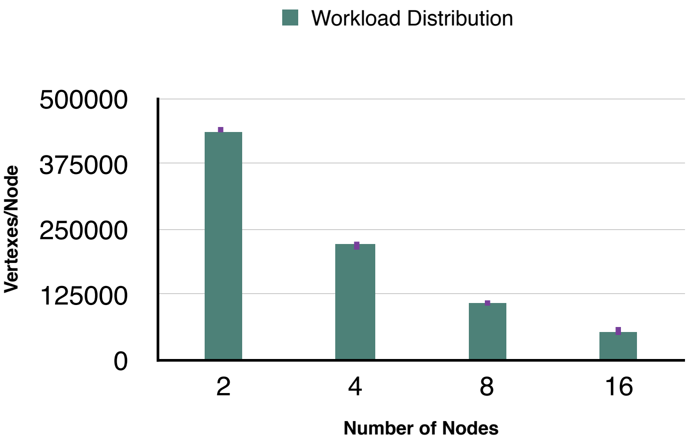


Figure 16: Workload Distribution

From the result you can see the purple line is very small, that means the number of vertexes each worker node need to handle is almost the same. This is because we use ZHT to handle load balance and the load balance of ZHT is very good.

IV. Conclusion

In this project, we design and implement a Graph database ZHT+ based ZHT. We have overcome several problems of using ZHT as the basic components and applying BSP Model. Even in such a short time the performance of our system is already very good.

We have also compare ZHT+ with one state-of-the-art graph database, graphlab and presented a very detailed evaluation.

At last in this project, we have learned a lot about graph database and distributed hash table(especially ZHT), and we have also learned a lot of algorithms used in graph area, such as PageRank, SSSP, BFS and DFS.

V. Future Work

We have a lot of ideas to improve the performance of our ZHT+ system.

Unbalanced workload: this is an interesting topic because the open source datasets are usually very balanced, however at the real world, in some situation like we talked before, the balance is very poor. We believe that if we can solve this problem, there will be a huge improvement of the performance of our system.

Pre-handle in Loading Process: graphlab shows eincredible performance of computing, if we can find some better method to pre-handle in the loading process, our system's performance will be much better.

Divide and Group Vertexes: this is also an interesting approach, in the future, we want to see whether this approach will work.

VI. Related Work

There are so many graph databases in the world, and the three most important graph databases: Neo4j, Giraph and GraphLab.

Neo4j is an open-source graph database, implemented in Java. It is an embedded, disk-based, full transaction supported Java persistence engine, and it stores data in graph instead of table. From version 2.0 which was released in December, 2013, it no longer supports node indexing.

Apache Giraph is an Apache project which leverages Apache Hadoop's MapReduce to handle graphs. It is used to perform graph processing on big data. The original design of Apache Giraph is from the paper "Pregel: a system for large-scale graph processing." which is published by Google. Facebook also used Apache Giraph to analyze one trillion edges. This can be done in only 4 minutes by 200 machines.

GraphLab is a graph-based, high performance, distributed computation framework which was written in C++.The GraphLab project started by Prof. Carlos Guestrin of Carnegie Mellon University in 2009. It is an open source project using Apache License. While GraphLab was originally developed for Machine Learning tasks, it has found great success at a broad range of other data-mining tasks; out-performing other abstractions by orders of magnitude.

VII. References

[1] Angles, Renzo, and Claudio Gutierrez. "Survey of graph database models." ACM Computing Surveys (CSUR) 40, no. 1 (2008): 1.

[2] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in Proceedings of IEEE International Symposium on Parallel and Distributed Processing, 2013.

[3] Dominguez-Sal, David, P. Urbón-Bayes, Aleix Giménez-Vañó, Sergio Gómez-Villamor, Norbert Martínez-Bazan, and Josep-Lluis Larriba-Pey. "Survey of graph database performance on the hpc scalable graph analysis benchmark." In Web-Age Information Management, pp. 37-48. Springer Berlin Heidelberg, 2010.

[4] Angles, Renzo. "A comparison of current graph database models." In Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on, pp. 171-177. IEEE, 2012.

[5] Batra, Shalini, and Charu Tyagi. "Comparative analysis of relational and graph databases." International Journal of Soft Computing and Engineering (IJSCE) 2, no. 2 (2012): 509-512.

[6] Ho, Li-Yung, Jan-Jan Wu, and Pangfeng Liu. "Distributed graph database for large-scale social computing." In Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, pp. 455-462. IEEE, 2012.

[7] He, Huahai, and Ambuj K. Singh. "Graphs-at-a-time: query language and access methods for graph databases." In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp. 405-418. ACM, 2008.

[8] Jiang, Haoliang, Haixun Wang, P. S. Yu, and Shuigeng Zhou. "Gstring: A novel approach for efficient search in graph databases." In Data

Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, pp. 566-575. IEEE, 2007.

[9] Robinson, Ian, Jim Webber, and Emil Eifrem. Graph databases. " O'Reilly Media, Inc.", 2013.

[10] Neo4j http://www.neo4j.com/

[11] Giraph http://giraph.apache.org/

[12] GraphLab https://dato.com/products/create/open_source.html

[13] Han, Minyang, Khuzaima Daudjee, Khaled Ammar, M. Tamer Ozsu, Xingfang Wang, and Tianqi Jin. "An experimental comparison of pregel-like graph processing systems." Proceedings of the VLDB Endowment 7, no. 12 (2014): 1047-1058.

[14] Malewicz, Grzegorz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing." In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135-146. ACM, 2010.

[15] Bollacker, Kurt, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. "Freebase: a collaboratively created graph database for structuring human knowledge." In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp. 1247-1250. ACM, 2008.

[16] Williams, David W., Jun Huan, and Wei Wang. "Graph database indexing using structured graph decomposition." In Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, pp. 976-985. IEEE, 2007.

[17] Vicknair, Chad, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. "A comparison of a graph database and a relational database: a data provenance perspective." In Proceedings of the 48th annual Southeast regional conference, p. 42. ACM, 2010.

[18] Zou, Lei, Lei Chen, and M. Tamer Özsu. "Distance-join: Pattern match query in a large graph database." Proceedings of the VLDB Endowment 2, no. 1 (2009): 886-897.

[19] Graves, Mark, Ellen R. Bergeman, and Charles B. Lawrence. "Graph database systems." Engineering in Medicine and Biology Magazine, IEEE 14, no. 6 (1995): 737-745.

[20] Dominguez-Sal, David, Norbert Martinez-Bazan, Victor Muntes-Mulero, Pere Baleta, and Josep Lluis Larriba-Pey. "A discussion on the design of graph database benchmarks." In Performance Evaluation, Measurement and Characterization of Complex Systems, pp. 25-40. Springer Berlin Heidelberg, 2011.

[21] Yan, Xifeng, and Jiawei Han. "gspan: Graph-based substructure pattern mining." In Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on, pp. 721-724. IEEE, 2002.

[22] Google Protocol Buffers: http://code.google.com/apis/protocolbuffers/2012