April 03, 2019

Change Detection for Higher Scale Rediscovery and BigNode Handling

Adhikari Samarjit

K Nagarajan

Shanmugasundaram Senthilnathan

Paranji Srirama Vinodkumar

Suriyanarayanan Muthukumar

Change detection for higher scale rediscovery and BigNode handling

Disclosed by: Adhikari Samarjit, K Nagarajan, Shanmugasundaram Senthilnathan, Paranji Srirama Vinodkumar, Suriyanarayanan Muthukumar

Abstract

Discovery is a process of collecting the configuration data from the network elements in any protocol like SNMP, REST, NetConf, SOAP, SSH etc. and processing the data to compute the logical network topology. And the topology is stored in the database in a normalized fashion.

The discovery and topology computation is typically a multi-step process where the data collected from one step could decide what need to be collected.

Once discovery of one node is completed the logical topology is persisted. As network continuously evolve or change due to changes to the nodes' configuration, automatic re-discovery and recomputation of the logical topology becomes necessary to maintain accurate logical topology of the network.

Thus it is fair to assert that the discovery/rediscovery is a continuous process leading to eventual consistency of the logical topology.

Similarly for the discovered node, state of various aspects of the node need to be continuously monitored in real or near real time.

In cases during the re-discovery when configuration data is collected from the devices, detecting what has changed between previous discovery to now is a challenge. Typically the process goes on to compute new topology and compare the old topology with the new topology and if there is a difference apply the difference and persist. If there is no difference between the topology nothing needs to be persisted. This means the entire topology compute process has to be done whether there is a change in the topology or not. There is no early indications whether something is changed or not.

This disclosure suggests usage of hash meta data derived from the raw data collected to compare and detect the changes ahead so that costly computation of logical topology could be avoided if there are no change.

Typically very less percentage of nodes in a network change in a daily basis and even within the node only less percentage of configuration change per day or across two discovery cycles. So, early detection of changes and avoiding computation if there is no change could help in a big way to scale the discovery of network.

Similarly state polling happens at near real time, even here across two polling cycles very less percentage of states would change. Detecting early whether a change has happened or not can help in handling state change at scale. The scheme proposed in this invention can help address the polling needs to scale high as well.

Problems Solved

1. Detecting the configuration changes in the network at the node level and at network level early in the discovery process so that when there is no change, the computations can be skipped, so that lot of compute power is saved and thus can increase the scale and performance of discovery.

2. Helps to address handing very big nodes with 100s of thousands of entities where is very minuscule change happens. It can help avoid costly computation of thousands of objects which where there is no change.

3. This is applicable for any type of continuous data collection either state polling or polling for performance or for configuration.

4. This scheme is applicable for any protocol communication through which the raw data is collected.

5. This scheme is agnostic of any device type of vendor.

Prior solutions

1. Prior solutions attempt to collect data, compute the logical topology which involves complex computation steps and finally decide nothing has changed. In this process the compute power is spent whether there is a change or not.

2. Current scheme cannot pin point the changes, and cannot collect data selectively.

3. Current solutions deployed in NNMi and other SPIs suffer huge performance issues in terms of number of nodes they can handle per hour.

4. Current solution suffers handling big nodes to detect what has changed.

5. Current solution has quadratic computation in dealing with index swap cases where two interfaces could swap their index only while other properties might remain same.

Description

The current solution views the data from the devices as a matrix of matrix where each matrix could represent one aspect of configuration. A matrix in other terms is a table. For instance a node could have an interface table, where each row could represent information about one interface. Each interface could have many columns of data for every interface. Basically an interface table could provide data about many interfaces and a cell in the table could be a piece of configuration data for a particular interface. e.g ifName or ifIndex and so on.

Similarly there could be many such tables in a node containing configuration information about many different objects like, ipAddress, physical components like CPU, fan, power supply etc.
Each table might have possibly different row and column counts based on the object they represent.

This organization is quite natural and agnostic of any protocol or format of data.

NNMi when collecting the data for the first time, as per this invention, creates table of table level, table level, row level and column level hash and store the hash meta data. These hash values are hierarchical. The row hash is computed based on all the cell values in the row. The column hash is computed based on all the cell values in the column. The table hash is computed based on all the row level hashes or column hashes. Row hash based table hash would be useful in dealing with config changes that could happen for a particular object like interface. While column level hash computation could help in cases where state polling is done. From the intersection of row and column level hashes the exact cells which are changed can be asserted. For e.g. if lot of attributes of one object is changed the row hash of that row will be changed/reflect the that it is changed and hence processing is required only for this object and the other objects for which row hash hasn't changed can be skipped from processing. In addition all the columns hashes of the cells of all the attributes will reflect that. From the column and row combination it is easy to ascertain the cells that have changed.

