An Oracle White Paper
February 2009

A Load-On-Demand Approach to Handling Large Networks in the Oracle Spatial Network Data Model

ORACLE

Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Executive Overview

Network modeling, management, and analysis are common tasks for enterprise applications such as geographic information system (GIS), customer relationship management (CRM), social network analysis, and in semantic web technologies such as the resource description framework (RDF). Oracle10*g* introduced the Oracle Spatial network data model (NDM), which lets users model and analyze networks. In Oracle10g, NDM uses an in-memory approach to pre-load the whole network into memory before analysis; however, this approach cannot handle networks that are too large to fit in memory. To address this scalability issue, we developed a load-on-demand approach (LOD) in Oracle11*g*. Large networks are first divided into manageable parts called network partitions. Only partitions that are needed are automatically loaded during analysis. In this paper we discuss how LOD works and how to use it.

Introduction

The network data model helps users analyze network connectivity relationships. It is commonly used in transportation, utilities, life sciences, and semantic technologies. It simplifies network modeling, analysis, and management so that users can focus on application logic. It provides an open, generic data model with many common network analysis capabilities. Application information is separated from connectivity information so that the model can be applied to many network applications without customization. NDM further provides a constraint mechanism, to let users guide analysis based on application rules and attributes. For more information on the Oracle Spatial Network Data Model, see references 1,2,3.

On Oracle10g, NDM APIs use an in-memory approach to analyzing networks. The entire network is loaded into memory from the database. Once the network is loaded in memory, users can query and edit it. This approach works well for networks that can be completely loaded into memory; however, it cannot handle networks that are too big to be fit in memory. To address this scalability issue, Oracle11g introduces a load-on-demand approach to handle large networks. Instead of loading the whole network into memory, the network is first divided into manageable subnetworks (network partitions), and only partitions that are needed during analysis are loaded into memory. Loading and unloading of network partitions are automatically managed, thus removing memory as a limiting factor.

Figure 1 shows a US major highway road network. The complete US road network contains around 20 million nodes and 50 millions links. (source: Navteq 2006). Networks of this size cannot be handled in-memory and need to be handled by a load-on-demand approach.

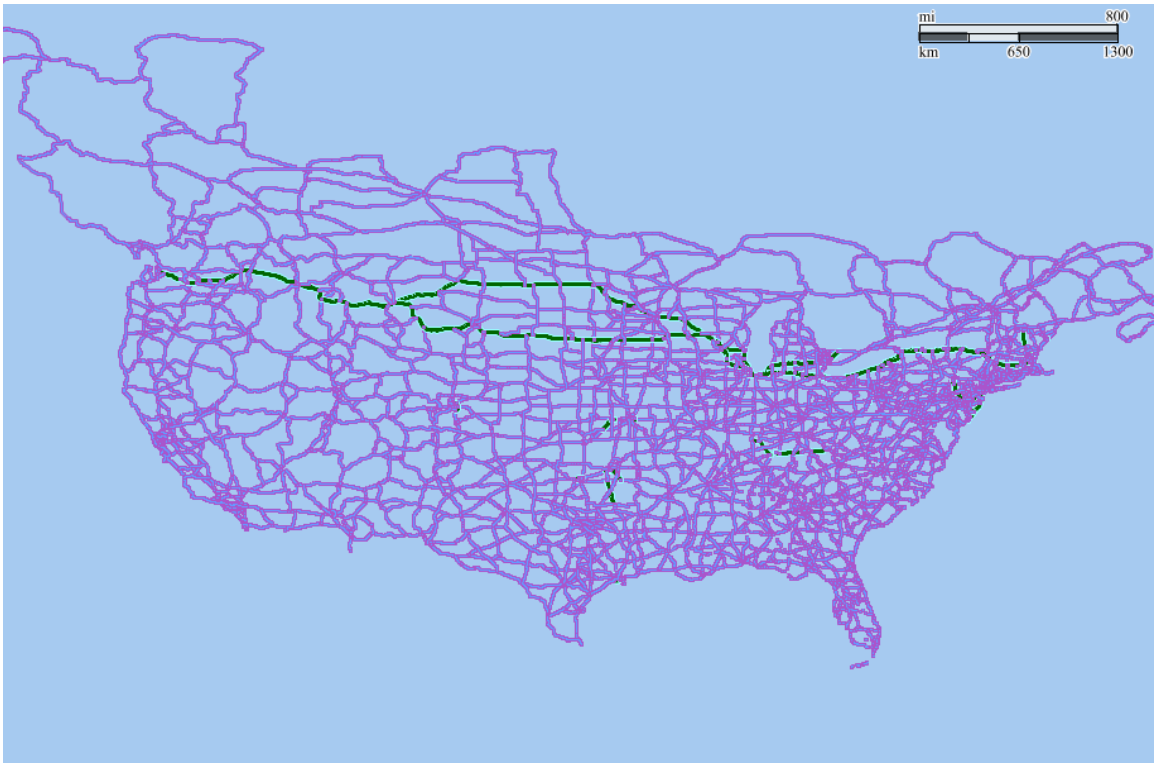

Figure 1. US road network (20 million nodes and 50 millions links, source: Navteq)

LOD uses the same network data model in database as the in-memory approach: network node, link, and path tables, and the network metadata. In addition, though, LOD requires networks to be partitioned first. NDM provides a partitioning procedure for spatial networks, which enables users to partition their spatial networks into network partitions and store the result in a partition table. To further speed up partition loading, this procedure can also create blob representation for each network partition in a partition blob table.

LOD provides the following analysis functions: shortest path and hierarchical shortest path, nearest neighbors, within cost, and reachable and reaching nodes. Additional features such as network constraints, multiple cost support, partial-link paths, and user-defined data are also supported.

