Relational Queries Using ODBC White Paper

VERSION 2020

jade

Contents

Relational Queries Using ODBC

The JADE ODBC (Open Database Connectivity) standard driver and thin client driver enable you to use SQL statements to access a relational view of your JADE database. You can use the JADE ODBC drivers with tools that access databases using ODBC (for example, MS Query or Crystal Reports). The JADE ODBC drivers are installed as part of the JADE installation process.

The JADE Relational View wizard enables you to create a relational view of your JADE database. This relational view can include JADE classes, properties, and methods. It can also include user schema-defined (soft) attributes and entities.

To use an ODBC driver, configure an ODBC Data Source (DSN) using the Data Source Administrator. The DSN defines the ODBC driver to use and any driver-specific configuration information. Usually, a DSN specifies a specific database to use. For JADE, this configuration information includes the relational view to use and connection information.

Typically, the query tool allows you to select the DSN to be used when using the ODBC interface. The query tool will retrieve catalog information defining tables and columns and allows you to define the data required. The retrieval of the data is done using a **SELECT** statement.

The JADE ODBC thin client establishes a TCP/IP connection with a user-defined ODBC server application running in a JADE node. The JADE ODBC standard (fat) client runs as a JADE client node, establishing communication with a JADE database server in the same way as any other JADE standard client.

The JADE ODBC driver is a Core-level implementation of an ODBC version 3.51 driver.

The JADE ODBC drivers are available in 32-bit and 64-bit versions. If running on a 64-bit machine, the driver used must match the third-party tool being used; for example, it may be necessary to install 32-bit JADE ODBC drivers for use with 32-bit tools.

For more details, see the following subsections.

Relational Views

The relational view allows you control over the visibility of data from the JADE schema.

The following table shows the mapping in a relational view between JADE components and ODBC entities.

JADE Component	ODBC Entity
Class	Table
Primitive attribute, value method	Column
Inverse (1:M relationship)	Foreign key / primary key OID
Inverse (M:M relationships)	Derived table
Collection method	Derived table
Non-inverse collection, primitive array	Derived table
External-keyed dictionary	Additional table

Using the JADE ODBC Driver

The JADE ODBC driver is installed as part of the JADE installation process.

The specific drivers installed depend upon the type of installation. For example, for a JADE 7.1 **Development** installation, the following drivers are installed.

- JADE ODBC Driver: a generic standard driver that is updated with each JADE release
- JADE ODBC Driver 7.1: a standard driver that is specific to the JADE release
- JADE ODBC Thin Client: a thin client driver that is updated with each JADE release

Note The JADE ODBC drivers are available in 32-bit and 64-bit versions. If installing the 64-bit version, both 64-bit and 32-bit drivers are installed. (64-bit version driver names have a suffix of **x64** suffix.

When using a JADE ODBC driver on a 64-bit machine, the bit version of the JADE node executing the JADE ODBC access must match the bit version of the JADE ODBC driver that is being used.

If you are running JADE on a 64-bit machine under Windows in 64-bit mode and you are configuring a 32-bit ODBC driver, run the following program.

```
<\windows-directory>\SysWOW64\odbcad32.exe
```

This runs the 32-bit version of the Microsoft Data Source Administrator program rather than the 64-bit version.

Using the Microsoft Data Source Administrator program, create a DSN that uses the appropriate JADE ODBC driver and configure it as required.

Sample Schemas

The sample schema for this white paper is built on the Erewhon sample schemas found in the JADE **examples\erewhon** directory. You can download the Erewhon sample schemas and the white paper sample schemas, if required, from the **JADE-Erewhon** link at https://github.com/jadesoftwarenz.

» To load the ODBC sample schema

1. Load the Erewhon example schemas, using the **ErewhonInvestments.mul** file.
2. Load the modifications to the **ErewhonInvestmentsModel** schema for this white paper sample, using **ModelODBC.scm** and **ModelODBC.ddb**.
3. Perform a reorganization of the **ErewhonInvestmentsModel** schema.
4. Load the **ErewhonInvestmentsODBC** schema, using **ErewhonInvestmentsODBC.scm** and **ErewhonInvestmentsODBC.ddb**.

JADE ODBC Thin Client

The JADE ODBC thin client driver allows JADE ODBC access from any machine that has the JADE thin (presentation) client installed.

The JADE ODBC thin client driver communicates with a JADE ODBC server application using a TCP/IP connection and the JADE multiple worker TCP/IP transport mechanism. The query to retrieve the JADE objects is executed in the ODBC server application, which can be run on a standard client, application server, or database server node.

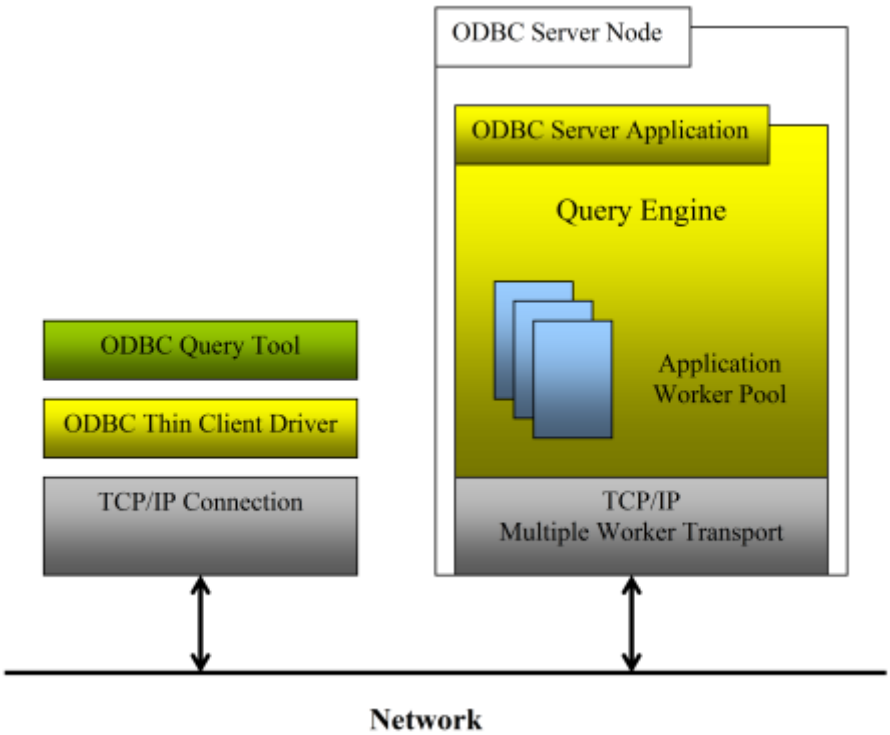
The multiple worker infrastructure enables a number of client connections to be supported by a smaller number of worker processes.

The following outlines the steps required to run an ODBC thin client. The details of the steps are covered in the following sections.

In the schema in which the relational view is defined:

1. Create the non-GUI ODBC server application.
2. On the ODBC server application node:
    a. Specify the JADE initialization file options for the ODBC server application. This can be done directly in the JADE initialization file or in an ODBC server application Extensible Markup Language (XML) file, which is specified in the JADE initialization file options.
    b. Run the ODBC server application.
3. On the ODBC thin client machine:
    a. Install the JADE ODBC thin client driver (normally done as part of **JADE Thin Client** installation).
    b. Create the DSN to connect to the ODBC server application.
    c. Run the ODBC query tool and then select the DSN.

The following image illustrates the execution of the ODBC query tool on the thin client machine and the execution of the ODBC server application in the ODBC server node.



The selection and location of the JADE node in which the ODBC server application is run will affect the performance of the ODBC queries. As this is the node in which the JADE code is executing and the JADE objects are being accessed, the same considerations apply as those for any application server or standard (fat) client node.

ODBC Server Application Definition

The sample **ErewhonInvestmentsODBC** schema contains the **OdbcServer** application, which has **initializeOdbcServer** and **finalizeOdbcServer** as the **initialize** and **finalize** methods. These methods call **app.odbcWorkerInitialize** and **app.odbcWorkerFinalize**.

The parameters for this application are defined in the JADE initialization file using the following.

```
[JadeOdbcServer]
ApplicationConfigFile=D:\jade\odbcconfig.xml
```

You can create and maintain the **odbcconfig.xml** file using the **OdbcServerConfigurator** application in the **JadeMonitorSchema**; for example:

```
jade.exe path=c:\jade\system ini=c:\jade\system\jade.ini schema=JadeMonitorSchema
app=OdbcServerConfigurator
```

The sample configuration file contains the following.

```
<?xml version="1.0" ?>
<jade_config>
    <application schema="ErewhonInvestmentsODBC" name="OdbcServer">
        <odbc_config>
            <listen_host_name>localhost</listen_host_name>
            <listen_port_number>65434</listen_port_number>
            <listen_protocol_family>TcpIPv4</listen_protocol_family>
            <minimum_workers>2</minimum_workers>
            <maximum_workers>5</maximum_workers>
            <read_timeout>0</read_timeout>
            <queue_depth_limit>1</queue_depth_limit>
            <queue_depth_limit_timeout>2</queue_depth_limit_timeout>
            <worker_idle_timeout>120</worker_idle_timeout>
        </odbc_config>
    </application>
</jade_config>
```

The defined host (**localhost**) and TCP/IP port number (**65434**) are used when defining the DSN for the ODBC thin client connection.

