ZHT+ : Design and Implementation of a Graph Database Using ZHT

Gagan Munisiddha Gowda
Benjamin L. Miwa
Anirudh Sunkineni
Department of Computer Science
Illinois Institute of Technology
Chicago, IL

Abstract— **Graph databases use the concepts of nodes, edges, and properties to represent and store data. Graph databases provide index-free adjacency, with every element containing a direct pointer to its adjacent element, removing the need for index lookups. As they depend less on a rigid schema, they are more suitable to manage ad hoc and changing data with evolving schemas.**

ZHT is a zero-hop distributed hash table Key-Value Store (KVS). It is designed to handle the requirements of high-end computing systems, providing rapid, decentralized access to key/value pairs.

In this project we built a limited implementation of a graph database - ZHT+, which leveraged the ZHT KVS. We then ran benchmarks including Depth First Search (DFS), Breadth First Search (BFS), and PageRank on a standard data set with ZHT+ and Neo4j, as well as limited testing on Giraph and GraphLab, to compare the performance of ZHT+ on various configurations of up to 50 virtual machines (VM) on Amazon EC2.

At the scales we tested, ZHT+ handled up to 16 clients with no degradation in performance. With Neo4j there was some degradation in performance as the number of clients was increased. However, since both systems are meant for high-end computing systems with thousands of clients, further testing will be needed at much larger scales.

Keywords—ZHT; zero-hop; graph database

I. Background Information

Graph databases provide a different way of storing information when compared to the traditional relational database management systems (RDBMS) or even some NoSQL databases. These databases need to store the data in a distributed fashion while exposing a unified API to perform queries.

Compared with relational databases, graph databases are often faster for associative data sets, and map more directly to the structure of object-oriented applications. They can scale more naturally to large data sets as they do not typically require expensive join operations.

There are well known graph databases available like Neo4j, Giraph, GraphLab, Allegro, and GraphBase. All of them provide similar features with few distinctive features over others. All of these databases use their own underlying KVS which provides considerable performance benefits.

It is clear in the performance evaluation of ZHT [1][2] that it fares better than other KVSs, especially at large scales. We have leveraged the advantages ZHT provides, like fault-tolerance, high-performance, and optimization for high performance computing, to develop the ZHT+ graph database to provide a distributed graph database especially suited to high performance computing at very large scales.

II. Problem Statement

Graph databases generally use a KVS to store node and edge information. The performance of the graph database is therefore directly dependent on the speed of that KVS, especially at large scales. Currently, no graph database has been implemented to take advantage of ZHT's performance advantages.

The contributions of this paper are as follows:

- Design and implementation of ZHT+, a graph database built on top of ZHT, a light-weight, high performance, fault tolerance, persistent, dynamic and highly scalable distributed hash table, optimized for high-end computing.
- Support for graph specific operations like add and get for nodes, edges, node properties and edge properties.
- Benchmarks using DFS, BFS, and PageRank on up to 49 server nodes and up to 16 concurrent clients.
- Evaluation and comparison with commercial graph databases like Neo4j, GraphLab and Giraph.

III. Related Work

There are several existing graph databases such as Neo4j, Giraph, GraphLab, Allegro, GraphBase and there have been previous experiments to benchmark such databases [3][4][5]. We leveraged this previous work in making design choices and attempted to re-use tools for the benchmarking experiments.

However, the largest portion of related work is ZHT itself. We used this fully functional KVS system to implement our graph database system. As such, we did not implement any of the work related to the actual storage and retrieval of data.

IV. ZHT+ Design and Implementation

We leveraged the work done previously with ZHT in order to build a graph based storage system on top of its distributed key-value store. Given that ZHT already handles the distributed storage and offers excellent availability and fault tolerance along with minimal latencies, we explored how to best represent a graph by using ZHT and take advantage of the design considerations made in ZHT.

A. Scope Limitations

We did not develop a full-fledged graph database with features to support a query language, transactions, runtime failover, monitoring, etc. The main goal of this project was to implement a basic graph database that is able to store data in the form of a graph (nodes and relationships) by using ZHT and implement some basic algorithms like DFS, BFS and PageRank to see how well it fares in comparison to other graph databases, purely in terms of the execution time of these algorithms.

ZHT is designed to target large distributed systems of thousands of nodes or more with similar numbers of clients. For our project we were limited to a total of 50 nodes, distributed in various ways between client and server nodes. This means that in many cases we were unable to load the system sufficiently to measure performance degradation.

B. Design Considerations

Graph databases need to store nodes, edges, and properties for them. We considered different ways to store these - separate ZHT KVS's for each, separate entries in a single ZHT KVS, or as a compound object with a single entry in one ZHT KVS. Since the goal was to maximize performance, we decided that minimizing the number of trips across the network would be the primary goal. A network transaction would be in the msec range while serialization/deserialization would be in the usec range.

The second major design decision was how to store edges. And edge goes from node A to node B. This can be store in either node A, node B, or both. Depending on the usage of the graph database, any of the three options could be appropriate. For quickly traversing a graph, storing the edge in the source node A is important. For quickly determining what other nodes point to a node, storing the edge in the target node B is important. For easy removal of nodes and associated links, storing the edge in both nodes would be easiest. Since our benchmarks consisted of inserting nodes and edges and traversing the trees, storing the edges in only the source node A was sufficient.

C. Protocol Buffer

To store multiple elements in a single key/value pair, we decided to use Google's Protocol Buffer [6] to serialize the data. Protocol buffers allow the developer to specify the structure in a way similar to C++, Java, and other structured languages. These structures - messages - are then compiled into language specific code that allows serialization/deserialization and simple access to the elements. Messages are composed of elements that are required, optional, or repeated and can be basic types or other messages types.

