Implementing Temporal Graphs with Apache TinkerPop and HGraphDB

When most people think of Big Data, often they imagine loads of unstructured data. However, in most cases there is some sort of structure or relationships within this data. Based on these relationships there is one or more representation scheme that best suits to handle this type of data. A common pattern seen in field is hierarchal/relationship representation. This form of representation is adept in handling scenarios like complex business models, chain of event or plans, chain of stock orders in banks, making it popular among financial institutions.

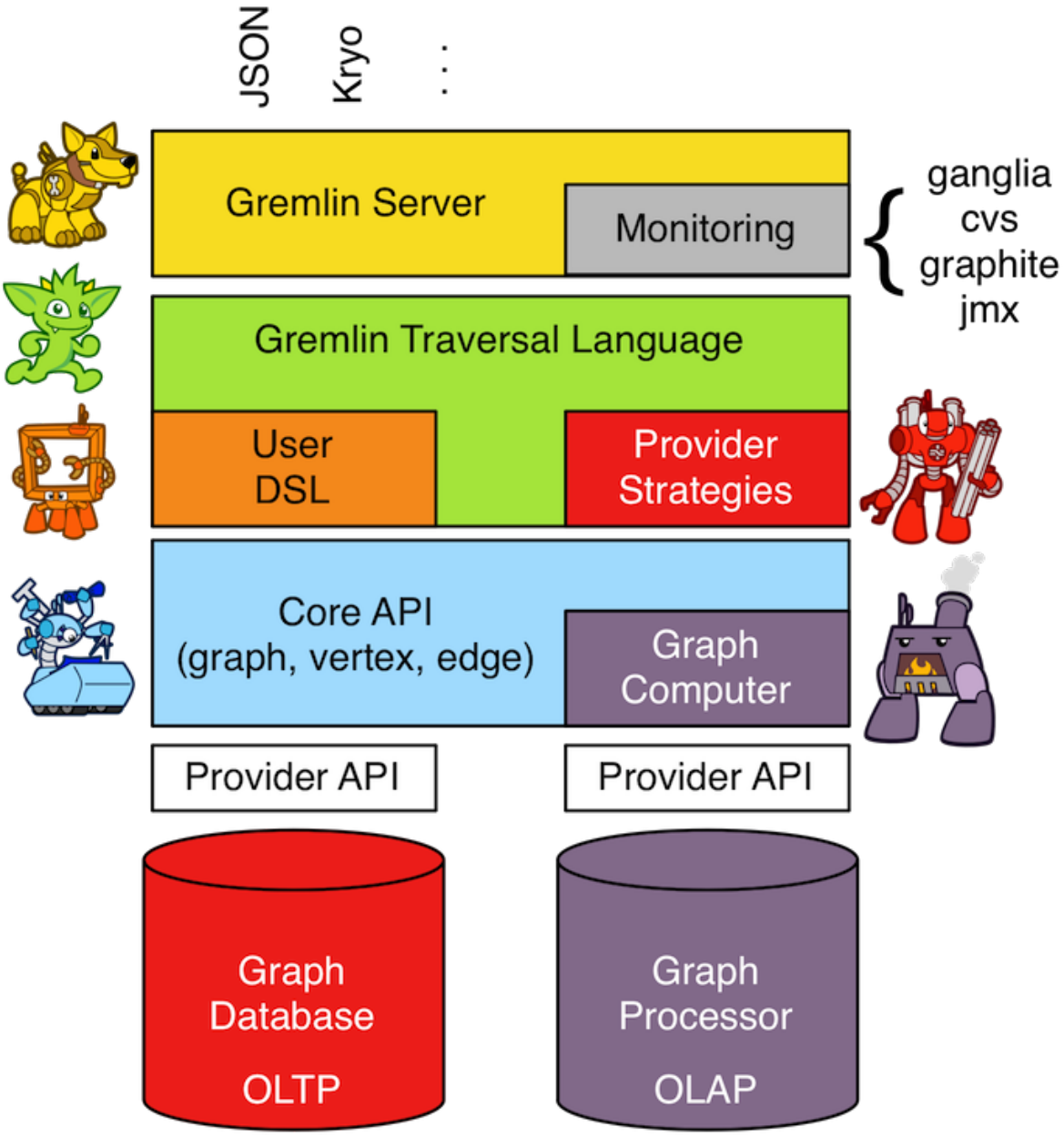
Storing and changing these hierarchical data sets can be a challenge, especially if data is in temporal form (Parts of it changing over time). One simple way to effectively store this data can be using graphs. A lot of the time these hierarchical data sets are in a form of a tree which is just a specialized version of graph. Storing data in this form has some inherent advantages such as easy representation /visualization of data, fast and well known traversal algorithms proven across industry, easy manipulation of parts of data without changing or duplicating the entire datasets.

