Edge Intelligence

Tutorial

Contents

Introduction

This tutorial walks you through the use of the Edge Intelligence platform and it covers a number of areas including:

- Creating a new network and adding nodes to it.
- Creating tables.
- Defining message formats.
- Injecting message data.
- Executing queries.
- Managing global data.
- Data replication and synchronisation.
- Availability of nodes.
- Administration of users, roles and grants.

By the end of this tutorial, you should have a good understanding of all of the areas above.

Note that it is important to follow the flow presented as some steps are dependent on others.

Install the Platform

This tutorial assumes that you have the Edge Intelligence platform already installed. If not, please follow the Installation Guide.

After installation, you may either have one catalog (for evaluation purposes) or 3 servers (for a standard system); but for simplicity, this tutorial assumes that you have a standard installation of 3 catalog servers.

Vanilla System

Immediately after successfully installing the Edge Intelligence software, you will have three catalog servers each running the Edge Intelligence platform. At this point, you have an empty system with no networks created yet.

The catalog servers provide the portal to any networks you wish to create or access and there are 3 of them to provide durability and high-availability – operations are possible provided any two of the catalog servers are currently up-and-running and available. To perform any operation you need access any one of the available catalog servers.

So start off by connecting to one of your catalog servers using ssh and whatever credentials you have available to you.

User Interface

For this tutorial, we will perform all of the operations using josql which provides an SQL command based interface.

On the catalog server you are connected to, use the following command to start a SQL command session:

```
$ josql star\$administration star\$administrator
```

(Note the use of backslash character to escape the dollar character).

This command opens a SQL session in the star$administration database as the star$administrator user and it will give you a SQL prompt like the following:

```
star$administration=#
```

The star$administration database is the database you use to create and drop networks, and manage users and roles. (More on managing users and roles later.)

The star$administration database provides a star$networks view which lists all of the networks that exist. From the SQL prompt, use the following to list the networks available:

```
star$administration=# SELECT * FROM star$networks;
```

(Note the use of a semicolon character to terminate a SQL command).

You will see the following response:

```
 name | description
------+-------------
(0 rows)
```

This shows that no networks exist.

Now, let's create our first network...

Creating a Network

We can create a network, by simply calling a create_network() function as follows:

```
star$administration=# SELECT create_network('myNetwork','My first network');
```

(Note the use of a single quotes to enclose names and text strings).

This creates an empty network with no nodes, no tables and no data.

Now, when we query the star$networks view, we see:

```
star$administration=# select * from star$networks;

   name    |   description
-----------+-----------------
 myNetwork | My first network
(1 row)
```

We can just as easily drop our network using the drop_network() function:

```
star$administration=# SELECT drop_network('myNetwork');
```

```
star$administration=# select * from star$networks;

 name | description
------+-------------
(0 rows)
```

If you dropped the network, re-create it again using the create_network() function as shown above.

You can use the \df command (describe functions) to list the functions available. For example:

```
star$administration=# \df public.*network
```

Will list the publicly available functions for performing network operations and will show details for the three functions used to create, drop and alter a network.

As we have seen. the create_network() and drop_network() functions do exactly what you would expect them to do. The third function, alter_network() can be used to change the description of a network. For example:

```
star$administration=# SELECT alter_network('myNetwork','My second network');
```

```
star$administration=# select * from star$networks;

   name    |   description
```

```
-----------+--------------------
 myNetwork | My second network
(1 row)
```

Quit from the SQL session in star$administration database using the \q command.

Now that you have created a network, you need to connect to the network to able to anything with it.

Managing a Network

On the catalog server you are connected to, use the following command to start a SQL command session in the network:

```
$ josql myNetwork star\$administrator
```

(Note the use of camel case in the network name).

The network we have created is currently empty and contains no nodes (servers) and no tables (data).

This can be verified using:

```
myNetwork=# select * from star$nodes;

 id | name | parent | description | type | host | ...
----+------+--------+-------------+------+------+ ...
(0 rows)


myNetwork=# select * from star$tables;

 schema_name | table_name | scope
-------------+------------+-------
(0 rows)
```

Nodes

A network typically consists of a tree of interconnected nodes where one or more of the nodes correspond to physical servers located in geographically dispersed locations or data centres.