We created three message types - Node, Edge, and Property. A Node consists of an optional nodeID of type string, an optional name of type string, a repeated edge_source of type Edge, a repeated edge_target of type Edge, and a repeated property of type Property. An Edge consists of an optional edgeID of type string, an optional name of type string, an optional source of type string, an optional target of type string, optional directed of type bool, and a repeated property of type Property. A Property consists of an optional propertyID of type string, an optional name of type string, an optional value of type string.

Some of these elements were redundant or for possible future use. The node ID is the key for the ZHT KVS, but we also stored it in the Node nodeID. While we supported both a nodeID and a name, for testing we used the same string for both. We did the same thing with the edgeID and name and the propertyID and name. Since we were working exclusively with directed graphs, we always created nodes with Edge.directed set to true. As previously noted, we did not implement node removal, so the Node.edge_target element was not used.

D. Private Methods

Low level private methods were implemented to serialize/serialize the protocol buffers and to access the various elements of the structure using ZHT's insert, lookup, append, and remove API's and the methods supported by Google Protocol Buffers. These were then wrapped in the public APIs of the ZHTplusClient class.

For repeated elements in protocol buffers, direct access to specific elements is only supported based on the order of insertion. Therefore to get the value of a specific element, the list of elements must be traversed until the desired element is found. While this may not be the most desirable method for random access, traversal of the entire graph consists of visiting every element in the list, so there is no performance cost for this method of access.

Likewise, removal of only the last element in the list is supported. Thus removal of a random element could only be implemented by swapping the item to be removed with the last element and then removing it from the end of the list. This operation was not implemented as part of our project as the removal of nodes, edges, and properties was not required.

E. Public APIs

In addition to the constructor and destructor for the class, we implemented the following basic methods: addNode, addNodeProperty, addNodeEdge, addNodeEdgeProperty, getNodePropertyValue, getNodeEdgeTarget, and getNodeEdgePropertyValue. These allowed us to create the required benchmarks to insert and traverse a graph. They would also provide the basis for searching the graph based on node or edge properties.

Remove methods were not implemented for this project for two reasons. At the low level, removing properties and edges from repeated elements in protocol buffers is not easy. But at a higher level, removing nodes requires one of two strategies.

The first strategy is to simply remove the node and any edges that originate from it. This could be easily accomplished by simply deleting the node from the ZHT KVS. However, this would leave any edges that point to that node hanging. Without the Node.edge_target there would be no way to find these edges stored as part of the nodes pointing to it. Thus traversing an edge would require verifying that the target node still exists and optimally, removing the edge if it did not. Node removal, edge insertion, and edge removal would all be single node operations, but edge traversal would be slower.

The second strategy is to store an edge in both the source and target nodes. This would require modifying two nodes when inserting an edge, doubling the time for edge insertion. Edge removal would also require modifying two nodes. Removing a node would require modifying all the nodes with edges pointing to the removed node. This would mean that all remaining edges would be valid, but there could be a significant performance penalty at node removal time.

The choice between these two methods depends on the specific use case for a particular graph database instance.

F. *Traversal Algorithms*

Three algorithms were implemented - DFS, BFS, and PageRank. These were used to benchmark ZHT+ against other graph databases. Well documented designs exist for all three of these algorithms, so an appropriate design was selected based on our ZHT+ implementation. All three were implemented as single-threaded functions but using private methods for best performance.

DFS was implemented as a recursive function with a start node specified. The result was the total number of nodes visited, the number of unique nodes visited, and the amount of time required to complete the traversal. The recursive function incremented the nodes visited count and returned immediately if the node had already been visited. Otherwise, the node was marked as visited using a hash table, the unique nodes visited count was incremented, the edges were retrieved, and the recursive function was called for each of the nodes pointed to by the edges.

BFS was implemented using a queue with the start node push onto the queue. The result was the total number of nodes visited, the number of unique nodes visited, and the amount of time required to complete the traversal. While the queue was not empty, a node was popped from the queue, the nodes visited count was incremented, and if the node had already been visited, the queue was checked for another node. Otherwise, the node was marked as visited using a hash table, the unique nodes visited count was incremented, the edges were retrieved, and each of the nodes pointed to by the edges was pushed onto the queue.

PageRank is an iterative algorithm requiring multiple passes through the graph beginning from a start node. While the same counts and time were produced, the result also included the top twenty pages ordered by pagerank. Usually the iterations would continue until the desired top number of results no longer change. However, since we did not know how many iterations this would take, we limited our passes to ten. The code could easily be modified to continue until the top results no longer changed. The algorithm consists of three parts: 1 - traversal and initialization, 2 - traversal and calculation, 3 - top pagerank selection and sorting. Part one, which was done only once, was a simple BFS with each node's ID inserted into a map with its value initialized to 1.0. Part two, repeated each iteration, was a BFS with each node's previous value distributed the the new value for each of the nodes pointed to by its edges. Part three iterated through the map collecting and sorting the top results. We chose twenty for our benchmark, but any number up to the total number of nodes could be sorted and displayed. Part three could either be included on every iteration to display the progressive calculation, or after all the part two iterations were completed to display only the final results.

V. Evaluation

In this section, we will describe the performance of ZHT+ through latencies and throughput of general ZHT+ operations followed by latencies for DFS, BFS and PageRank algorithms. Firstly, we will introduce the testbed and benchmark configuration. Secondly a comprehensive performance evaluation will be presented. We will compare ZHT+ with Neo4j, a graph database offering similar features followed by GraphLab and Giraph, more advanced graph processing frameworks.