This blog explains how to use an up and coming Graph framework like Apache TinkerPop over big data storage like Apache Hbase to simulate the hierarchical structures. Hbase was chosen as a storage layer for fast retrieval and scalability.

Apache TinkerPop

What is apache TinkerPop?
Apache TinkerPop is an open source graph computing framework composed of various interoperable components. TinkerPop uses simple and intuitive terms like Vertices (Nodes of a graph representing data points) and Edges (connection between these vertices representing relationships between data points) to represent complex hierarchical data structure of nodes and relationships between them.
TinkerPop also comes with its own graph traversal language called Gremlin, which makes traversing nodes and searching for relations between them as easy as specifying vertices and looking for specific edges between them.
More details for Apache TinkerPop is available at
http://tinkerpop.apache.org/docs/current/reference/

As mentioned earlier Apache TinkerPop comprises interoperable components. Following diagram from Apache TinkerPop documentation will show this in more detail.



This interoperability means any underlying storage and processing engine can be used to support scalability and meet required SLA's with industry standard tools.
For instance, Apache Spark can be used as the Graph processor which can take load of processing large and complex graphs using standard Hadoop clusters.
Similarly, the underlying graph database can also be any storage platform like Apache Hbase which supports fast updates and retrieval, together with storage of vast amounts of data.
This interoperability is all possible because of the provider API's, which means if there is an implementation of these API for required platform then it can be used.

There are various implementations for the provider API to support different underlying storages.

HGraphDB

One such implementation discussed in this blog is HGraphDB[2] implemented by Robert Yokota at Yammer. HGraphDB is an implementation of Apache TinkerPop 3 interface that uses Apache HBase as the underlying graph database. Like TinkerPop HGraphDB stores nodes and relationship between them as vertices and edges in Hbase Tables.
HGraphDB uses a tall table approach to store graph data in Apache Hbase i.e. each of the graph's nodes and relationships between them are stored in one Hbase row as vertex and edge. Unique Key (row IDs) to these rows are VertexID and EdgeID. HGraphDB also stores "created" and "updated" time for all vertices and edges along with multiple user-defined property columns for better management and searching of vertices and edges. These user-defined properties are what makes storing and updating temporal graphs easier without duplicating the entire graph.
Apart from this, HGraphDB also provides indexes for faster traversal of graphs. These indexes are based on user-defined properties, one per index.
HGraphDB uses five tables in Apache HBase and their structure is explained in more detail below.

Vertex Table:
This table stores all the vertices. Columns and their represented values for this table are as follows:

Label: Text label assigned to vertices for multiple types of vertices
Created: Timestamp when a vertex was created
Updated: Timestamp of last update for this vertex
Property #: User-defined property for this vertex and its associated value. There can be as many property columns as required
Example of one entry for a graph node is as below

Key	Label	Created	Updated	Property 1
Vertex_ID	~l	~c	~u	Created_on
TempVertice_1	Invoce	1480522438042	1480522438042	1229600000

```
hbase(main):002:0> scan "TemporalGraph:vertices"
ROW                                    COLUMN+CELL
 \x00)\x834TempVertice_1\x00           column=f:create_on, timestamp=1480522438042, value=)\x87,\x80\x00\x00\x00x0\xE7\x80
 \x00)\x834TempVertice_1\x00           column=f:~c, timestamp=1480522438042, value=)\x87,\x80\x00\x01X\B6\x03\x95N
 \x00)\x834TempVertice_1\x00           column=f:~l, timestamp=1480522438042, value=)\x834vertex\x00
 \x00)\x834TempVertice_1\x00           column=f:~u, timestamp=1480522438042, value=)\x87,\x80\x00\x01X\B6\x03\x95N
```