This paper is organized as follows: it presents the LOD network data model schema, APIs and architecture; shows how to use the data model; discusses how to use features such as network constraints, a dynamic data set, and multiple link costs to enhance modeling and analysis capabilities; compares major differences between the in-memory APIs and LOD APIs; and describes a tool for visualizing LOD networks and analysis results.

Oracle Spatial Network Data Model

The network data model consists of two parts: a network schema and network APIs. The network schema is the persistent data storage for storing network information. LOD uses the same NDM tables but with additional tables for partitions. The network APIs contain a PL/SQL package for data management in the database, a Java API for data management and analysis on the client side or middle tier, and an XML API for middle-tiered integration.

Network Data Model Schema

A network contains network metadata that includes network tables (node, link, path, and sub-path tables). User-defined data can also be managed by NDM in Oracle11*g*. In addition to NDM network schema, LOD requires additional partition tables. Figure 2 shows the NDM schema.

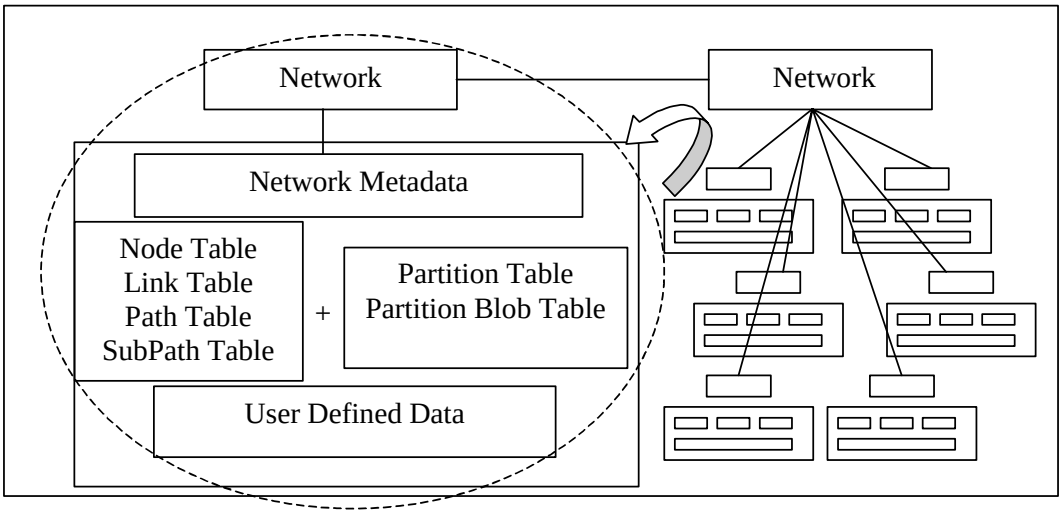

Figure 2. Oracle Network Data Model (Schematic View)

Network Metadata

Network metadata provides general information about networks. Partition information needs to be inserted into the network metadata after a network is partitioned.

Network Tables

An Oracle network contains at least a node table and a link table, and a path table can be added if needed. Figure 2 shows the schema for the network data model, which includes these tables. The schema represents the information necessary for network management and analysis. User-defined data (application attributes) can be added to these tables. Node views and link views are also supported.

The network data model can also handle geometry information. That is, the network data model can represent both logical and spatial network applications. Adding geometric data to a logical network will allow the logical network to be displayed.

User Defined Data

In Oracle11*g*, users can define their own data in the user data metadata view, and NDM will manage the user data as well as the connectivity information. Each specified user-defined data contains the following information:

The following example shows how to insert link user data into the metadata:

- DATA_NAME: the name of the user data. It corresponds to the column name of the data in node/link/path table.
- TABLE_TYPE: the table type of the user data {NODE, LINK, PATH, SPATH}
- DATA_TYPE: the data type of the user data {VARCHAR2, NUMBER, INTEGER, SDO_GEOMETRY}. The corresponding data types in Java are: {String, Double, Integer, JGeometry}.
- DATA_LENGTH: length for VARCHAR2 user data

The following example shows how to insert link user data into the metadata:

```
-- Insert link user data named 'interaction' of type varchar2 (50) in
-- network 'bi_test'.
-- 'interaction' is a column of type varchar2(50) in the link table of
-- network 'bi_test'.
insert into user_sdo_network_user_data
      (network,table_type,data_name,data_type,data_length)
values ('bi_test','LINK','interaction','VARCHAR2',50) ;
-- Insert link user data named 'PROB' of type Number.
--'PROB' is a column of type NUMBER in the link table of network 'bi_test'.
insert into user_sdo_network_user_data(network,table_type,data_name,data_type)
values ('bi_test','LINK','PROB','NUMBER') ;
```

Once a network or network partition is loaded, user-defined data is available in NDM Java representations. You can access user-defined data through the getUserData and setUserData methods for the Node, Link, Path, and SubPath interfaces. For example:

```
String interaction = (String)link.getUserData("interaction");
double prob = ((Double)link.getUserData("PROB")).doubleValue();
```

Partition Tables

LOD requires networks to be partitioned. Once a network is partitioned, the partition result is stored in the partition tables, which are described as follows:

Network Partition Table

- NODE_ID (NUMBER): network node ID
- PARTITION_ID (NUMBER): network partition ID
- LINK_LEVEL (NUMBER): network link level

Network Partition Blob Table

- PARTITION_ID (NUMBER): network partition ID
- LINK_LEVEL (NUMBER): network link level
- BLOB (BLOB): partition Blob
- NUM_INODES (NUMBER): number of internal partition nodes
- NUM_ENODES (NUMBER): number of external partition nodes
- NUM_ILINKS (NUMBER): number of internal partition links
- NUM_ELINKS (NUMBER): number of boundary partition links
- NUM_INLINKS (NUMBER): number of incoming partition links
- NUM_OUTLINKS (NUMBER): number of outgoing partition links
- USER_DATA_INCLUDED (VARCHAR2(1)): 'Y' if the partition contains user-defined data, 'N' otherwise

The partition table name and partition blob table name are stored in network metadata.

