Building the

Izenda Custom Data Source Adapter

Contents

[Introduction 5](#_Toc505356789)

[Izenda Data Source Adapter Overview 5](#_Toc505356790)

[The Concept in Data Source Adaptor 8](#_Toc505356791)

[Query Source Field 8](#_Toc505356792)

[Query Source 8](#_Toc505356793)

[Query Source Category 8](#_Toc505356794)

[DBSource 9](#_Toc505356795)

[Query Source Parameter 9](#_Toc505356796)

[SQL Query Source Type 9](#_Toc505356797)

[Relationship 9](#_Toc505356798)

[The connection 9](#_Toc505356799)

[Mapping between RDBMS Concept and Izenda Data Source Provider Concept 9](#_Toc505356800)

[Izenda Fusion Module and Query Tree 10](#_Toc505356801)

[Query tree Node 10](#_Toc505356802)

[SQL Command Generators 11](#_Toc505356803)

[Operator Command Generator 11](#_Toc505356804)

[Filter Command Generator 14](#_Toc505356805)

[Calculated Command Generator 15](#_Toc505356806)

[Izenda Raw Query 17](#_Toc505356807)

[Naming Identifier 18](#_Toc505356808)

[IZENDA\_CONCAT function 18](#_Toc505356809)

[DATETUNCATE function 19](#_Toc505356810)

[IDataSourceAdaptor Interface 19](#_Toc505356811)

[Crossing Concern Members 19](#_Toc505356812)

[LogManager Property 19](#_Toc505356813)

[Get connection information 20](#_Toc505356814)

[TestConnection(Guid, string) 20](#_Toc505356815)

[GetDatabaseServer(string) 20](#_Toc505356816)

[GetDatabaseName(string) 20](#_Toc505356817)

[GetConnectionStringWithServerAndDatabaseName 20](#_Toc505356818)

[Load Data Source Schema Definition 21](#_Toc505356819)

[LoadDatabaseSchema(string) 21](#_Toc505356820)

[LoadFields(string, string, string, string, bool, List<QuerySourceParameter>, bool, int, ILog) 21](#_Toc505356821)

[GetQuerySourceParameters(string) 22](#_Toc505356822)

[GetQuerySourceParameters(List<QuerySourceParameter>, string, string, string, QuerySource) 22](#_Toc505356823)

[LoadConnectionDetail(Connection, bool, string, ILog) 22](#_Toc505356824)

[GetExtendedProperties(string) 23](#_Toc505356825)

[GetSupportedDataSourceType(string) 23](#_Toc505356826)

[GetBaseDataTypes(string) 23](#_Toc505356827)

[LoadCustomQuerySourceFields(string, string) 23](#_Toc505356828)

[Execute SQL commands and query data 24](#_Toc505356829)

[Query(string, string, object, int) 24](#_Toc505356830)

[Query(string, string, string, string, string) 24](#_Toc505356831)

[Query(stirng, QueryTreeNode, FusionContextData) 25](#_Toc505356832)

[CanQuery(QueryTreeNode, FusionContextData, string) 25](#_Toc505356833)

[Execute(string, string, List<dynamic>) 25](#_Toc505356834)

[GetFirstValueInFilteredValue(string, string) 26](#_Toc505356835)

[QueryMultiple(string, string, object, int, Action<SqlMapper.GridReader>) 26](#_Toc505356836)

[Specific Configuration for RDBMS in Adaptor Component 27](#_Toc505356837)

[Data Types 27](#_Toc505356838)

[DatabaseSupportDataType class 28](#_Toc505356839)

[Database Functions 29](#_Toc505356840)

[Database Functions Format 31](#_Toc505356841)

[Database Constants 34](#_Toc505356842)

[Select Field Command Format 35](#_Toc505356843)

[Query Normalizer 35](#_Toc505356844)

[Preparation for New Data Source Adaptor 36](#_Toc505356845)

[Download dependency components 36](#_Toc505356846)

[Create Query Normalizer Component 37](#_Toc505356847)

[General Purpose 37](#_Toc505356848)

[Analysis to implement Query Normalizer Activity 37](#_Toc505356849)

[Create Izenda.BI.QueryNormalizer.Redshift Library 37](#_Toc505356850)

[Create abstracted class RedshiftQueryNormalizerActivity 39](#_Toc505356851)

[Add class RedshiftEscapseTableNameActivity 39](#_Toc505356852)