Edge Table:
This table stores all the edges between graph nodes and weights or values (Property) assigned to these edges are stored as user defined property column. Columns and their represented values for this table are as follows:

Label: Text label assigned to vertices for multiple types of vertices
Created: Timestamp when an Edge was created
Updated: Timestamp of last update for this Edge
From Vertex: Vertex where this edge originates from
To Vertex: Destination vertex for this edge
Property #: User defined property for this Edge and its associated value. There can be as many property columns as required
Example of one such edge entry is as below.

Key	Label	Created	Updated	From Vertex	To Vertex	Property 1	Property 2
Vertex_ID	~l	~c	~u	~f	~t	since	till
Edge_1	Parent	1479600000	1479600000	TempVertice_1	TempVertice_2	1479600000	1479780000

```
hbase(main):002:0> scan "TemporalGraph:edges"
ROW                                COLUMN+CELL
 \x00)\x834Edge_1\x00              column=f:Till, timestamp=1480522438042, value=)\x87,\xFF\xFF\xFF\xFF\xFF\xFE\xAE\7F
 \x00)\x834Edge_1\x00              column=f:Since, timestamp=1480522438042, value=)\x87,\x80\x00\x00\x00x0\xE7\x80
 \x00)\x834Edge_1\x00              column=f:~c, timestamp=1480522438042, value=)\x87,\x80\x00\x01X\B6\x03\xc3\x18
 \x00)\x834Edge_1\x00              column=f:~f, timestamp=1480522438042, value=)\x834TempVertice_1\x00
 \x00)\x834Edge_1\x00              column=f:~l, timestamp=1480522438042, value=)\x834Parent\x00
 \x00)\x834Edge_1\x00              column=f:~t, timestamp=1480522438042, value=)\x834TempVertice_2\x00
 \x00)\x834Edge_1\x00              column=f:~u, timestamp=1480522438042, value=)\x87,\x80\x00\x01X\B6\x03\xc3\x18
```

Vertex Index table:
This table stores indexes of all the vertices for faster searching vertices based on their property values.
Columns and their represented values for this table are as follows:

Key: combination of vertex label, property for which this index is created, value of the property and vertex ID
Created at: Timestamp of creation of this index value
Vertex ID: ID of the vertex that is being indexed
Following is an example of the table:

Key	Created at	Vertex ID
Vertex_Label,Property_Key,Property_value,Vertex_ID	~c	~c
Vertex,Created_on,1229600000, TempVertice_1	1479600000	TempVertice_1

Edge Index table:
Similarly, vertex index table, this table stores indexes for all edges for faster searching based on property values.
Columns and their represented values for this table are as follows:

Key: combination of vertex ID of first vertex associated with this edge, direction of edge related to first vertex, property for which this index is created, label of this edge, value of the property and vertex ID for second vertex associated with this edge
Created at: Timestamp of creation of this index value
Following is an example of same.

Key	Created at
Vertex_1_ID,Direction,Property_Key,Edge_label,Property_value,Vertex_2_ID	~c
TempVertice_2,IN,Since,Parent,1229600000, TempVertice_1	1479600000

```
hbase(main):002:0> scan "TemporalGraph:edgeIndices"
ROW                                    COLUMN+CELL
 \x00)\x834TempVertice_2\x00)\x80)\x8  column=f:~c, timestamp=1480522438042, value=)\x87,\x80\x00\x01X\xCCH\xB4\xD8
 04Till\x004Parent\x00)\x87,\xFF\xFF\
 xFF\xFF\xFF\xFE\xAE\7F)\x834TempVert
 ice_1\x00)\x834Edge_1\x00
```

Index Metadata table:
This table contains metadata related to the indexes created on each property and specifies whether the index is currently active or not.
Columns and their represented values for this table are as follows:

Key: A unique value
Created at: Timestamp of creation of this index metadata value
Updated at: Timestamp of last update for this index metadata value
Id Unique: Id value for this index if index is unique
State: Current state of this index if it's active or not
Following is an example of the table.