When the **OdbcServer** application is started and calls **app.odbcWorkerInitialize**, the initialization file options in the [JadeOdbcServer] section are read, to define the start-up options. This section can specify an XML file that defines the options, or the options can be specified directly in the initialization file. The use of the XML file allows multiple worker applications to be defined for the node.

In the sample, the initialization file specifies the XML file to be used. When the XML file is read, two worker processes are started (the minimum workers value) and these workers wait for connections on **localhost::65434**.

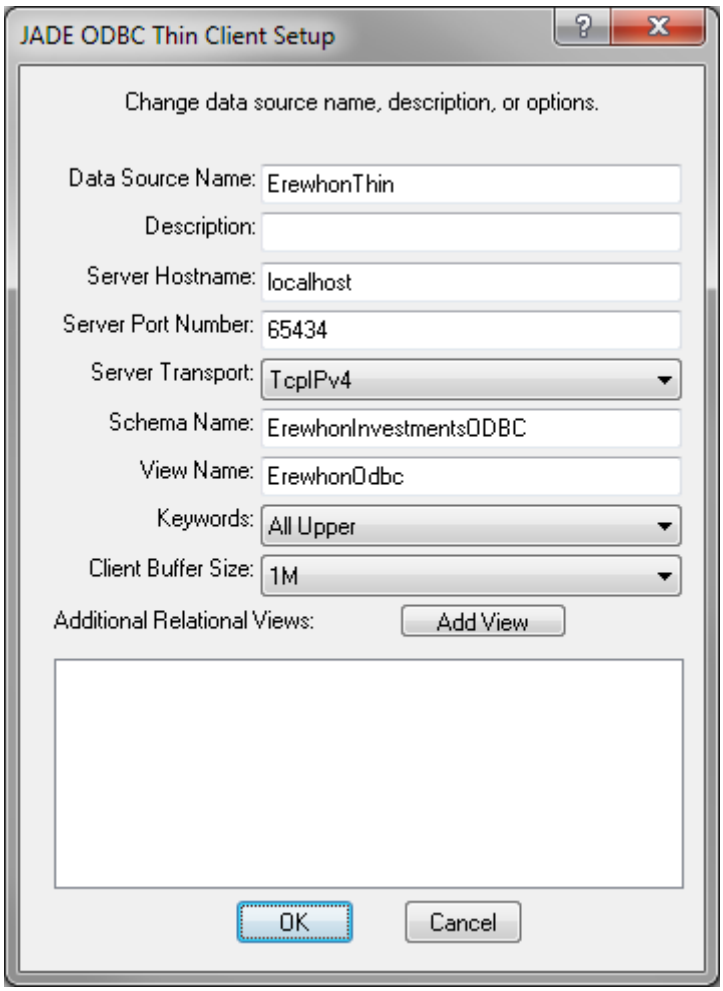
The worker processes also respond to management event call-backs from the multiple worker TCP/IP transport framework to start and stop workers, as required. If all of the workers are busy executing query requests, there will be no workers available to start new workers, if required. For this reason, we recommend that the minimum number of workers be set to one more than the queue depth limit.

In the sample, initially two workers are started. When a connection to a thin client is established, each request is queued and allocated to a worker process. If another connection is established and the requests overlap, the queue depth is exceeded, which sends a management request to start a new worker process. A third worker process is started and continues as long as the worker idle timeout is not exceeded. Up to five workers can be started (the maximum workers value).

Thin Client DSN Definition

To define a DSN for the ODBC thin client connection, run the Microsoft ODBC Administrator on the thin client machine. If running on a 64-bit machine, the DSN must be defined using the appropriate 64-bit or 32-bit administrator and JADE thin client driver for the ODBC query tool that will use the DSN.

Add a User Data Source using the JADE thin client driver. The following screen defines the fields, including the required TCP/IP connection information and relational view information.



ODBC Query Tool Execution

Run the ODBC query tool to be used to access the JADE data.

Specify the **ErewhonThin** DSN to be used for ODBC connection.

When the tool calls **SQLConnect**, the TCP/IP connection to the ODBC server application is opened. The versions of the JADE DLLs being used by the JADE ODBC thin client must match the versions in the ODBC server node. If the versions do not match, an error is reported and the connection is dropped. The JADE thin client binaries must be updated by establishing a standard JADE presentation client to an application server connection and using the automatic download, or by manually updating the thin client binaries.

The connection from the JADE ODBC thin client passes in a user-supplied user name and password. This connection information is validated by a call to the **Global::isUserValid** method. You should implement this to validate the user name and password for ODBC access.

User Context-Dependent Application Method Code

User logic is executed when columns in a **SELECT** statement are mapped to methods. This user logic is executed from within the **OdbcServer** application that was started by a specific user code.

Each **SELECT** query queued for a worker can be from a different thin client, each of which connected with its own user code. Before the user logic is called, the value of **process.userCode** is set to the user name for the thin client connection for which the query is being executed.

In the sample schema, the method **SaleItem**::**getMySaleStatus** returns a string which, based on the current **app.zMyClient** value, returns whether or not the client (the logged in user) has bought this item or bid on this item and the status of the bid. Normally, the **app.zMyClient** would be set in the application initialization code.

To allow the application context to be set for the appropriate thin client user, the ODBC server calls the following routines.

- `Application::startOdbcSession(rv: RelationalView; username: String);`

  This method is called when the connection from the thin client is established, after **Global::isUserValid** has been called to verify the validity of the user name. You can use this method to create a shared transient session object to save any required user context information for this session.

  The code can call **app.setOdbcSessionObject(sessionObject)** to save the session object for later retrieval using **app.getOdbcSessionObject**. This is required because **app** cannot be used to save the session, since the next call for this thin client can be on a different worker and the next call on this worker can be for a different thin client.

  In the sample schema, the following code in **startOdbcSession** creates and saves the application-specific session object.

```
vars
    os : OdbcSession; // class defined for session context
begin
    beginTransientTransaction;
    create os sharedTransient;
    os.zMyClient := Company.firstInstance.allClients[username];
    commitTransientTransaction;
    setOdbcSessionObject(os);
end;
```

- `Application::initializeOdbcSelect(rv: RelationalView; username: String);`

  This method is called when a **SELECT** query is executed. All user logic for the columns will be called on this worker thread. You can use this method to set up the application data required for this user.

  The code can call **app.getOdbcSessionObject** to retrieve the appropriate session object for this user, if this has be created and saved previously.

In the sample schema, the following code in **initializeOdbcSelect** gets the saved context and sets **app.zMyClient** for the correct client (if any).

```
vars
    os : OdbcSession; // class defined for session context
begin
    os := getOdbcSessionObject.OdbcSession;
    if os <> null then
        app.setClient(os.zMyClient);
    endif;
end;
```

If the **SaleItem::getMySaleStatus** method is called during the **SELECT** query, it will return the correct value based on the correct **app.zMyClient** value, which can be null if the ODBC user name is not a valid client.

- `Application::finalizeOdbcSelect(rv: RelationalView; username: String);`

  This method is called when the **SELECT** query processing is completed. You can use this method to clear any **Application** user context that may have been set for the user. The next call on this worker thread can be for a different user.

  In the sample schema, the following code in **finalizeOdbcSelect** clears the **app.zMyClient** value.

```
begin
    app.setClient(null);
end;
```

- `Application::endOdbcSession(sessionObject: Object);`

  This method is called when the connection from the thin client is disconnected. You can use this method to delete the shared transient session object and perform any other clean-up code required.

  In the sample schema, the following code in **endOdbcSession** deletes the session information for the thin client user name.

```
begin
    if sessionObject <> null then
        beginTransientTransaction;
        os := sessionObject.OdbcSession;
        delete os;
        commitTransientTransaction;
    endif;
end;
```

You can also use the ODBC security method defined in the relational view to restrict the visibility of objects to the ODBC user. The security method is used in both standard client and thin client connections. This method is invoked for each object that satisfies the current query before it is added to the result set.

The security method has the following signature.

```
odbcSecurity(relViewName: String): bool;
```

The security method returns **true** if the object can be included in the result set; otherwise it returns **false**.

The security method must be defined for every class that can have instances used in the relational view. In most cases, the method would be defined at the **Object** class level, and reimplemented in the subclasses, as required.

Improving Query Performance

The following subsections discuss techniques for improving query performance. This discussion assumes that the user schema-supplied query provider is not being used for these queries.

The sample **ErewhonInvestmentsODBC** schema contains the relational view **ErewhonOdbc**, which you can use to run the queries.