A. *Testbed, Metrics and Workload*

We used Amazon AWS c3.large EC2 instances powered by 2 vCPUs and 3.75 GB of memory. We have evaluated ZHT+ at different scales of server instances (upto 49) and also at different scales of concurrent clients (upto 16).

We have created cluster setup and execution scripts which uses AWS API along with IAM profiles which can be used to setup ZHT+ at different scales and collect evaluation results from multiple clients.

The basic operations ZHT+ supports include addNode, addEdge, addNodeProperty, addEdgeProperty, lookupNodeProperty and lookupEdgeProperty. One each node, one of the ZHT server-client pairs are deployed, namely ZHT instances. Clients sequentially send all the requests through a ZHT+ Client API for the operations mentioned above.

The dataset used for evaluation is LiveJournal Social Network [12].

The metrics measured and reported are:

- Latency: The time taken for a request to be submitted from a client and a response to be received by the client, measured in milliseconds. Since the latencies of various operations (insert/lookup/remove) are fairly close, we use average of the three operations to simplify results presentation. Note that the latency includes the round trip network communication and storage access time.
- Throughput: The number of operations (insert/lookup/remove) the system can handle over some period of time, measured in Kilo Ops per second/s.
- Ideal Throughput: Measured throughput between two nodes times the number of nodes.
- Efficiency: Ratio between measured throughput and ideal throughput.

B. *Latencies*

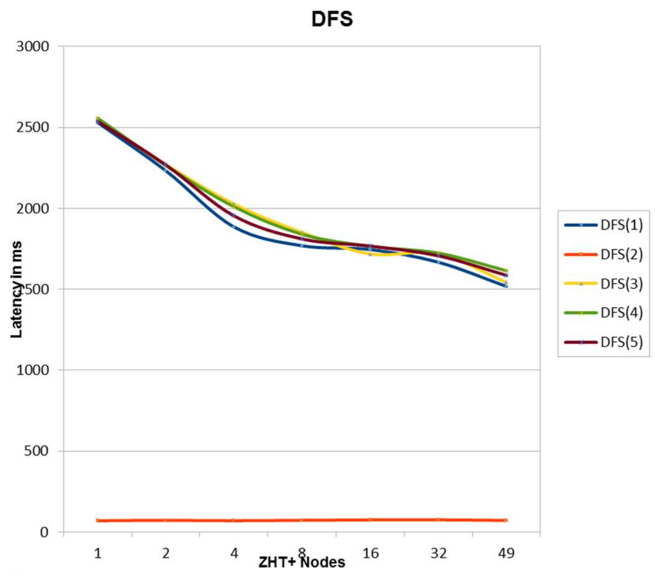

Figure 1 – DFS Latency

We ran DFS algorithm multiple times starting from different start nodes. DFS(1), DFS(3), DFS(4) and DFS(5) traversed through 2729 nodes each whereas DFS(2) traversed through 98 nodes. Figure 1 shows that the latency is around 2.5s at 1-node scale and drops down to 1.5s at 49-node scale. Looking at the trend we should see very less latency at very large scales indicating the high performance nature of underlying ZHT key-value store.

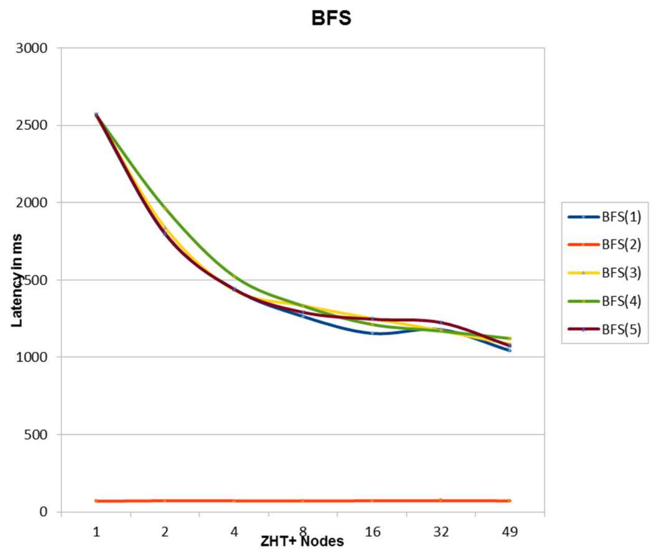

Figure 2 – BFS Latency

Similarly, we also ran BFS algorithm multiple times starting from different start nodes. Figure 2 shows that the latency is around 2.5s at 1-node scale and drops down to 1.1s at 49-node scale.

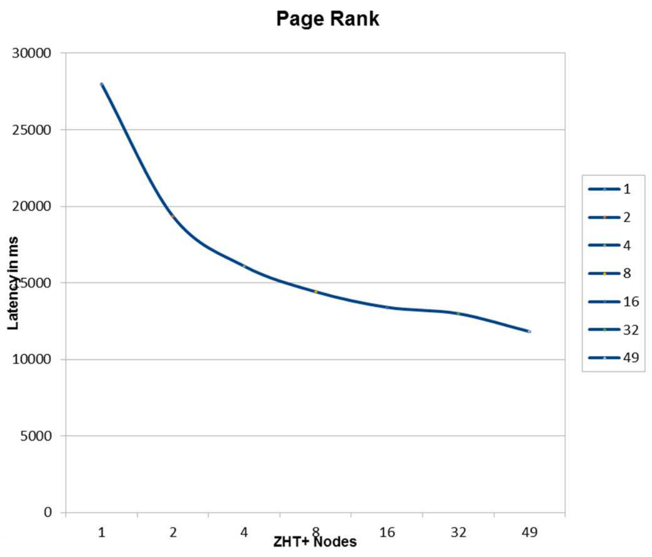

Figure 3 – PageRank Latency

We ran PageRank algorithm starting from node 5 which would traverse around 2729 nodes. Our implementation of PageRank internally uses DFS and performs 10 iterations in one single run. Figure 3 shows a very similar trend as DFS but the latency being 11 times (1 initialization plus 10 calculations).