Network Partitioning

The NDM node-based partition approach divides network nodes into partitions. Each node is associated with a partition ID. In addition, NDM considers link priorities, so that link priority levels (link levels) can be considered. Each link is assigned a priority to its link_level column. Link priorities have positive integer values starting from 1.

For example, in road networks links are assigned with priorities based on their speed limits. One way to represent such link priorities is to designate local roads as link_level = 1 and state or interstate highways as link_level = 2, thus creating a two-level network. In such multilevel networks, routing can be computed based on the target link level where primary path computation occurs. Note that lower link levels contain all nodes and links of its higher levels.

NDM provides a spatial partitioning procedure to help users partition networks. This procedure generates network partitions based on the maximum number of nodes per partition as specified by the user. A bisecting approach is performed recursively on a network until the desirable number of nodes per partition is achieved. The number of partitions is always a power of 2. In the beginning there is only one big partition that contains all nodes. Each subsequent bisecting will divide the partitions into halves until the number of nodes in each partition is smaller than or equal to the given maximum number of nodes per partition. The partitions are of equal size for each link level.

For example, partitioning a network of 1,000,000 nodes with the maximum number of node per partitions equal to 5000 will generate 256 partitions (3907 nodes/partition). The following formula can be used to obtain the number of partitions, P based upon the total number of nodes, T_v, and maximum number of nodes per partition, P_v:

$P = 2^N$ where $N = CEIL(LN(T_v/P_v)/ LN(2))$, the number of bisections

Note:

- CEIL returns the smallest integer value that is greater than or equal to the argument and LN returns the natural logarithm of the argument.
- The actual nodes per partition = T_v/P_v

The result is stored in a partition table. To further speed up partition loading, blob representations can be generated and stored in a partition blob table. This procedure automatically inserts partition information into its network metadata.

NDM LOD Architecture and APIs

LOD can be deployed in a simple client-server or a multi-tier environment. Figure 3 shows a typical 3-tiered architecture for LOD.

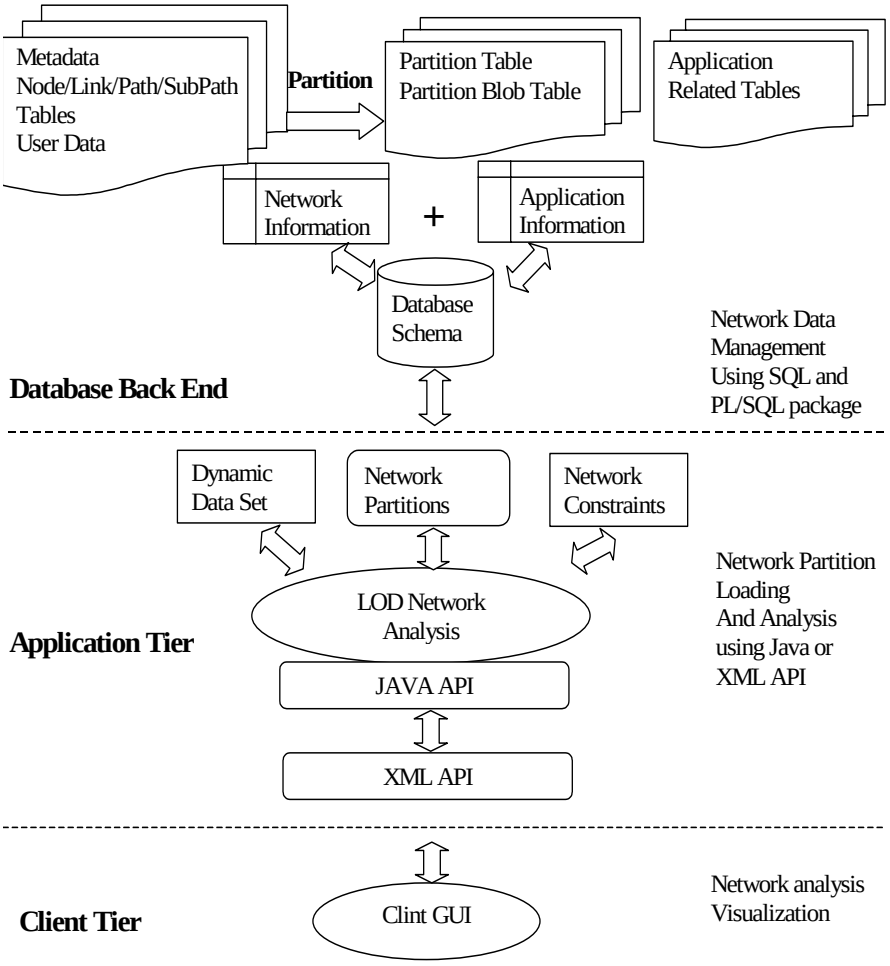

Figure 2. Oracle Network Data Model Architecture

LOD Network Analysis Capabilities

The following analysis functions are supported in the LOD API:

- Shortest Path: The shortest path from node A to node B
- Accessibility Analysis: Is node A accessible from node B?
- Within Cost Analysis: What nodes are within a given cost from (to) a given node?
- Nearest Neighbors: What are the N nearest neighbors of a given node?
- Connected Components Analysis: Label connected components with component IDs.
- Hierarchical Shortest Path: The shortest path based on link priority (link level)

Java Representations of LOD Network Elements

The Java network representations (network, nodes, links, paths, and sub-paths) are defined as Java interfaces and can therefore be extended. These interfaces specify the necessary behaviors for the network and its elements.

The following are some common LOD Java interfaces in the package oracle.spatial.network.lod:

- LogicalNode, LogicalLink, LogicalPath, and LogicalSubPath

  Representations of logical node, link, path and subpath

- LogicalPartition and SpatialPartition

  Representations of network partitions

- LODNetworkManager

  For getting the NetworkIO, NetworkAnalyst, etc.