The following sections apply when using the thin client or standard client JADE ODBC drivers. When using the thin client driver, the query is actually executed in the ODBC server. For clarity, this section refers to the query execution code as the *query engine*.

Using Collections on Joins

Understanding how the query engine utilizes collections and inverses for a **SELECT** query that contains a **WHERE** clause can help you to construct the query in a way that improves performance.

In a relational database, performing a **SELECT** query with a **WHERE** clause that utilizes an index over the table is usually quick; for example:

```
SELECT * FROM "Client" WHERE "name" = 'Roger Boeing'
```

In JADE, the only index over the table **Client** is **Client.allInstances**. To find the required object, the query engine must go through **Client.allInstances** and do a **getProperty** of **name** for each instance.

In real JADE code, of course, this is not how the equivalent operation would be done. If this is a selection that occurs frequently, there will be a collection that would be keyed by **Client.name** on **Company**; for example, **Company.firstInstance.allClients**.

To utilize this collection, the **SELECT** query must "tell" the query engine about this relationship between **Company** and **Client**; for example:

```
SELECT Client."name", Client."address1", Client."address2"
    FROM "Client", "Company"
    WHERE Client.myCompany = Company.oid AND Client.name = 'Roger Boeing'
```

In this case, the **Client.myCompany = Company.oid** clause allows the query engine to check the inverses of **Client.myCompany** for a collection that has **Client.name** as a key. Since this collection exists, the query engine uses the **Company.allClients** collection **getAtKey** method to select the required instance.

Note The **OidFieldSeparator** parameter in the [JadeOdbc] section of the JADE initialization file on the server enables you to customize the OID field separator with a single punctuation or similar character (for example, **@**). This can be useful when the default OID String values that are produced are misinterpreted by a third-party tool as a different ODBC type such as a Decimal.

Trace output from the query engine is useful in checking the paths used for the **SELECT** queries. To enable path execution trace output in the query engine:

- Set the following option in the JADE initialization file.

  ```
  [JadeOdbc]
  QueryExecutionTraceOn=true
  ```

- When running the thin client driver and ODBC server application, set the option in the JADE initialization file for the node on which the ODBC server application is executing. This option is read when the thin client connection is established.

  The ODBC tool may need to be restarted to re-establish the connection if the option is changed.

- When running the standard client ODBC driver, set the option in the JADE initialization file specified in the DSN.
- Trace information is output to the **jommsg.log** file.

You can also set debug output for the thin client and ODBC server. This debug output logs information about all ODBC calls made from the tool and all calls made to the server.

- To set the call tracing debug output for the ODBC server application, set the following option in the JADE initialization file for the JADE node in which the application is run.

```
[JadeLog]
Odbc-s=0x0FFFF
```

- To set the call tracing debug output for the thin client side, set the following option.

```
[JadeLog]
OdbcDriver=0x0FFFF
```

  For the thin client, this option must be set in the **jade.ini** file in the **bin** directory in which the **jadodbc_c** DLL is located.

- For the standard client, set both options in the JADE initialization file specified in the DSN.

```
[JadeLog]
Odbc-s=0x0FFFF
OdbcDriver=0x0FFFF
```

- Debug output is logged to the **jommsg.log** file.

The path execution trace output for the first **SELECT** query above would be as follows.

```
Prepare Query :
SELECT * FROM "Client" WHERE "name" = 'Roger Boeing'
Table Costs :
    Client = 20
Add TableSoftClass SoftLookupAll Client
LookUpTableAll:  Read 20 of 20 : Included in Query : 1
End Execute Query: ResultsReturned=1
```

The table costs indicate the number of instances in the table. In this case, the **LookupTableAll** option (equivalent to **allInstances**) is the only option available, so 20 instances are read and 1 is returned.

The path execution trace output for the second **SELECT** query above would be as follows.

```
Prepare Query :
SELECT Client."name", Client."address1", Client."address2"
    FROM "Client", "Company"
    WHERE Client.myCompany = Company.oid AND Client.name = 'Roger Boeing'
Table Costs :
    Client = 20
    Company = 1
Edges :
    Company :     --- (0.5) ---> Client
    Client :     --- (1) ---> Company
Path 0 (21) =
    All Instances --- (1) ---> Company
    All Instances --- (20) ---> Client
Path 1 (1.5) =
    All Instances --- (1) ---> Company
```

```
    Company --- (0.5) ---> Client
Path 2 (40) =
    All Instances --- (20) ---> Client
    Client --- (1) ---> Company
Minimum Path =
    All Instances --- (1) ---> Company
    Company --- (0.5) ---> Client
Add TableClass LookupAll Company
Add LookupJoinMemberKeyDict Company.allClients
Join Using Equality : myCompany = oid
Optimised Out Comparison : name = Roger Boeing
LookUp Join MemberKey: allClients Read 1 of 20 : Included in query 1
LookUpTableAll: Company Read 1 of 1 : Included in Query : 1
End Execute Query: ResultsReturned=1
```

The table costs indicate the number of instances in the table. The edges indicate the paths found between **Company** and **Client**, using **Company.allClients** and **Client.myCompany**.

Using the **allClients** dictionary to look up the required instances has a lower cost than traversing in the opposite direction. After calculating the cost of all possible paths, the lowest-cost path is selected, using **Company.allInstances** (in this case, one instance) and using the **allClients** dictionary. This results in one instance being read and returned.

Using Collection Methods

A collection method in a relational view is a method that returns a collection that is mapped to a table in the relational view. The method can have input parameters, which are mapped to columns in the table. The value of the column as defined in the query is passed to the method as the parameter value.

Collection methods are useful in a number of scenarios. They can be used to allow the use of persistent collections with constraints. When the query engine calculates the minimum path, collections with constraints are not used. However, a collection method returning the constrained collection is an efficient method of using a persistent constrained collection in the query, since the table is explicitly included in the query.

Collection methods are also useful in hiding complex relationship paths from the ODBC user. The method table joins the tables of the receiver of the method and the membership of the collection.

In the JADE system, the collection may actually be a multiple-level path. Without the collection method, each of these tables must be included in the query and the joins defined. In other cases, the collections a **SELECT** query needs may not exist in the persistent model. Alternatively, the collection may need to be constructed based on specific parameter values. In these cases, it may be more efficient to use a method that returns the collection for the query, rather than have the query engine search for the required instances.

In the sample schema, the **Company.getSaleItemsByRegionByCost** method has been included as an example of using a collection method. This method has the following signature.

```
getSaleItemsByRegionByCost(regionName : String;
                           minPrice   : Decimal;
                           maxPrice   : Decimal): SaleItemsByRegionAndPrice;
```

The method creates a transient instance of the **SaleItemsByRegionAndPrice** external key dictionary, which is keyed by **Region.name** and **SaleItem.getPrice**. If the parameters are null (that is, the user has not specified a value), all values are returned; otherwise the results are constrained by the parameter values. When the method is included in the relational view, the **Company_getSaleItemsByRegionByCost** table is defined with the **Company_oid**, **SaleItem_oid**, **regionName**, **minPrice**, and **maxPrice** columns.

You can use the following **SELECT** query to select all sale items in the London region that cost between $1000 and $100,000, ordered by price.

```
SELECT SaleItem_0.getPrice, SaleItem_0.shortDescription
FROM Company_getSaleItemsByRegionByCost RBC, Company Company_0, SaleItem SaleItem_0
WHERE
    RBC.saleItem_oid = SaleItem_0.oid AND
    RBC.company_oid = Company_0.oid AND
    ((RBC.regionName='London') AND
     (RBC.maxPrice=100000) AND
     (RBC.minPrice=1000))
```

The columns mapped to the method parameters have been given constant values that are passed to the method.