[Add class RedshiftReplaceConcatFunctionActivity 40](#_Toc505356853)

[Add class RedshiftReplaceDateTruncateFunctionActivity 42](#_Toc505356854)

[Write your custom Query Normalizer Activity 42](#_Toc505356855)

[Create Data Source Adaptor for Redshift 43](#_Toc505356856)

[Identify Data Provider 43](#_Toc505356857)

[Dapper Usage 44](#_Toc505356858)

[Create new Adaptor Project 44](#_Toc505356859)

[Add Project Dependency 45](#_Toc505356860)

[Configure Data Types and Build-in Functions 47](#_Toc505356861)

[Data Types 47](#_Toc505356862)

[Database Functions 49](#_Toc505356863)

[Database Function Format 50](#_Toc505356864)

[Database Constants 51](#_Toc505356865)

[Implement connection handler class 51](#_Toc505356866)

[Implement schema loader class 54](#_Toc505356867)

[ISchemaLoader interface and implementation 54](#_Toc505356868)

[GetQuerySourceParameters(string) 56](#_Toc505356869)

[GetQuerySourceParameters(string, string, string) 56](#_Toc505356870)

[LoadCustomQuerySourceFields(string, string) 57](#_Toc505356871)

[LoadQuerySourceField(string) 58](#_Toc505356872)

[LoadSchema(string) 59](#_Toc505356873)

[LoadFields(string, string, string, string, bool, List<QuerySourceParamter>, bool, int, ILog) 61](#_Toc505356874)

[LoadRelationship(string, List<string>) 63](#_Toc505356875)

[Implement Data Source Adaptor class 64](#_Toc505356876)

[Declare adaptor class and exported configuration 64](#_Toc505356877)

[Override SchemaLoader and Connection Property 65](#_Toc505356878)

[Override NormalizeQuery(string) Method 65](#_Toc505356879)

[Override GetConnectionStringWithServerAndDatabaseName(string) Method 66](#_Toc505356880)

[Override GetFirstValueInFilteredQuery Property 66](#_Toc505356881)

[Override GetBaseDataTypes() Method 66](#_Toc505356882)

[Override Query(string, string, object, int) Method 67](#_Toc505356883)

[Override QueryMutiple(string, string, object, int, Action<SqlMapper.GridReader>) Method 67](#_Toc505356884)

[Override QueryTreeCommandGenerator Propery 68](#_Toc505356885)

[Implement Custom Operator Command Generators 70](#_Toc505356886)

[Operator Command Generators Are Required to Override 70](#_Toc505356887)

[Overriding Other Operator Command Generators 75](#_Toc505356888)

[Register Operator Command Generator in Visitor Class 76](#_Toc505356889)

[Implement Expression Command Generator 78](#_Toc505356890)

[Analysis Overriding Expression Command Generator 78](#_Toc505356891)

[Add generator and visitor class 79](#_Toc505356892)

[Implement Filter Generator 79](#_Toc505356893)

[Create Redshift Data Source Adaptor Using ODBC Driver 79](#_Toc505356894)

[Introduction and Analysis 79](#_Toc505356895)

[Install Amazon Redshift ODBC Driver 79](#_Toc505356896)

[Implement Adaptor with OdbcConnection 79](#_Toc505356897)

[RedhisftConnection class 79](#_Toc505356898)

[RedhisftSchemaLoader class 79](#_Toc505356899)

[RedshiftDataAdaptor class 79](#_Toc505356900)

[Troubleshooting When Developing New Data Source Adaptor 79](#_Toc505356901)

[Debugging on Data Source Adaptor 79](#_Toc505356902)

[Investigate Issue by Analysis Izenda Log 79](#_Toc505356903)

[Troubleshooting for Limitation on ODBC Driver 79](#_Toc505356904)

[Testing New Data Adaptor From Izenda UI 79](#_Toc505356905)

[Connection String 80](#_Toc505356906)

[Data Model 81](#_Toc505356907)

[Report 82](#_Toc505356908)

[Case 1: Global report contains tenant field 83](#_Toc505356909)

[Case 2: Report with filters 86](#_Toc505356910)

[Case 3: Report with grouping, aggregated function and sub total 87](#_Toc505356911)

[Case 4: Report with Calculated Field 89](#_Toc505356912)

[Case 5: Drilldown chart report part 90](#_Toc505356913)

[Case 6: Exporting report 91](#_Toc505356914)

# Introduction

This document guideline will give you concept and guideline to create an Izenda Data Source Adaptor.