Each node in the network has a unique name within the network and a node type that indicates its position and function in the network, where the type is one of:

- Instance Node. This is a physical server with an IP address which contains data.
- Edge Node. This is a parent node to one or more instance nodes which replicate data between themselves and operate as redundant mirrors for their edge parent.
- Area Node. This is a grouping node that operates as a parent to one or more edge nodes and/or other area nodes. Area nodes allow a tree of arbitrary depth, width and shape to be built.

Note that an instance node will always appear as a leaf in the network tree and must always be connected to a parent edge node; while an area node will always appear above an edge node.

We can create an instance node using the create_node() function as follows:

```
myNetwork=# select create_node('Instance-A',
                               'My first node',
                               'I',
                               'localhost',
                               null,null,null,null);
```

The following describes each of the first four parameters in order:

- The name of the node
- A description of the node
- The node type
- The address of the host server

The node type of 'I' indicates an instance node (which requires the address of a physical server) and the address of 'localhost' is the current host. Hence this function creates an instance node and deploys it on our current server.

You can see the new node by using:

```
myNetwork=# select name,parent,type,host from star$nodes;

    name    | parent | type |   host
------------+--------+------+-----------
 Instance-A |        | I    | localhost
(1 row)
```

As you would expect, we can create another instance node (with a different name) as follows:

```
myNetwork=# select create_node('Instance-B',
                               'My second node',
                               'I',
                               'localhost',
                               null,null,null,null);
```

Now star$nodes shows the following nodes:

```
myNetwork=# select name,parent,type,host from star$nodes;

    name    | parent | type |   host
------------+--------+------+-----------
 Instance-A |        | I    | localhost
 Instance-B |        | I    | localhost
(2 rows)
```

We can now create an edge and area node as follows:

```
myNetwork=# select create_node('Edge-1',
                               'My edge node',
                               'E',
                               null,null,null,null,null);

myNetwork=# select create_node('Area-1',
                               'My area node',
                               'A',
                               null,null,null,null,null);
```

Note that for these nodes, we do not need to specify a server address and they do not relate to physical servers.

Now star$nodes shows the following:

```
myNetwork=# select name,parent,type,host from star$nodes;

    name    | parent | type |   host
------------+--------+------+-----------
 Instance-A |        | I    | localhost
 Instance-B |        | I    | localhost
 Edge-1     |        | E    |
 Area-1     |        | A    |
(4 rows)
```

Note that in all cases, the parent of the node is shown empty, indicating that the node has no parent associated with it.

We can now start attaching nodes to each other using the attach_node() function. First we can make the instance nodes children of the edge node as follows:

```
myNetwork=# select attach_node('Instance-A','Edge-1');

myNetwork=# select attach_node('Instance-B','Edge-1');
```

Then, we can attach the edge node as a child of the area node:

```
myNetwork=# select attach_node('Edge-1','Area-1');
```

Now star$nodes shows the following:

```
myNetwork=# select name,parent,type,host from star$nodes;

    name    | parent | type |   host
------------+--------+------+-----------
 Instance-A | Edge-1 | I    | localhost
 Instance-B | Edge-1 | I    | localhost
 Edge-1     | Area-1 | E    |
 Area-1     |        | A    |
(4 rows)
```

We can see that all of the nodes now have a parent node apart from the area node which is at the top of the tree. A more visually intuitive view is provided by using the star$topology function as follows:

```
myNetwork=# select rpad(' ',depth)||name as name,type,host
            from star$topology('Area-1')
            order by path;

     name     | type |   host
--------------+------+-----------
 Area-1       | A    |
  Edge-1      | E    |
   Instance-A | I    | localhost
   Instance-B | I    | localhost
(4 rows)
```

Here the name of each node is indented by its depth in the tree and the nodes are presented in depth first order with the area node picked as the root of the tree.

We can also limit the query to just the edge node as follows:

```
myNetwork=# select rpad(' ',depth)||name as name,type,host
            from star$topology('Edge-1')
            order by path;

     name    | type |   host
-------------+------+-----------
 Edge-1      | E    |
  Instance-A | I    | localhost
  Instance-B | I    | localhost
(3 rows)
```

As you might expect, you can also detach nodes from each other. For example,

```
myNetwork=# select detach_node('Instance-B');
```

Now we have a very simple network with one node of each type:

```
myNetwork=# select rpad(' ',depth)||name as name,type,host
            from star$topology('Area-1')
            order by path;

     name     | type |   host
--------------+------+-----------
 Area-1       | A    |
  Edge-1      | E    |
   Instance-A | I    | localhost
 (3 rows)
```

Of course, we can create any number of nodes and connect them in
various arbitrary topologies, but for now, we will keep it as a simple network of 3 nodes.