The following query engine trace output shows the execution plan for this **SELECT** query.

```
Minimum Path =
    All Instances --- (1) ---> Company
    Company --- (Unknown) ---> Company_getSaleItemsByRegionByCost
    Company_getSaleItemsByRegionByCost --- (1) ---> SaleItem
Add TableClass LookupAll Company
Add LookupJoinCollMethod Company.getSaleItemsByRegionByCost
Add LookupReference Company_getSaleItemsByRegionByCost.oid
Join Using Equality : saleItem_oid = oid
Join Using Equality : company_oid = oid
Join Using Equality : myRegion = oid
LookUp Join Reference: SaleItem Read 1 of 1 : Included in query 1
LookUp Join Collection: getSaleItemsByRegionByCost Read 1 of 1 :Included in query 1
LookUpTableAll: Company Read 1 of 1 : Included in Query : 1
```

The equivalent query without using the collection method would be as follows.

```
SELECT SaleItem_0.getPrice, SaleItem_0.shortDescription
FROM Country Country_0, Company Company_0, Region Region_0, SaleItem SaleItem_0
WHERE Country_0.myCompany = Company_0.oid AND
      Region_0.myCountry = Country_0.oid AND
      SaleItem_0.myRegion = Region_0.oid AND
      ((Region_0.name='London') AND
       (SaleItem_0.getPrice>=1000) AND
       (SaleItem_0.getPrice<=100000))
ORDER BY SaleItem_0.getPrice
```

In this case, the number of tables included in the equivalent query is not that significant. In a more-complex JADE system, the difference could be significant. The execution path for this query would be as follows.

```
Minimum Path =
    All Instances --- (1) ---> Company
    Company --- (1) ---> Country
    Country --- (0.5) ---> Region
    Region --- (1) ---> SaleItem
Add TableClass LookupAll Company
Add LookupJoinMemberKeyDict Company.allCountries
Add LookupJoinMemberKeyDict Country.allRegions
Add LookupJoinMemberKeyDict Region.allSaleItems
Join Using Equality : myCompany = oid
Join Using Equality : myCountry = oid
Join Using Equality : myRegion = oid
Optimised Out Comparison : name = London
```

```
LookUp Join MemberKey: allRegions Read 0 of 2:Included in query 0
LookUp Join MemberKey: allRegions Read 0 of 4:Included in query 0
LookUp Join MemberKey: allRegions Read 0 of 5:Included in query 0
LookUp Join MemberKey: allRegions Read 0 of 4:Included in query 0
LookUp Join MemberKey: allSaleItems Read 4 of 4:Included in query 1
LookUp Join MemberKey: allRegions Read 1 of 6:Included in query 1
LookUp Join MemberKey: allRegions Read 0 of 6:Included in query 0
LookUp Join MemberKey: allCountries Read 6 of 6:Included in query 6
LookUpTableAll: Company Read 1 of 1 :Included in Query : 1
End Execute Query: ResultsReturned=1
```

For this query, the query engine must read more instances to determine the result set. In the collection method case, the reading of the instances is done in the JADE method code.

Soft Entities and Soft Attributes

Often it is desirable for an administrative user of a JADE application to be able to extend the schema-defined data model by defining additional attributes for a data entity. An administrative user may also be allowed to define new data entities. These user-defined entities and attributes are not hard-coded directly in the JADE metadata, but are soft-coded using JADE classes created specifically for that purpose within the user schema. These user-defined metadata entities and attributes are referred to as *soft entities* and *soft attributes*.

The mapping of soft entities and soft attributes is done using JADE interfaces defined in the RootSchema and implemented in the user schema JADE classes that define the entities and attributes. Methods in the RootSchema **RelationalView** class enable you to add the soft entities and soft attributes to the relational view and define the required mappings.

The following subsections apply when using the thin client or standard client JADE ODBC drivers. When using the thin client driver, the query is actually executed in the ODBC Server. For clarity, this section refers to the query execution code as the *query engine*.

Soft Entity and Soft Attribute Definition

In order to explain how to map soft entities and soft attributes in a relational view, we will first implement the following classes: **SoftEntity**, **SoftAttribute**, and **SoftValue**. This example will use the Erewhon sample schemas as the development base.

For simplicity, in this example all creation and deletion of these classes will be done using JADE scripts. In a useful implementation of soft entities and attributes, you would develop a more-flexible user interface.

See the **ErewhonInvestmentsModelSchema** and **ErewhonInvestmentsODBC** sample schemas for more information. The sample **ErewhonInvestmentsODBC** schema contains additional methods on the classes discussed later in this section for creating and initializing instances, and so on, as required.

The **SoftEntity** class defines each soft entity. Each **SoftEntity** instance defines either a real JADE class in the user schema that has soft attributes added to it or a soft entity that is not represented by a real JADE class.

The **SoftEntity** class contains the following properties.

- `tableName`

  The name of the table in the relational view.

- `classNumber`

  If the entity is mapped to a real JADE class, the value is the JADE class number.

  If the entity is not mapped to a real JADE class, the value is zero (**0**).

The abstract **SoftAttribute** class defines each soft attribute. It contains the following properties.

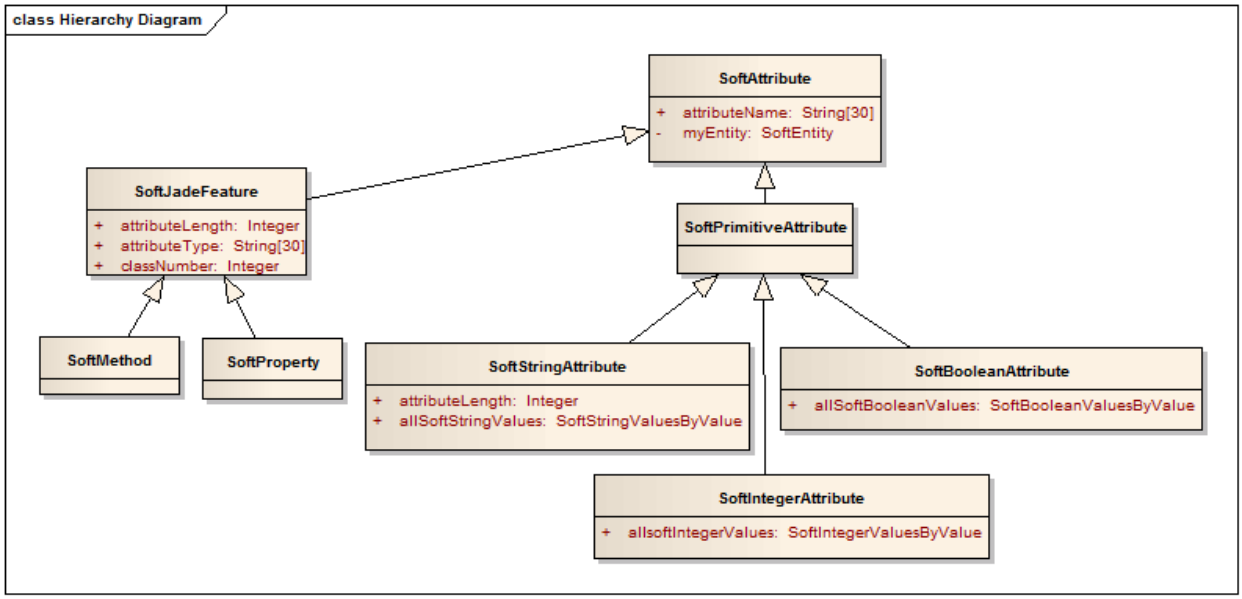
- `attributeName`

  The name of the attribute. This is used as the **Column** name in the relational view.

- `attributeLength`

  The length of the attribute. This is used for variable length types (**String**, **Binary**, and **Decimal**) only.

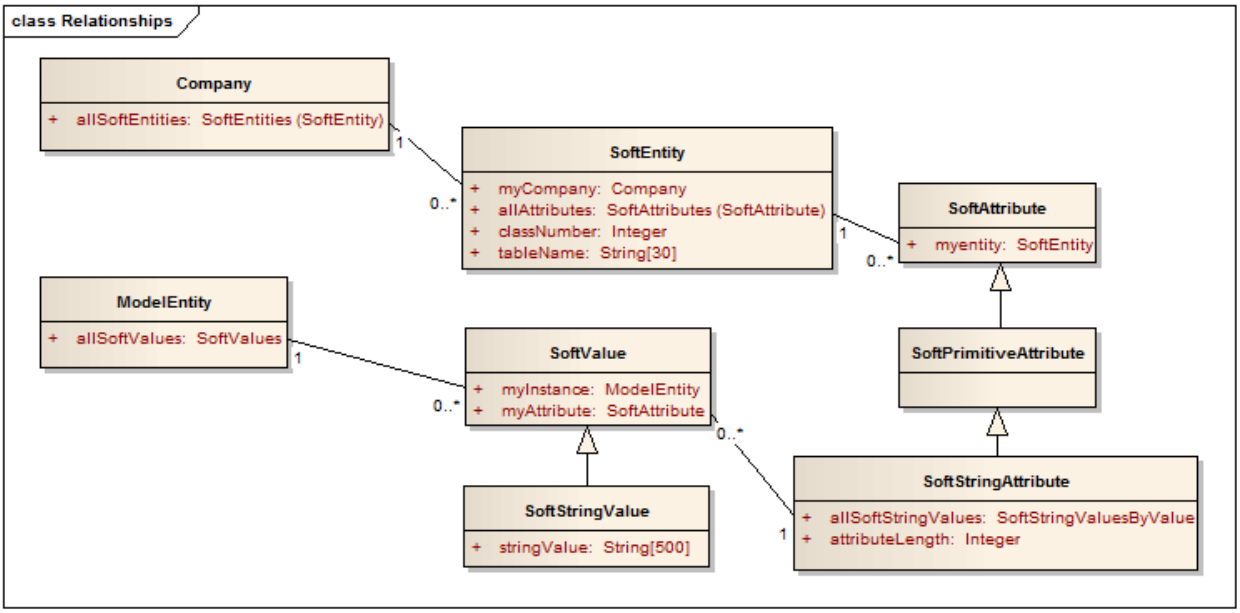
The following image shows the class hierarchy for soft attributes.