# Izenda Data Source Adaptor Overview

Izenda Data Source Adaptor is part of Izenda system provides communication between RDBMS Data Source and Izenda reporting logic. It takes responsibility for querying report data and calculating data aggregation and combine with Izenda Business Layer to enable data access and data manipulation capabilities on specific RDBMS data server.

The Data Source Adapter component handles transformation Izenda reporting logic to SQL command of specific RDBMS then specify SQL query command that provide elementary CRUD functionality on data source.

An Adaptor also is using to take care of data source connection in Izenda system and explore data source schema structure and configuration then convert to Izenda abstraction concepts to save on Izenda system database for management data connection as well as process for reporting logic.

Data Source Adapter in Izenda system:

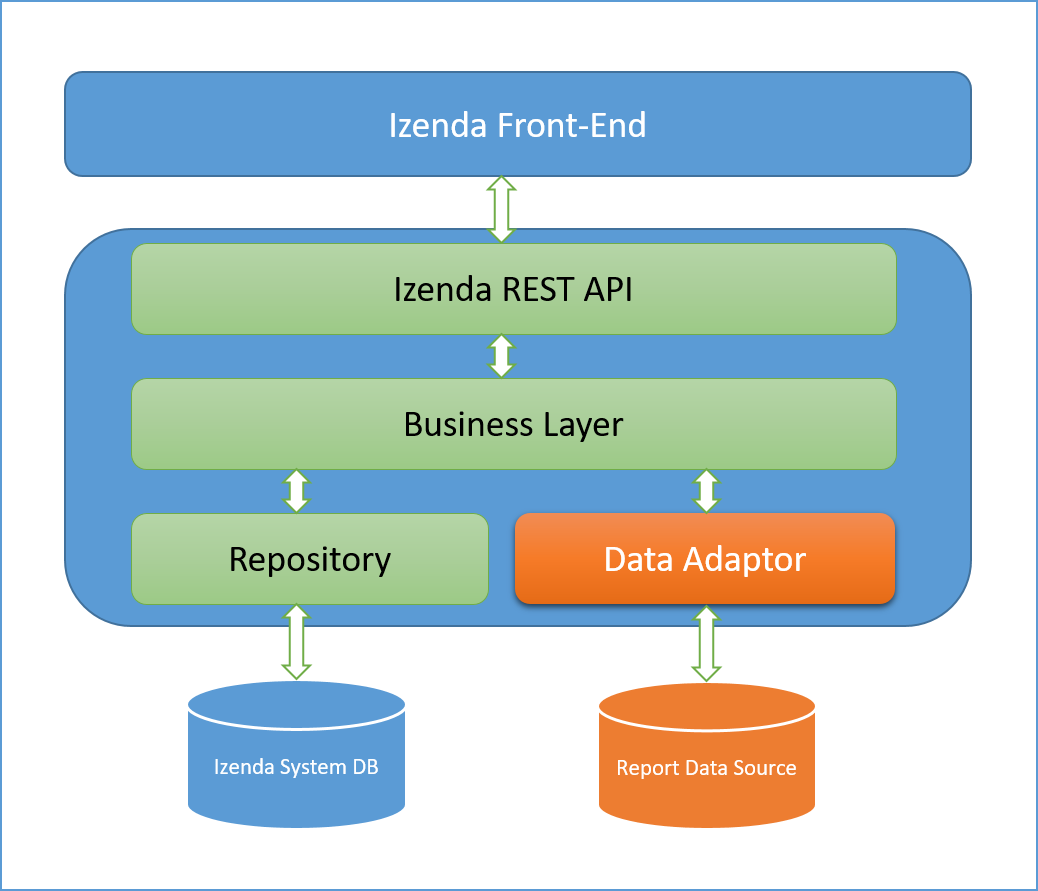


Figure 1: Data Source Adaptor in Izenda System

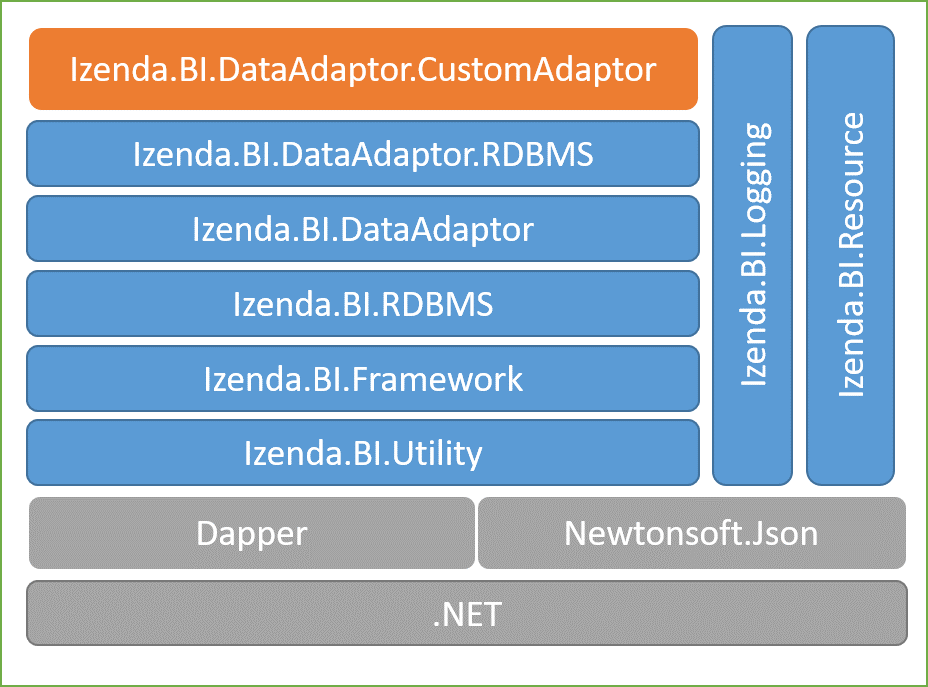
Below diagram demonstrate dependency of a Data Source Adapter component. The orange component is your custom data source adaptor.

Figure 2: Dependency component of a Data Source Adaptor

A custom Data Source Adaptor is able to implement with any architecture as long as it can get report data and manipulate aggregation calculation on Data Source of a database connection. However, look through this guideline; it will be following on architecture that we have used to implement Data Source Adaptor for Izenda system, and a specific Adaptor component should have logical units similar other Izenda Adaptor such as:

* Main Data Source adapter component, which is inheriting from IDataSourceAdapter interface.
* Connection handler unit to verify and open data source connection as well as collect more information on the connection.
* Data Source schema loader to explore data source schema structure and convert to abstraction object that Izenda system can understand.
* The logical for converting reporting logic processing on Izenda Logic Layer to specific SQL command for a database server, that will generate SQL query statement which will be able to run completely on the RDBMS that you are going to develop Adaptor for.

The Data Source Adaptor usually has logical units as diagram below:

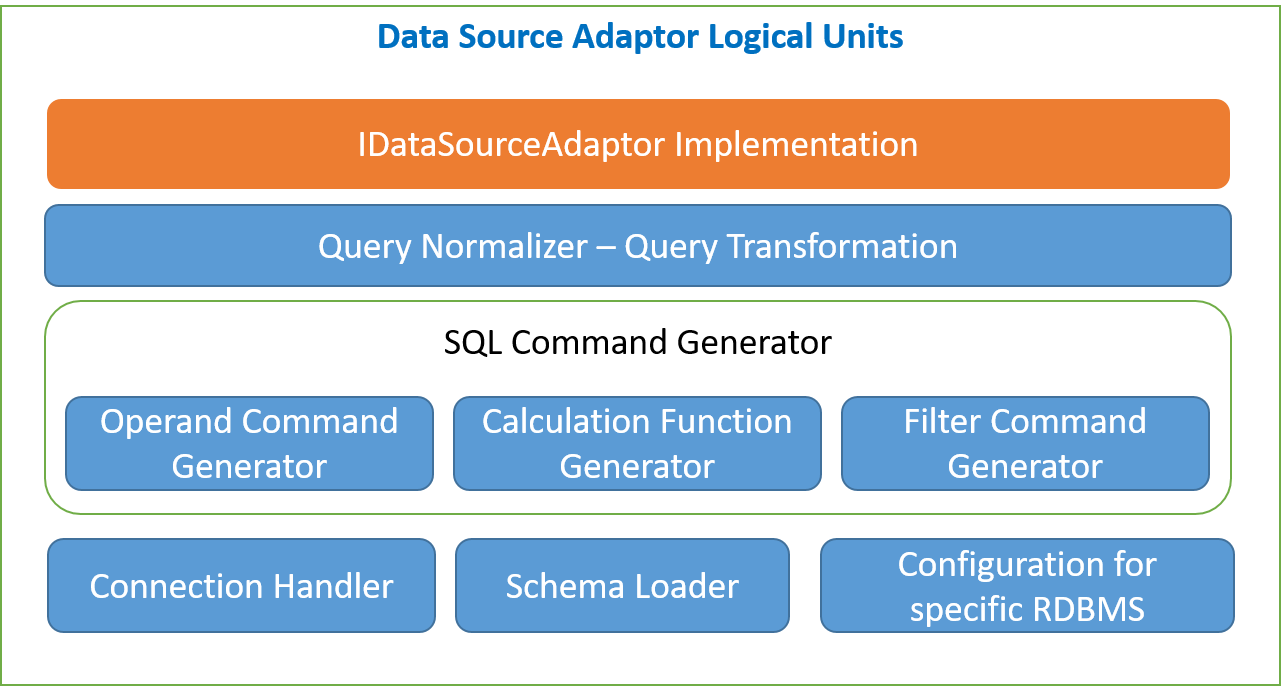


Figure 3: Data Source Adaptor logical units

This table describe usage purpose for each logical unit in above diagram:

|  |  |  |
| --- | --- | --- |
| .No | Logical Unit Name | Description |
| 1 | The class implement IDataSourceAdaptor | The most important unit for an Adaptor can work with Izenda BI system |
| 2 | Query Normalizer – Transformation | Convert or transform Izenda “raw” query to SQL syntax that target RDBMS can execute completely without issue |
| 3 | Operand Command Generator | Unit to generate SQL command for operand that Izenda uses on report. Each operand has corresponding concreate SQL statement |
| 4 | Calculation Function Generator | Izenda uses some built in function in RDBMS on report processor, but if RDBMS does not contain some of that function, we have to create generator to create new corresponding function. We also called as Expression Command Generator. |
| 5 | Filter Command Generator | On Izenda Report, there are many filters can be applied, this logical unit is using to generate filtering statement in syntax of your RDBMS |
| 6 | Connection Handler | Each RDBMS will have specific data provider to handle opening connection as well as executing data query and data modifying, this component logic will use that data provider to open and verify connection as well as get more information about data source connection |
| 7 | Schema Loader | The unit for exploring data source schema structure convert to Izenda abstraction object such as Query Source, Query Source Field… etc. |
| 8 | Configuration for RDBMS | Each RDBMS has its own separated configuration and feature, this unit will provide configuration to distinguish your RDBMS in concept of supported datatype, build-in function and data formatting |

Table 1: Description of Logical Units should be contained in a custom Data Source Adaptor

This document will provide detail description for each logical unit in above diagram in separated section, in generally it will describe about concept, the definition and how to implementation as well as how each unit works and effect on Izenda system.

# The Concept in Data Source Adaptor

Beyond introduction, definitions, before going to practice how to create a custom driver, there’s of course an expectation that you need to know some concepts are using in Izenda data source adapter what you might implement in your custom driver.

## Query Source

This concept presents for a table, view, store procedure or function in database.

The class QuerySource in Izenda.BI.Framework.Models namespace is abstraction class for this concept. From the view of Data Source Adaptor we just care about related properties that include Query Source Name, Query Source Type and list of Query Source Field, which are belong to concreate Query Source object.

|  |  |  |
| --- | --- | --- |
| # | Property Name | Description |
| 1 | Name | The query source name, either name of table, view, procedure or function |
| 2 | Type | The value indicate the Query Source is presentation object of either table, view, procedure or function. |
| 3 | QuerySourceFields | The column list of either table, view, procedure or function. |

Table 2: Concerned properties of Query Source from Data Source Adaptor Component

Image below is presentation of QuerySource object on Izenda UI:



Figure 4: The Query Source object's presentation on Connection Info View

## Query Source Category

This concept maps to schema in database. Cause of that each Query Source Category contains multiple Query Source, which is present for a table, view, procedure or function in database. It is container of Query Source objects.

From Adaptor aspect, we only take care name of schema that is Query source Category name.

The abstraction class for this concept is Izenda.BI.Framework.Models.QuerySourceCategory.