Key	Created at	Updated at	Id Unique	State
Label,Property_key,Element_Type	~c	~u	~q	~x
Parent,Since,Edge	1479600000	1479600000	NO	Active

```
hbase(main):002:0> scan "TemporalGraph:indexMetadata"
ROW                                COLUMN+CELL
\x804Parent\x004Till\x00)\x80      column=f:~c, timestamp=1480522438042, value=)\x87,\x80\x00\x01X\B6\x03\xc3\x18
\x804Parent\x004Till\x00)\x80      column=f:~q, timestamp=1480522438042, value=)\x82)\x80
\x804Parent\x004Till\x00)\x80      column=f:~u, timestamp=1480522438042, value=)\x87,\x80\x00\x01X\B6\x03\xc3\x18
\x804Parent\x004Till\x00)\x80      column=f:~x, timestamp=1480522438042, value=)\x834ACTIVE\x00
```

More details and source code for HGraphDB is available at
https://github.com/rayokota/hgraphdb

Working Project

By this point you should have a good understanding of base components and their working, the rest of the blog will describe an example data generator and query module to see how to implement hierarchical structure using Apache HBase as graph database and performance of such setup.

Code for this project is available at
https://github.com/ashishtyagicse/TemporalGrapsUsingHBase-TinkerPop

Cluster used for this project
- Hardware: 5 EC2 instances with 4 core and 15GB memory for each node
- CDH version: 5.6.2

This project has two main components
- Data generator: Generates test data / graph
- Query engine: Queries the generated data to test performance and functionality

Data Generator:
Code in java class with name "datagen.java" generates multiple hierarchies in a form of a balanced binary tree. It persists the data into Apache HBase using HGraphDB.

Following is a call to HGraphDB for generating a graph in HBase

```
this.cfg = new HBaseGraphConfiguration()
.setInstanceType(HBaseGraphConfiguration.InstanceType.DISTRIBUTED)
                .setGraphNamespace(HIERARCHY_PREFIX + HIERARCHY_NAME)
                .setCreateTables(Create)
                .setRegionCount(REGION_COUNT)
                .set(ZOOKEPER_QORUM, ZOOKEPER_QORUM_NODE)
                .set(ZOOKEPER_PARENT, ZOOKEPER_PARENT_NODE);
this.graph = HBaseGraph.open(cfg);
```

Parameters for graph generated can be controlled by constants file[3] which defines constants like no of nodes, suffixes and prefixes used for vertex and edge names etc.
Graph generated by the datagen has following property's
- Each vertex has a VertexID, a combination of prefix defined in constants file and node number
- All Vertices have one user defined property called Created_on, a value of seconds from Epoch. This is a test property to demonstrate how to define properties of a vertex.

```
Vertex v = VertexBulkLoad.addVertex(T.id, NodePrefix + "_" + NodeNum, T.label,
"Node", "created_on", HIERARCHY_CREATE_TIME);
```

- Same as vertices, edges have an EdgeID, a combination of prefix defined in constants file and edge number
- Edges in this graph has two user defined property "Since" and "Till"
- "Since" property defines when an edge was created

- “Till” property defines till when an edge was active
- Together these two properties help maintain the temporal nature of a graph
- Apart from a static base graph, the datagen also generates some updates to the base graph which acts as temporal data and shows a different state of graph at a given point of time.
- These temporal data points are created by creating new edges and manipulating “Till” property of existing edges.

Running this datagen will generate required number of hierarchal data sets in HBase each with a binary tree/ graph complete with random temporal data.

Query Engine:

After data is generated in HBase by datagen, queryHierarchy java file code query’s graphs sitting in HBase using Apache TinkerPop.

