# Horton – Distributed Graph Database

Horton is a distributed graph database designed for efficient handling of very large graph data structures. It provides interfaces for defining the data as graph entities, such as node and edges, uploading the data into a cluster of machines, retrieving the uploaded data and executing reachability queries against the data. Horton is written as an Orleans application (and as such we sometimes refer to it as the graph library) and leverages Orleans for distribution, messaging, persistence and all the distributed management of the graph database. The client uses Horton via a user-side library (which we sometimes refer to as the graph library façade) which hides the distributed nature of the database from the user.

## Data model: node and edges

Nodes and edges are the data units of the graph database. A node has a type, unique identifier, and a variable number of attributes: key-value pairs of application specific data. An edge has a type, unique identifier, references to the nodes it connects, direction and a variable number of attributes. For both nodes and edges the keys of the attributes are strings and the values of the attributes are arbitrary serializable objects.

## Node and NamedNode:

Horton supports two types of nodes: regular Nodes and NamedNodes. The only difference between the two types is the way unique identifier is created. For the regular node the developer only specifies the node type and the runtime automatically allocates an identifier which is guaranteed to be unique across all nodes of the graph. The identifier appears as an opaque object to the user (it wraps a Guid). For NamedNode, the developer specifies herself the identifier for the node. This identifier must be unique across all nodes of the same type in order for this node to be part of the graph. We sometimes refer to the identifier given explicitly by the user as a “semantic key”. In general, if the developer is not able to guarantee uniqueness across all nodes of the same type, he should use regular nodes and not NamedNodes. The advantage of using NamedNode is the ability to provide meaningful identifiers and the improved performance gained by removing the need for indirection through the opaque identifier.

## Graph Schema:

Horton supports an optional schema for the graph. As opposed to a relational data base schema, a Horton schema does not specify the organization of the data (i.e. in tables), but just the types of the data that can be stored in nodes and edges. The schema is used by the runtime to perform data validation and to provide hints for performance optimizations, such as which attribute indices to create. For every node and edge type, the schema can specify the set of key-value attributes, the C# type of the value, whether the attribute is required or not, and whether an index should be created for this attribute or not.

## Creating a graph and adding nodes and edges:

A developer starts by creating a graph. The developer specifies a name for the graph that must be unique in the same deployment, along with number of machines and number of partitions. Typically, the number of partitions should be equal to the number of machines. By specifying more than one partition per machine the developer can tune the performance of Horton by tuning the amount parallelism in processing querying (which influences throughput) with the amount of cross partition messages (which influences latency).

Once the graph is created, the nodes can be added. The edges are added after the nodes. By default, if one of the end nodes of an edge does not appear in the graph when this edge is added, the edge addition will fail (there is a way to opt out of this behavior by specifying it on the schema). After nodes and edges are successfully added, the nodes and edges can be retrieved from the graph and the graph can be queried.

Added nodes and edges can be updated later on. Only the attributes of node and edges can be updated - the identifier, type, edge direction and edge endpoints are immutable. The developer can either specify a new list of attributes to fully replace the existing list, or merge new attributes with existing ones. In the latter case, if a new attribute is specified for an existing key, the new value will replace the old one.

In the example below, we create a new graph with nodes of type Contact and Company. Contact nodes represent a person which has a name and phone number, Company has name, address and size. The graph also has 3 directed edges of types: Works, Manages and Friend. Works edge has start date attribute.

Graph graph = Graph.CreateGraph(“MySampleGraph”, 2, 2);   
// 2 machines, 2 partitions

List<Node> NodeList = new List<Node>();

List<Edge> EdgeList = new List<Edge>();

Node node1 = new NamedNode(GraphSampleTypes.NODE\_TYPE\_CONTACT, (uint)1);

node1["Name"] = "PersonOne";

node1["Phone"] = 12345;

NodeList.Add(node1);

Node node2 = new NamedNode(GraphSampleTypes.NODE\_TYPE\_CONTACT, (uint)2);

node2["Name"] = "PersonTwo";

node2["Phone"] = 33333;

NodeList.Add(node2);

// Node will get its unique id allocated automatically by the Graph.

Node node3 = new Node(GraphSampleTypes.NODE\_TYPE\_COMPANY);

node3["Name"] = "Microsoft";

node3["Size"] = 1024;

NodeList.Add(node3);

// Wait() blocks the remote call until the reply is received.

// Instead, use ContinuewWith for async execution.

distrGraph.AddNodes(NodeList).Wait();

// Edges should be created only after nodes have been added.