Tables

We now have nodes in our network, but still no data. To add data, we need to create tables.

A network typically consists of one or more tables that can be queried. Each table in the network has a name and a scope that indicates how data within the table is acquired and replicated around the network:

- Local. This is a table that contains data particular to an edge node – that is, the content of the table will vary between edge nodes. Data is acquired on each instance server and is replicated between sibling instance nodes with the same edge parent. For example, a local table may contain messages from geographically distributed devices. Rows in a local table represent fact data and are considered immutable which can be inserted or removed from the table, but cannot be updated.
- Global. This is a table that contains data universally common across the network. Data in a global table is defined and managed centrally from a catalog server and is replicated between the catalog servers and across all instance nodes in the network. For example, a global table may contain reference data that is common to all instance nodes. Rows in a global table can be inserted, updated or deleted as required.
- Central. This is like a global table, except that the data it contains is perceived to be too sensitive to be replicated around the network.

When performing a query, the scope of the tables involved determine how the query is distributed across the network and where particular query operations are performed.

Let's create some tables around some temperature sensor data.

First, we create our first local table to record temperature readings from temperature sensors

```
myNetwork=# select create_table(
            'temperature_readings',
            'device_id bigint, datetime timestamp, temperature float',
            'L');
```

Here we have created a local table called 'temperature_readings' with three columns to record the identity of the device reporting the reading, when the reading was made and the temperature reported.

Your also list the tables defined using:

```
myNetwork=# select * from star$tables;

 schema_name |      table_name      | scope
-------------+----------------------+-------
 public      | temperature_readings | Local
(1 row)
```

Note that the table is within the default 'public' schema because we did not specify a schema when the table was created. It is possible to create different schemas and place tables within them – but we shall only use the public schema for now.

To list the columns of a particular table use a query like:

```
myNetwork=# select column_name,type
            from star$columns
            where table_name='temperature_readings';

column_name |            type
```

```
-------------+-----------------------------
 device_id   | bigint
 datetime    | timestamp without time zone
 temperature | double precision
(3 rows)
```

Now we have one local table with three columns - but still no data.

We have created a table whose definition is universal and applies to every node in our network. So we can query the table from the perspective of any of the area or edge nodes in the network by using the 'USE NODE' clause in our query. For example,

```
myNetwork=# select *
            from temperature_readings
            use node 'Edge-1';

 device_id | datetime | temperature
-----------+----------+-------------
(0 rows)
```

(Note that the node name is quoted in the USE NODE clause.)

Unsurprisingly, the table is currently empty and contains no rows.

To acquire data for a local table, we need to create a port to map from a message format expected from the source of the data to the columns of the table that will be used to contain the data.

Let us assume that the messages arrive in the following JSON format:

```
{"id":16582178991,"timestamp":"20170101 10:11:06.056","temp":12.5}
```

There is an obvious mapping from the message values to columns in the table and we can configure this with the following port definition:

```
myNetwork=# select create_port('temperature_port',
                               'My temperature port',
                               'temperature_readings',
                               'json',
                               'device_id,datetime,temperature',
                               'id,timestamp,temp',
                               'timestamp',
                               'T',
                               'YYYYMMDD HH24:MI:SS.MS');
```

The parameters used are described below in order:

- The unique name of the port
- A description of the port
- The name of the table to map the messages to
- The type of messages handled by the port
- The columns in the table to map to
- The message keys to map from (in corresponding order of the table columns above)
- The message key of the message identifier to uniquely mark each message from a device
- The type of message identifier used (time).
- The format mask of the message identifier.

As you would expect, there is a view which lists all port definitions:

```
myNetwork=# select * from star$ports;
       name       |     description     |     append_table     | adaptor | ...
------------------+---------------------+----------------------+---------+ ...
 temperature_port | My temperature port | temperature_readings | json    | ...
(1 row)
```

We can see that the messages are being handled by a 'json' adaptor – this is a generic interface that deals with all json messages and performs the mapping work based on the configuration parameters passed to it. There is also an 'sv' adaptor for handling messages of delimited values (such as CSV and TSV) - but we will just use JSON in this tutorial.