The **SoftJadeFeature** subclasses are used when a soft entity that is not based on an existing JADE class maps real JADE properties or methods to a soft attribute.

The **SoftPrimitiveAttribute** (and the real type-specific subclasses) are used when the **SoftAttribute** is not mapped to a real JADE attribute. In this case, the abstract **SoftValue** class (and its real subclasses **SoftIntegerValue**, **SoftStringValue**, **SoftBooleanValue**, and others as required) is used to define the actual values for each object on which the **SoftAttribute** is defined.

The following image shows the relationships between these classes for a **SoftAttribute** of type **String**.



The root **Company** contains a collection of all **SoftEntity** objects. Each **SoftEntity** object contains a collection of all **SoftAttribute** objects. Each **SoftAttribute** object is an instance of **SoftJadeFeature** or **SoftPrimitiveAttribute**.

Each **SoftPrimitiveAttribute** object contains a collection of all **SoftValue** objects (of the appropriate subclass). Each **SoftValue** object contains a value (in this case, **stringValue**) and a reference to the **ModelEntity** object to which this **SoftAttribute** value is assigned.

To create some instances of these classes, run the **JadeScript::createAllSoftValues** method in the **ErewhonInvestmentsODBC** schema. This script will create the following **SoftEntity** (and **SoftAttribute** and **SoftValue**) instances.

- `Agent`

  Represents the JADE class **Agent**. Create and populate the **SoftIntegerAttribute** class **agentNumber** property, which is an **Integer** value. A value of this attribute is assigned to all existing **Agent** instances.

- `Client`

  Represents the JADE class **Client**. Create and populate the **SoftBooleanAttribute** class **current** property, which is a **Boolean** value. A value of this attribute is assigned to all existing **Client** instances.

- `Country`

  Represents the JADE class **Country**. Create and populate the **SoftStringAttribute** class **continent**, which is a **String** of length 50. A value of this attribute is assigned to selected **Country** instances.

- `Person`

  Represents instances of **Agent** or **Client** (not based on a single JADE class).

  Create and populate the **SoftStringAttribute** class **nickname**, which is a **String** of length 80. A value of this attribute is assigned to selected **Agent** and **Client** instances.

  Create the **SoftProperty** class, which maps to real JADE property called **name**. The property is defined on **AddressibleEntity**, which is a superclass of **Agent** and **Client**.

Create the **SoftMethod** property, which maps to real JADE method called **getNameAndAddress**, also on the **AddressibleEntity** class.

These classes define a sample implementation of soft entities, soft attributes, and soft values.

In the sample schema, there is no user interface to view, define, or modify these values. However, the implementation is sufficient for use as an example of mapping these entities and attributes to a relational view.

How to Add Soft Entities to a Relational View

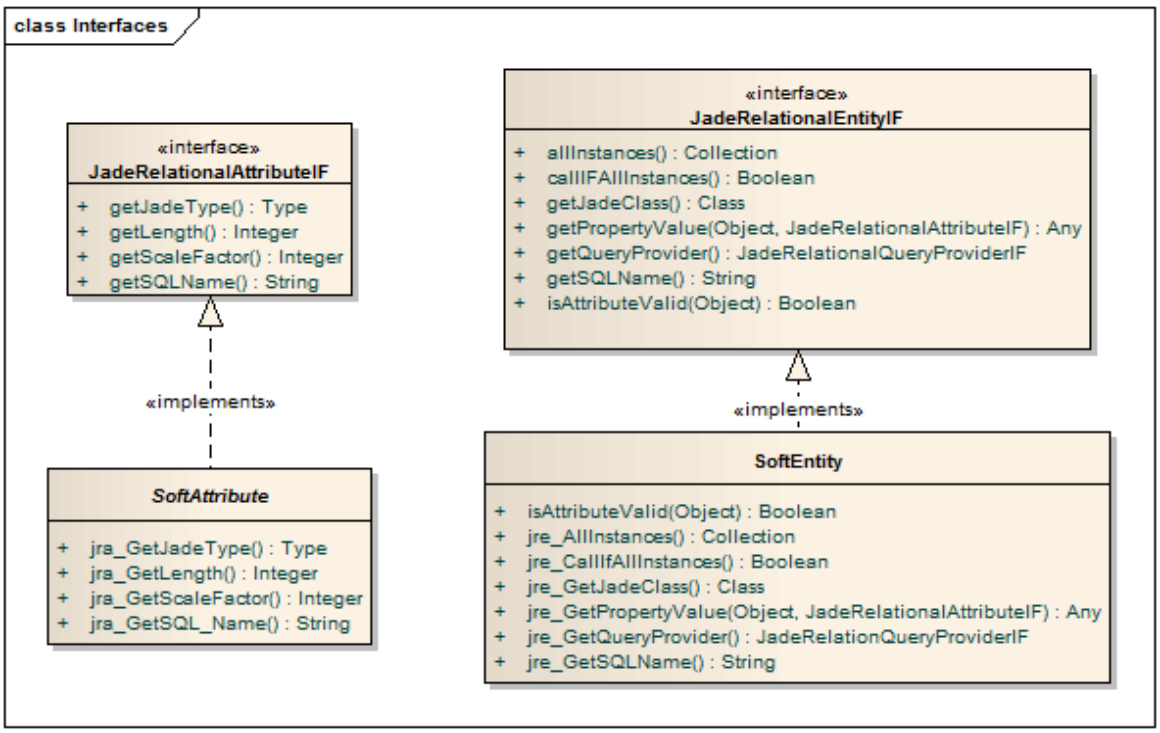
Using the Relational View Wizard, you can map JADE classes to tables and JADE properties and methods to columns in these tables.

Soft entities are administrator-defined metadata and not part of the JADE metadata, so cannot be added to the relational view in the same way.

To add soft entities to a relational view, you must implement the following interfaces from the RootSchema.

- **JadeRelationalEntityIF**, which defines the soft entities to be added to the relational view.
- **JadeRelationalAttributeIF**, which defines the soft attributes on the soft entities.

In the **ErewhonInvestmentsODBC** sample schema, the **SoftEntity** class implements the **JadeRelationalEntityIF** interface and the **SoftAttribute** class implements the **JadeRelationalAttributeIF** interface, as shown in the following image.



Once these interfaces are implemented as required, you can use the following methods on the RootSchema **RelationalView** class to add the soft entities and attributes to the relational view.

- ```
  addUserTable(entityDesc            : JadeRelationalEntityIF;
               includeRealProperties : Boolean;
               includeMethods        : Boolean);
  ```
  ```

Adds the table defined by the **entityDesc** interface object. Properties and methods on the JADE class to which the table maps (if any) can optionally be added to the table.

- ```
  addUserAttribute(entityDesc : JadeRelationalEntityIF;
                   attrDesc   : JadeRelationalAttributeIF);
  ```

  Adds the column defined by the **attrDesc** interface object to the table defined by the **entityDesc** interface instance (must already have been added).

Some important ideas to keep in mind are:

- All soft entity and soft attribute objects are user metadata (not JADE metadata).
- Your logic is responsible for adding, deleting, and maintaining the integrity of the soft entities and soft attributes used in a relational view.
- A relational view can have a mix of tables mapped directly to JADE classes and tables mapped to soft entities that may or may not map to a JADE class. Only the tables mapped directly to JADE classes are extracted during a schema extract. Tables mapped to soft entities must always be added in user logic.

  Soft attributes can be added only to soft entity tables, not to tables mapped directly to a JADE class.

  Soft entity tables that have been added to the relational view are visible in the Relational View wizard. Using the wizard, you can delete these tables but you cannot add or alter them.
- All tables in the relational view always have an **oid** column. This is not defined by user logic, but is always added on creation of the table.

The sample **ErewhonInvestmentsODBC** schema includes the **SoftEntitiesOdbc** relational view, which contains the class **Company**. This class is the root class for the relational view.

To add the previously created **SoftEntity** instances to the **OdbcSoftEntities** relational view, run the **JadeScript::addSoftEntitiesToRV** method in the **ErewhonInvestmentsODBC** schema. This script uses the methods described earlier in this section to add the **Agent**, **Client**, **Country**, and **Person** tables to the relational view.

JadeRelationalEntityIF Interface

A soft entity that is to be mapped to a table in a relational view must be an instance of a class that implements the RootSchema **JadeRelationalEntityIF** interface. This interface contains methods that:

- Define the mapping between the soft entity and the table. The following methods are called at the time the entity is added to the relational view and the values are saved in the JADE metadata relational view information.
  - ```
    callIFAllInstances(): Boolean;
    ```
  - ```
    getJadeClass(): Class;
    ```
  - ```
    getSQLName(): String;
    ```
- Provide information for the query engine when the query is executing. The following methods are called when the **SELECT** query is executed.
  - ```
    allInstances(): Collection;
    ```
  - ```
    getPropertyValue(entity: Object;attributeDesc: JadeRelationalAttributeIF): Any;
    ```
  - ```
    getQueryProvider(): JadeRelationalQueryProviderIF;
    ```
  - ```
    isAttributeValid(attributeDesc: Object): Boolean;
    ```

In the **ErewhonInvestmentsODBC** sample schema, the **SoftEntity** class implements the **JadeRelationalEntityIF** interface.

