# Edge Intelligence

## User Guide

# Contents

Edge Intelligence 7.0 Documentation

## What is the Edge Intelligence Platform?

The Edge Intelligence platform orchestrates a highly geographically distributed network of Edge Intelligence stores to enable 'single pane of glass' control and query access across the entire network.

The platform enables data to be retained at the edge of a network to overcome the issues that arise with shipping data to a central location. Retaining data at the edge has a number of virtues, including:

- Overcoming the bandwidth limitations of wide-area networks and the Internet.
- Avoiding the cost of shipping terabytes of data across networks.
- Enabling exhaustive data retention and avoiding loss of data from enforced filtering.
- Increased reliability and reduced latency by enabling autonomous data processing at the edge of the network.
- Avoiding the movement of sensitive data over public networks.
- Enables geo-political movement restrictions to be respected.

The Edge Intelligence Platform provides the infrastructure to enable hybrid clouds and geographically distributed data lakes for situations where location enforcement and/or network bandwidth limitations prevent the movement of bulk data over public networks.

The platform provides a number of key features in this architecture:

- Provides a central management console for an entire network.
- Changes are made once and are automatically rolled-out and autonomously implemented across a network.
- Agile database schemas that respond to evolving requirements without requiring any prior knowledge of those requirements to provide a future proof solution.
- Autonomous edge servers that perform at network speed and deliver excellent response times to arbitrary queries.
- Distributed resources can be provisioned, deployed and controlled at a distance.
- Distributed operations can be monitored and controlled at a distance.
- Networks can be arranged to provide high-availability and resilience against both network connectivity and hardware failures.
- Networks can be arranged into arbitrary regional hierarchies and queries applied to the network as a whole and to arbitrary areas of the network.
- Query performance is optimised by distributing queries in parallel across the network and pushing query processing to the data at the edge of the network.
- Data transit is minimised by shipping only queries and their results over the network.
- In-flight data is protected from exposure through encryption.
- Data is secured through a role based security framework which restricts who can see which data and where.

The Edge Intelligence Platform presents an industry standard SQL interface over ODBC or JDBC protocols using a standard PostgreSQL driver. Hence the platform is compatible with most standard query clients and business intelligence tools.

## What is an Edge Server?

An edge server operates at the edge of the network to collect exhaustive detailed data and retain it indefinitely while providing fast response to arbitrary queries. An Edge Intelligence server is capable of:

- Design-free operation
- Autonomous operation in a lights-out environment
- Data acquisition at network speeds
- Excellent response times to arbitrary, unplanned and unforeseen queries
- Retention of very large data volumes
- Exceptional hardware efficiency

A key characteristic of an Edge Intelligence server is its ability to rapidly respond to arbitrary queries without requiring any prior design for performance. Tables are defined as standard relational tables, but there is no need to design or implement any indexing or partitioning structures to achieve good performance – regardless of the nature of the queries submitted - from granular forensic queries to extensive analytical queries.

This characteristic removes any need to tune or optimise an edge server and allows the server to operate autonomously. This is important for edge servers operating at the edge of a network where a widely distributed deployment makes close supervision and regular intervention in the operation of each and every server impractical.

An Edge Intelligence server is exceptionally hardware efficient and is able to deliver excellent performance from a small number of high-capacity high-latency disks (8TB HDDs for example), which allow a single server to readily manage many terabytes of data and acquire that data at rates of hundreds of thousands of messages per second whilst retaining that data for many months or years.

## Architecture

An installation of the Edge Intelligent platform consists of three catalog servers that provide the meta-data repository and the portal for access to a network of edge servers.

All of the catalog servers typically contain identical data. There are three catalog servers to provide high-availability and operations are possible while any two of the catalog servers are up and available where users can connect to any one of the currently available catalog servers to perform operations or queries.

An important point to understand is that any number of independent networks can be created, managed and accessed from the same set of catalog servers. The catalog servers simply provide a central point of control and access for one or more networks.

Each network comprises a population of geo-distributed edge servers – all of which are managed, operated and queried through a catalog server. A single edge server can also be shared between multiple networks, if required.

Transparently to users, the platform will replicate changes made by the users between the catalog servers and propagate the changes to edge servers as required. The majority of propagation and synchronization work happens asynchronously so that changes can be performed while parts of the network are unavailable. Yet, from a user's perspective all changes are immediate and appear consistent across an entire network.

To guarantee consistency, any attempt to perform operations and queries when only one catalog server is available will cause an exception to be raised.

## Vanilla System

Immediately following a fresh installation, there will only exist three catalog servers, no edge servers and no networks. On each catalog server, a star$administration database and a star$administrator user will exist.

The star$administration database is used to perform the following operations:

- Create/drop networks
- Create/drop roles
- Create/drop users and alter their login credentials

Users with login credentials are able to connect to any of the networks, but their access rights will likely vary between each of the networks.

## Building a Network

The steps involved in building a network are covered in detail in the remainder of this document, but are summarized here where all of the following steps are performed from a catalog server connection.

- Define the names and credentials of users who will have access to the system.
- Remotely provision new edge servers, as and when required.
- Create one or more networks.
- In each network:
  - Create nodes to model the topology of the network
  - Create database objects to model the schema of the network
  - Create ports and sources to map from input message formats to relational tables
  - Define global reference data

A worked example of creating a network, node and tables is provided at the back of this document.

## User Interface

The user interface is API based for easy dev-ops integration and scripting and within a network, an operation is performed via a function call in simple SQL statement.

For example, to create a new user, the administration user can connect to the star$administration database and use the following command:

```
SELECT create_user('joe');
```

Where joe is a function argument identifying the name of the new user to be created.

Similarly, the following command gives joe login access and sets his password for connecting to the server.

```
SELECT alter_user('joe', true, 'password123');
```

Some functions permit optional arguments and where an argument is not supplied, a NULL argument value should be used. For example, to disable logins for user joe without changing his password, you would use the following command.

```
SELECT alter_user('joe', false, NULL);
```

Most of the command functions return a void result to indicate a success and will raise an exception if there is a problem.

For example, if there is an attempt to create another user with the name joe, we would see

```
SELECT create_user('joe');

ERROR: role "joe" already exists
```

These functions can be called programmatically from an application via a language interface (for example JDBC) or manually from a SQL command console, such as josql.

Using josql you can obtain a list of command functions by using the \df command.

## Identifiers

The names used for objects such as networks, nodes and tables are required to observe the following rules:

- They may contain letters, numbers and underscores only.
- They must begin with a letter or underscore.

Examples of valid object names include

- myTable
- table1
- table_1
- _table1

Examples of invalid object names include

- my.table
- "my table"
- 0_table
- table$

Note that database object names (such as table names) are typically case insensitive while networks and nodes are not. If uncertain, it is best to assume that an identifier is case sensitive.

**Note that some predefined system identifiers (such as standard user names and function argument names) are prefixed with "star$" (star$administration for example) while user defined identifiers are prohibited from containing the dollar character.**

## Messages

Users can control the severity level of messages they receive from functions they call by using the set_messages() function. This function accepts one of the following severity levels:

- INFO. Messages with a severity level of information or above are output.
- WARNING. Messages with a severity level of warning or above are output.
- ERROR. Messages with a severity level of error or above are output.

Information level messages can be ignored while messages with a severity level of WARNING or above require the attention of the user.

When a session is started, it is recommended that the following is used to avoid the output of informational messages:

```
SELECT set_messages('warning') ;
```

## Managing Users

After installation a single user called star$administrator exists for performing administrative tasks such as managing users and roles.

Users can be created and dropped as needed and their login rights and credentials can be altered as and when required. When a new user is created, they have no login rights and no password set so it is also necessary to alter the user to enable them to connect to a catalog server.

After installation, the following standard users are defined.

| User | Description |
|---|---|
| star$administrator | Administration user for managing users, roles and networks |
| star$source | Restricted user for connections from data collection agents |
| star$ | Proxy user for remote connections |

The following functions are used to manage users:

| Function | Description | Arguments |
|---|---|---|
| create_user | Creates a new user | user$ - name of the user to be created |
| drop_user | Drops an existing user | user$ - name of user to be dropped |
| alter_user | Changes the login right of the user and/or their authentication password | user$ - name of user to be altered<br>login$ - boolean setting for enabling login (optional)<br>password$ - optional password to be set (optional) |

Disabling login for a user prevents that user from connection to any of the catalog servers.

**Note that the alter_user function is the only function that requires all of the catalog servers to be available when the function is called. If any one of the catalog servers is unavailable, the function will return an error**.

There is a star$users view which provides a list of all known users:

| Column | Description |
|---|---|
| user_name | Name of the user |
| login | Indicates if the user has login enabled |

Users can have roles which bestow permissions for performing certain operations and certain queries.

After installation, the following standard roles are defined.

| Role | Description |
|---|---|
| star$administrate | Administrate networks, users and roles |
| star$grant | Grant or revoke roles to and from users and objects |
| star$network | Manage network topology |
| star$define | Define database objects (DDL) |
| star$modify | Modify database objects (DML) |
| star$query | Perform queries |
| star$append | Append data to local tables |
| star$purge | Purge data from local tables |

A standard user must be granted one or more of the above roles to perform the operations associated with those roles. For example, a user would be granted the star$query role to allow them to perform queries.

The star$administrator user is a super user who has access to all of the roles above and can be used to configure the users, roles and networks required immediately after installation. Thereafter, it is advisable to disable login access for the star$administrator to prevent any wide ranging super user access.

Additional roles can also be created for the purpose of controlling which data is visible to particular users. This is discussed in greater detail in the security section of this document, but for now it is important to understand that new roles can be created and that roles can be granted or revoked from users to control their data access rights.