So far, we have created a table and port that maps JSON messages to that table. These definitions are universal and apply to every node in the network. Now we need to define sources that will feed messages received on instance servers to the port. Typically, there will be multiple instance servers associated with an edge node and a source is defined for that edge node so that sibling instance nodes share that definition; one there is one source per edge node. In our network, we have just one edge node and we just need one source.

A source simply defines an identity tag for device and we create a source using the create_source() function as follows:

```
myNetwork=# select create_source('temp-1','my first source');
```

Now we can see the source using the star$sources view:

```
myNetwork=# select * from star$sources;

   tag    |   description    | port | node
---------+-----------------+------+------
  temp-1 | my first source |      |
(1 row)
```

Note that the source is not associated with a port or node currently and we need to attach the source to a port and edge node to make it usable. This is performed using the attach_source() function as follows:

```
myNetwork=# select attach_source('temp-1','temperature_port','Edge-1');

myNetwork=# select * from star$sources;

   tag    |   description    |       port       |  node
---------+-----------------+------------------+--------
  temp-1 | my first source | temperature_port | Edge-1
(1 row)
```

We can now feed messages to the 'temp-1' source and those messages will be parsed and injected into the temperature_readings table.

Quit from the SQL command session using \q.

At the OS prompt, use the following command to inject a single message into the 'temp-1' source:

```
$ echo '[{"id":16582178991,"timestamp":"20170101 10:11:06.056","temp":12.5}]' |
es_agent 'localhost' 'myNetwork' 'temp-1'
```

This command pipes a single json message to an agent which connects to an available instance node under the edge node the source is attached to and processes any messages it receives on standard input. Note that the agent expects an array of messages.

You should see something similar to the following in response to the above command:

```
Loading database driver...
Connecting to justonedb database...
Connecting to network database...
Connecting to instance database...
Connected to instance "myNetwork$4" using tag "device1"
Opening session...
Reading input...
Closing session...
Closing database connections...
```

Now that you have injected a single message, you can reconnect to the network using:

```
$ josql myNetwork star\$administrator
```

Then you can query the table again:

```
myNetwork=# select *
            from temperature_readings
            use node 'Edge-1';


  device_id  |        datetime         | temperature
-------------+-------------------------+-------------
 16582178991 | 2017-01-01 10:11:06.056 |    12.5
(1 row)
```

You can now see the one message that was injected.

Given that there is only one edge node, you will see the same result if you perform the query from the area node:

```
myNetwork=# select *
            from temperature_readings
            use node 'Area-1';


  device_id  |        datetime         | temperature
-------------+-------------------------+-------------
 16582178991 | 2017-01-01 10:11:06.056 |    12.5
(1 row)
```

When querying across a whole network, it is often useful to know which edge node each piece of data came from. We can include the edge$() function in our query to provide this information:

```
myNetwork=# select edge$(),*
            from temperature_readings
            use node 'Area-1';

 edge$   | device_tag |      datetime       | temperature
```

```
--------+------------+---------------------+-----------
--
 Edge-1 | temp-1     | 2017-01-01 10:11:06 |        12.5
(1 row)
```

Global Tables

Let's now add a global table that provides reference information for our devices.

```
myNetwork=# select create_table(
            'devices',
            'device_id bigint not null, type text not null, unit char(1) not null',
            'G');
```

This table identifies the device type and the unit of measurement for each device.

We probably want to create a primary key constraint on the device_id column to avoid any duplicate device definitions. We can do that with:

```
myNetwork=# select create_primary_constraint('devices','device_id');
```

You query integrity constraint definitions using the star$constraints view:

```
myNetwork=# select * from star$constraints;

 schema_name | table_name |    type     |  columns  | reference
-------------+------------+-------------+-----------+-----------
 public      | devices    | Primary Key | device_id |
(1 row)
```

Remember that global tables contain data that is universally common across the network and the data in a global table is managed centrally. So rather than injecting data via an agent at each edge, we can insert data into our global table from the catalog server using:

```
myNetwork=# select insert_rows(
                  'devices',
                  'device_id,type,unit',
                  '(16582178991,''temp'',''C''),
                   (16582199712,''temp'',''F'')');
```