JadeRelationalAttributeIF Interface

A soft attribute that is to be mapped to a column in a relational view must be an instance of a class that implements the RootSchema **JadeRelationalAttributeIF** interface. This interface contains methods that define the mapping between the soft attribute and the column.

The following methods are called at the time the attribute is added to the relational view and the values are saved in the JADE metadata relational view information.

- `getJadeType(): Type;`
- `getLength(): Integer;`
- `getSQLName(): String;`
- `getScaleFactor(): Integer;`

In the **ErewhonInvestmentsODBC** sample schema, the **SoftAttribute** class implements the **JadeRelationalAttributeIF** interface.

Adding a Soft Entity to a Relational View

A soft entity is added to a relational view when user logic calls the **RelationalView** class **addUserTable** method. The parameters to this method are:

- `entityDesc: JadeRelationalEntityIF`

  The user instance of the soft entity that is added to the relational view. This instance must implement the **JadeRelationalEntityIF** interface. Interface method calls will be made on this instance at the time of the **addUserTable** method call and when the table is used in the query engine.

- `includeRealProperties: Boolean`

  If **true** and the soft entity maps to a real JADE class, the real properties of the JADE class are added to the table.

- `includeMethods: Boolean`

  If **true** and the soft entity maps to a real JADE class, the methods of the JADE class that conform to ODBC method mapping requirements are added to the table.

The interface methods that are called when the soft entity is mapped in the relational view specify how the table will be used in the relational view. A table within a relational view can be a:

- JADE class containing real JADE attributes mapped using the Relational View wizard.

  When these tables are used in a query, the query engine accesses the class instances, properties, and methods using the JADE metadata. Properties are accessed using **getProperty** calls and methods are accessed using **sendMsg** calls.

  When the query contains joins between JADE classes, any existing collection and inverse information can be used to optimize the query. For example, if the query contains **Client.myCompany = Company.oid AND Client.name = 'Philip Jenkins'** in the **WHERE** clause, the query engine will use the **Company.allClients** collection to optimize the selection of the **Client** instance.

- Soft entity mapped using the soft entity class that does not equate directly with a JADE class and has only soft attributes mapped.

  When these tables are used in a query, the query engine has no intrinsic knowledge of how to access the instances for this table or how to access the column values for each instance. It must use the **JadeRelationalEntityIF** interface **allInstances** and **getPropertyValue** methods to retrieve this information.

  Additionally, it has no intrinsic knowledge of how to optimize any **WHERE** clauses specified for this table. The **JadeRelationalQueryProviderIF** (discussed later in this document) can be implemented in user logic to provide this optimization.

- JADE class mapped using the **SoftEntity** class with real JADE attributes mapped and additional soft attributes mapped. When these tables are used in a query, you can specify whether the table should be treated as a:

  - Real JADE class, with only the actual soft attribute values being accessed using the **JadeRelationalEntityIF** interface **getPropertyValue** method.
  - Soft entity mapping, assuming no knowledge about optimization and calling the **JadeRelationalEntityIF** interface **allInstances** and **getPropertyValue** methods.

  In most cases, you would want the table treated a real JADE class, in order to take advantage of the optimizations using collections and inverses. In some cases, however, you may want control over the instances included in the table and can choose to do any optimization yourself using a query provider.

In the **ErewhonInvestmentsODBC** sample schema, the **SoftEntity** class implements the **JadeRelationalEntityIF** methods, as:

- Mapping definition methods, as follows.

  - `jre_CallIFAllInstances(): Boolean;`

    This method returns whether or not the table should be treated as a soft entity or a JADE class.

    For a soft entity, this method returns **classNumber = 0**; that is, it treats the table as a soft entity if this table has no JADE class mapped to it; otherwise it treats it as a JADE class.

  - `jre_GetJadeClass(): Class;`

    This method returns the JADE class to which the table is mapped, if any.

    For a soft entity, this method returns null if **classNumber = 0**; otherwise it returns **currentSchema.getClassByNumber(classNumber)**.

  - `jre_GetSQLName(): String;`

    This method returns the name of the table. If mapped to a JADE class, this can be the class name, but is not required to be. The name must be unique in the relational view in which it is being defined.

    For a soft entity, this method returns **tableName**.

Adding a Soft Attribute to a Soft Entity in a Relational View

A soft attribute is added to a soft entity table in a relational view when user logic calls the **RelationalView**::**addUserAttribute** method. The parameters to this method are:

- `entityDesc : JadeRelationalEntityIF`

  The user instance of the soft entity that defines the soft entity table. This instance must implement the **JadeRelationalEntityIF** interface.

- `attrDesc : JadeRelationalAttributeIF`

  The user instance of the soft attribute that will be added to the table. This instance must implement the **JadeRelationalAttributeIF** interface. Interface method calls will be made on this instance at the time of the **addUserAttribute** method call and when the table is used in the query engine.

The interface methods that are called when a soft attribute is added to a soft entity in a relational view define how the column will be defined in the table.

In the **ErewhonInvestmentsODBC** sample schema, the **SoftAttribute** class implements the **JadeRelationalAttributeIF** methods, as the following mapping definition methods.

- `jra_GetJadeType(): Type;`

  This method returns the JADE type of the column.

  For **SoftAttribute**, this method returns **Integer** for **SoftIntegerAttribute**, and so on.

- `jra_GetLength(): Integer;`

  This method returns the length of the column for variable-length types; for example, **String**, **Binary**, and **Decimal**.

  For **SoftAttribute**, this method returns **attributeLength** for **String** types; else zero (**0**).

- `jra_GetSQLName(): String;`

  This method returns the name of the column. The name must be unique in the table in which it is being defined.

  For **SoftAttribute**, this method returns **attributeName**.

- `jra_GetScaleFactor(): Integer;`

  This method returns the number of decimal digits for columns of type **Decimal**.

Query Execution with Soft Attributes

When a **SELECT** query that includes soft entities is executed, the query execution time methods on the **JadeRelationalEntityIF** interface will be called by the query engine.

For more details, see the following subsections.

SELECT with Soft Entity Mapped to JADE Class

Execute the following query on the sample schema.

```
SELECT name, agentNumber FROM Agent
```

The table **Agent** in the relational view is a soft entity table that is defined as mapped to the JADE **Agent** class, containing all **Agent** properties as columns and the soft attribute **agentNumber**.

When validating the statement, the query engine calls the following interface method.

- `isAttributeValid(attributeDesc: Object): Boolean;`

  This method returns **true** if the attribute defined by **attributeDesc** is still valid. If the method returns **false**, the column will be set to null in the query output.

The call made is equivalent to the following code.

```
vars
    softEntity : SoftEntity;
    jre        : JadeRelationalEntityIF;
    softAttrib : SoftAttribute;
begin
    softEntity := Company.firstInstance.allSoftEntities["Agent"];
    jre := softEntity;
    softAttrib := softEntity.allAttributes["agentNumber"];
    return jre.isAttributeValid(softAttrib);
end;
```

Since the table is mapped to a real JADE class and the **jre_CallIFAllInstances** method call returned **false**, the query engine will retrieve all instances of the **Agent** class from the JADE database.

For each instance of the **Agent** class, the query engine will call the following interface method.

- `getPropertyValue(entity: Object; attributeDesc: JadeRelationalAttributeIF): Any;`

  This method returns the value of the soft attribute defined by **attributeDesc** for the entity object. For soft entities that are mapped to a JADE class, the entity object will be an instance of that JADE class.

The type of the returned value must match the type defined by the **attributeDesc**. For example, if the attribute has been defined as an **Integer**, the value returned must be an **Integer**. If the types do not match, an exception is raised by the query engine.

The **SoftEntity** implementation executes:

```
return entity.ModelEntity.getSoftValue(attributeDesc.Object.SoftAttribute);
```

ModelEntity.getSoftValue looks up the **SoftAttribute** in the **allSoftValues** dictionary and if the entry is found, returns the value; else it returns a null value.

```
softValue := allSoftValues.getAtKey(attribute.attributeName);
if softValue = null then
    return attribute.getNullValue;
endif;
return softValue.getValue;
```

SELECT with Soft Entity Not Mapped to JADE Class

Execute the following query on the sample schema.

```
SELECT Nickname, getNameAndAddress, name, oid FROM Person
```

The table **Person** in the relational view is a soft entity table that is not mapped directly to a JADE class. **Nickname** is a **SoftStringAttribute** instance with a **SoftStringValue** instance, **getNameAndAddress** is a **SoftMethod** instance mapped to the **AddressibleEntity::getNameAndAddress** method, and **name** is a **SoftProperty** instance mapped to the **AddressibleEntity::name** property. The table is defined in the user logic as containing all instances of **Client** and **Agent**.