The following functions are used to manage roles:

| Function | Description | Arguments |
|---|---|---|
| create_role | Creates a new role | role$ - name of the role to be created |
| drop_role | Drops an existing role | user$ - name of role to be dropped |
| grant_user | Grants a user access to a role | user$ - name of user to be granted a role<br>role$ - name of role to be granted |
| revoke_user | Revokes access to a role from a user | user$ - name of user to revoke role from<br>role$ - name of role to be revoked |

The star$roles view provides a list of known roles:

| Column | Description |
|---|---|
| role_name | Name of the role |

There is also a star$roles function which provides a list of roles for a given user and returns one row per role. This function can be used as follows:

```
select * from public.star$roles(<user>);
```

Where <user> is the name of the user to get the roles for. To use this function you must be a superuser or be a member of the star$grant role.

## Managing Networks

The Edge Intelligence platform can manage multiple networks and these can be created and dropped as required. Each network operates independently and multiple networks are only used where the physical separation of discrete data sets is required. In particular, note that:

- Multiple networks must be managed separately.
- Queries can only operate within the context of one network at a time.
- Data cannot be shared between multiple networks.

Networks are created and dropped by connecting to the star$administration database and using the following functions:

| Function | Description | Argumentsdrop |
|---|---|---|
| create_network | Creates a new network | network$ - name of the network to be created<br>description$ - description of the network (optional) |
| drop_network | Drops an existing network | network$ - name of network to be dropped<br>cascade$ - indicates if dependent nodes should be dropped |
| alter_network | Changes the description of a network | network$ - name of the network to be altered<br>description$ - new description of the network (optional) |

The cascade option of the drop_network function allows all of the nodes and databases associated with a network to be dropped in a single command.

**Note that an attempt to use drop_network without the cascade option will fail if there are any dependent nodes in the network.**

There is a star$networks view in the star$administration database which provides a list of all known networks:

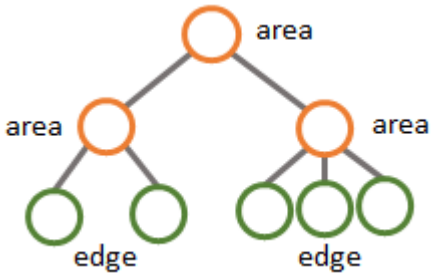
| Column | Description |
|---|---|
| name | Name of the network |
| description | Description of the network |

Following installation, no networks will exist and at least one network will need to be created as described above.

## Network Topology

A network comprises of one or more edge nodes which denote the edge points of the network.

Edge nodes can be grouped under area nodes for the purposes of management, visibility and query access. Area nodes can also be grouped under other area nodes to create hierarchies of nodes. For



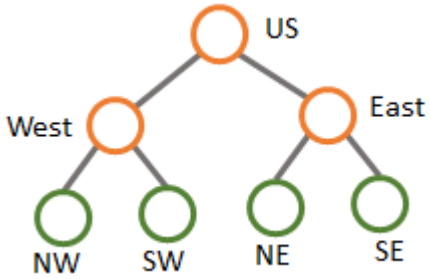

example, a network of nodes may be arranged as:

The topology and shape of a network is completely arbitrary except for the following rules:

- Each node can only have at most one parent node
- A parent node must be an area node (not an edge node).

Each node is given a unique name and it is possible to control and query regions of the network by qualifying operations with a node name.

For example, a simple topology could be named as follows.




Queries can be submitted to any of the nodes in this topology to provide a view of all of the edge nodes at or under the queried node. For example, a query submitted to the US node will provide a view of the entire network; while a query submitted to the East node provides a combined view of the NE and SE edge nodes; and a query can also be submitted to an individual edge node.

Similarly, node specific operations can be targeted at regions of the network as well as individual edge nodes.

To provide high-availability and resilience to network and hardware failures, data is collected and stored in one or more physical servers connected beneath each edge node. These are represented as instance nodes where each instance node represents a replica of the data available at the edge

node. For example, the data for an edge node may be replicated across two instance nodes as illustrated below.

The data stored on the instance nodes under an edge node is considered to be identical and,


typically, each instance node will be hosted on a different physical server to the other instance nodes under the same edge node. A query reaching an edge node will use one of the currently available instance nodes under that edge point - to afford high-availability for the query.

Each edge node can have any number of instance nodes beneath it - but each instance node can only be attached directly to one edge node. In particular, instance nodes cannot be attached to each other and cannot be attached to area nodes.

The following summarizes the rules governing network topology:

- A node is one of an area node, edge node or instance node.
- Each node can only be the child of one parent node at most.
- An instance node can only be the child of an edge node.
- An edge node can only be the child of an area node.
- An area node can only be the child of another area node.

Nodes are managed by connecting to the network database and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_node | Creates a new node | node$ - name of the node to be created<br>description$ - description of the node (optional)<br>type$ - type of node (A,E,I)<br>host$ - server host address for an instance node<br>address$ - address information (optional)<br>location$ - location information (optional)<br>latitude$ - latitude of the node (optional)<br>longitude$ - longitude of the node (optional) |
| drop_node | Drops an existing node | node$ - name of node to be dropped |
| alter_node | Changes the details for a node | node$ - name of the node to be altered<br>name$ - new name for the node (optional)<br>description$ - new description of the node (optional)<br>address$ - new address information (optional)<br>location$ - new location information (optional)<br>latitude$ - new latitude of the node (optional)<br>longitude$ - new longitude of the node (optional) |
| attach_node | Attaches a node to a parent node | node$ - name of node to be attached<br>parent$ - name of parent node to attach to |
| detach_node | Detaches a node from its parent node | node$ - name of node to be detached |

| | | |
|---|---|---|
| set_node | Sets an instance node as readable and/or writable | node$ - name of instance node to be set<br>readable$ - readable state to be set (optional)<br>writable$ - writable state to be set (optional) |
| grant_node | Grants query access for a role to a node | node$ - name of node to grant access to<br>role$ - name of role to be granted access |
| revoke_node | Revokes query access for a role from a node | node$ - name of node to revoke access from<br>role$ - name of role to be revoked access |

Networks can be built by defining the constituent nodes in any order and by attaching defined nodes in any order. Moreover, existing networks can be re-shaped by detaching and re-attaching nodes as required.

### Changing Network Topology

The topology of a network can be any hierarchical shape and can be changed at any time by detaching and/or attaching child nodes to parent nodes. However, the following should be borne in mind:

- Area nodes can be detached and attached to area nodes at any time; similarly, edge nodes can also be detached and attached to area nodes at any time.
- An instance node can only receive data and be queried if it is attached to an edge node.
- Detaching an instance node from an edge node effectively drops the data from that instance node. Hence detaching the instance node from an edge node which only has one instance node attached will drop all of the data collected at that edge.
- An instance node can only be attached to an edge node if it is not already attached.
- After attaching an instance node to an edge node it will synchronize its data with any sibling instance nodes under the same edge node.

There is a star$nodes view in a network database which provides the following information for the nodes in the network:

| Column | Description |
|---|---|
| id | Unique node identity |
| name | Name of the node |
| parent | Name of the parent node |
| description | Description of the node |
| type | Node type. A, E or I for area, edge or instance node respectively |
| host | The server host address for an instance node |
| address | Address information for the node |
| location | Location information for the node |
| latitude | Latitude of the node |
| longitude | Longitude of the node |
| readable | Indicates if an instance node is currently readable |
| writable | Indicates if an instance node is currently writable |

There is a star$topology function in a network database which provides the following topology information about nodes in the network below a given starting node:

| Column | Description |
|---|---|
| depth | Depth of the node relative to the starting node |
| path | Node path from the starting node to this node |
| node_id | Identity of this node |
| parent_id | Identity of parent node |
| name | Name of this node |
| type | Node type. A, E or I for area, edge or instance node respectively |
| host | The server host address for an instance node |
| readable | Indicates if an instance node is currently readable |
| writable | Indicates if an instance node is currently writable |
| address | Address information for the node |
| latitude | Latitude of the node |
| longitude | Longitude of the node |
| location | Location information for the node |
| description | Description of the node |

This function requires a single node parameter which identifies the starting node and the function returns information for every node at and below the starting node.

The depth column can be used with the rpad() function to indent the node name based on the depth of the node; and the path column can be used to order the results in depth first order. For example:

```
SELECT rpad(' ',depth)||name AS name, type, description
FROM star$topology('US')
ORDER BY path ASC;
```

Might return something like the following, which provides a visual outline of the network hierarchy:

```
  name   | type | description
---------+------+-------------
 US      |  A   | United States
  West   |  E   | Western edge
   NW    |  I   | Northwest instance
   SW    |  I   | Southwest instance
  East   |  E   | Eastern edge
   NE    |  I   | Northeast instance
   SE    |  I   | Southeast instance
```

## Managing Databases

Each network presents a relational data model in which data is stored in tables of columns and rows and where the definition of any database object, such as table, can be changed dynamically at any time.

The definition of database objects is managed independently for each network; but within a single network, object definition is common across all nodes and is managed centrally and singularly from a connection to a catalog server. For example, creating a table in a network simply involves creating the table once and the existence and definition of that table is automatically propagated out to all of the instance nodes in the network.

The following objects can be created, altered or dropped in a network:

- Schema (namespace for database objects).
- Table.
- Constraint (primary key, unique key and foreign key constraints)
- View.
- Function.
- Sequence.

**Note that there are no features provided to manage indexes, partitioning or sharding because the edge stores do not require them - regardless of the volume data being managed.**

The following sections describe the management operations possible for each object type.

## Managing Schemas

A schema is a namespace for other database objects and allows the same object name to be used in different schemas without conflict; for example, schemas schema1 and schema2 can both contain a table named mytable. A user can access objects in any of the schemas in the network he is connected to, if he has privileges to do so.

Schemas are typically used to organize database objects into logical groups to make them more manageable.

Schemas are managed by connecting to the network and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_schema | Creates a new schema | schema$ - name of the schema to be created |
| drop_schema | Drops an existing schema | schema$ - name of schema to be dropped<br>cascade$ - indicates if drop cascades to dependent objects. |
| rename_schema | Renames an existing schema | old$ - current name of the schema<br>new$ - new name for the schema |

Naturally, if you attempt to create a schema using the name of an existing schema an exception will be raised; and similarly if you attempt to drop or rename a non-existent schema.