C. *Throughput*

We have performed experiments on every basic operation ZHT+ supports i.e. addNode, addEdge, addNodeProperty, addEdgeProperty, lookupNodeProperty and lookupEdgeProperty. The throughput increases increases near-linearly with scale, reaching nearly 220 ops/sec at 49-node scale. One thing to note here is addNode internally performs 2 different ZHT operations (lookup and add) and addNodeProperty and addEdgeProperty internally performs 3 different ZHT operations (lookup, delete and add).

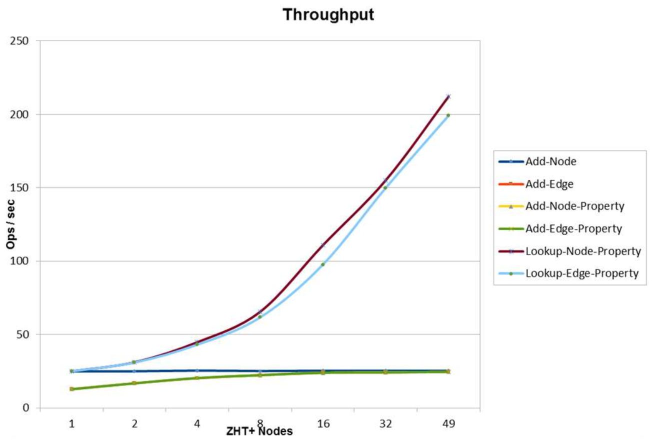

Figure 4 – Throughput (single client)

We have performed similar experiments on concurrent client-server pairs ranging from 1 to 16 nodes. We observed that there was no degradation in performance at 1 and 16 clients. We see similar throughput trend when run with 1-client. Since all add operations delivers around the same throughput, we have considered average of all add operations. Therefore, we will see only Read and Insert operations plotted in this graph.

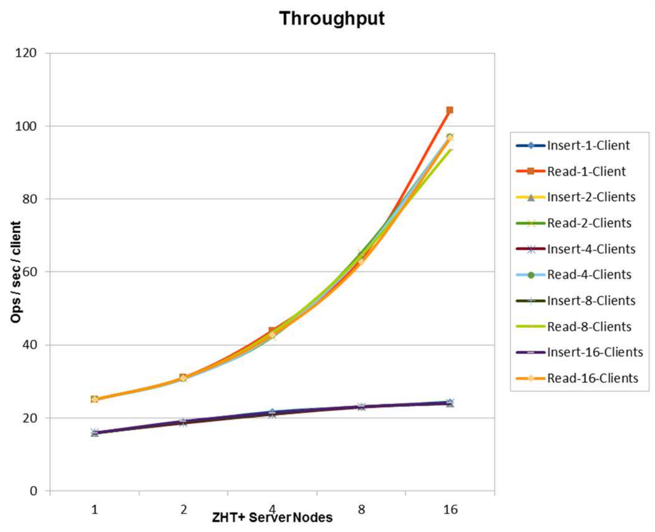

Figure 5 – Throughput (multiple clients)

D. *Scalability and Efficiency*

We have investigated the efficiency of ZHT+ when compared to the performance with multiple server nodes. Efficiency is measured throughput divided by ideal throughput. Figure 6 considers the 1-node scale will perform at 100% efficiency and eventual change in efficiency as we scale the servers. We feel that the drop in efficiency we see is due to being unable to fully load the servers. We can see that addNode performs the worst as the time remains exactly the same even as more server nodes are added. The lookupNodeProperty and lookupEdgeProperty perform somewhat better as times drop somewhat as server nodes are added, but this is not proportional. We would need to evaluate our system at extreme scale to understand the true impact on efficiency.

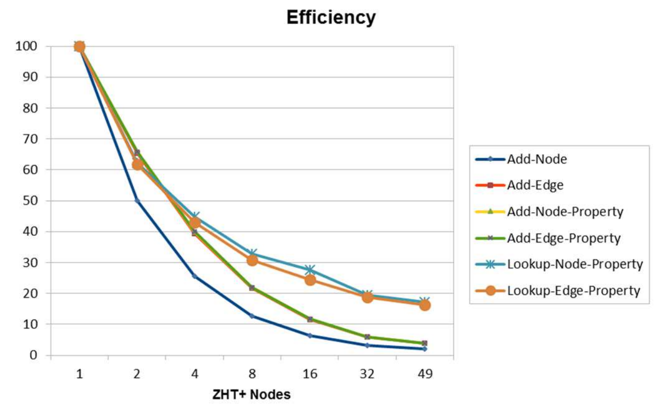

Figure 6 – Efficiency

E. *Neo4j*

Neo4j is a persistent graph database that can be used for enterprise deployments. It features fully ACID transactions, meaning that it enforces that all operations that modify data occur within a transaction to guarantee that data is consistent. It can scale to billions of node and relationships and work with multiple instances of Neo4j to form a high availability cluster, thus improving data availability and featuring data redundancy through replication. A single instance can itself handle billions of nodes and relationships however we can scale it to improve throughputs with little impact on performance of the database. The queries work through 'traversals' in the graph which are analogous to 'join' operations in relational databases. Neo4J can perform millions of traversals per second.