Query engine in this code can get two types of data from graphs in HBase

- Get the required root of a node at any given point in time
- Get the entire hierarchy up till a given level as it appears at a given point in time

Query Root:

To get the all the nodes in path from a given node till its root at any given point in time the code performs following series of steps

- First get a graph instance. Use same HGraphDB command as a create graph but this time with create as false.
- Once we have the graph instance we can then use Gremlin commands to find the requested vertex in question.

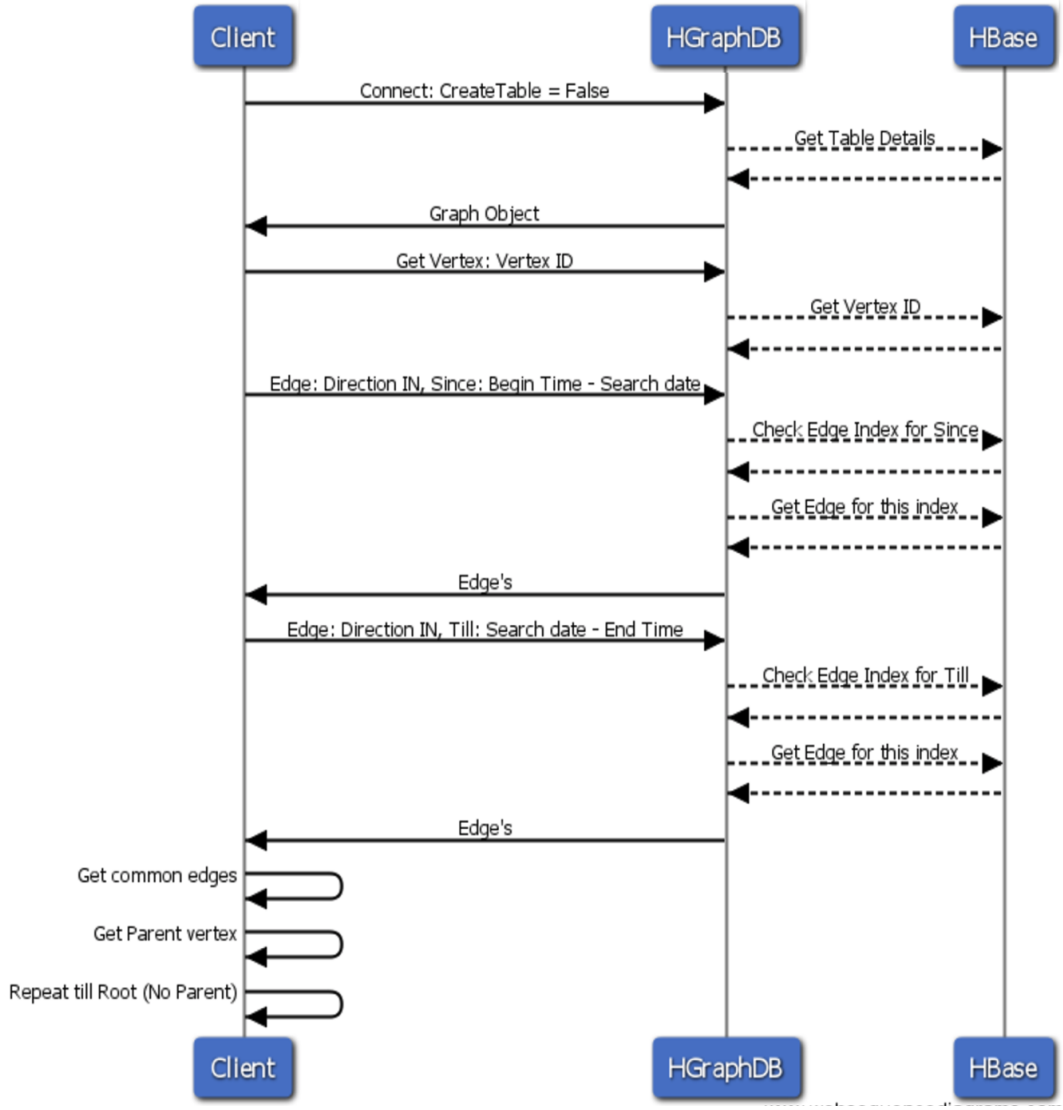
```
return Graph.vertex(NodeName);
```

- Next, find all the incoming edges of this vertex and check their since and till property to find the one edge that was active for the requested time.
- This will give an active parent for the requested node at specified time.

```
E1 = V.edges(D, "Parent", "since", HIERARCHY_CREATE_TIME,  TimeInterval[1]);
E2 = V.edges(D, "Parent", "Till", TimeInterval[0], HIERARCHY_END_TIME +
UPDATE_INTERVAL);
```