This creates two rows in our global table:

```
myNetwork=# select * from devices;

  device_id   | type | unit
--------------+------+------
 16582199712 | temp | F
 16582178991 | temp | C
(2 rows)
```

Now we can join the local and global tables together in a query:

```
myNetwork=# select t.*,d.unit
            from temperature_readings t
            join devices d on d.device_id=t.device_id
            use node 'Area-1';

  device_id  |        datetime         | temperature | unit
-------------+-------------------------+-------------+------
 16582178991 | 2017-01-01 10:11:06.056 |        12.5 | C
(1 row)
```

Taking a Closer Look

At this point it is worth taking a brief look behind the scenes to see which data lives where.

Firstly, when a new network is created a database with the same name as the network is created on the catalog server.

You can list the databases on your current host using the \l command. You will see a database named 'myNetwork' in the list. This is the database you connected to when you connected to the network.

You will also see two other databases with myNetwork in the name followed by a dollar character and a number. These databases correspond to the instance nodes that were created – remember that all of the instance nodes were created on the same catalog server. The number that appears as the suffix of the instance database corresponds to the node identity that appears in the star$nodes view.

For example,

```
myNetwork=# select id,name,parent,status
            from star$nodes;

 id |    name    | parent | status
----+------------+--------+--------
 16 | Area-1     |        | L
  4 | Instance-A | Edge-1 | L
 14 | Edge-1     | Area-1 | L
  6 | Instance-B |        | D
(4 rows)

myNetwork=# \l
                                  List of databases
    Name                |  Owner   | Encoding |   Collate   |    Ctype    | ...
------------------------+----------+----------+-------------+-------------+-----
 star$administration    | justone  | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 |
 justonedb              | justone  | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 |
 myNetwork              | star$    | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 |
 myNetwork$4            | star$    | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 |
 myNetwork$6            | star$    | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 |
 template0              | justone  | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 |
 template1              | justone  | UTF8     | en_GB.UTF-8 | en_GB.UTF-8 |
(7 rows)
```

In the example list, we can see 'Instance-A' has an identity number of 4 and a corresponding database of 'myNetwork$4'.

The identity numbers you see on your own system may differ from the examples above and that's fine.

In the star$nodes view we can see that 'Instance-A' is connected to the edge node whereas 'Instance-B' is not. Similarly, we can see that 'Instance-A' has a status of 'L' (Live) meaning it is connected and active; while 'Instance-B' has a status of 'D' (Detached) meaning it is unconnected to an edge node.

Connect to 'Instance-A' by connecting to the database with the identity suffix for that node using the \c command as follows:

```
myNetwork=# \c myNetwork$4
```

(Note the node identity may be different on your system).

Use the star$tables view to list the tables in that database.

```
myNetwork$4=# select * from star$tables;

 schema_name |      table_name      | scope
-------------+----------------------+--------
 public      | devices              | Global
 public      | temperature_readings | Local
(2 rows)
```

Check that the data in the global table 'devices' that was inserted in the network database (myNetwork) also exists in this instance database:

```
myNetwork$4=# select * from devices;
  device_id  | type | unit
-------------+------+------
 16582199712 | temp | F
 16582178991 | temp | C
(2 rows)
```

Now connect to 'Instance-B' by connecting to the database with the identity suffix for that node using the \c command as follows:

```
myNetwork$4=# \c myNetwork$6
```

(Note the node identity may be different on your system).

Again, use the star$tables view to list the tables in that database.

```
myNetwork$4=# select * from star$tables;

 schema_name |      table_name      | scope
-------------+----------------------+--------
 public      | devices              | Global
 public      | temperature_readings | Local
(2 rows)
```

Check that the data in the global table 'devices' that was inserted in the network database (myNetwork) also exists in this instance database:

```
myNetwork$4=# select * from devices;
  device_id  | type | unit
-------------+------+------
 16582199712 | temp | F
 16582178991 | temp | C
(2 rows)
```

So both instance nodes got a copy of global data because they are both members of the same network (even though 'Instance-B' is currently detached).

Now reconnect to the network database 'myNetwork' using \c.