- NetworkIO

  For reading network information from the database

- NetworkAnalyst

  For handling network analysis

- LODNetworkConstraint and LODAnalysisInfo

  Representations of network constraint and analysis information

For information about each interface, see the Javadoc.

Software Requirements

The load-on-demand feature in network data model is shipped with Oracle Database release 11g. The PL/SQL package is pre-loaded in the database and required Java .jar files are provided; the Java API supports JDK (or JRE) version 1.5 or later. The LOD viewer is included as a utility tool. For more information, see the Oracle Spatial Topology and Network Data Models manual.

Using LOD in Network Data Model

This section explains how to use the LOD approach. There are four major steps: network creation, network partition, partition cache configuration, and load-on-demand analysis.

Creating a Network

Create the network in the database by creating and populating node, link, and (optionally) path tables.

Partitioning a Network

Partition the network by using a spatial partitioning procedure, specifying the maximum number of nodes in each partition. The partition result is stored in a partition table, which is automatically generated, and partition metadata information is inserted into the network metadata. To enhance the performance of network loading, you can store partitions as blobs in a network partition blob table. (Note that LOD will still work on networks that are not partitioned -- these networks will be treated as single-partition networks; however, the use of multiple partitions is recommended for LOD analysis.)

A good partition strategy is to minimize the number of links between partitions, which reduces the number of partitions that need to be loaded and the probable number of times that the same partitions need to be reloaded. Moreover, partitions that are too small require excessive loading and unloading of partitions during analysis.

The recommended maximum number of nodes per partition, assuming 1 GB of memory, is between 5,000 and 10,000. You can tune the number and see what is best for your applications, considering the available memory, type of analysis, and network size. You should also consider configuring the partition caching size.

The following PL/SQL example code shows how to partition a spatial network:

```
exec sdo_net.spatial_partition
(network->'NYC_NET', -- network name
partition_table_name->'NYC_PART$', -- partition table name
max_num_nodes->5000, -- max. number of nodes per partition
log_loc->'MDDIR', -- partition log directory
log_file->'nyc_part.log', --partition log file name
open_mode->'w', -- partition log file open mode
link_level->1); -- link level
The following example creates partition blobs:
exec sdo_net.generate_partition_blobs(
network->'NYC_NET', ,-- network name
link_level ->1, -- link level
partition_blob_table_name->'NYC_PBLOB$', -- partition blob table name
includeUserdata->FALSE, --whether to include user data in partition blobs
log_loc->'MYDIR',  -- partition log directory
log_file->'nyc_part.log', --partition log file name
open_mode->'a'); -- partition log file open mode
```

The preceding two examples generate the necessary partition tables for the NYC_NET network. After executing these examples, you can check the nyc_part.log file for the current status or any errors encountered during partitioning or blob generation.

Configuring the Partition Cache

Before you perform network analysis, you can configure the network partition cache to optimize performance, by modifying an XML configuration file to override the default configuration. You can specify the following:

- Cache size: the maximum number of nodes in partition cache
- Partitions source: from a partition table or a partition blob table
- Resident partitions: ID of partitions that will not be flushed out of the cache
- Cache flushing policy: least recently used (LRU) or frequency based

A sample configuration is shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<LODConfigs xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.oracle.com/spatial/network/lodLODConfigs.xsd"
    xmlns="http://www.oracle.com/spatial/network/lod">
 <!-- default configuration for networks not configured -->
 <defaultLODConfig>
   <LODConfig>
    <readPartitionFromBlob>false</readPartitionFromBlob>
    <partitionBlobTranslator>
     oracle.spatial.network.lod.PartitionBlobTranslator11g
    </partitionBlobTranslator>
    <userDataIO>oracle.spatial.network.lod.LODUserDataIOSDO</userDataIO>
    <cachingPolicy>
     <linkLevelCachingPolicy>
      <linkLevel>1</linkLevel>
      <maxNodes>500000</maxNodes>
      <residentPartitions>-1</residentPartitions>
      <flushRule>oracle.spatial.network.lod.LRUCachingHandler</flushRule>
     </linkLevelCachingPolicy>
    </cachingPolicy>
   </LODConfig>
 </defaultLODConfig>


 ...
 <!-- network to be configured -->
 <networkLODConfig>
<networkName>NYC_NET</networkName>
   <LODConfig>
    <!- read partitions from partition table or from partition blob table -->
    <readPartitionFromBlob>true</readPartitionFromBlob>
    <partitionBlobTranslator>
```

```
      oracle.spatial.network.lod.PartitionBlobTranslator11g
    </partitionBlobTranslator>
    <userDataIO>oracle.spatial.network.lod.LODUserDataIOSDO</userDataIO>
    <cachingPolicy>
     <linkLevelCachingPolicy>
      <linkLevel>1</linkLevel>
      <!-- Maximum number of nodes allowed in cache -->
      <maxNodes>500000</maxNodes>
      <!-- resident partitions -->
      <residentPartitions>-1</residentPartitions>
      <flushRule>oracle.spatial.network.lod.LRUCachingHandler</flushRule>
     </linkLevelCachingPolicy>
     <linkLevelCachingPolicy>
      <linkLevel>2</linkLevel>
      <maxNodes>500000</maxNodes>
      <residentPartitions>*</residentPartitions>
      <flushRule>oracle.spatial.network.lod.LRUCachingHandler</flushRule>
     </linkLevelCachingPolicy>
    </cachingPolicy>
   </LODConfig>
  </networkLODConfig>
</LODConfigs>
```

The configuration can be reloaded if necessary.

LOD provides functions to help you estimate the best caching size. The recommendation, however, is based on logical network information; so if additional user-defined information is stored in network partitions, caching size should be reduced accordingly.

The following PL/SQL function returns the estimated size in bytes for a given network partition:

```
SELECT SDO_NET.GET_PARTITION_SIZE (
  NETWORK->'NYC_NET',
  PARTITION_ID->1,
  LINK_LEVEL ->1,
  INCLUDE_USER_DATA->false,
  INCLUDE_SPATIAL_DATA->'Y') FROM DUAL;
```

Analyzing the Network

After you have created and partitioned the network, and optionally configured the partition cache, you can issue analysis queries. Analysis results are returned in Java representation or XML responses, depending on whether you used the Java or XML API. See the LOD Javadoc or XML schemas for details.

Java API (oracle.spatial.network.lod)z

The following sample code uses the LOD Java API to issue a shortest path query on a network:

```
// load LOD Configuration (XML)
InputStream config = LODTest.class.getResourceAsStream(configXmlFile);
```

```
LODNetworkManager.getConfigManager().loadConfig(config);
// get database connection
Connection conn = LODNetworkManager.getConnection(
  dbUrl, dbUser, dbPassword);
// get LOD network IO Adapter
String networkName = “NYC_NET”;
NetworkIO reader = LODNetworkManager.getCachedNetworkIO(
  conn, networkName, networkName, null);
// get analysis module
NetworkAnalyst analyst = LODNetworkManager.getNetworkAnalyst(reader);
// compute the shortest path
LogicalSubPath path = analyst.shortestPathDijkstra(
  new PointOnNet(startNodeId),
  new PointOnNet(endNodeId), null);
…
```

XML API (oracle.spatial.network.xml)

The following example shows the LOD XML query and result (request and response) for a shortest path query:

```
<?xml version="1.0" encoding="UTF-8"?>
<ndm:networkAnalysisRequest
  xmlns:ndm="http://xmlns.oracle.com/spatial/network"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:gml="http://www.opengis.net/gml">
  <ndm:networkName>NYC_NET</ndm:networkName>
  <ndm:shortestPath>
    <ndm:startPoint>
      <ndm:nodeID>65</ndm:nodeID>
    </ndm:startPoint>
    <ndm:endPoint>
      <ndm:nodeID>115</ndm:nodeID>
    </ndm:endPoint>
    <ndm:subPathRequestParameter>
      <ndm:isFullPath> true </ndm:isFullPath>
      <ndm :startLinkIndex> true </ndm:startLinkIndex>
      <ndm:startPercentage> true </ndm:startPercentage>
      <ndm:endLinkIndex> true </ndm:endLinkIndex>
      <ndm:endPercentage> true </ndm:endPercentage>
      <ndm:geometry>false</ndm:geometry>
      <ndm:pathRequestParameter>
        <ndm:cost> true </ndm:cost>
        <ndm:isSimple> true </ndm:isSimple>
        <ndm:startNodeID>true</ndm:startNodeID>
        <ndm:endNodeID>true</ndm:endNodeID>
        <ndm:noOfLinks>true</ndm:noOfLinks>
        <ndm:linksRequestParameter>
        <ndm:onlyLinkID>true</ndm:onlyLinkID>
        </ndm:linksRequestParameter>
        <ndm:nodesRequestParameter>
          <ndm:onlyNodeID>true</ndm:onlyNodeID>
```

```
      </ndm:nodesRequestParameter>
      <ndm:geometry>true</ndm:geometry>
     </ndm:pathRequestParameter>
   </ndm:subPathRequestParameter>
  </ndm:shortestPath>
</ndm:networkAnalysisRequest>

<?xml version = '1.0' encoding = 'UTF-8'?>
<ndm:networkAnalysisResponse
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ndm="http://xmlns.oracle.com/spatial/network"
  xmlns:gml="http://www.opengis.net/gml">
  <ndm:networkName>NYC_NET</ndm:networkName>
  <ndm:shortestPath>
   <ndm:subPathResponse>
     <ndm:isFullPath>true</ndm:isFullPath>
     <ndm:startLinkIndex>0</ndm:startLinkIndex>
     <ndm:startPercentage>0.0</ndm:startPercentage>
     <ndm:endLinkIndex>17</ndm:endLinkIndex>
     <ndm:endPercentage>1.0</ndm:endPercentage>
     <ndm:pathResponse>
      <ndm:cost>6173.212694405703</ndm:cost>
      <ndm:isSimple>true</ndm:isSimple>
      <ndm:startNodeID>65</ndm:startNodeID>
      <ndm:endNodeID>115</ndm:endNodeID>
      <ndm:noOfLinks>18</ndm:noOfLinks>
      <ndm:linkIDs>145477046 145477044 …… 145476905</ndm:linkIDs>
      <ndm:nodeIDs>65 64 …… 115</ndm:nodeIDs>
      <ndm:geometry>
       <gml:LineString>
        <gml:coordinates>-71.707462,43.555262 -71.707521,43.555601…
        </gml:coordinates>
       </gml:LineString>
      </ndm:geometry>
     </ndm:pathResponse>
   </ndm:subPathResponse>
  </ndm:shortestPath>
</ndm:networkAnalysisResponse>
```

Modeling and Analysis Enhancements

Several features enhance the flexibility of modeling and analysis, and greatly simplify the customization of application requirements. These features and described in this section.

Network Constraints

A network constraint is a user-implemented interface to interact with and guide the LOD analysis engine, based on application information and logic. The interface contains a simple Boolean function that must be implemented by users. Analysis information is passed to this implementation to help determine the feasible links and nodes. The constraint implementation also requires methods that indicate if the constraint requires complete path information and user-defined data.

The following demonstrates how to implement the LODNetworkConstraint interface.