Neo4J can be setup in HA cluster mode, enabling replication between nodes in the cluster. The configuration follows Master-Slave architecture. When a new cluster is started each node is configured on how to reach all the other nodes of the cluster and after joining a master is elected. Any new nodes created and joining past this point, join in as a slave. Slaves will automatically synchronize with the master at a regular interval which can be configured with ha.pull_interval. When a node goes down or is unreachable, it is marked as "unavailable". If the master goes down, a new master is elected which will then broadcast its availability.

The architecture is designed in master-slave format, the master always needs to have a complete view of the database. In order for the master to maintain a complete view, any writes that are going through the slaves needs to be replicated to the master before an acknowledgement is made. This means that locks will be acquired on both master and slave. When the transaction commits it will first be committed on the master and then, if successful, on the slave, thus making the master a bottleneck for writes. The slaves also need to update themselves with the maser before attempting a write transaction and as such write performance is much better if the writes happen at the master. When updates are made on the master, by default it tries to replicate to at least 1 node (this can be changed to 0 for better performance / a higher number for higher data redundancy) before completing the transaction. The rest of slaves are optimistically updated i.e. even if some slaves fail to receive updates (as long as at least one receives it) we still succeed the transaction. HA configuration is usually complemented by a load balancer which load balances between all nodes. The reads in this configuration scale linearly but the writes are still bottlenecked at the master.

We created a VPC and configured all the private interfaces in the VPC (these interfaces were defined and used for all the inter-cluster communication, there was no other traffic on this subnet). We set the ha.pull_interval = 10s (defines how often the slaves will sync up with the master). Replication was set to 1 node, i.e. master succeeds the transaction if at least one slave receives the update. All benchmarks were run directly on the master node for consistency.

The graph was loaded using a cypher query with periodic commits. Being an ACID database, all transactions are completely flushed down to the disk and when loading large graphs this might cause an issue with performance so we can instead force periodic commits where the transactions are held in memory until a certain threshold and then flushed to the disk. In this case we set the periodic commits to 1000. This can be made much larger for large graphs with millions of entries, this is only limited by JAVA heap space which can be changed. In this case the graph load times for 3-32 nodes are slightly worse than 1 node case but very close to each other, this is because in HA mode, the updates are synced with at least one other node on flush to disk [if the replication was set to 0, we would have received near 1 node performance stats but lost out on data redundancy]. For loading the graphs, we used periodic commits with cypher queries, which means that the 'n' (configurable parameter) transactions were held in memory before flushing them down to the disk. This helped improve performance for graph loading. For multiple node cases, we ran multiple instances of the benchmark scripts from multiple clients and then average the throughput performance to plot the scaling trend.

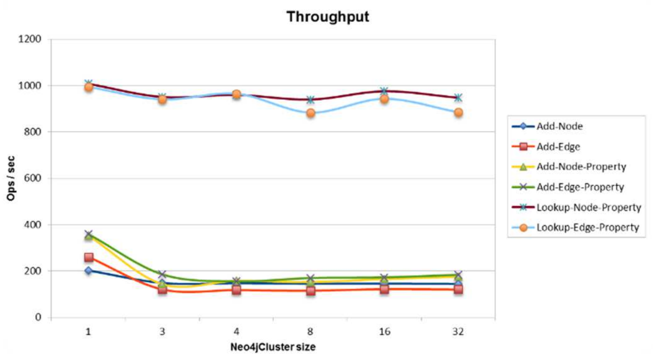

Figure 7 – Neo4j Throughput

In Figure 7 the top 2 lines (lookups / read operations) are much faster than write operations and that is expected, with the difference being even more drastic in HA configurations because writes are even slower.

The graph shows the expected trend where reads are pretty much similar performing across different node configurations, this is because all reads were from one node. Though we don’t have a graph for multiple node , multiple client configurations, that config with a load balancer would have yielded a close to linear scale up [This was verified with 1,3 and 4 node configuration]

For the bottom add/write operations we can clearly see the performance dropping from 1 node and then staying relatively constant. This was expected to due the addition of replication to at least 1 slave and remains constant because, it is always at least 1 slave irrespective of the cluster size.

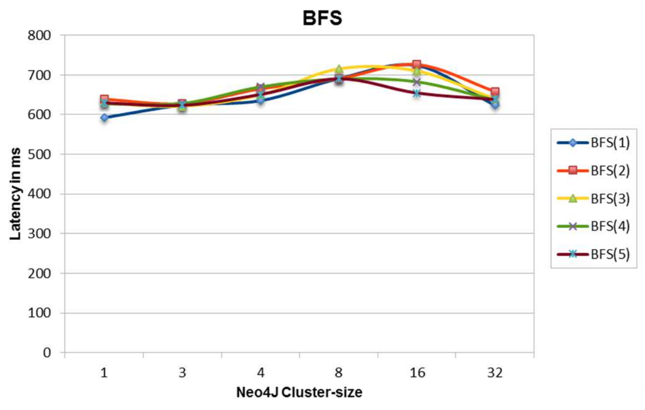

Figure 8 – Neo4j BFS

Figure 8 shows Breadth first search starting at different nodes. The performance stays relatively the same because in this case there are no writes and therefore there are no updates to the slave nodes and since the benchmarks were run on the master which has the complete view of the database, this is just like running it on a single node.