## Query Source Field

The concept presents for column of either table, view, procedure or function. It store information to distinguish between columns of a Query Source in database.

From Adaptor component, the concerned info include Name, Data Type, Izenda Data Type, Allow Distinct, ExtendedProperty, Position, Query Source Name and Category Name. Detail description of properties as table below:

|  |  |  |
| --- | --- | --- |
| # | Property Name | Description |
| 1 | Name | Corresponding column name |
| 2 | DataType | The column data type in database. This is data type name, which is supported by database system. |
| 3 | IzendaDataType | The Izenda Data Type name, because Izenda supports multiple database system, it group database data type into some group to handle calculating manipulation, such as numeric, date time, currency…etc. |
| 4 | AllowDistinct | The property indicates that a data type whether must be unique or not. |
| 5 | ExtendedProperties | Not required property, it is extension holder for additional information about data source field. |
| 6 | Position | The position of field in data source, this map to position number of column in table, view, result set of store procedure. |
| 7 | CategoryName | The schema that the field is belong. |
| 8 | QuerySourceName | This is the name of container query source. |

Table 3: The concerned property of Query Source Field from Adaptor component

Except ExtendedProperty, other properties have to have value. In schema loader function, it will load all of above information from database structure.

The abstraction class for this concept is Izenda.BI.Framework.Models.QuerySource.

Image below demonstrate Query Source and Query Source Filed on Izenda Report Designer UI:

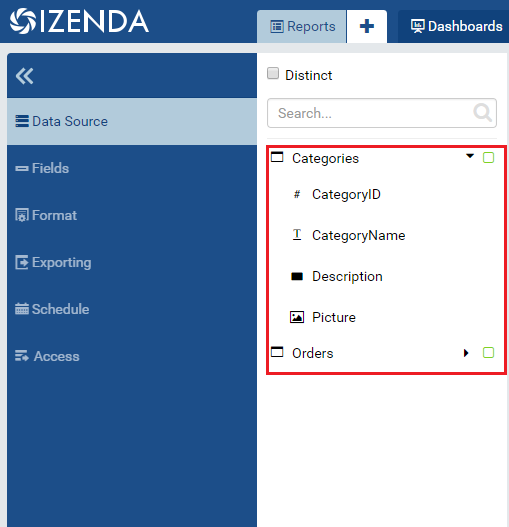


Table 4: Query Source and Query Source Field on Izenda Report Designer

## Query Source Parameter

In the case the Query Source is function or store procedure. The concept Query Source Parameter will be abstraction for parameter in function or procedure.

Below table list concerned properties of a Query Source Parameter in Data Source Adaptor component:

|  |  |  |
| --- | --- | --- |
| # | Property Name | Description |
| 1 | Name | Corresponding column name |
| 2 | DataType | The column data type in database. This is data type name, which is supported by database system. |
| 3 | IzendaDataType | The Izenda Data Type name, because Izenda supports multiple database system, it group database data type into some group to handle calculating manipulation, such as numeric, date time, currency…etc. |
| 4 | InputMode | Indicates whether parameter is input or not |
| 5 | Result | Indicates parameter is output of result after executed or not. |
| 6 | Position | The position of field in data source, this map to position number of column in table, view, result set of store procedure. |
| 7 | Category | The schema that the field is belong. |
| 8 | QuerySourceName | This is the name of container query source. |
| 9 | Value | Default value of function or procedure parameter |
| 10 | AllowDistinct | The property indicates that a data type whether must be unique or not. |

Table 5: Concerned property of a Query Source Parameter

In Izenda system, the class Izenda.BI.Framework.Models.QuerySourceParameter will present for this concept.

## SQL Query Source Type

Izenda separate Query Source into four types that are Table, View, Store Procedure and Function. They are corresponding defined as constant member of Izenda.BI.RDBMS.Constants. SQLQuerySourceType class.

|  |  |  |
| --- | --- | --- |
| # | Query Source Type Name | Description |
| 1 | Table | The table in database |
| 2 | View | The view in database |
| 3 | Store Procedure | The procedure in database |
| 4 | Function | The function in database |

Table 6: Predefine Query Source Type in Izenda System

The Connection Info on UI groups database schema structure into four group corresponding with four Query Source Types above:

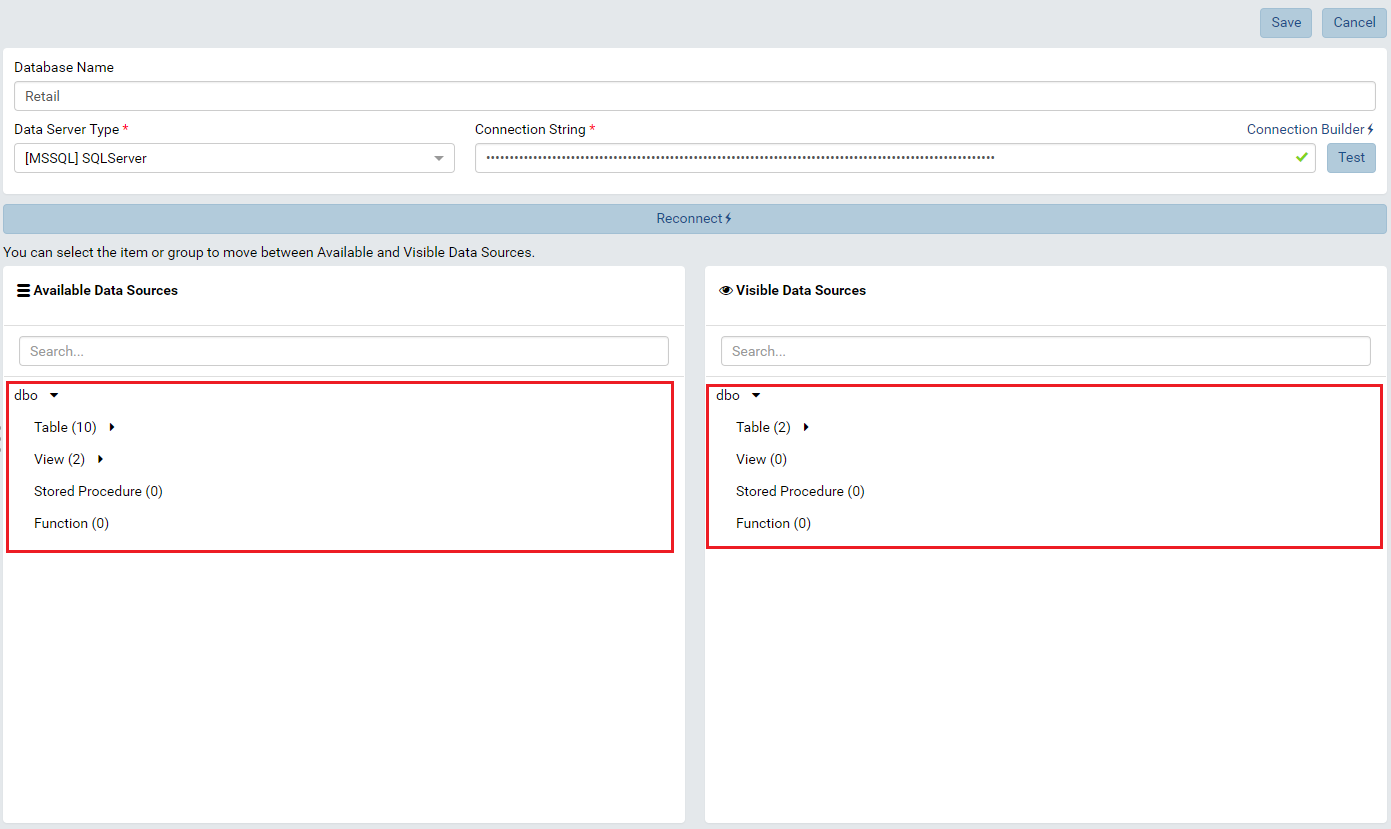


Figure 5: Corresponding Query Source Type on UI

## DB Source

The DB Source is concept present for entire database structure. It contains Query Source Category, Query Source, Query Source Field and Query Source Parameter. Like the database contains multiple schema, table, view, function and procedure.

The class Izenda.BI.Framework.Models.DBSource present for this concept in Izenda Framework.

## Relationship

Relationship object present for a relationship between tables or table and view. For each relationship in database, we have one relationship instance in Izenda system.

We will query database relationship then convert to this concept object then store into Izenda system database.