There is always a 'public' schema in which objects are created by default unless otherwise specified. Objects can be qualified by their schema name by prepending the object name with the schema name and a dot. For example, to refer to the table mytable in the schema myschema, you can use myschema.mytable to qualify it. If you omit a schema qualification, the object is assumed to exist in the public schema - that is, 'mytable' implicitly refers to 'public.mytable'.

## Managing Tables

A table is used to store data and is defined with one or more columns. Each column has a data type and a number of optional constraints associated with it, such as whether a column value is allowed to be null (undefined).

Each table has a scope which specifies how the data within a table is distributed across the network. Currently, the scope of a table can be either:

- Local. The data differs between edge nodes in the network. This data is collected, stored and replicated between instance nodes under an edge node and this data is effectively partitioned by edge node.
- Global. This data is common across the network. This data is created centrally on the catalog servers and replicated across all instance nodes such that every instance node contains a copy of the same global data.
- Central. This data only exists on the catalog servers and is never propagated to any instance node.

The scope of a table is set when a table is created and the scope parameter is a letter which may be one of the following:

- L - local scope
- G - global scope
- C - central scope

**Note that the scope of a table cannot be changed after it has been created.**

The mutability of data varies depending on the scope of a table

- Local. Data in local tables is collected at the edge and treated as immutable fact data that can only be appended to a table - where rows cannot be updated or individually deleted.
- Global and Central. Data in non-local tables is managed centrally and treated as mutable so that rows can be inserted, updated and deleted at will.

The life-cycle of data is also managed differently depending on the scope of the table.

- Local. Local tables can be purged such that data inserted into a local table before a specified receipt date/time may be dropped. Note that this an operation used to reclaim storage from local tables and does not guarantee to drop any specific rows - it only guarantees to retain data received at or after the given purge date/time.
- Global and Central. Non-local tables can be truncated such that all data in a table is guaranteed to be removed.

**Note that local tables cannot be truncated because local tables collect data independently at the edge of the network and therefore the effect of a central truncation operation would depend on the timing of truncation request relative to any data collection processing and also on the network connectivity at the time of the request. This would cause the effect of a truncation operation on a local table to be non-deterministic.**

Tables are managed by connecting to the network and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_table | Creates a new table | table$ - name of the table to be created<br>columns$ - columns specification for the table<br>scope$ - scope of the table (G or L (or C in future)) |
| drop_table | Drops an existing table | table$ - name of table to be dropped<br>cascade$ - indicates if drop cascades to dependent objects. |
| grant_table | Grants a role query access to a table | table$ - name of table to grant access to<br>role$ - name of role to grant query access to |
| purge_table | Purges data from a local table | table$ - name of table to be purged<br>datetime$ - timestamp to purge up to |
| rename_table | Renames an existing table | old$ - current name of the table<br>new$ - new name for the table |
| revoke_table | Revokes query access to a table from a role | table$ - name of table to revoke access from<br>role$ - name of role to revoke query access from |
| truncate_table | Truncates a non-local table | table$ - name of table to be truncated |

In each of the above functions, the name of a table can be optionally qualified with a schema name. If not qualified, the public schema is assumed.

Naturally, if you attempt to create a table using the name of an existing table an exception will be raised; and similarly if you attempt to drop or rename a non-existent table.

The column$ parameter for the create_table function is a comma separated list of column specifications as described below.

An example of creating a table is:

```
SELECT create_table('my_table',
                    'column_1 INTEGER, column_2 TEXT',
                    'G');
```

Which creates a global table called my_table with two columns:called column_1 and column_2 of type integer and text respectively.

A column can also be added, dropped or renamed in an existing table using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_column | Creates a new column | table$ - name of the table to create the column in<br>column$ - column specification |
| drop_column | Drops an existing column | table$ - name of table to drop the column from<br>column$ - name of the column to be dropped<br>cascade$ - indicates if drop cascades to dependent objects. |
| rename_column | Renames an existing column | table$ - name of table to rename column in<br>old$ - current name of the column<br>new$ - new name for the column |

A column specification takes the form:

<name>  <type>  [ [[NOT] NULL] [ CHECK (<check>)] [DEFAULT <default>] ]

where:

<name> is the name of the column
<type> is the datatype of the column
<check> is a predicate for checking the value of the column
<default> is a default value for populating a column

For example:

```
SELECT create_column('my_table','new_column INTEGER NULL CHECK(extra>0)');
```

Creates a new column in my_table called  new_column of type integer with a constraint that requires the new column to contain positive integers.

## Managing Constraints

Constraints can be defined for tables to enforce data integrity. The constraints that can be applied vary by table scope:

- Global and Central. Primary key, unique key and foreign key constraints can be defined for non-local tables such that any attempt to insert, update or delete data which might violate a constraint raises an exception.
- Local. The data being collected at the edge of the network is factual event data and is never rejected or discarded - therefore the constraints described above cannot be applied to a local table. However, a distribution constraint can be applied to local tables which is used to enforce a join condition between local tables involved in a join query. If there is no distribution constraint defined for a local table - it cannot be joined with another local table.

### Primary Key Constraints

A primary key constraint defines a combination of one or more columns that define a unique key for every row. If there is an attempt to insert or update a row which would have the same key as another existing row, the operation raises an exception.

Primary key constraints can be defined for global and central tables - but only one primary key constraint can be defined per table.

Primary key constraints are managed by connecting to the network and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_primary_constraint | Creates a primary key constraint | table$ - name of the table to create constraint on<br>columns$ - list of columns to apply constraint to |
| drop_primary_constraint | Drops a primary key constraint | table$ - name of the table to drop constraint from<br>cascade$- indicates drop should cascade to dependent foreign key constraints |

### Unique Key Constraints

A unique key constraint defines a combination of one or more columns that are required to be unique for every row. If there is an attempt to insert or update a row which would have the same values as another existing row, the operation raises an exception.

Unique key constraints can be defined for global and central tables and there may be multiple unique key constraints per table - but only one unique key constraint can be defined per column combination.

Unique key constraints are managed by connecting to the network and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_unique_constraint | Creates a unique key constraint | table$ - name of the table to create constraint on<br>columns$ - list of columns to apply constraint to |
| drop_unique_constraint | Drops a unique key constraint | table$ - name of the table to drop constraint from<br>columns$ - list of columns constraint is applied to |

### Foreign Key Constraints

A foreign key constraint defines a combination of one or more columns that are required to have a corresponding primary key value in another table. If there is an attempt to insert or update a row which would have a value that does not correspond to a known primary key value, the operation raises an exception.

Foreign key constraints can be defined for global and central tables and there may be multiple foreign key constraints per table - but only one foreign key constraint can be defined per column combination and it must be related to a compatible primary key in another table. The primary key is deemed compatible if it has the same number of columns and castable column types as the foreign key.

Foreign key constraints are managed by connecting to the network and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_foreign_constraint | Creates a foreign key constraint | table$ - name of the table to create constraint on<br>columns$ - list of columns to apply constraint to<br>reference$ - name of table with the associated primary key |
| drop_foreign_constraint | Drops a foreign key constraint | table$ - name of the table to drop constraint from<br>columns$ - list of columns constraint is applied to |

## Distribution Constraints

A distribution constraint specifies the column in a local table that is known to partition the data by edge node. Any query attempting to join local tables requires that those local tables must have compatible distribution constraints.

Distribution constraints can be defined for local tables only and there may be only one distribution constraint per table.

**Note that distribution constraints need only be defined if you have multiple local tables and you intend to join between them in queries. However, distribution constraints are not fully operational in the current release and as such this requirement is not yet enforced.**

A join between a local table and a global table does NOT require a distribution constraint to be defined.

Distribution constraints are managed by connecting to the network and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_distribution_constraint | Creates a distribution constraint | table$ - name of the table to create constraint on<br>column$ - column to apply constraint to |
| drop_distribution_constraint | Drops a distribution constraint | table$ - name of the table to drop constraint from |

## Managing Views

A view defines a virtual table that contains rows specified by a query.

Views are managed by connecting to the network and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_view | Creates a new view | view$ - name of the view to be created<br>query$ - query that specifies the rows in the view |
| drop_view | Drops an existing view | table$ - name of view to be dropped<br>cascade$ - indicates if drop cascades to dependent objects. |
| grant_table | Grants a role query access to a view | table$ - name of view to grant access to<br>role$ - name of role to grant query access to |
| rename_view | Renames an existing view | old$ - current name of the view<br>new$ - new name for the view |
| revoke_table | Revokes query access to a view from a role | table$ - name of view to revoke access from<br>role$ - name of role to revoke query access from |

In each of the above functions, the name of a view can be optionally qualified with a schema name. If not qualified, the public schema is assumed.

Naturally, if you attempt to create a view using the name of an existing view an exception will be raised; and similarly if you attempt to drop or rename a non-existent view.

## Managing Functions

A function defines processing logic that can be applied to query results.

A function can accept zero, one or more arguments and return a result and functions can be overloaded such that the same function can be defined for different argument sets.

Functions are managed by connecting to the network and using the following command functions:

| Function | Description | Arguments |
|---|---|---|
| create_function | Creates a new function | function$ - name of the function to be created<br>arguments$ - list of arguments accepted by the function<br>return$ - return type of the function<br>language$ - language the function body is written in<br>body$ - the body of the function |
| drop_function | Drops an existing function | function$ - name of function to be dropped<br>arguments$ - list of arguments accepted by the function<br>cascade$ - indicates if drop cascades to dependent objects. |
| rename_function | Renames an existing function | old$ - current name of the view<br>arguments$ - list of arguments accepted by the function<br>new$ - new name for the view |

In each of the above functions, the name of a function can be optionally qualified with a schema name. If not qualified, the public schema is assumed.