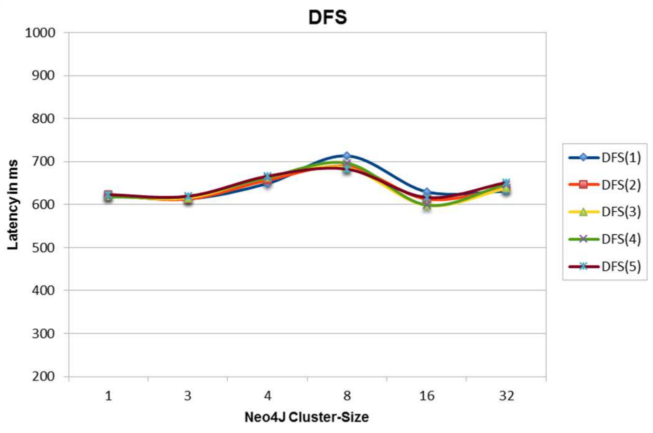

Figure 9 – Neo4j DFS

Figure 9 shows very similar results for Depth first search.

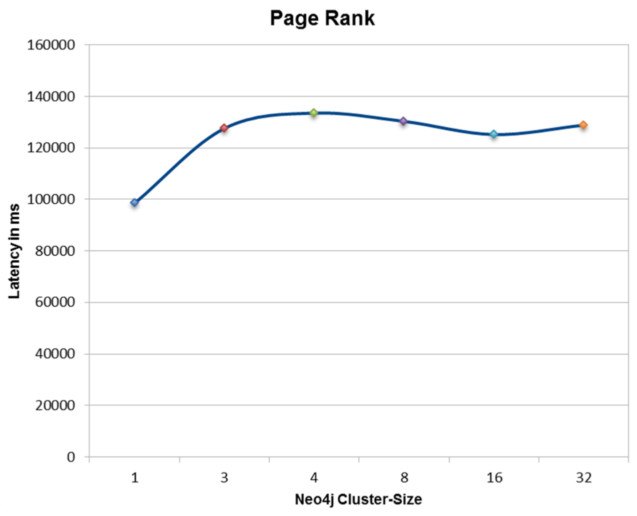

Figure 10 – Neo4j PageRank

Pagerank shown in figure 10, unlike BFS and DFS involves updating the edges of the nodes with new rank information and therefore the performance for HA is not in-line with the 1-node case. The time to run it increases from 1-node to 3-nodes and then stays more or less the same.

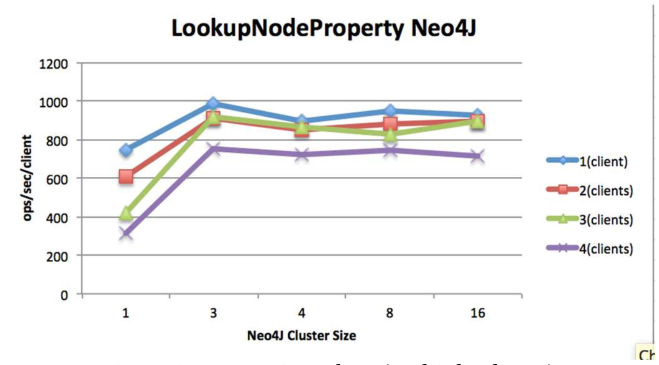

Figure 11 – Neo4j Lookup (multiple clients)

In Figure 11 we see that for 1 node we see an improvement in read performance when increasing the clients from 1 to 2, this was surprising. This could be because, since the calls were over REST, they weren't fast enough to saturate the server. From 2 clients onwards, we see that there is not much performance gain. For 3 node cluster size, we see that we keep getting performance gains all the way until 3 nodes after that point it starts to level. For 4 nodes and greater we see improvements all the way until 4 nodes, this shows that the reads are actually scaling linearly with cluster size.

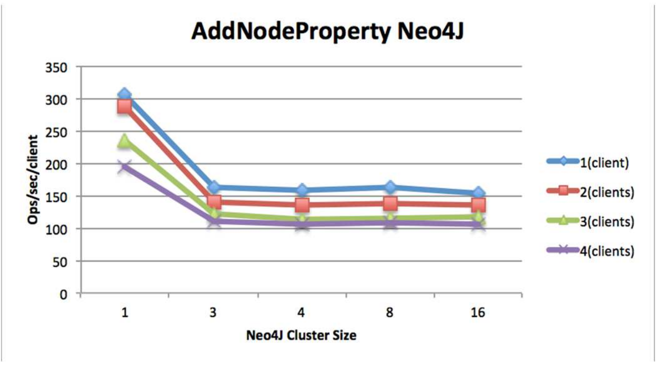

Figure 12 – Neo4j Add Node (multiple clients)

In Figure 12 we see that for write performance, the bottleneck is the master. For 1 node we see that the write performance is the fastest, this is because there is no replication overhead in the background. It was surprising to see an improvement in performance with writes with multiple clients, it is also worth noting that all the clients were trying to run the same operations, so not sure if there is some caching in play here. However the improvement in performance is definitely not linear like in the case of reads.

F. GraphLab

We set up graphlab and loaded the graph using an internal API which loads the graph from csv format. We ran pagerank on the graph with 10k edges and it took less than 3s. It seemed to us that the whole computation was in memory and given that it took so little time, we did not see much value in distributing the load. We also experimented with an inbuilt API (in graphlab) to spawn multiple ec2 instances and distribute the load. It is vastly different from ZHT+ (not even a persistent database and more of a processing framework).

Benchmarking throughput, reads and writes doesn't make sense. We can measure the execution times of various algorithms. Unfortunately we couldn't find much documentation around for BFS / DFS (if it is inbuilt) but we did run PageRank and saw that it was extremely fast (probably because the computation was completely in memory). It took less than 3s for the PageRank

It also has an execution framework to define an environment and parallelize tasks. Has support for AWS where it can automatically spawn instances and parallelize the work (this is still needs to be experimented / benchmarked upon) however given that it is so ridiculously fast [took less than 3s for the PageRank], is not a persistent database and everything is in memory, it doesn't make sense to compare with ZHT+.