Edge edge1 = new Edge(node1, node3, GraphSampleTypes.EDGE\_TYPE\_WORKS, true); // directed

edge1["Weight"] = 12.56;

edge1["Start"] = 123456;

EdgeList.Add(edge1);

Edge edge2 = new Edge(node1, node2,GraphSampleTypes.EDGE\_TYPE\_FRIEND,false);// undirected

edge2["Weight"] = 10.9;

EdgeList.Add(edge2);

distrGraph.AddEdges(EdgeList).Wait();

## Finding an existing graph and retrieving nodes and edges:

A developer can find an existing graph by looking it up by its name. Once the graph is found and its reference is retrieved, the developer can use it for accessing nodes and edges and for querying. The developer can:

1. Retrieve all nodes and edges of the graph
2. Retrieve nodes and edges by node or edge id
3. Retrieve nodes and edges by node or edge type

When retrieving a node or an edge, the developer can specify an optional projection. Projection specifies which attributes out of all node/edge attributes should be retrieved. Projections allow saving network traffic by only retrieving necessary attributes.

## Regular expression queries:

Horton provides the developer with the ability to specify and execute regular expression reachability queries. A query is a sequence of predicates over node and edges types and attributes in the following form:

(NodePredicate, EdgePredicate, NodePredicate,...)

Each predicate can contain conjunctions and disjunctions on node and edge attributes as well as closures such as regular language operators “\*” (zero or more), and “+” (one or more). The query specifies a pattern over the graph. While executing the query Horton will match the query pattern to the graph data and return all paths in the graph that match the given pattern. By default, Horton returns all paths in the graph that satisfy the regular expression of the query. The user can limit the number of paths returned or the maximum length of the paths. The returned query result includes the identifiers of the nodes and edges (but not nodes and edges themselves) that constitute the answer path. If the developer is interested in the nodes and edges themselves, she should retrieve the nodes and edges by using the retrieval APIs described above.

## Query examples:

We continue with our example from above.

A developer can write the following queries:

query1 = "%% PathQuery = {Contact{Name=\"John\"} Works> Company{Name=\”Microsoft\”}}";

query2 = "%% PathQuery = {Contact{Name=\"Bob\"} Manages> Contact }";

query3 = "%% PathQuery = {Contact{Name=\"Alice\"} Friend<> Contact}";

query4 = "%% PathQuery = {Contact{Name=\"Alice\"} Friend<> Contact Works>  
Company{Name=\”Microsoft\”} Manages< Contact{Name=\"Bob\"}}";

The first query will find all people names John who work for Microsoft, the 2nd query will find all people who are managed by Bob, the 3rd query will find all friends of Alice (in our example graph friendship is a directional relationship and this query will retrieve all friendship relationships in both directions), the 4th query will find all friends of Alice who also work in Microsoft and are also managed by Bob. The last query demonstrates the power of Horton to express complex queries over the graph that span long paths and execute them efficiently.

In the example code below we create a new parser object from our custom application-specific type converter which is used to convert node and edge type strings to int. We parse the query into final state machine and pass it as an argument to the Horton. Horton executes the query and returns query result, which is a collection of answer paths. Each answer path is a list of node id, edge id, …   
After receiving the query result, the developer can either selectively use the node and edges ids I the result to retrieve the actual nodes and edges ort she can retrieve all nodes and edges referenced in the query result by materu8alizing the query result. The materialized query result is a collection of materialized answer paths. Each materialized answer path is a list of node, edge, …  
  
QueryParser CQP = new QueryParser(new GraphSampleTypeConverter());

string query = "%% PathQuery = {" + "Contact Works> Company" + "}";

QueryResult queryResult = distrGraph.ExecuteQuery(CQP.Parse(query)).GetValue();

MaterializedQueryResult materializeResult =   
distrGraph.MaterializeResult(queryResult).GetValue();

## Statistical queries:

Developer can extract various useful statistics from the graph, such as:

1. Number of nodes and edges
2. Histogram of node degrees – number of nodes of every degree
3. Histogram of node types – number of nodes of every type
4. Histogram of edge types – number of edges of every type
5. Histogram of node-edge tuple types – number of instances of every node-edge type combination.
6. Histogram of node-edge-node triplets types – number of instances of every node-edge-node type combination.

## Extensibility framework:

We envision Horton as a platform for graph management and analysis. As such, it must support a rich set of graph algorithm. Our current release supports only regular language reachability, but many more graph algorithms will be supported in the future.

We also provide a developer with a way to extend Horton’s functionality by plugging in his own graph algorithm. In this mode the developer will write code that runs inside Horton (similar to the concept of stored procedures in the database, and probably more similar to the concept of implementing map and reduce functions for map/reduce frameworks). To simplify writing this code, Horton exposes a traversal framework. The traversal framework is a design pattern in which the developer expresses his algorithm in steps that operate on nodes (steps can be synchronous or asynchronous) and implements a visit function that accepts a single node and returns a set of nodes for the next step. The traversal framework automatically invokes the specified function on nodes at every steps and care of distributing the steps across machines, synchronizing the steps (if synchronous algorithm is required), and optimizing performance by merging the steps, when possible.

An interested developer can get more information about the traversal framework by contacting the Horton team.