- Once the parent is found, repeat the same process as above to get the entire path till root.

Following sequence diagram shows above mentioned process steps for query for root



This simple setup demonstrated how to use properties for edges to maintain and manipulate same vertex for creating different hierarchies in time.

In terms of performance if requested node is found at last level of binary tree then search for root in a graph of 100,000 vertices will take roughly around 300 ms

Query Hierarchy:
Let's say you would like to find information about full lifecycle of a stock order. (ie you need to reconstruct all parent orders for this order). In this case, a query for entire hierarchy is needed for that day.

To get the entire hierarchy, code performs following series of steps

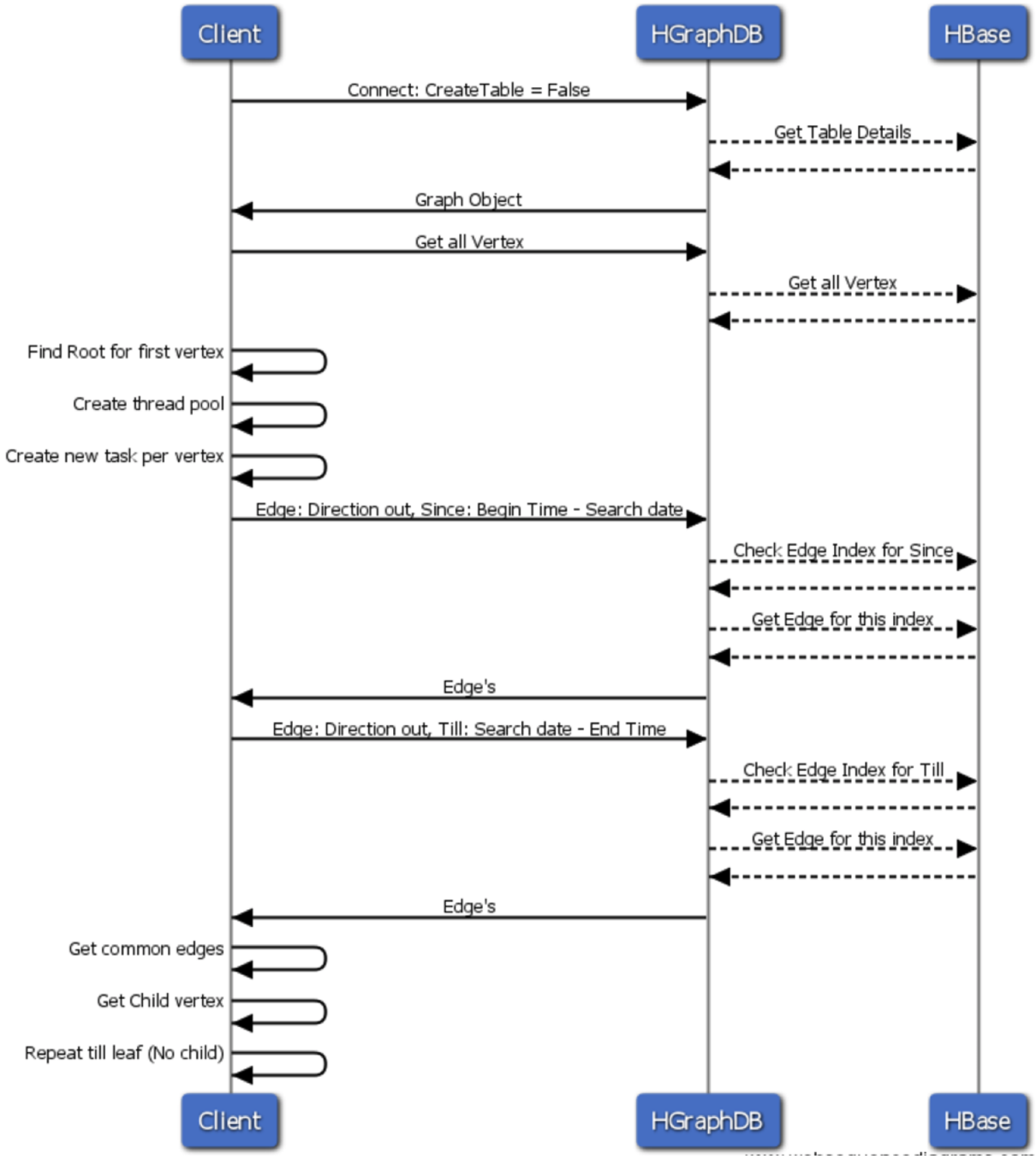
- Same as with root query, first get a graph instance. This can be done by same HGraphDB command as a create graph but this time with create as false.
- Then find anyone random vertex of this graph that was active for given moment in time.