If more than one object change then all the row hashes of those objects will be changed. In this case going by the row hash gives better hit to reflect what is changed while column based hash would give a poor indication of what is changed. And only those objects corresponding to those rows needs to be processed.

In case where one attribute has changed in all the objects or a group of objects then column based hash can reflect which column/attribute has changed. For e.g. ifOperStatus only has changed in a group of interfaces, then the column hash would be changed. And other column hashes would be intact. With that going over the rows it is easy to ascertain the rows who's hash is also changed can be inspected which will deduce for which set of interfaces ifOperStatus hash changed.

The same approach can be applied to a range if required where column hash could be computed for a range for 100 cells if the table has long columns. So that the locating few change in a column of 100s of thousands could be restricted to few range hash miss.

Thus with the combination of row and column hashes it is easy to arrive at what has changed at the table level. If nothing has changed in this table, then the table hash will indicate that and the entire table processing can be skipped.

Table of table hash is computed from all the table hashes. Table of table hash hasn't changed then it indicates nothing has changed on the node and no further processing is required for that node.

Use cases

1. Rediscovery of a node. When a node is rediscovered, all the three levels of hashes are computed.
   - If table of table hash is unchanged -no further processing is required -node is not re-configured.
   - If a specific table hash is changed, only the objects related to that table needs processing - Only IfTable is changed.
   - Within the table which of the row hashes that has changed will give hint as to which of the rows are effected and only objects corresponding to those rows needs to be worked upon -Only certain particular ifEntries are changed.
2. Addition of new objects -when new ipAddresses are added to the node.

   a. During re-dicovery, if the table-of-table hash will be changed from previous discovery hash.
   b. ipAddress table's table level hash will be changed from previous discovery hash.
   c. But all the row hash in ipAddress table from previous discovery will match and new rows with new hashes are discovered.
   d. In this case, only new row's specific new ipaddress objects are created and the rest of the objects are not recomputed.

3. One attribute of all the objects are changed -interface renumbering.
   a. During re-dicovery, if the table-of-table hash will be changed from previous discovery hash.
   b. IfTable table level hash will be different from previous discovery hash
   c. Many or all the interface row hashes are changed because interface ifIndex got renumbered.
   d. Only the column hash for ifIndex is not matching from previous discovery hash values while other column hashes are matching
   e. Now, the rediscovery can pick up new index and re-compute the column hash for ifindex and row hash for all the rows.
   f. This could happen for entire set of rows or could happen for a range of interfaces if range level hashes are implemented.

Advantages

1. Networks are rediscovered every day to capture configuration changes on the network, or more than once per day. This means every node in the network is rediscovered. This means every object in every node is rediscovered and the rediscovery computation is performed.