```
public class ProhibitedTurnConstraint implements LODNetworkConstraint
{
 // prohibited turns information
 private HashMap<Long, long[]> pTurnMap = null;

 // construct prohibited turns information
 public ProhibitedTurnConstraint() {
  pTurnMap = new HashMap();
  // add prohibited turns as start link : prohibited end links
  long [] startLinkIDs = {145485662, 145483032};
  long [][] prohibitedEndLinkIDs = {{145477663}, {145477866}};
  for ( int i = 0; i < startLinkIDs.length ;i++)
   pTurnMap.put(startLinkIDs[i], prohibitedEndLinkIDs[i]);
 }

 // check if the given turn (start link ID, end link ID) is allowed
 private boolean validTurn(long startLinkID, long endLinkID) {
  if ( pTurnMap == null ) // no prohibited turns
   return true;
  else {
   long [] prohibitedEndLinks = pTurnMap.get(startLinkID);
   if ( prohibitedEndLinks == null )
    return true;
   else {
    for ( int i = 0; i < prohibitedEndLinks.length;i++) {
     if ( prohibitedEndLinks[i] == endLinkID)
      return false; // prohibited turn found
    }
    return true; // OK
   }
  }
 }
```

```
  public boolean isSatisfied(LODAnalysisInfo info) {
    LogicalLink currentLink = info.getCurrentLink();
    if ( currentLink == null )
      return true; // start node, current link == null
    LogicalLink nextLink    = info.getNextLink();
    return validTurn(currentLink.getId(), nextLink.getId());
  }

  public boolean requiresUserData(){ return false; }
}
```

Multiple Cost Support

In Oracle11g, you can specify multiple costs on a node or a link, whereas in Oracle10g you can associate only one set of costs (major cost) with links and nodes. The ability to specify multiple costs, combined with the use of user-defined data, enables you to switch from different costs in analysis. For example, travel time and travel distance are two commonly used costs for route computation in road networks. In this scenario, you would simply implement a method that returns a specific cost for a node or link and use it in an analysis function. The interface has the following method:

```
public double getLinkCost(LogicalLink link);
```

With such an implementation, the user-specified cost will override the default cost specified in network metadata. This interface can be applied on node cost as well.

The following shows how users can implement their own link cost function for use in cost-related analysis. In this example, the default link cost is the value of the cost column of the link table, and the user defined link cost is the travel time computed from the default link cost and the speed limit for the link level.

```
 public class TravelTimeCalculator implements LinkCostCalculator
 {
   double SPEED_LIMIT_1 = 30;
   double SPEED_LIMIT_2 = 60;

   public double getLinkCost(LogicalLink link)
   {
     int linkLevel= link.getLevel();
     double linkCost = link.getCost();
     switch(linkLevel)
     {
       case 2:
         return linkCost/SPEED_LIMIT_2;
       default:
         return linkCost/SPEED_LIMIT_1;
     }
   }
 }
```

```
public static void main(String[] args)
{
  ...
  //get network analyst
  NetworkAnalyst analyst=LODNetworkManager.getNetworkAnalyst(reader);

  //Shortest path analysis using default cost
  LogicalPath path =
    analyst.shortestPathDijkstra(new PointOnNet(startNodeId),
     new PointOnNet(endNodeId), null);

  //Set travel time as cost
  LinkCostCalculator lcc = new TravelTimeCalculator();
  analyst.setLinkCostCalculator(lcc);

  //Shortest path analysis using travel time as cost
  LogicalPath path =
    analyst.shortestPathDijkstra(new PointOnNet(startNodeId),
     new PointOnNet(endNodeId), null);
  ...
}
```

Precomputed Connected Components

Checking if two nodes are connected is a common query in network applications. You can use this information to eliminate unnecessary computations if two nodes are not connected. In Oracle11g, you can pre-compute connected component information and store it in a connected component table. Each node at the various link levels is given a component ID.

Once the connected component table is generated, you can quickly decide if two nodes are connected by checking the component IDs of these two nodes. (If node A can reach node B, nodes A and B are considered to be connected.) If two nodes are not connected, there is no possible path between node A and B. This information can be used as a filter to avoid unnecessary path computations.

The connected component table is described as follows:

- NODE_ID (NUMBER): network node ID
- COMPONET_ID (NUMBER): network component ID
- LINK_LEVEL (NUMBER): network link level

Note that the information in connected component table is pre-computed and needs to be recomputed if network connectivity is changed.

The PL/SQL procedure for generating the connected components of a network is:

```
exec sdo_net.find_connected_components(
network->'NYC_NET', -- network name            I
link_level->1, -- target link level
component_table_name->'NYC_COMP$', -- component result table
```

```
log_loc->'MYDIR', -- partition log directory
log_file->'nyc.log', -- partition log file name
open_mode ->'a'); -- partition file mode
```

This procedure automatically inserts the connected component table name in the network metadata.

Dynamic Data Set

The dynamic data set is network data (not metadata) that users can temporarily modify in order to affect network analysis. For example, if you want to disable some links in the network temporarily (such as to indicate road segments closed for repairs or by bad weather), you can put the disabled link information in the network dynamic data set and pass the dynamic data set to the analysis function. In this case, for example, when a shortest path is computed, any temporarily disabled links will not be included in the path.

The dynamic data set contains element state, including any changes that need to be reflected in subsequent analysis requests. Users first obtain network elements from the database, but they can then modify attributes of these elements. During network analysis, the dynamic data set overrides any matching element in the data originally loaded from the network partitions. The dynamic data set can be used on a per-query basis or be reused for multiple queries.

The following example shows how the dynamic data set is used to make some links inactive during analysis:

```
...