When validating the statement, the query engine will call the **isAttributeValid** interface method for each column (other than oid) in the select query.

The query engine calls the interface method to retrieve the instances to return for the select.

- `allInstances : Collection;`

  This method returns a collection of instances for this table.

For **SoftEntity**, this method is expected to only be called for the **SoftEntity** "Person". An exception is raised if this method is called for any other instance of **SoftEntity**. For **Person**, it executes the following code

```
create coll transient;
// Person is defined by all instances of Client and Agent
Company.firstInstance.allClients.copy(coll);
Company.firstInstance.allAgents.copy(coll);
return coll;
```

This **coll** transient collection will be deleted by the query engine when it is no longer required.

For each object in the collection returned from **allInstances**, the query engine will call the **getPropertyValue** interface method for each column (other than oid) in the select query. The **SoftEntity** implementation for **SoftMethod** and **SoftProperty** retrieves the values for the entity object using **sendMsg** and **getPropertyValue**, as required.

SELECT with Soft Entity with WHERE Clause

Execute the following query on the sample schema:

```
SELECT Agent.agentNumber, Agent.name, Company.name FROM Agent, Company
    WHERE Agent.myCompany=Company.oid AND Agent.name='Hank Williams'
```

Because this query includes a selection on the soft entity table **Agent** and a **WHERE** clause for that table (**Agent.name='value'**), the query engine calls the interface method to determine whether or not the implementation wants to use its own query provider to optimize this **WHERE** clause or let the query engine do the optimization. The call is made to:

- `getQueryProvider : JadeRelationalQueryProviderIF;`

  This method returns an instance of a class that implements the **JadeRelationalQueryProviderIF** interface if the user logic provides the query provider to optimize the selection for this table (based on the **WHERE** clause).

The **SoftEntity** implementation returns a null value for soft entities that are mapped to a JADE class (as **Agent** is).

The query engine optimization will use the collection **Company::allAgents**, which has **Agent.name** as the key to optimize the selection of the instance to be output.

This is possible because the **Agent** and **Company** tables are joined in the query using the reference **Agent.myCompany** to **Company.oid**. The query engine will check all inverses of **Agent::myCompany** for key matches on the remaining **WHERE** clause. In this case, it finds an appropriate match.

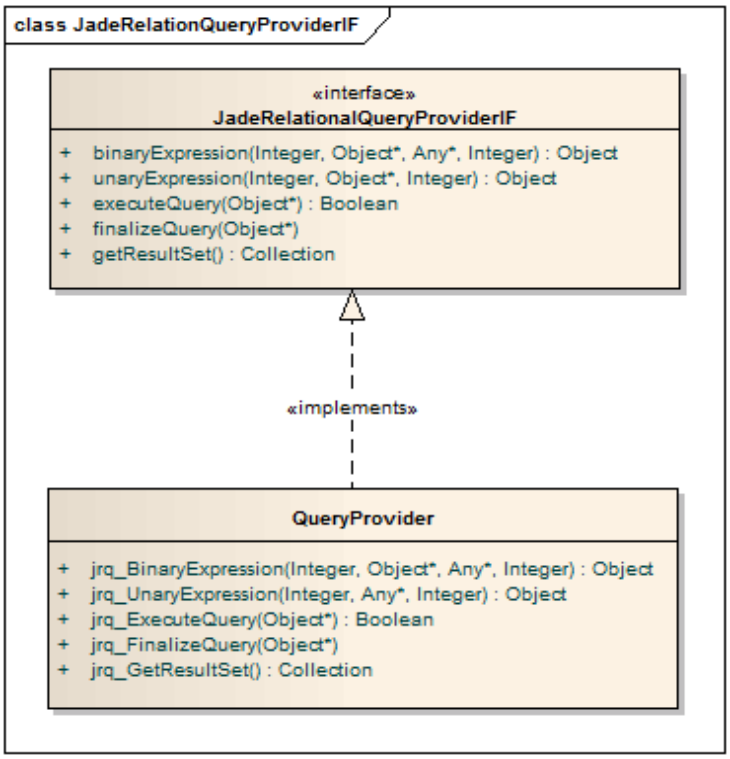
Query Provider Interface

When a soft entity is mapped to a table in a relational view, you have the option of providing your own query provider to optimize the retrieval of results when the **SELECT** query includes a **WHERE** clause that includes values in the soft entity.

This query provider must implement the **JadeRelationalQueryProviderIF** interface from the RootSchema. When a soft entity table is used in a **WHERE** clause, the query engine will call the **JadeRelationalEntityIF** interface **getQueryProvider** method.

If this method returns a non-null value, this object will be used to call the **JadeRelationalQueryProviderIF** interface methods to process the **WHERE** clause expressions for this table. The result of this is a collection that contains the objects that satisfy the **WHERE** clause expression.

In the **ErewhonInvestmentsODBC** sample schema, the **QueryProvider** class is a query provider that implements the **JadeRelationalQueryProviderIF** interface, as shown in the following diagram.



It is not a complete implementation and as implemented, is intended to be used only for **SoftEntity** instances that do not map to a JADE class. In the case of the **ErewhonInvestmentsODBC** sample schema, this is the **Person** table that maps to **Agent** and **Client** instances.

To illustrate the process of using the query provider from the query engine, we will describe the calls made to process a simple **SELECT** with **WHERE** clause using the **Person** table.

```
SELECT Nickname, getNameAndAddress, name, oid FROM Person WHERE Nickname > ''
```

Since the **WHERE** clause includes an attribute on a soft entity table, the query engine calls the **JadeRelationalEntityIF** interface **getQueryProvider** method to determine if a user-defined query provider should be used for this query. This method is called using the instance of the **SoftEntity** class that was associated with the **Person** table when the table was added to the relational view.

In the **ErewhonInvestmentsODBC** sample, the **SoftEntity::jre_GetQueryProvider** implementation creates and returns a transient instance of **QueryProvider** if the soft entity is not mapped to a JADE class. It saves this transient instance on the **app** instance, so it can be deleted when no longer required.

```
if classNumber = 0 then
    if app.myQueryProvider = null then
        create app.myQueryProvider transient;
    endif;
    return app.myQueryProvider;
endif;
return null;
```

Using the instance of **QueryProvider** returned by this method, the query engine next calls the **JadeRelationalQueryProviderIF** interface **binaryExpression** and **unaryExpression** methods, as required, to define the **WHERE** expression.

In our example **WHERE** clause, the query engine calls the **jrq_BinaryExpression** method, as follows.

```
binaryExpression(operator: Integer; leftOperand: Object io;
                 rightOperand: Any io; level: Integer): Object updating;
```

The **binaryExpression** method has the following parameters.

```
binaryExpression(JadeRelationalEntityIF.Op_GreaterThan,
                 SoftEntity object for the Nickname attribute,
                 String with value null string (''),
                 1);
```

The sample implementation builds a temporary tree to save the expression using the subclasses of **QueryNode**, **OperatorNode**, and **LeafNode**. The return value is the instance of **OperatorNode** for this tree.

The query engine then calls the **JadeRelationalQueryProviderIF** interface **executeQuery** method, as follows.

```
executeQuery(expression: Object io): Boolean updating;
```

The **executeQuery** method has the following parameter.

```
executeQuery(instance of OperatorNode from binaryExpession);
```

In the sample schema, the **jrq_ExecuteQuery** implementation builds a result set of objects that match the expression passed in to the execution. The result set is built in the **myResultSet** on **QueryProvider** collection.

The **jrq_ExecuteQuery** method interprets the **LeafNode** left and right operands. For an **OperatorNode**, the following combinations are possible.

- Operator is a boolean operator **Op_And**, **Op_Or**, or **Op_Not**
  - Left and right operands are **OperatorNode** instances or Boolean **SoftAttribute**, **Property**, or **Method** instances.
- Operator is a comparison operator **Op_Equal**, **Op_NotEqual**, **Op_GreaterThan**, **Op_LessThan**, **Op_GreaterThanOrEqual**, **Op_LessThanOrEqual**, **Op_Like**, or **Op_NotLike**.
  - Left operand is a **SoftAttribute**, **Property**, or **Method** instance.
  - Right operand is a **SoftAttribute**, **Property**, **Method**, or **SoftValue** (literal) instance.

For this **WHERE** clause (**Nickname > ‘’**), the **leftOperand** is a **SoftPrimitiveAttribute** and the **rightOperand** is a literal **String** value. To process this input, the query provider **jrq_ExecuteQuery** method calls the **executeSoftAttrValueToLiteral** method. This method creates an iterator for the **allSoftStringValues** collection on the **SoftStringAttributeNickName** instance and starts the iterator at the **getAtKeyGtr('')** instance. All instances in the collection from that point on are put into the results collection.

The query engine then calls the **JadeRelationalQueryProviderIF** interface **getResultSet** method, to retrieve the set of objects that match the expression. In the sample, the **jrq_GetResultSet** implementation returns the **myResultSet** value built in the **executeQuery** call.