- Once this vertex is found perform a Query root operation as described above to get the root for this vertex for that given moment in time.
- After root for hierarchy is found for given time the entire hierarchy can be rebuild
- Rebuilding the entire hierarchy tree is split in multiple threads using a threadpool. One vertex at a time is assigned to a thread from this pool for processing.

```
ExecutorService executorService = Executors.newFixedThreadPool(NUMBER_OF_THREADS);
while (graphQueue.size() > 0){
   Vertex node = graphQueue.poll();
   if (node == null){
   if (graphQueue.size() > 0) {
      System.out.println("Level: " + Level + " > " );
      Level++;
      ArrayList<Future<ArrayList<Vertex>>> ProcessNodes = new ArrayList<>();
      final ArrayList<Vertex> VResFuture = new ArrayList<>();
      while(graphQueue.peek() != null) {
        final Vertex LevelNode = graphQueue.poll();
        ProcessNodes.add(executorService.submit(new Callable<ArrayList<Vertex>>() {
           @Override
           public ArrayList<Vertex> call() throws Exception {
               ArrayList<Vertex> VRes = new ArrayList<>();
               for (Vertex Child : getChild((HBaseVertex)LevelNode,TimeInterval)) {
                    VRes.add(Child);
                    VResFuture.add(Child);
                }
                return VRes;
           }
         }));
      }
      try {
         graphQueue.add(null);
         for (Future<ArrayList<Vertex>> TaskFuture: ProcessNodes ) {
             VList = TaskFuture.get();
             for (Vertex Child : VList) {
                 if (ProcessedNodes.containsKey(Child.id().toString())) {
                     continue;
                 }
                 ProcessedNodes.put(Child.id().toString(), true);
                 graphQueue.add(Child);
             }
```

- Within this thread, check all the out edges of this vertex and then find the once that were active at required point of time.
- After edges are found check their out vertex and continue pushing those vertex to thread pool till end of hierarchy is not reached.

Following sequence diagram shows above mentioned process steps for query of hierarchy



Conclusion

Some takeaways from this blog
Handling temporal hierarchical data is tricky and can take large amount of processing power, storage and subject expertise.
But with help of tools like Apache TinkerPop and HGraphDB, it can be done with relative ease while still taking advantage of scalable storage engines like Apache HBase and processing power of Apache spark on a Hadoop cluster. In addition, using TinkerPop framework and HGraphDB could be more productive as Gremlin style traversal language is more intuitive for people across industry, and reduces effort for building custom traversal logic.

Resources

Apache TinkerPop
HGraphDB
Datagen and Query Engine