G. Apache Giraph

Giraph is a an iterative graph processing system built for high scalability. The system is built based on Bulk Synchronous Parallel (BSP) model of distributed computing. In Giraph, graph-processing programs are expressed as a sequence of iterations called supersteps. During a superstep, the framework starts a user-defined function for each vertex, conceptually in parallel. The user-defined function specifies the behaviour at a single vertex V and a single superstep S. The function can read messages that are sent to V in superstep S-1, send messages to other vertices that are received at superstepS+1, and modify the state of V and its outgoing edges. Messages are typically sent along outgoing edges, but you can send a message to any vertex with a known identifier. Each superstep represents atomic units of parallel computation. Figure 13 illustrates the execution mechanism of the BSP programming model.

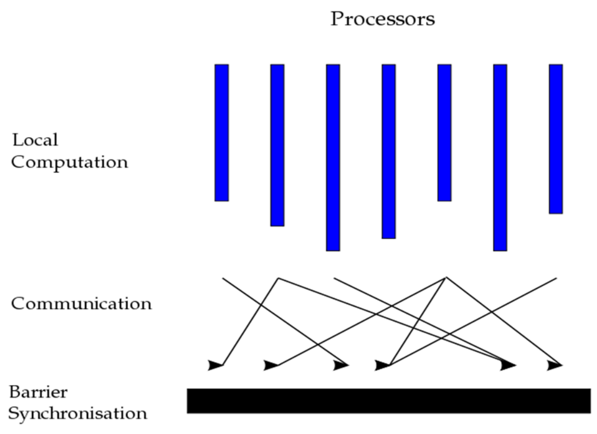

Figure 13 – Bulk Synchronous Parallel model

Giraph jobs run on Hadoop infrastructure as it leverages the map phase of mapreduce. Giraph does in-memory processing which speeds up performance. This can also lead to a memory bottleneck when dealing with very large dataset leading to a lot of inter-vertex messages.

Giraph is a very different graph processing system compared to a graph database like ZHT+ or Neo4j. [13]

H. *ZHT+ vs Neo4j*

We feel that Neo4j is the most direct comparison with ZHT+. Therefore the following graphs compare the performance of these two graph databases.

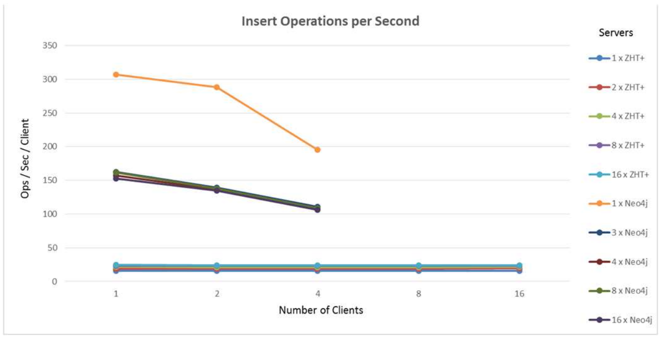

Figure 14 – Insert Operations

Figure 14 shows the insert operations per second per client for various numbers of clients and servers. A horizontal line would indicate that the server scales linearly with no degradation as the number of clients increases. The ZHT+ lines are horizontal with somewhat greater performance as the number of servers increases. The Neo4j lines all slope downward with the most dramatic fall in the single server configuration. However, at the limited scales tested, Neo4j significantly outperforms ZHT+.

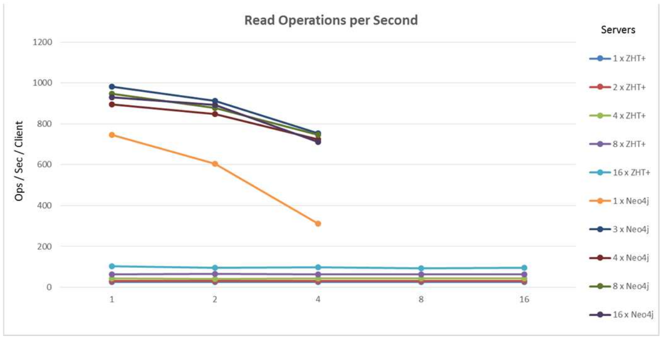

Figure 15 – Read Operations

Figure 15 shows the read operations per second per client for various numbers of clients and servers. A horizontal line would indicate that the server scales linearly with no degradation as the number of clients increases. The ZHT+ lines are horizontal with four times greater performance per client as the number of servers increased from 1 to 16. The Neo4j lines all slope downward with the most dramatic fall in the single server configuration. At the limited scales tested, Neo4j still outperforms ZHT+, but the single server performance is rapidly falling towards that of ZHT+.

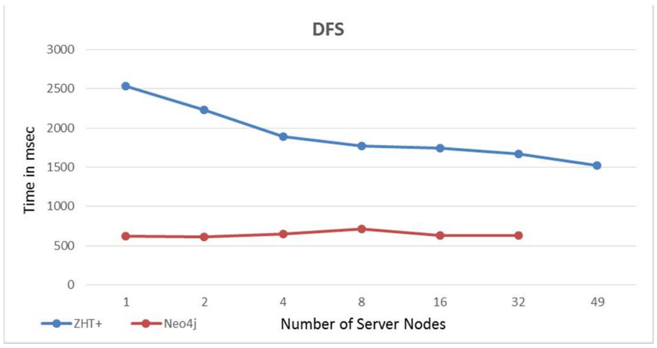

Figure 16 – DFS Operations

Figure 16 shows the time in msec for one client to complete the benchmark DFS for various numbers of servers. The DFS for ZHT+ is being done on the client while the DFS for Neo4j is an inbuilt function. The time for Neo4j is significantly faster than ZHT+, though it does not improve with additional server nodes while ZHT+ does improve somewhat.