For each object in the results collection, the query engine will retrieve the column values for the **SELECT** query by calling the **JadeRelationalEntityIF** interface **getPropertyValue** method.

When the **SELECT** output rows are complete, the query engine calls the **JadeRelationalEntityIF** interface **finalizeQuery** method with the same expression parameter as passed to the **executeQuery**.

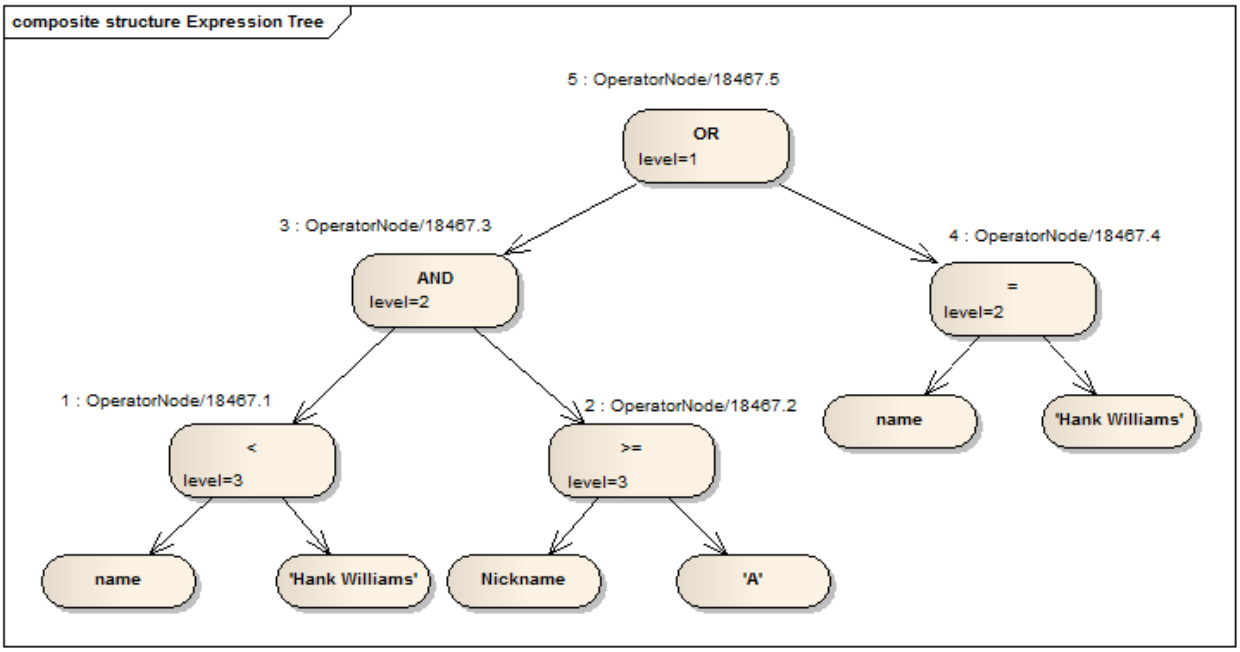
At this point, the implementation cleans up the transients created for this query. It deletes the transient **QueryNode** expression and deletes itself; that is, the **QueryProvider** instance. The transient set **myResultSet** is deleted by the query engine.

In a more-complicated **SELECT** when the **WHERE** clause contains multiple expressions, the **binaryExpression** method will be called multiple times to process the expression. The output of a **binaryExpression** call can be passed in as an operand to a subsequent call.

The final output is passed to the **executeQuery** method; for example, with the following **SELECT** statement.

```
SELECT Nickname, getNameAndAddress, name, oid FROM Person
WHERE name < 'Hank Williams' AND Nickname >= 'A' OR
            name = 'Hank Williams'
```

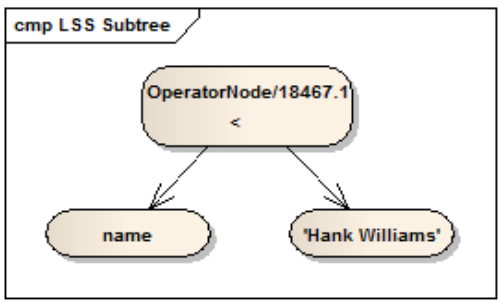
The following diagram shows the expression tree built to represent this **SELECT** statement.



The **binaryExpression** method will be called five times with the following parameters (oid values are example values only).

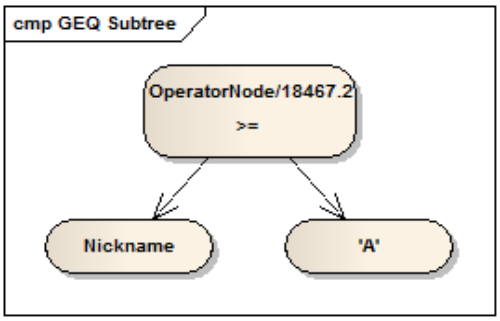
1. `(Op_LessThan,SoftProperty/2078.2,"Hank Williams",3)`

   SoftProperty/2078.2 represents the **name** property. The return value is **OperatorNode/18467.1**.



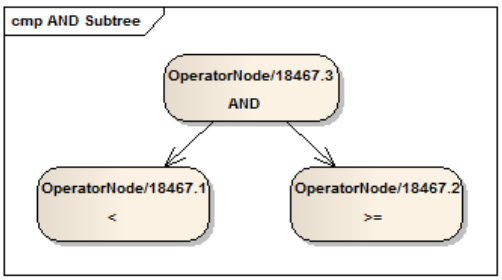
2. `(Op_GreaterThanOrEqual,SoftStringAttribute/2061.12,"A",3)`

   SoftStringAttribute/2061.12 represents the **Nickname** soft attribute. The return value is **OperatorNode/18467.2**.



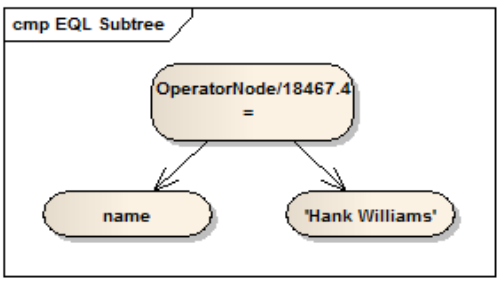
3. `(Op_And,OperatorNode/18467.1,OperatorNode/18467.2,2)`

   The operands are the **OperatorNode** objects returned from calls 1 and 2. The return value is **OperatorNode/18467.3**.



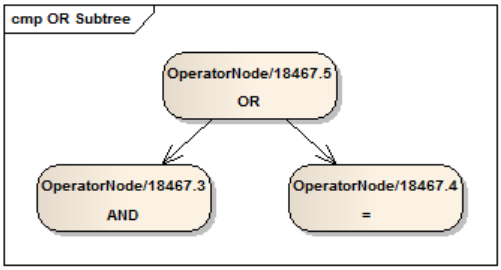
4. (Op_Equal,SoftProperty/2078.2,"Hank Williams",2)

   SoftProperty/2078.2 represents the **name** property. The return value is **OperatorNode/18467.4**.



5. `(Op_Or,OperatorNode/18467.3,OperatorNode/18467.4,1)`

The operands are the **OperatorNode** objects returned from calls 3 and 4. The return value is **OperatorNode/18467.5**.



The **OperatorNode** object returned in the last call (**18467.5**) will be passed into **executeQuery** for execution.

JADE Documentation on Relational Views and ODBC Queries

The following JADE documentation should be read for additional information about defining relational views and using the ODBC drivers in queries.

- Chapter 9, "Defining ODBC Inquiry Relational Views and Ad Hoc Indexes", in the *JADE Development Environment User's Guide*, for details about using the Relational View wizard to define a relational view.
- "Obtaining a Relational View of your JADE Database", in Chapter 2 of the *JADE External Interface Developer's Reference*, for details about using the JADE ODBC driver.

Conclusion

When using the JADE ODBC driver to access JADE data using a query tool that uses ODBC, the following may be useful.

- Using the ODBC thin client will improve performance, especially if the server application is executed on the same machine as the database server. Some JADE development is required to set up the application and ensure the correct session context for method execution.
- Queries that define joins that efficiently utilize existing JADE collections will perform better. Using the ODBC Execution trace allows you, as the query developer, to check the query engine path selection.
- Collection methods may be useful to improve performance when:
  - The collection to be used is constrained.
  - The collection to be used must be constructed for the query.
  - The paths to the required instances are long and difficult to set up in the query.
- Soft attributes can be mapped to ODBC tables using the JADE interfaces supplied in the RootSchema. Some JADE development is required to implement the interfaces and include the soft attributes and soft tables in the relational view.
- The query provider may be useful for JADE systems that use soft attributes extensively. Significant JADE development may be required to implement the query provider JADE Interface supplied in the RootSchema, especially if the search engine must be built from scratch.