Availability

At the moment, we have one instance node in our network and if that node were to become unavailable then we would be unable to perform any queries

We can demonstrate this by making node 'Instance-A' unreadable. We can do this with the set_node() function as follows:

```
myNetwork=# select set_node('Instance-A',false,true);

 set_node
----------

(1 row)
```

We can see the status of each node using the star$nodes view:

```
myNetwork=# select name,type,status,readable,writable
            from star$nodes;

    name     | type | status | readable | writable
-------------+------+--------+----------+----------
 Area-1      | A    | L      |          |
 Edge-1      | E    | L      |          |
 Instance-B  | I    | D      | t        | t
 Instance-A  | I    | L      | f        | t
(4 rows)
```

Note that only instance nodes are readable and writable as they are the only node type that are hosted on physical servers – whereas edge and area nodes are purely logical. In the list above, we can see that 'Instance-A' is not currently readable while 'Instance-B' is readable but has a status of D (detached) – meaning it is isolated and not connected to anything.

Next, we need to set our query goal to 'Available' to indicate we want to query from any available instance nodes:

```
myNetwork=# select set_goal('Available',null);
```

(We'll be looking at other goals later on).

Now, if we attempt to query from 'Area-1' we get an error as the edge node is unavailable because it has no readable instance nodes connected to it.

```
myNetwork=# select *
            from temperature_readings
            use node 'Area-1';

ERROR:  No edge nodes found
```

We can make the edge node available by connecting 'Instance-B' to the edge node using attach_node():

```
myNetwork=# select attach_node('Instance-B','Edge-1');
```

Now, when we query the temperature readings table we no longer get the 'No edge node' message:

```
myNetwork=# select *
```

```
            from temperature_readings
            use node 'Area-1';

device_id | datetime | temperature
-----------+----------+-------------
(0 rows)
```

Instead, we see that 'Instance-B' returns no rows from the temperature readings table. Previously, a record was injected into 'Edge-1' when only 'Instance-A' was connected to it and clearly that record has not been replicated onto 'Instance-B'.

The replication of rows between instance nodes operates on blocks of rows and replication only occurs for blocks that are deemed complete. With only one row in the table the current block is still deemed incomplete.

Now, let's inject another message into 'Instance-A'...

Quit from the SQL command session using \q.

At the OS prompt, use the following command to inject another message into the 'temp-1' source:

```
$ echo '[{"id":16582178991,"timestamp":"20170101 10:12:02.016","temp":12.4}]' |
es_agent 'localhost' 'myNetwork' 'temp-1'
```

Let's reconnect to the network and query the temperature readings table:

```
$ josql myNetwork star\$administrator

myNetwork=# select instance$(),*
            from temperature_readings
            use node 'Edge-1';


 instance$  |  device_id  |        datetime         | temperature
------------+-------------+-------------------------+-------------
 Instance-A | 16582178991 | 2017-01-01 10:12:02.016 |        12.4
 Instance-A | 16582178991 | 2017-01-01 10:11:06.056 |        12.5
(2 rows)
```

We can now see the two messages that were injected.

Note the use of instance$() function in the query - this function identifies the instance node that is source of the row. We can see above that both rows came from 'Instance-A'.

Let's re-attach 'Instance-B' – which currently has no rows in its temperature readings table:

```
myNetwork=# select attach_node('Instance-B','Edge-1');
```

Set 'Instance-A' as non-readable:

```
myNetwork=# select set_node('Instance-A',false,true);
```

Now, running the query again we see one row being returned from 'Instance-B':

```
myNetwork=# select instance$(),*
            from temperature_readings
            use node 'Edge-1';


 instance$  |  device_id  |        datetime         | temperature
```

```
------------+-------------+-------------------------+--
-----------
 Instance-B | 16582178991 | 2017-01-01 10:11:06.056 |        12.5
(1 row)
```

We can see that the earliest message that was injected has now been replicated from 'Instance-A' to 'Instance-B'.

As things currently stand, we have two rows in the temperature reading tables on 'Instance-A' and one row on 'Instance-B' with the common row being the earliest.

When we run a query we may receive rows from either of the instance nodes – depending on which node is available at the time of the query; and when both nodes are available, we can set a query goal to influence which node is used by the query - this is explained next.

Query Goals

When one or more instance nodes are collecting data from the same devices, it is likely that those instance nodes will not be in perfect synchronization and the data they held will vary between them from time to time. If one or more instances nodes under an edge node are available at query time, the results we get back from a query depends on which instance node is chosen to service the query.