When creating, renaming or dropping a function, the argument list is a comma separated list of argument name and argument type pairs, for example:

```
'left integer, right integer'
```

The return type of a function should be a standard database type such as integer, bigint, text etc.

The body of a function can be written in SQL or pl/pgSQL and the language parameter should specify one of these.

A SQL function can execute a SQL statement to return the function result. For example, to return the sum of two number arguments called left and right from a SQL function, the body would be:

```
select left+right;
```

To create this SQL function with the name 'exampleA' using create_function:

```
SELECT create_function('exampleA',
                       'left integer, right integer',
                       'integer',
                       'SQL',
                       'select left+right;');
```

A pl/pgSQL function is procedural and can contain conditional, loop and control structures - see PostgreSQL documentation for details about the pl/pgSQL language. Each pl/pgSQL body must start

with a BEGIN and finish with an END and include a RETURN statement which returns the function result. For example, the body for a procedural version of the function above would be:

```
begin
  return left+right;
end
```

To create this pl/pgSQL function with the name 'exampleB' using create_function:

```
SELECT create_function('exampleB',
                       'left integer, right integer',
                       'integer',
                       'pl/pgSQL',
                       'begin return left+right; end');
```

**Note that functions are not allowed to contain statements that would change the state of the database and an exception is raised if such statements are included.**

## Managing Sequences

A sequence is a sequential number generator typically used to populate a primary key column or as a counting sequence. Once a sequence is defined it can be referenced by a column definition wishing to use it.

All sequences start at 1 and increment by 1 up to a maximum value of 2^63 - 1.

Sequences are managed by connecting to the network and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_sequence | Creates a new sequence | sequence$ - name of the sequence to be created |
| drop_sequence | Drops an existing sequence | sequence$ - name of sequence to be dropped<br>cascade$ - indicates if drop cascades to dependent objects. |
| rename_sequence | Renames an existing sequence | old$ - current name of the sequence<br>new$ - new name for the sequence |

In each of the above functions, the name of a sequence can be optionally qualified with a schema name. If not qualified, the public schema is assumed.

Naturally, if you attempt to create a sequence using the name of an existing sequence an exception will be raised; and similarly if you attempt to drop or rename a non-existent sequence.

## Managing Interfaces

Interfaces are used to parse messages received from external collection agents and append rows to local tables; where each agent collects messages from a source such as a device, socket or file stream.

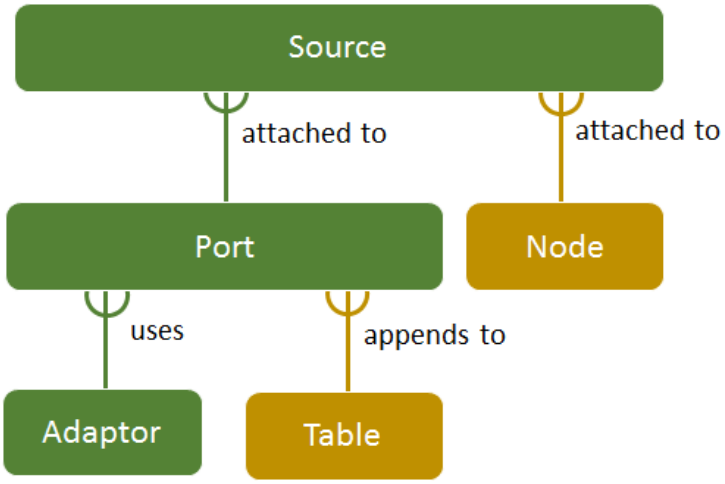
The following framework is used to provision and configure those interfaces:

- Adaptors provide generic message parsing capabilities and can be configured to parse specific message formats. For example, there is a JSON adaptor that can be configured to parse JSON messages of a specific format.
- A port connects an adaptor to a local table using a specific adaptor configuration. Messages received on a port are parsed using the adaptor configuration and appended to the connected table.
- A source definition identifies a source of data such as a device, file stream or socket.
- A source can be attached to one port and one edge node and multiple sources can be attached to the same port.

This framework allows:

- Messages from a single source to be collected by multiple instance nodes under the same edge node.
- Messages from different sources may be appended to the same local table.

The entity relationship diagram below describes the relationships between sources, ports and adaptors:



## Message State

The same message may be sent to multiple instance nodes under the same edge node to provide message replication and high-availability; and sibling instance nodes under an edge node perform multi-master synchronization to achieve a common data set on all instance nodes under an edge.

To enable instance nodes to synchronize their data, messages from each source must contain a strictly monotonically increasing message state to uniquely identify every message. This state may be a timestamp or a sequence number and must be convertible to a BIGINT type. The sequence of message states from a source does not need to be contiguous - just strictly monotonic.

If the message state is discovered to be not strictly monotonic, then an exception will be thrown during message parsing and the message will be rejected.

## Adaptors

An adaptor is a general purpose parser that converts messages to relational rows. To use an adaptor, it is associated with a local via a port definition that specifies a configuration for how the adaptor behaves.

There are two standard adaptors provided:

- json – parses and converts json messages.
- sv - parses and converts separated value messages (such as CSV).

These adaptors are described in more detail below.

### JSON Adaptor

The JSON Adaptor takes multiple messages structured as an array of objects and converts each object to a relational row. The adaptor configuration specifies:

- A list of table columns to populate.
- A list of object keys to map to the columns above in corresponding positional order.
- The object key used for message state.
- The type of message state used (time or number)
- The format mask to be applied to message state when converting to BIGINT.

For example, consider the following two JSON messages:

```
[{"timestamp":"20161201:190501.123","temperature":45,"pressure":15},
 {"timestamp":"20161201:191002.291","temperature":44,"pressure":16}]
```

To configure the JSON adaptor to convert these to rows with a "temp" and "press" columns would require the following configuration:

| Parameter | Setting |
|---|---|
| table columns | temp,press |
| JSON keys | temperature,pressure |
| state key | timestamp |
| state type | T (timestamp) |
| state mask | YYYYMMDD:HH24MISS.MS |

### SV Adaptor

The SV Adaptor takes multiple messages structured as a records with delimited fields and converts each record to a relational row. The adaptor configuration specifies:

- List of  table columns to populate
- List of field positions that correspond to the columns above
- Field delimiter character
- Record delimiter character
- Position of the message state field
- Type of message state used (time or number)
- The format mask to be applied to message state when converting to BIGINT.

For example, consider the following two records:

20161201:190501.123,45,15;

20161201:190501.123,44,16;

To configure the SV adaptor to convert these records to rows with a "temp" and "press" columns would require the following configuration:

| Parameter | Setting |
|---|---|
| table columns | temp,press |
| field positions | 2,3 |
| field delimiter | , |
| record delimiter | ; |
| state position | 1 |
| state type | T (timestamp) |
| state mask | YYYYMMDD:HH24MISS.MS |

For non-printable delimiter characters, use E'\xNN' notation to specify a hexadecimal ASCII code for the character, eg. use E'\x0D' to specify ASCII code 13 (decimal). The following escape sequences can also be used to indicate special characters:

- \b - backspace
- \f - form feed
- \n - new line
- \r - carriage return
- \t - tab

### Format Masks

The JSON and SV Adaptors allow an optional format mask to be used for converting message content into message state. The format mask used depends on the type of the content (number or timestamp) and the following mask patterns are available:

| Type | Pattern | Meaning |
|---|---|---|
| Timestamp | H | hour of day (01-12) |
| | HH12 | hour of day (01-12) |
| | HH24 | hour of day (00-23) |
| | MI | minute (00-59) |
| | SS | second (00-59) |
| | MS | millisecond (000-999) |
| | US | microsecond (000000-999999) |
| | SSSS | seconds past midnight (0-86399) |
| | AM, am, PM or pm | meridiem indicator (without periods) |
| | A.M., a.m., P.M. or p.m. | meridiem indicator (with periods) |
| | Y,YYY | year (4 or more digits) with comma |
| | YYYY | year (4 or more digits) |
| | YYY | last 3 digits of year |
| | YY | last 2 digits of year |
| | Y | last digit of year |
| | IYYY | ISO 8601 week-numbering year (4 or more digits) |
| | IYY | last 3 digits of ISO 8601 week-numbering year |
| | IY | last 2 digits of ISO 8601 week-numbering year |
| | I | last digit of ISO 8601 week-numbering year |
| | MONTH | full upper case month name (blank-padded to 9 chars) |

| | | |
|---|---|---|
| | Month | full capitalized month name (blank-padded to 9 chars) |
| | month | full lower case month name (blank-padded to 9 chars) |
| | MON | abbreviated upper case month name (3 chars in English, localized lengths vary) |
| | Mon | abbreviated capitalized month name (3 chars in English, localized lengths vary) |
| | mon | abbreviated lower case month name (3 chars in English, localized lengths vary) |
| | MM | month number (01-12) |
| | DAY | full upper case day name (blank-padded to 9 chars) |
| | Day | full capitalized day name (blank-padded to 9 chars) |
| | day | full lower case day name (blank-padded to 9 chars) |
| | DY | abbreviated upper case day name (3 chars in English, localized lengths vary) |
| | Dy | abbreviated capitalized day name (3 chars in English, localized lengths vary) |
| | dy | abbreviated lower case day name (3 chars in English, localized lengths vary) |
| | DDD | day of year (001-366) |
| | IDDD | day of ISO 8601 week-numbering year (001-371; day 1 of the year is Monday of the first ISO week) |
| | DD | day of month (01-31) |
| | D | day of the week, Sunday (1) to Saturday insert_rowcreate(7) |
| | ID | ISO 8601 day of the week, Monday (1) to Sunday (7) |

| | | |
|---|---|---|
| | W | week of month (1-5) (the first week starts on the first day of the month) |
| | WW | week number of year (1-53) (the first week starts on the first day of the year) |
| | IW | week number of ISO 8601 week-numbering year (01-53; the first Thursday of the year is in week 1) |
| | CC | century (2 digits) (the twenty-first century starts on 2001-01-01) |
| | J | Julian Day (days since November 24, 4714 BC at midnight) |
| | TZ | uppercase time-zone name |
| | tz | lowercase time-zone name |
| Number | 9 | value with the specified number of digits |
| | D | expected position of decimal point |
| | S | expected position of sign |
| | V | shift value by number of digits |

To convert identifiers with a mixture of digits and letters and symbols, by extracting only the digits, use a number type with a format mask containing a series of "9" to the length of the maximum expected identifier. For example, to convert 'A12/34' to '1234' use a format mask of '999999'.