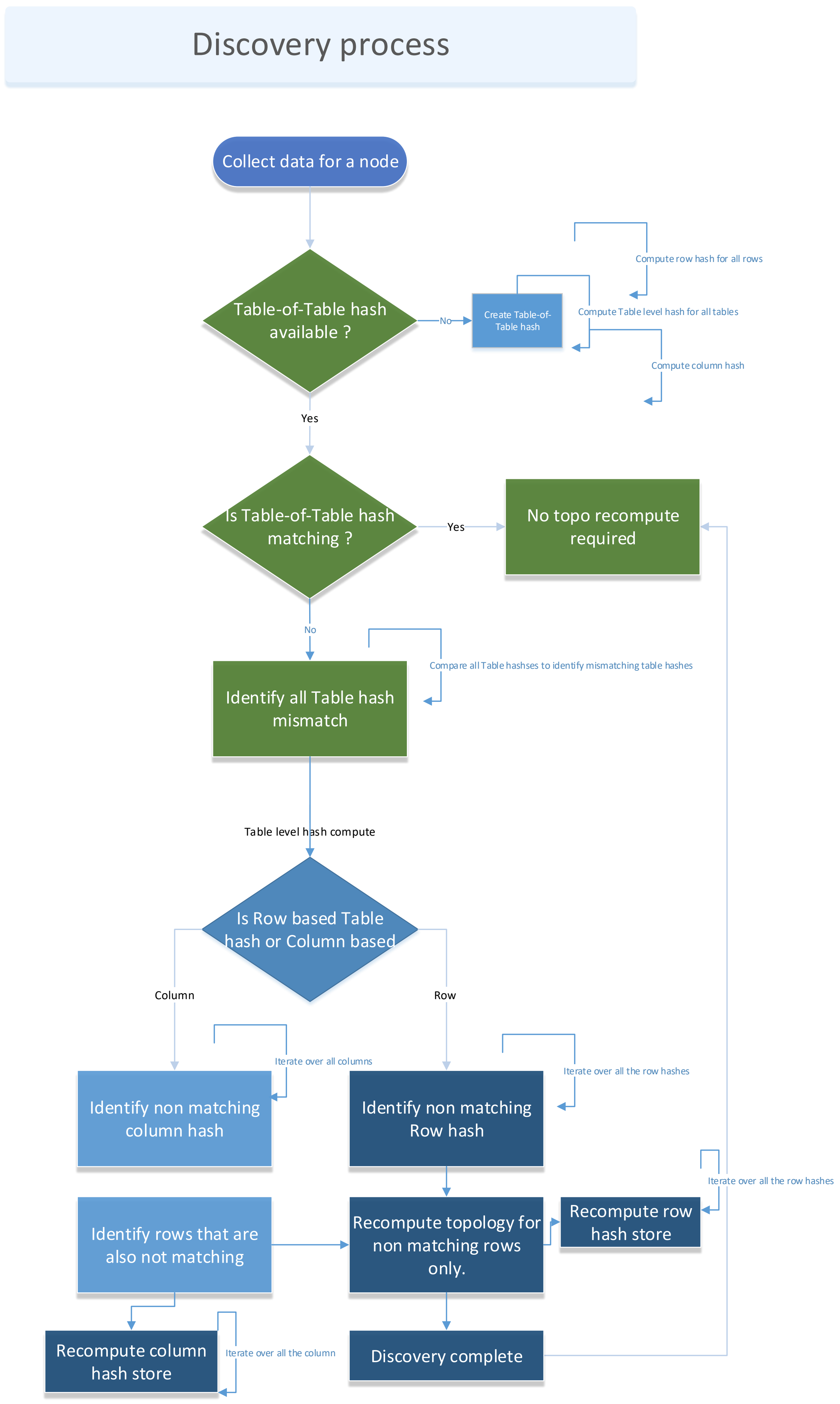
2. But in a typical network only a certain low single digit percentage of nodes get reconfigured, so by detecting smartly whether a change or reconfiguration is done for the node will help in avoiding rediscovery for > 90% nodes per day. This means the discovery process can scale multiple levels.

3. Even within a node, even if reconfigured, only certain objects only will get reconfigured. Detecting those objects that needs to be processed for topology change reduce the system load and time it takes to complete the rediscovery for a node.

4. Selective rediscovery either at object level or at a table level can be very useful for handling 'BigNodes' which can contain upto million objects, detecting the respective change and performing rediscovery on those changed objects only can improve 'BigNode' rediscovery time multi fold.

5. Similarly on status polling, objects like ifOperState, ifAdminState etc are polled near realtime, During every poll when the results arrive every object state is compared with previous poll value to detect a state change. But when millions of objects are polled every minute, but most of the cases there wont be any change in the devices/objects between the polls, so detecting intelligently what has not changed in a node or a group of objects can help in achieving higher scale. In this invention if the column level hash hasn't changed for a node, then that is good enough to conclude nothing has changed which saves millions of comparisons and db lookup. With the row and colum hash the changed objects can be easily detected and only processing is required for those objects.

6. So, this invention helps in smartly detecting the change and improves rediscovery both at node and network level as well as status polling to reach very high scale, which is the need for products like NNM.


Discovery process
Collect data for a node
Table-of-Table hash available ?
No
Create Table-of-Table hash
Compute row hash for all rows
Compute Table level hash for all tables
Compute column hash
Yes
Is Table-of-Table hash matching ?
Yes
No topo recompute required
No
Identify all Table hash mismatch
Compare all Table hashses to identify mismatching table hashes
Table level hash compute
Is Row based Table hash or Column based
Column
Row
Iterate over all columns
Identify non matching column hash
Identify non matching Row hash
Iterate over all the row hashes
Iterate over all the row hashes
Identify rows that are also not matching
Recompute topology for non matching rows only.
Recompute row hash store
Recompute column hash store
Iterate over all the column
Discovery complete