This can be influenced by setting a query goal to indicate if the node selection should prioritize the node selection based on the most recent data or the most complete data etc.

If we want the fastest possible response time and we don't care which node the data comes from, we can set the goal as 'response' with the 'USE GOAL' clause in a query, for example:

```
myNetwork=# select instance$(),*
            from temperature_readings
            use node 'Edge-1'
            use goal response;


 instance$   |  device_id  |        datetime         | temperature
-------------+-------------+-------------------------+-------------
 Instance-B | 16582178991 | 2017-01-01 10:11:06.056 |        12.5
(1 row)
```

Let's make 'Instance-A' available and query the table again for the fastest response:

```
myNetwork=# select set_node('Instance-A',true,true);

myNetwork=# select instance$(),*
            from temperature_readings
            use node 'Edge-1'
            use goal 'Response';


 instance$   |  device_id  |        datetime         | temperature
-------------+-------------+-------------------------+-------------
 Instance-B | 16582178991 | 2017-01-01 10:11:06.056 |        12.5
(1 row)
```

We know that 'Instance-A' contains two rows while 'Instance-B' contains one row (the oldest row), hence 'Instance-A' contains both the most recent and the most comprehensive data set. Hence, if we are most interested the latest data, we can set our goal as 'latest':

```
myNetwork=# select instance$(),*
            from temperature_readings
            use node 'Edge-1'
            use goal latest;


 instance$   |  device_id  |        datetime         | temperature
-------------+-------------+-------------------------+-------------
 Instance-A | 16582178991 | 2017-01-01 10:12:02.016 |        12.4
 Instance-A | 16582178991 | 2017-01-01 10:11:06.056 |        12.5
(2 rows)
```

Users, Roles & Permissions

So far, we performed all of our queries as the star$administrator user – a user with many privileges; but for general use, we will want to create standard users and create roles for them so that we can limit their access to certain tables, columns and/or nodes in the network.

To manage users and roles we need to connect to the administration database and create users, roles and grant roles to specific users.

To connect to the administration database directly from the network database we can use the \connect command:

```
myNetwork=# \connect star$administration star$administrator
```

Then we can create a user:

```
star$administration=# select create_user('me');


star$administration=# select * from star$users;

     user_name      | login
--------------------+-------
 justone            | t
 star$              | t
 star$administrator | t
 star$source        | t
 star$super         | t
 me                 | f
(6 rows)
```

We can see the new user has been created, but that user is not able to login currently. This is the default for all new users. After creating the user, we can alter the user to allow logins and set their login password using:

```
star$administration=# select alter_user('me',true,null);
```

The second parameter to alter_user() indicates if logins are allowed and the third parameter is the password assigned to the user – but we'll avoid using passwords for now. Now we can see the user is able to login:

```
star$administration=# select * from star$users;

     user_name      | login
--------------------+-------
 justone            | t
 star$              | t
 star$administrator | t
 star$source        | t
 star$super         | t
 me                 | t
(6 rows)
```

The defined user will be able to login onto any network we define. For example, the user can connect to myNetwork:

```
star$administration=# \connect myNetwork me;
```

But the new user has no privilege to query the temperature table:

```
myNetwork=# select instance$(),*
            from temperature_readings
            use node 'Edge-1';

ERROR: permission denied for relation temperature_readings
```

So, we have to explicitly grant permission to the user to enable them to query the table.

But rather than manage granular permissions for every individual user, we do this via roles whereby roles are granted permissions to specific objects and users can be assigned to one or more roles.

Let's create a role and grant it to the user:

```
myNetwork=# \c star$administration star$administrator
star$administration=# select create_role('myrole');
star$administration=# select grant_user('me','myrole');
```

Now grant the role access to the table:

```
star$administration=# \c myNetwork star$administrator
myNetwork=# select grant_table('temperature_readings','myrole');
```

Now the new user can query the temperature table:

```
myNetwork=# \c myNetwork me
myNetwork=# select instance$(),*
            from temperature_readings
            use node 'Edge-1'
            use goal latest;

instance$   |  device_id  |        datetime         | temperature
------------+-------------+-------------------------+-------------
 Instance-A | 16582178991 | 2017-01-01 10:12:02.016 |        12.4
 Instance-A | 16582178991 | 2017-01-01 10:11:06.056 |        12.5
(2 rows)
```