## Additional Adaptors

Adaptors other than the standard JSON and SV adaptors can be defined to extend the library of available adaptors.

Adaptors are created and dropped by connecting to the network database and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_adaptor | Creates a new adaptor | adaptor$ - name of the adaptor to be created<br>description$ - description of the adaptor (optional)<br>check$ - name of function used to validate an adaptor configuration<br>open$ - name of the function used to open an adaptor<br>close$ - name of the function used to close an adaptor<br>append$ - name of the function used to append messages<br>arguments$ - list of arguments required for an adaptor configuration |
| drop_adaptor | Drops an existing adaptor | adaptor$ - name of adaptor to be dropped |

As can be seen from the create_adaptor() function, each adaptor is required to provide the following four functions:

| Function | Description |
|---|---|
| Check | Checks that the configuration parameters provided by a user when deploying the adaptor are valid. |
| Open | Opens an adaptor session to prepare for parsing one or more messages. This may allocate resources required for parsing purposes. |
| Close | Closes an adaptor session to enable any allocated adaptor resources to be released |
| Append | Receives messages, parses them and appends their rows to a temporary data table |

The check function is called when the adaptor is being configured and deployed. It validates that the configuration specified is sufficient and valid for the adaptor. Each adaptor has its own specific set of configuration parameters.

The open, append and close functions are called when the adaptor is being used to process messages during a message collection session. The open function is called when the session starts and allocates any resources required by the adaptor. The close function is called at the end of the session and de-allocates the resources no longer required by the adaptor. In between, the open and close function calls the append function will be likely called multiple times to process a stream of messages. Each append function call is a passed one or more messages and these are parsed, translated into rows and appended to a temporary data table.

All of the adaptor functions receive the configuration parameter settings specified when an adaptor was deployed.

**Note that the functions are created separately from the adaptor. This means that an adaptor being used with a port can be dropped and the port will continue to operate as the configuration remains with the port and the adaptor functions will continue to exist. Dropping an adaptor simply removes the adaptor definition so that it cannot be used with future port definitions.**

The parameters required by the adaptor functions varies according to the needs of each adaptor, but the following parameters are always provided prior to any adaptor specific parameters:

| Function | Parameter | Descriptionstate$rea |
|---|---|---|
| Check | table$ | Name of the table to be appended to |
| Open | source$ | Identity of the source of the messages |
| | state$ | The starting state of the table to be appended to |
| Close | source$ | Identity of the source of the messages |
| | state$ | The closing state of the table that has been appended to |
| Append | message$ | The message to be parsed |
| | source$ | Identity of the source of the messages |
| | state$ | The state of the table from the previous append |

The adaptor append function is required to implement the following functionality:

- Parse the messages provided in the function call.
- Insert each message as a row into a temporary table called $data.

The temporary table $data is created and dropped by the interface framework and the adaptor function does not need to create or drop it.

The $data table will contain all of the columns of the target local table and the following additional columns:

- The identity of the source - $source (BIGINT)
- The message state - $state (BIGINT)

It is the responsibility of the adaptor append function to populate all of the columns of the $data table. The identity of the source is passed into the append function call and can be used directly, but the $state column value must be parsed out of the message.

The following functions are used by the standard adaptors and can be used as examples for implementing new adaptor functions:

| Adaptor | Function | Description |
|---|---|---|
| JSON | json_adaptor_check | Checks adaptor configuration |
| | json_adaptor_open | Opens adaptor and creates temporary table for JSON type mapping |
| | json_adaptor_close | Closes adaptor and drops temporary table for JSON type mapping |
| | json_adaptor_append | Parses and appends messages to $data table |
| SV | sv_adaptor_check | Checks adaptor configuration |
| | sv_adaptor_open | Opens adaptor (does not allocate any resources) |
| | sv_adaptor_close | Closes adaptor (does not deallocate any resources) |
| | sv_adaptor_append | Parses and appends messages to $data table |

**Note that currently it is not possible to create new adaptor functions - but this feature will be provided in a future release.**

The star$adaptors view provides a list of installed adaptors:

| Column | Description |
|---|---|
| name | Name of the adaptor |
| description | Description of the adaptor |
| check_function | Name of the adaptor check function |
| open_function | Name of the adaptor open function |
| close_function | Name of the adaptor close function |
| append_function | Name of the append adaptor function |
| arguments | List of adaptor configuration arguments |

## Ports

A port provides a channel for appending messages of a specific format to a local table. A port associates an adaptor with a local table via a specific adaptor configuration.

Ports are created and dropped by connecting to the network database and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_port | Creates a new port | name$ - name of the port to be created<br>description$ - description of the port (optional)<br>table$ - name of local table to receive messages<br>adaptor$ - name of the adaptor used for parsing<br>arguments$ -arguments used for the adaptor configuration |
| drop_port | Drops an existing port | port$ - name of port to be dropped |
| test_port | Tests a port configuration | port$ - name of the port to be tested<br>messages$ - message text to be tested |

Once a port is defined, one or more sources can be attached to it to provide a stream of messages.

The test_port() function can be used to test a port configuration against sample message text to verify that the configuration works for the message supplied. If the configuration is invalid for the sample message then an exception is thrown; otherwise the function returns the content of the target table obtained by executing the port configuration. For example:

```
select * from test_port('port','1,1;');
        result
------------------------
 (value,$source,$state)
 (1,0,1)
(2 rows)
```

The first row returned by test_port() is always the list of columns in the table and the $source column is always shown as zero. Note that none of the message data supplied during a test_port function call ever reaches the target table - the function simply verifies the configuration for the sample messages.

The star$ports view provides a list of ports:

| Column | Description |
|---|---|
| name | Name of the port |
| description | Description of the port |
| append_table | Name of the local table that receives rows from the port |
| adaptor | Name of the adaptor used for parsing |
| arguments | List of adaptor argument settings |

## Sources

A source identifies a source of messages such as a device, socket or file stream.

A source must be attached to a specific port at a specific edge node to collect and direct data from the source to a local table at an edge point in the network. Note that multiple instance nodes under an edge node can collect data from the same source.

Sources are managed by connecting to the network database and using the following functions:

| Function | Description | Arguments |
|---|---|---|
| create_source | Creates a new source | tag$ - resource tag for the source<br>description$ - description of the source (optional) |
| attach_source | Attaches a source to a port and/or edge node | tag$ - resource tag of the source<br>port$ - name of the port to attach to (optional)<br>node$ - name of the edge node to attach to (optional) |
| drop_source | Drops an existing source | source$ - name of source to be dropped |

A source is defined with a resource tag which is the identity which uniquely identifies the device, socket or file stream.

The star$sources view provides a list of sources:

| Column | Description |
|---|---|
| tag | Resource identity tag of the source |
| description | Description of the source |
| port | Name of the attached port |
| node | Name of the attached edge node |

## Agents

Data is collected from a source via an agent process which connects to one of the available instance node databases and sends the messages for parsing via an append_source() function call.

Agents are coded and deployed as required and should operate as described below.

The agent process should obtain a connection to its target instance node as follows:

- Connect to the administration database as star$source user on any server within the network.
- Call network_connection(<network>) to get a connection string for the network.
- Close the connection to the administration database.
- Open the network connection using the connection string returned from network_connection().
- Call instance_connection(<tag>) to get a connection string for an instance node.
- Open the assigned instance connection string.

Note that in the above, each connection prior to the final instance connection simply provides a directory service as follows:

- The administration database provides a directory for the available network databases.
- The network database provides a directory for the available instance databases.

The call to instance_connection allocates an available instance node to the source and locks the source tag on the instance node to prevent any other agent processes connecting to the same instance for the same source. Note that the network connection has to be kept open for the duration of the agent process to maintain the lock on the allocated instance node.

Once the connection is made, the agent process should:

- Call open_source(<tag>) to start a source session
- Repeatedly call append_source(<messages>) to append messages received

When an agent process gracefully terminates, it should

- Call close_source()
- Close the instance database connection.
- Close the network database connection.

The following table describes each of the functions used by an agent:

| Function | Description | Arguments |
|---|---|---|
| network_connection | Returns a connection string for a network database | network$ - name of the network to connect to |
| instance_connection | Returns a connection string for an instance database | tag$ - tag name of the source to be locked |
| open_source | Starts a source session | tag$ - name of source to be opened |

| | | |
|---|---|---|
| append_source | Appends one or messages | messages$ - text string containing one or more messages |
| close_source | Closes the source session | None |

Normally, the messages supplied to append_source() will be in the format defined by the port configuration - but if any of the messages fails to conform to the port configuration the call to append_source() will raise an exception.

An exception is also raised by append_source() if the message state is deemed out of order or repeated by one or more messages in the call.

The call to append_source() will either process and commit all of the messages in the call or will reject all messages in the call if an exception is raised.

if you need to achieve message processing rates of 10,000 messages/second or above, it is recommended that calls to append_source() be no more frequent than every 1 second.

# Standard Agents

## Input Agent

A standard agent is provided which receives records on standard input and streams that data via a configured source.

This agent runs in Java and requires 3 parameters:

- Host name of the database server to connect to
- Name of the network to connect to.
- Name of the source tag to be streamed.
- Optional record separator character

This input agent can executed using:

es_agent [--streamer] <host> <network> <tag> [ <record separator> ]"

Where <host> is the database server host name or address; <network> is the name of the network to connect to; <tag> is the tag of source to be streamed; and <record separator> is an optional record separator (otherwise new line is assumed). See File Transactions below for details about the optional --streamer switch.