//get network input/output object
NetworkIO reader = LODNetworkManager.getCachedNetworkIO(
                conn, networkName, networkName);

//construct dynamic data set
NetworkUpdate networkUpdate = new NetworkUpdate();
LogicalLink oldLink = reader.readLogicalLink(linkId, true);
LogicalLink newLink = (LogicalLink) oldLink.clone();
newLink.setIsActive(false);
int pid = reader.readPartitionId(newLink.getStartNodeId(), 1);
networkUpdate.updateLink(newLink, pid);
pid = reader.readPartitionId(newLink.getEndNodeId(), 1);
networkUpdate.updateLink(newLink, pid);

//get network analyst
NetworkAnalyst analyst = LODNetworkManager.getNetworkAnalyst(reader);

//Set network update in the analyst
HashMap<Integer, NetworkUpdate> networkUpdates =
  new HashMap<Integer, NetworkUpdate>();
networkUpdates.put(1, networkUpdate);
analyst.setNetworkUpdate(networkUpdates);
```

```
//find the shortest path with updated network information
LogicalSubPath path = analyst.shortestPathDijkstra(
  new PointOnNet(startNodeId), new PointOnNet(endNodeId), null);

...
```

Partial Link Paths (Sub-paths)

Oracle11g introduces the partial link path (sub-path) feature, to represent a part of a reference path in which the start and end points are specified as, if each case, the start node of a link and optionally a percentage of the distance along the link. A sub-path refers to an existing path by the following parameters:

- Reference path ID: the path ID of the reference path
- Start link index: the start link index on the reference path
- Start percentage: the percentage of the sub-path's start node on the start link
- End Link index: the end link index on the reference path
- End percentage: the percentage of the sub-path's end node on the start link

The cost and geometry information is computed based on the percentage information.

Figure 4 shows a sub-path. It starts at 50% of the first link of the reference path and ends at 50% of the last link of the reference path.

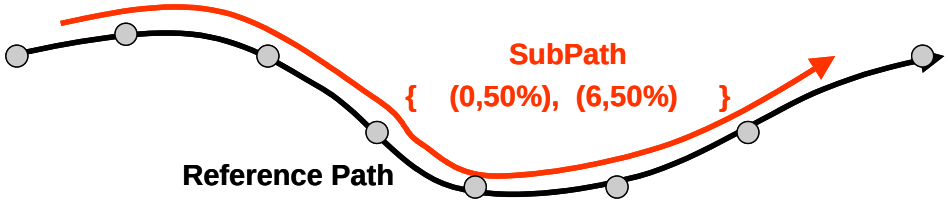

Figure 4. Sub-path Representation

Hierarchical Shortest Path Computation

Shortest path computation could be expensive if the network is large and the path is long. In some applications, sub-optimal solutions are acceptable if the performance (computation time) is good. One common approach to trading path quality for performance is hierarchical path computation.

For hierarchical path computation, a network is divided into different levels based on link priorities. Links of higher priority will be traversed first. This approach dramatically reduces the search space, and usually the results are good compared to the optimal solution. A typical example is the route computation on a road network. A road network is classified into local roads (link level 1), and state and interstate highways (link level 2). The speed limit indicates link priority for computing travel time.

Figure 5 shows a two-level network. If the start and end nodes (denoted by Sn and En) are on link level 1, the hierarchical path will first compute the nearest node (HSn) on link level 2 from the start node (Sn) and the nearest node (HEn) of link level 2 to the end node (En). The hierarchical path is just a path from SP(Sn -> HSn) + SP(HSn -> HEn) + Sp(HEn -> En), where SP(A->B) is the shortest path from A to B. Note that hierarchical paths always try to traverse links of the target link level if possible.

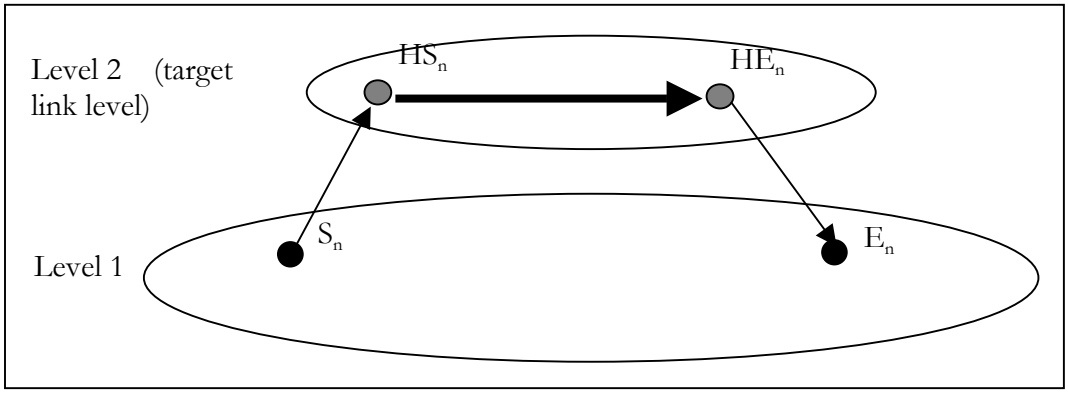

Figure 5. Hierarchical Shortest Path from Sn to En

The following assumptions apply for hierarchical path computation:

- The network is classified into different hierarchical link levels.
- Link level i ,Li needs to connect to link level L(i-1) and L(i+1) if it exists.
- A hierarchical path will always try to traverse links of the target link level.

The following example shows how to compute a hierarchical shortest path on link level 2:

```
...
int linkLevel = 2;
//Compute shortest path using the prefered link level
LogicalPath path = analyst.shortestPathDijkstra(
  new PointOnNet(startNodeId), new PointOnNet(endNodeId), linkLevel, null);
...
```

Comparisons between NDM in-memory API and NDM LOD API

This section explains the differences between the in-memory API (using a Java API and its associated SDO_NET_MEM PL/SQL package) and the LOD API (introduced in Oracle11*g*). It compares their approaches, data model, and analysis functions:

Approaches

The in-memory and LOD APIs take the following basic approaches:

- The in-memory API requires that the entire network be loaded into memory before analysis can be conducted. The in-memory API can be found under the Java package oracle.spatial.network. A PL/SQL interface (SDO_NET_MEM) is built on top of the in-memory API.
- The LOD API requires that the network be partitioned first, and then partitions are loaded into memory whenever needed during LOD analysis. The LOD API can be found under the Java package oracle.spatial.network.lod and in an XML API. The XML schema information is described in the following file:

  $ORACLE_HOME/md/doc/sdondmxml.zip

The in-memory and LOD APIs cannot be mixed in an application. Choose the appropriate API according to your application requirements and network size.

Data Model

The in-memory and LOD APIs use the same network data model, but the LOD API requires additional partition information. (Although the LOD API can handle networks without partition information, such networks will be treated as networks with a single partition.)

- The in-memory API can perform read, write, and edit operations on a network. All network elements are available after a network is loaded.
- The LOD API can only perform read-only operation on a network. Network updates need to be done at database level, and partitions (and partition blobs, if used) need to be updated accordingly. Access to network elements needs to be done through the LOD API.

Analysis Functions

Both the in-memory and LOD APIs support shortest path, nearest neighbors, within cost, and reachable and reaching analysis operations. The in-memory API also supports minimum cost spanning, traveling salesman problem (TSP), and k-shortest paths. The LOD also supports hierarchical shortest path computation, the resulting path of which prefers high-priority links. Network constraints and user-defined data are supported in both the in-memory and LOD APIs.

The NetworkManager class is the main entry for in-memory analysis, and the LODNetworkManager class is the main entry for LOD analysis. Supported analysis functions can be found in the Javadoc for each class.

LOD Analysis Viewer

A viewer tool is provided to help users visualize LOD analysis results. This viewer is delivered on Oracle companion CD, and is not officially supported. It is a standalone Java application built on top of the LOD Java API and Oracle MapViewer client API. It supports most LOD analysis functions and provides simple viewing capabilities.

To use the LOD viewer, first enter database connection information for the database where the network data model (including partition information) is stored. Next, enter MapViewer connection information. For information on how to install and use MapViewer, see the OracleAS MapViewer User's Guide [4]. After the connections are validated, you can perform various analysis operations on the selected network. Zoom-in, zoom-out, and zoom-window functions are available for viewing analysis results. A text window prints out analysis result and statistics. Figure 6 shows a shortest path analysis result.

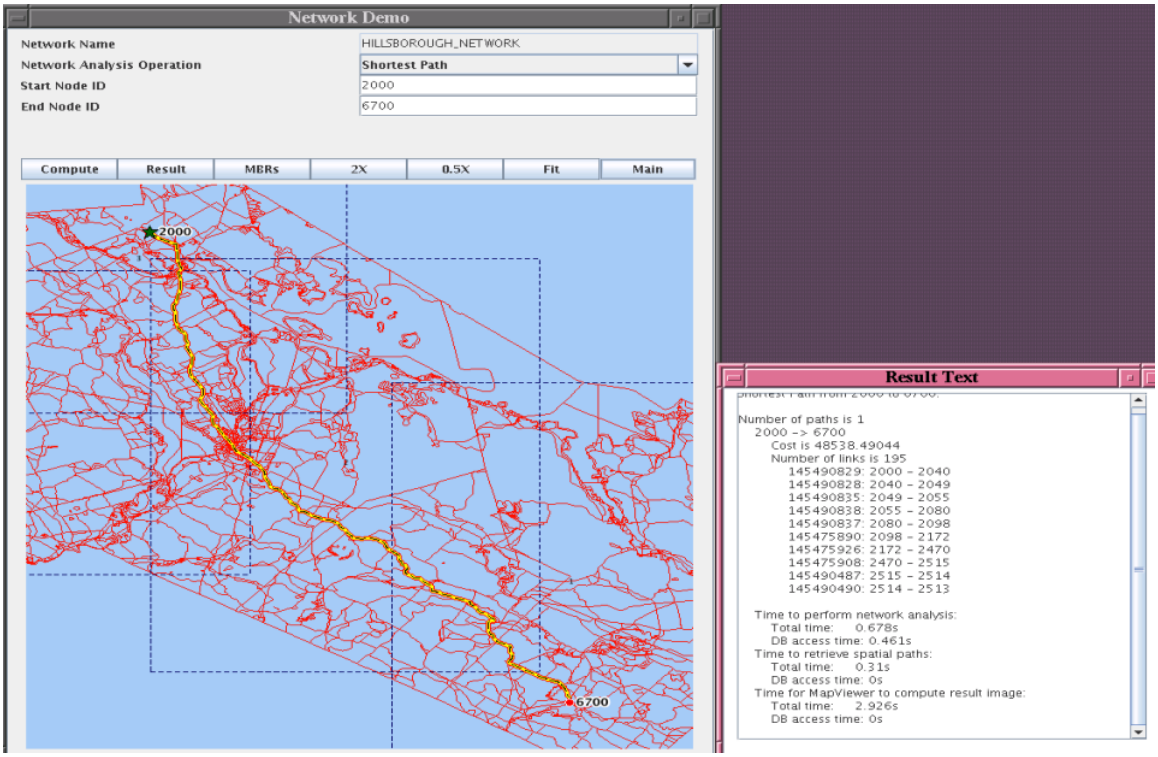

Figure 6. NDM LOD Analysis Viewer

Conclusion

The load-on-demand feature in the Oracle Spatial network data model, available in Oracle Database release 11g, is designed to handle analysis operations on large networks. It provides a scalable and a flexible solution to many network applications. We are currently working with our customers and partners to extend the modeling and analysis capabilities of the network data model.

References

1. Oracle 10g, Oracle Spatial Network Data Model, A Technical White Paper, Oracle Corporation, May 2005

2. Building GIS Applications Using the Oracle Spatial Network Data Model: A Technical White Paper, Oracle Corporation, May 2005

3. Oracle Spatial Topology and Network Data Models, Oracle Corporation.

4. OracleAS MapViewer User's Guide, Oracle Corporation

A Load-On-Demand Approach to Handling Large Networks in the Oracle Spatial Network Data Model
Feb 2009
Author: Jack Chenghua Wang
Contributing Author: Huiling Gong

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Oracle is committed to developing practices and products that help protect the environment

0109