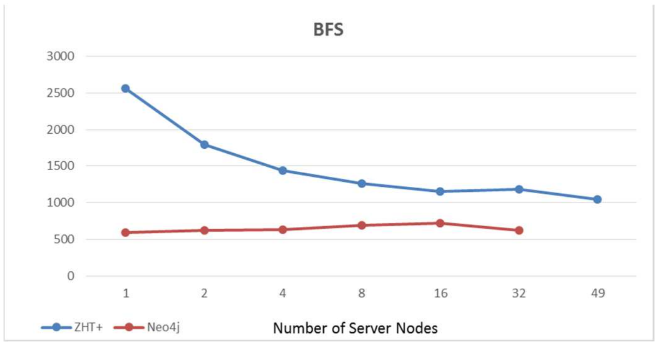

Figure 17 – BFS Operations

Figure 17 shows the time in msec for one client to complete the benchmark BFS for various numbers of servers. Again, the BFS for ZHT+ is being done on the client while the BFS for Neo4j is an inbuilt function. The time for Neo4j is significantly faster than ZHT+ with a single server node, but ZHT+ improves dramatically with additional server nodes while Neo4j remains about the same.

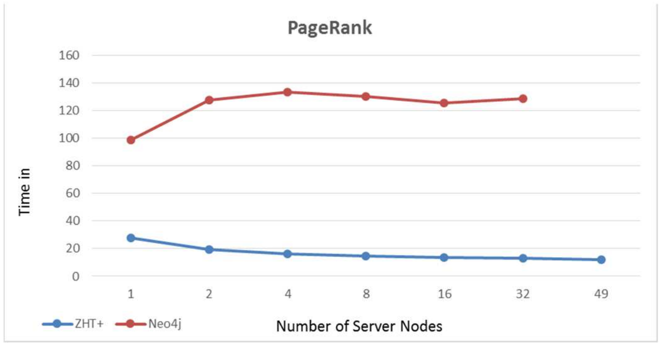

Figure 18 – PageRank Operations

Figure 18 shows the time in seconds for one client to complete the benchmark PageRank for various numbers of servers. PageRank is not built into Neo4j, so PageRank is being done on the client for both ZHT+ and Neo4j. The time for ZHT+

is exactly what is expected - the 11 BFS passes needed means that PageRank for ZHT+ takes almost exactly 11 times as long as BFS. However, when PageRank is run on a client through a REST interface to Neo4j, it is significantly slower than ZHT+.

I. ZHT+ Performance Summary

In our testing, ZHT+ was slower than Neo4j in every test except PageRank. We checked the underlying ZHT calls and found that they also did not perform as well as expected. We checked the benchmark included with ZHT and it showed similar performance. We do not have a good explanation for this.

ZHT+ did perform as expected regarding scalability. It showed no decline in performance as additional clients were added. The per client operations stayed the same meaning that N times as many clients produced N times the number of total operations. However, we do not feel that the loads on the server nodes were anywhere near capacity so this is expected for lightly loaded systems.

VI. Future Work

Higher loads: At the scales that we tested, ZHT+ scaled perfectly - access time per client node was not diminished as additional client nodes were added. But with a limit of 50 nodes, we feel that we were unable to sufficiently load the ZHT KVS. Additional benchmark testing should be conducted at larger scales so that the ZHT server nodes are being fully utilized and performance begins to degrade.

Additional features: Support for node, edge, and property removal should be added. Ideally, both scenarios should be supported - full edge removal at node removal and dead edge removal at edge traversal. This would allow optimization of the specific ZHT+ instance by selecting the desired removal method.

Query language: An interface should be added by which a user could query ZHT+ for nodes and edges based on their properties. The most useful results would probably be either an iterator of all matches or sorted results based on pagerank value.

VII. Conclusion

ZHT+ is a limited scope implementation of a graph database using ZHT. Due to its distributed nature, there is no central point where performance bottlenecks will occur. There is no master for reads or writes, no replication of data across all nodes, and no load balancer. Each client goes directly to the server node containing the desired data. We have shown that this model scales extremely well as long as the distribution of access (either reads or writes) is relatively uniform. However, if many clients all require access to the same piece of data, the server node on which that piece of data resides could become overloaded while other server nodes are underutilized. In a general purpose graph database, this may not be an issue. But, if used for a social network type graph, this could be an issue with “trending” nodes.

References

[1] Li, Tonglin, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table." In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 775-787. IEEE, 2013.

[2] Li, Tonglin, Antonio Perez de Tejada, Kevin Brandstatter, Zhao Zhang, and Ioan Raicu. "ZHT: a Zero-hop DHT for High-End Computing Environment." (2012).

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

[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] 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.

[6] Cogo, Vinicius Vielmo “Serializing Data with Protocol Buffers” In *Smalltalks*, Universidade de Lisboa, February 12, 2014.

[7] Miller, Justin J. "Graph Database Applications and Concepts with Neo4j." In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA March 23rd-24th*. 2013.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in Proceedings of the 2010 international conference on Management of data, ser. SIGMOD ’10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[9] Jouili, Salim, and Valentin Vansteenberghe. "An empirical comparison of graph databases." In *Social Computing (SocialCom), 2013 International Conference on*, pp. 708-715. IEEE, 2013.

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

[11] Redis Graph: https://github.com/tblobaum/redis-graph

[12] https://snap.stanford.edu/data/soc-LiveJournal1.html

[13] http://www.ibm.com/developerworks/library/os-giraph/

[14] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Ozsu, Xingfang Wang, Tianqi Jin. An Experimental Comparison of Pregel-like Graph Processing Systems