This agent will wait on standard input indefinitely and will execute until it is aborted via a process or command termination.

The agent will only append records it has received when either of the following occurs since the last append:

- More than 5000 records have been received.
- It has waited longer than 5 seconds for the next record.

### File Streamer

A file streamer is also provided for use with the input agent above to process records from a file stream. A file stream is a directory where new files are regularly deposited for processing. The file streamer requires the following:

- Files deposited for processing are complete and closed files
- All files are of the same format and use the same field delimiter
- The file names are alphabetically ordered in the sequence they are to be processed.

Hence a source directory for a file stream may contain files named data001.dat, data002.dat etc.

The file streamer will process files in name order and stream the records from each file to standard output. Each record streamed is prepended with the modification time of the file and the record number as the first field of each record output. This first field can be used a message state field and should be configured in the adaptor as a number field with a format mask of 19 digits (i.e. all 9s). The modification time of a file is calculated relative to 1 Jan 2000 and resolved down to seconds and is presented in the first 10 digits while the record number is presented in the last 9 digits.

For example, the first two data records from a file may be prepended with 0535635255000000001

and 0535635255000000002 respectively.

As each file is processed, it is moved to a specified fulfilment directory to avoid a file being processed more than once. Note that if you edit a file that has been processed, its modification time will change and if the file is moved back to the source directory, it will be treated and processed as new data.

This streamer runs in Java and requires 3 parameters:

- Glob expression for finding files
- Field delimiter used in the source files.
- Fulfilment directory for receiving processed files.
- Optional number for passes

In the glob expression for finding files, glob characters (? and *) must only appear within a file name and not in the directory path. It is also possible to run the FileStream with an explicit file path (without any glob characters) to process a single file.

This file streamer can executed using:

```
es_streamer {--streamer] <glob> <delimiter> <fulfilment> [ <passes> ]
```

Where <glob> is a glob expression for the source files; <delimiter> is the field delimiter used in the file records; <fulfilment> is the fulfilment directory for receiving processed files; and <passes> can be used to limit the number of files processed. See File Transactions below for details about the optional --streamer switch.

This agent will wait on files to appear in the source directory indefinitely and will execute until it is aborted via a process or command termination.

The file streamer can be used as input the standard input agent by piping its output to it.

### File Transactions

Normally, when es_streamer output is piped to the es_agent, they communicate on a record by record basis, but this means that the agent commits occur asynchronously with file boundaries. This can make recovery from failed or aborted file streaming difficult.

Both the file streamer and agent can accept a --streamer switch which links commits with the end of each file read. The --streamer option should only be used when piping es_streamer output to the es_agent and when the streamer and agent both use the option.

## Managing Data

This section describes the management of data in global and central tables which is created and managed centrally in the catalog.

Data in non-local tables is considered mutable and rows can be inserted into, updated in and deleted from those tables. Any changes made to data in global tables are also propagated out to the instance nodes across the network.

The following functions can be used to manage data in non-local tables:

| Function | Description | Arguments |
|---|---|---|
| insert_rows | Inserts rows into a table | table$ - name of the table to insert into<br>columns$ - list of columns to insert into<br>rows$ - row data to be inserted |
| delete_rows | Deletes rows from a table | table$ - name of table to delete rows from<br>where$ - where clause to identify rows to be deleted |
| update_rows | Updates rows in a table | table$ - name of table to update rows in<br>columns$ - list of columns to be updated<br>values$ - list of expressions to update columns with<br>where$ - where clause to identify rows to be updated |
| copy_rows | Copies rows into a table | table$ - name of table to copy rows into |

The insert_rows() function requires a rows$ parameter that specifies the rows to be inserted. This parameter accepts a comma separated list of one or more rows where each row is expressed as:

(<value> [, <value>]...)

Where each value corresponds in positional order with the list of columns in the columns$ parameter.

Any text or character values should be enclosed in single quotes - which need to be double escaped because the parameter itself is a string. For example,

```
SELECT insert_rows('my_table',
                   'id,name,salary',
                   '(1,''billy'',50000),(2,''bobby'',45000)');
```

The delete_rows() and update_rows() functions require a where$ parameter to identify the rows to be deleted or updated. For example,

```
SELECT delete_rows('my_table',
                   'id=2');
```

The update_rows() function requires a values$ parameter to list the expressions used to update the columns with. For example,

```
SELECT update_rows('my_table',
                   'salary',
                   'salary*1.02',
                    'id=2');
```

The copy_rows() function is used to copy rows into a table from a temporary table with the same name. This may be used in conjunction with a loader to copy a large number of rows from a file into a temporary table and then into the target table.

For example, joloader can be used to copy a file into a global table using the command:

```
joloader <network> <file> <table>
     -U<user> -E~"select copy_rows('<table>');"
```

Where <network> is the name of the network; <file> is the name of the file to be loaded; <table> is the name of the global table; and <user> is the name of a user with star$modify role granted.

See the Edge Intelligence Loader user guide for detailed information about the features and capabilities of the file loader.

## Collecting Data

Data for local tables is collected on instance nodes via agents, adaptors and ports and these are described in the Managing Interfaces section.

Multiple instances under the same edge node collect the same data to provide high availability. This data has to be synchronized and replicated between instance nodes to mitigate against possible communication and hardware failures at instance nodes and a regular job runs on each instance node to synchronize its data with its siblings.

## Migrating Data

If you need to migrate data from files into a local table, you can do so on each edge server using the following loader command:

```
joloader <network>$<instance> <file> <table> -Ujustone
      c'<columns>,"$source","$state"' J'0,%I%'
```

Where <network> is the name of the network; <instance> is the node identity of the instance node to load into; <file> is the name of the file to be loaded; <table> is the name of the local table and <columns> is the list of table columns to load into.

This command uses the J switch to populate the source with a zero value and the state with a unique identifier.

Note you will need root access to the edge server to do this and that you should only migrate data into one instance node under each edge node. If you attempt to migrate the same data into two or more instance nodes under the same edge node you will duplicate data because the message state numbers assigned by the loader are unique.

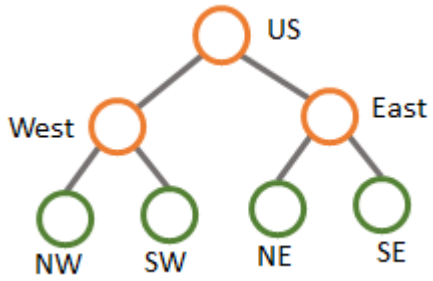
## Performing Queries

Users are able to perform queries by establishing a connection to an available catalog server - direct connections to edge servers are not permitted.

Queries are performed using the standard SQL query language to select from one or more tables and or views that have been defined as described above.

Each query can be submitted in the context of an area or edge node within the network. For example, in the network below, a query can:

- Include the entire network by issuing a query in the context of node 'US'.
- Cover the 'NE' and 'SE' edges by issuing the query in the context of node 'East'
- Include 'SE' only by issuing the query in the context of node 'SE'




The node context for a query can be specified using the USE NODE clause at the end of a query statement. This clause requires a known area or edge node name enclosed in single quote marks. For example:

```
SELECT count(*),column_1
FROM my_table
GROUP BY column_1
ORDER BY 1 DESC
LIMIT 10
USE NODE 'US';
```

## Join Queries

A query can perform inner joins between tables of any scope; except that inner joins between two local tables requires that there is a compatible distribution constraint on both local tables. See the section on Distribution Constraints for further details.

A query can perform outer joins between tables of any scope, except that a local table is not allowed to return null rows when joined with a global table. For example, for tables local_table and global_table with local and global scope respectively, the following query would raise an exception:

```
SELECT count(*)
FROM local_table l
```

```
RIGHT JOIN global_table g ON (g.column_1=l.column_1);
```

This query would raise an exception because the right join can potentially return null rows from the local table for non-matching rows from the global table.

## Query Goals

Each edge node node may be linked to multiple instance nodes below it and during a query, only one of those instance nodes will be chosen to service the query at that edge point. The choice of instance node can be influenced by setting one of the following query goals before issuing the query:

- Response – provide the fastest response time
- Available - randomly choose any one of the currently available instance nodes
- Complete - choose the instance node with the most rows in a particular table
- Recent - choose the instance node with the most recently received rows in a particular table
- Balance - choose the instance node which is least busy

The default query goal is to provide the fastest response time.

The goal for a query can be specified using the USE GOAL clause at the end of a query statement. This clause requires one of the above goals to be specified and is case insensitive. For example:

```
SELECT count(*),column_1
FROM my_table
GROUP BY column_1
ORDER BY 1 DESC
LIMIT 10
USE NODE 'US'
USE GOAL recent;
```

## Query Syntax

Queries are performed using the standard SQL query language, except for the following restrictions:

- Use of recursive queries are not currently supported.
- Window queries are not currently supported.
- Use of FETCH is not supported
- Use of UPDATE is not supported

Hence a valid SQL query syntax is described by the following:

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
  [ expression [ [ AS ] output_name ] [, ...] ]
  [ FROM table_name | (query)  [ [ AS ] alias ]  [, ...]
  [ INNER | LEFT | RIGHT | FULL [OUTER] ] JOIN table_name | (query)  [ [ AS ] alias ] ON condition [, ...]
  [ WHERE condition ]
  [ GROUP BY position | element [, ...] ]
  [ HAVING condition [, ...] ]
  [ ORDER BY position | expression [ ASC | DESC ] [, ...] ]
  [ LIMIT { count } ]
  [ USE NODE 'node' ]
  [ USE GOAL  goal ]
```

## Managing Jobs

Jobs are processes that execute regularly on nodes in a network to perform background tasks.

An example of a job is the process that executes regularly on each instance node to perform data synchronization activities.

The star$jobs view provides a list of jobs and their status on the server you are connected to:

| Column | Description |
|---|---|
| name | Name of the job |
| description | Description of what the job does |
| command | The command executed by the job |
| period | Period of execution |
| enabled | Indicates if the job is currently enabled |
| scheduled | Time at which job is scheduled for its next run |
| failed | Indicates if the last run failed |
| executed | Date and time last run completed |
| duration | Duration of last run |
| error | Error message associated with a failed run |

There is also a star$jobs function that provides a list of jobs and their status on remote instance nodes which can used as follows from a network connection on a catalog server:

```
SELECT * FROM star$jobs(<node>);
```

Where <node> is a node in the network at or above an edge node. This function returns the following information:

| Column | Description |
|---|---|
| edge_id | Identity of the parent edge node |
| edge_name | Name of the parent edge node |
| connection | Connection string for the instance node |
| state | Connection state (0 indicates OK) |
| error | Error message (from either a connection or job failure) |
| instance | Name of the instance node on which the job executes |
| name | Name of the job |
| enabled | Indicates if the job is currently enabled |
| scheduled | Time at which job is scheduled for its next run |
| failed | Indicates if the last run failed |
| executed | Date and time last run completed |
| duration | Duration of last run |

The following table describes the standard jobs that run on every instance node:

| Job | Period | Description |
|---|---|---|
| Synchronize | 1 minute | Synchronizes instance node with the catalog and sibling instance nodes |

## Managing Resources

Views and functions are provided for querying resource usage by instance nodes across the network in the following areas:

- Configuration settings
- Node connection status
- Job status
- Log file messages

The following functions on each catalog node provide information about one or more remote instance nodes in the network.

| Function | Description | Arguments | Returns |
|---|---|---|---|
| star$configuration | Returns configuration settings for each instance node | node$ - node to query from | edge_id - edge node identity<br>edge_name - name of edge node<br>connection - instance node connection string<br>state - connection state<br>error - connection error<br>identity - instance node identity<br>key - configuration key<br>value - configuration setting |
| star$connections | Returns connection state for each instance node | node$ - node to query from | edge_id - edge node identity<br>edge_name - name of edge node<br>connection - instance node connection string<br>state - connection state<br>message - error message<br>detail - error detail |
| star$jobs | Returns job states for each instance node | node$ - node to query from | edge_id - edge node identity<br>edge_name - name of edge node<br>connection - instance node connection string<br>state - connection state<br>error - error message<br>instance - name of instance node<br>name - name of the job,<br>enabled - indicates if job is enabled<br>scheduled - when scheduled to run next<br>failed - indicates if last run failed<br>executed - date and time of last execution<br>duration - duration of last execution |
| star$logs | Returns recent log messages for each instance node | node$ - node to query from<br>size$ - maximum number of characters to return from the log | edge_id - edge node identity<br>edge_name - name of edge node<br>connection - instance node connection string<br>state - connection state<br>information - most recent log messages |

Note the following for the functions above:

- One or more rows will be returned for each instance node underneath the query node
- For each instance node currently unavailable, it will return a single row with a non zero connection state and an error message detailing the reason for its unavailability.

The following are examples of function usage:

```
mynw# SELECT edge_name,connection,state,message
FROM star$connections('Denmark');

 edge_name  |          connection             | state | message
------------+---------------------------------+-------+---------
 Denmark    | host=localhost dbname=mynw$31 | 0     |
 Denmark    | host=localhost dbname=mynw$59 | 0     |
(2 rows)

mynw# SELECT edge_name,identity,key,value
FROM star$configuration('Denmark');
 edge_name | identity  |    key    |   value
-----------+-----------+-----------+-----------
 Denmark   |        31 | $trace    | false
 Denmark   |        31 | $catalogs | localhost
 Denmark   |        31 | $job      | t
 Denmark   |        59 | $trace    | false
 Denmark   |        59 | $catalogs | localhost
 Denmark   |        59 | $job      | t
(6 rows)
```

**Functions will also be provided for returning CPU, memory and disk utilisation and process and OS information from instance nodes in a future release.**

# Security

## Server Security

Database connections between servers are authenticated via SSL certificates and only connection requests with a valid certificate are accepted. Each server can operate as a client or server with respect to any other server in the same network - therefore every server has both a client and server side SSL certificate the installation and provisioning process automatically creates the certificate pairs required between servers.

Instance servers operate in potentially hostile environments outside of data centre and are locked down as follows to prevent accidental or malicious exposure of data:

- Only the root user is permitted SSH access and this user is authenticated using a strong and secure password. This SSH access is only used for occasional software installation and upgrade purposes.
- An instance server only permits database connections from other servers using a defined system user with the required client SSL certificate.
- An instance server can accept also connection requests via the "star$source" user from an agent processes for the purposes of receiving input messages for local data. The star$source user is restricted to a limited set of functions used solely for the collection of new messages.

Catalog servers should operate in a secure environment and should enforce appropriate authentication methods on users wishing to connect to them. The following standard authentication methods are available:

- Password.
- GSSAPI.
- SSPI.
- Kerberos.
- LDAP.
- RADIUS.
- PAM.

## Object Security

Access to nodes and database objects is managed via a conventional role based framework whereby:

- Roles are explicitly granted or revoked from users
- Access for database objects is explicitly granted to roles (and implicitly to users who have that role granted).

Users and roles can be defined and granted as required for the purposes of controlling query access to the following objects.

- Schemas
- Tables
- Views
- Columns
- Nodes

The following functions are used to manage roles:

| Function | Description | Arguments |
|---|---|---|
| create_role | Creates a new role | role$ - name of the role to be created |
| drop_role | Drops an existing role | user$ - name of role to be dropped |
| grant_user | Grants a user access to a role | user$ - name of user to be granted a role<br>role$ - name of role to be granted |
| revoke_user | Revokes access to a role from a user | user$ - name of user to revoke role from<br>role$ - name of role to be revoked |

**Note that the above functions are only available in the star$administration database.**

Roles are granted to or revoked from database objects using the following functions:

| Function | Description | Arguments |
|---|---|---|
| grant_column | Grants query access to a column | table$ - table containing the columns<br>columns$ - list of columns grant access to<br>role$ - role to be granted query access |
| grant_node | Grants query access to a node | node$ - name of node to grant access to<br>role$ - name of role to be granted access |
| grant_schema | Grants query access to a schema | schema$ - schema to grant access to<br>role$ - role to be granted query access |
| grant_table | Grants query access to a table | table$ - table to grant access to<br>role$ - role to be granted query access |
| revoke_column | Revokes query access to a column | table$ - table containing the columns<br>columns$ - list of columns to revoke access from<br>role$ - role to be revoked query access |
| revoke_node | Revokes query access to a node | node$ - name of node to revoke access from<br>role$ - name of role to be revoked access |
| revoke_schema | Revokes query access to a schema | schema$ - schema to revoke access from<br>role$ - role to be revoked query access |

| | | |
|---|---|---|
| revoke_table | Revokes query access to a table | table$ - table to revoke access from<br>role$ - role to be revoked query access |

**Note that by default roles are denied query access to schemas, tables and columns until explicitly granted; while a node is available for query to all roles until access is explicitly limited to one or more roles.**

The following standard roles are predefined and may be granted to users .

| Role | Description |
|---|---|
| star$administrate | Administrate networks, users and roles |
| star$grant | Grant or revoke roles to and from users and objects |
| star$network | Manage network topology |
| star$define | Define database objects (DDL) |
| star$modify | Modify database objects (DML) |
| star$query | Perform queries |
| star$append | Append data to local tables |
| star$purge | Purge data from local tables |

A standard user must be granted one or more of the above roles to perform the operations associated with those roles. For example, a user would be granted the star$query role to allow them to perform queries.

**In a future release it will be possible to limit a schema to only exist on a subset of nodes in a network.**

## Auditing

An audit trail is maintained on the catalog servers for all metadata changes and queries and this audit trail records:

- Who submitted the request
- When the request was submitted
- The SQL used to effect the request

The audit trail can be queried using the star$audit view which contains the following columns:

| Column | Description |
|---|---|
| username | Name of the user who applied the change |
| datetime | Timestamp when change was applied |
| sql | The function call invoked to effect the change |

An example of an audit trail is given below:

```
 username |      datetime       |                                sql
----------+---------------------+-----------------------------------
 billy    | 2017-03-23 10:15:52 | SELECT _star."_create_node"
(109,'Denmark','','E','','','',NULL,NULL,NULL,NULL,'D')
 billy    | 2017-03-23 10:22:22 | SELECT
_star."_create_table"(9,'public.region','r_regionkey integer,r_name
varchar(25),r_comment varchar(152)','G','jo_tablespace')
 bob      | 2017-03-23 10:22:22 | SELECT r_name,r_comment FROM region
...
```

It is possible to purge the audit to drop entries that occur before a given date using the following function:

| Function | Description | Arguments |
|---|---|---|
| purge_audit | Purge old audit entries | datetime$ - date to retain entries from |

**Note that purge_audit() does not guarantee to drop all entries before the specified date, but rather guarantees to keep entries for or after the specified date.**

Only a user granted the star$administrate role is permitted to query or purge the audit trail.

# Metadata Views

Metadata views are provided in each network database and the star$administration database on a catalog server.

## Network Views

The following metadata views are provided in each network on a catalog server:

| View | Description |
|---|---|
| star$adaptors | Defined adaptors |
| star$audit | Audit trail of metadata changes |
| star$columns | User defined table columns |
| star$configuration | Current configuration settings |
| star$constraints | User defined table constraints |
| star$functions | User defined functions |
| star$jobs | Status of background jobs |
| star$locks | Active locks |
| star$nodes | User defined nodes in the network |
| star$ports | User defined ports |
| star$schemas | User defined schemas |
| star$sequences | User defined sequences |
| star$sources | User defined sources |
| star$tables | User defined tables |
| star$trace | Query tracing |
| star$views | User defined views |

Each of these views is described below:

### star$adaptors

The star$adaptors view provides a list of installed adaptors and contains the following columns:

| Column | Description |
|---|---|
| name | Name of the adaptor |
| description | Description of the adaptor |
| check_function | Name of the adaptor check function |
| open_function | Name of the adaptor open function |
| close_function | Name of the adaptor close function |
| append_function | Name of the append adaptor function |
| arguments | List of adaptor configuration arguments |

### star$audit

The star$audit view provides an audit trail of metadata changes and queries:

| Column | Description |
|---|---|
| username | Name of the user who submitted the request |
| datetime | Timestamp when request was submitted |
| sql | The SQL statement used to effect the request |

### star$columns

The star$columns view lists all user defined table columns:

| Column | Description |
|---|---|
| schema_name | Name of the schema to which the column belongs |
| table_name | Name of table to which column belongs |
| column_name | Name of the column |
| type | Data type of the column |

### star$configuration

The star$configuration view lists current configuration settings:

| Column | Description |
|---|---|
| key | Name of configuration parameter |
| value | Current parameter value |

### star$constraints

The star$constraints view lists all user defined constraints:

| Column | Description |
|---|---|
| schema_name | Name of the schema to which the constraint belongs |
| table_name | Name of table on which constraint is defined |
| type | Constraint type (Distribution, Foreign Key, Primary Key, Unique) |
| columns | List of columns in the constraint |
| reference | Primary key table referenced by a foreign key |

### star$functions

The star$functions view lists all user defined functions:

| Column | Description |
|---|---|
| schema_name | Name of the schema to which the function belongs |
| function_name | Name of the function |
| return | Return data type |
| arguments | List of function argument names and types |

### star$jobs

The star$jobs view provides a list of jobs and their status:

| Column | Description |
|---|---|
| name | Name of the job |
| description | Description of what the job does |
| command | The command executed by the job |
| period | Period of execution |
| enabled | Indicates if the job is currently enabled |
| scheduled | Time at which job is scheduled for its next run |
| failed | Indicates if the last run failed |
| executed | Date and time last run completed |
| duration | Duration of last run |
| error | Error message associated with a failed run |

### star$locks

The star$locks view provides a list of active locks:

| Column | Description |
|---|---|
| lock_id | Internal lock identity |
| resource | Name of the locked resource |
| pid | Identity of the process that is requesting the lock |
| mode | Mode of the lock requested |
| granted | Indicates if the lock has been granted |

### star$nodes

The star$nodes view provides a list of user defined nodes:

| Column | Description |
|---|---|
| id | Unique node identity |
| name | Name of the node |
| parent | Name of the parent node |
| description | Description of the node |
| type | Node type. A, E or I for area, edge or instance node respectively |
| host | The server host address for an instance node |
| address | Address information for the node |
| location | Location information for the node |
| latitude | Latitude of the node |
| longitude | Longitude of the node |
| readable | Indicates if an instance node is currently readable |
| writable | Indicates if an instance node is currently writable |

### star$ports

The star$ports view provides a list of user defined ports:

| Column | Description |
|---|---|
| name | Name of the port |
| description | Description of the port |
| append_table | Name of the local table that receives rows from the port |
| adaptor | Name of the adaptor used for parsing |
| arguments | List of adaptor argument settings |

### star$schemas

The star$schemas view provides a list of user defined schemas:

| Column | Description |
|---|---|
| schema_name | Name of the schema |

### star$sequences

The star$sequences view provides a list of user defined sequences:

| Column | Description |
|---|---|
| schema_name | Name of the schema to which the sequence belongs |
| function_name | Name of the sequence |

### star$sources

The star$sources view provides a list of user defined sources:

| Column | Description |
|---|---|
| tag | Resource identity tag of the source |
| description | Description of the source |
| port | Name of the attached port |
| node | Name of the attached edge node |

### star$tables

The star$sources view provides a list of user defined sources:

| Column | Description |
|---|---|
| schema_name | Name of the schema to which the table belongs |
| table_name | Name of the table |
| scope | Scope of the table (Central, Global, Local) |

### star$trace

The star$trace view provides the results from traced queries:

| Column | Description |
|---|---|
| trace | Unique trace identity |
| stage | Stage of query (0 indicates synchronisation stage) |
| step | Step of query |
| datetime | Timestamp when step started |
| tick | System tick when step started |
| connection | Connection when step was executed |
| rows | Number of rows returned by the step |
| elapsed | Duration of step |
| sql | SQL statement executed |

### star$views

The star$views view provides a list of user defined views:

| Column | Description |
|---|---|
| schema_name | Name of the schema to which the view belongs |
| scope | Scope of the table (Central, Global, Local) |

## Administration Views

The following metadata views are provided in the star$administration database on a catalog server:

| View | Description |
|---|---|
| star$audit | Audit trail of metadata changes |
| star$jobs | Status of background jobs |
| star$networks | User defined networks |
| star$roles | All defined roles |
| star$users | All defined users |

Each of these views is described below:

### star$audit

The star$audit view provides an audit trail of metadata changes in the administration database:

| Column | Description |
|---|---|
| username | Name of the user who applied the change |
| datetime | Timestamp when change was applied |
| sql | The function call invoked to effect the change |
| arguments | List of adaptor configuration arguments |

### star$jobs

The star$jobs view provides a list of jobs and their status:

| Column | Description |
|---|---|
| name | Name of the job |
| description | Description of what the job does |
| command | The command executed by the job |
| period | Period of execution |
| enabled | Indicates if the job is currently enabled |
| scheduled | Time at which job is scheduled for its next run |
| failed | Indicates if the last run failed |
| executed | Date and time last run completed |
| duration | Duration of last run |
| error | Error message associated with a failed run |

### star$networks

The star$networks view lists all user defined networks:

| Column | Description |
|---|---|
| name | Name of the network |
| description | Description of the network |

### star$roles

The star$roles view lists all defined roles:

| Column | Description |
|---|---|
| role_name | Name of the role |

### star$users

The star$users view lists all defined users:

| Column | Description |
|---|---|
| user_name | Name of the user |
| login | Indicates if login is enabled |

## Example of Deploying a Network

The following provides a complete example of deploying a network on vanilla install. The example uses the josql SQL command interface to execute SQL commands. In these examples, capitalization of SQL reserved words has been used for clarity - but is not required.

```
#connect to the star$administration database
josql star\$administration star\$administrator

-- create analyst role
star$administrator=# SELECT create_role('Analyst');

-- create analyst user
star$administrator=# SELECT create_user('myuser');

-- alter analyst user to allow login
star$administrator=# SELECT create_user('myuser',true,'password');

-- grant analyst user query access
star$administrator=# SELECT grant_user('myuser','star$query');

-- create a new network
star$administrator=# SELECT create_network('demo','Demo network');

-- quit from administration database
\q

#connect to the demo network as administrator
josql demo star\$administrator

-- create an instance node
demo=# SELECT SELECT create_node('Instance', 'Instance node', 'I',
'52.59.208.12',null,null,null,null);

-- create an edge node
demo=# SELECT SELECT create_node('Edge', 'Edge node', 'E',
null,null,null,null,null);

-- create an area node
demo=# SELECT SELECT create_node('Area', 'Area node', 'A',
null,null,null,null,null);

-- attach nodes
demo=# SELECT attach_node('Instance', 'Edge');
demo=# SELECT attach_node('Edge', 'Area');
```

```
-- check network topology is as expected
demo=# SELECT rpad(' ',depth)||name AS name, type, description FROM
star$topology('Area') ORDER BY path ASC;

     name      | type | description
---------------+------+-------------
 Area          |  A   | Area node
  Edge         |  E   | Edge node
   Instance    |  I   | Instance node


-- create a global temperature sensor table
demo=# SELECT create_table('sensors','sensor_id BIGINT, latitude
NUMERIC, longitude NUMERIC','G');

-- create a local temperature measurement table
demo=# SELECT create_table('temperatures','datetime TIMESTAMP,
sensor_id BIGINT, temperature FLOAT','L');

-- load the global sensor table from a sensor csv file
demo=# \! joloader demo 'sensors.csv' sensors -Ustar$administrator -
d',' -E~"select copy_rows('sensors');"

-- check the number of rows in the sensor table
demo=# SELECT count(*) FROM sensors;
 count
-------
   1000
(1 row)

-- create a port for parsing messages of the form
{"timestamp":"20170101:120000","sensor":1093,"temp":12.3}
demo=# SELECT create_port('temperature messages','Temperature
measurements','temperatures','json','datetime,sensor_id,temperature'
,'timestamp,sensor,temp','timestamp','T','YYYYMMDD:HH24MISS.MS');

-- create a source for the temperature sensor
demo=# SELECT create_source('T:1985219','Temperature sensor');

-- attach sensor source to port and edge node
demo=# SELECT attach_source('T:1985219','temperature
messages','Edge');

-- grant analysts access to the sensor and temperature tables
star$administrator=# SELECT grant_table('sensors','Analyst');
star$administrator=# SELECT grant_table('temperatures','Analyst');

-- quit from the network database as administration user
```

```
\q

#connect to the demo network as analyst user
josql demo myuser

-- set my role
demo=# SELECT set_role('Analyst');

-- open a query session for the Edge node
demo=# SELECT open_query('Edge');

-- query data from the tables
demo=# SELECT count(*) FROM temperatures t JOIN sensors s ON
s.sensor_id=t.sensor_id WHERE s.latitude=51.4545 AND s.latitude=-
2.5879;

...

-- quit from the network database
\q
```

## Known Issues

The following lists a number of known issues that will be fixed in subsequent releases.

### Distribution constraints

Distribution constraints are not currently enforced.

### Creating adaptor functions

Creating new adaptor functions is not currently possible.

## Forthcoming Features

The following lists a number of features planned for forthcoming releases.

### Bounded Schemas

This feature will allow schemas to be limited to particular nodes in the network. Bounded schemas can be used to provide additional security where a single network is required to handle separate data sets.

### Explicit Purge Criteria

This feature will allow purging to be performed on a nominated column rather than only on the implied transaction timestamp and will provide greater control over data retention.