On an Adaptor component, we will concern to properties below:

|  |  |  |
| --- | --- | --- |
| # | Property Name | Description |
| 1 | JoinQuerySourceName | The combine of schema name and table name of join table in format [SchemaName.TableName]. The join table is the table contains foreign key in relationship. |
| 2 | ForeignQuerySourceName | The combine of schema name and table name of foreign table in format [SchemaName.TableName]. The foreign table is the table contains primary key in relationship. |
| 3 | JoinFieldName | The join column name, join column is foreign key column in relationship. |
| 4 | ForeignFieldName | The join column name, join column is primary key column in relationship. |

Table 7: Concerned properties of Relationship object from Adaptor component

For example, we have relationship between table Products and Categories in dbo schema. The Products table contain CategoryID column is foreign key to Categories table. The Categories table have ID column is private key. According that, we will have the relationship object with corresponding properties are:

* Join Query Source Name: dbo.Products
* Foreign Query Source Name: dbo.Categories
* Join Field Name: CategoryID
* Foreign Field Name: ID

The class Izenda.BI.Framework.Models.Relationship presents for this concept.

Image below show relationship on Data Model View:

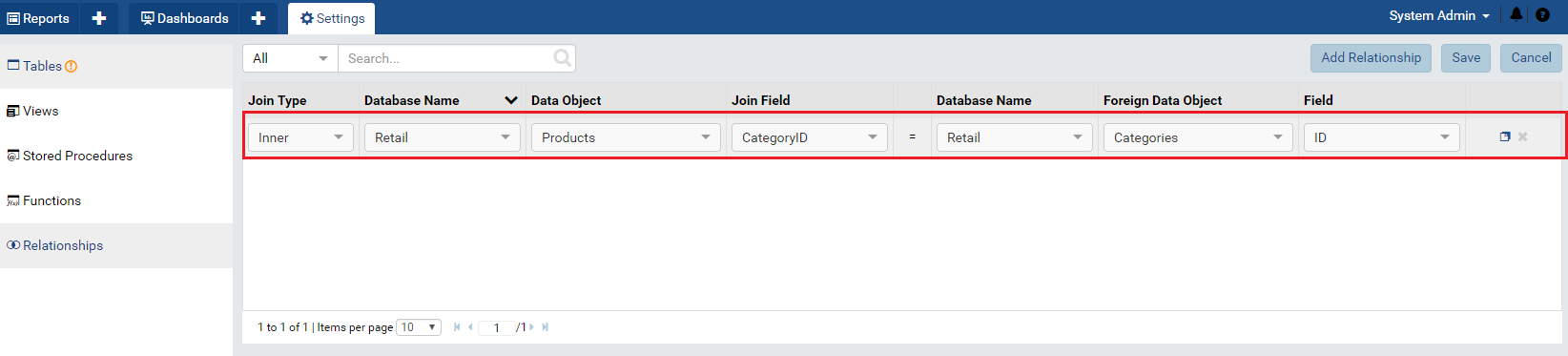


Figure 6 Relationship on Izenda UI

## The connection

The database connection will be stored into Izenda database system to manipulate reporting and calculation.

Each connection will be managed by an IConnection instance on specific Data Source Adaptor class.

## Mapping between RDBMS Concept and Izenda Data Source Adaptor Concept

According understanding of above concepts, looking to the table below of mapping between RDBMS and Izenda Data Source Provider Concept

|  |  |  |
| --- | --- | --- |
| **RDBMS** | **Izenda Data Source** | **Description** |
| Database | DB Source | Exclude relationship |
| Schema | Query Source Category |  |
| Table | Query Source | SQL Query Source Type = Table |
| View | Query Source | SQL Query Source Type = View |
| Function | Query Source | SQL Query Source Type = Function |
| Store Procedure | Query Source | SQL Query Source Type = Procedure |
| Field | Query Source Field |  |
| Parameter | Query Source Parameter |  |
| Relationship | Relationship |  |

Table 8: The mapping between Izenda Query Source concept and RDBMS concept

## Izenda Fusion and Query Tree

When end user building report on Izenda UI, they are creating report definition according to operations on set theory such as union, intersection, set different, symmetric different, Castersian product and power set. To understand how the Izenda Query Tree structure data operation, you have to have knowledge about query tree in Relation Database.

Izenda Business Layer will organize reporting definition to Query Tree Structure, which contains multiple Operator in tree structure with each node in tree presents for an operand. In Izenda Framework, the class QueryTreeNode abstract for an operand node in query tree.

The building process for query tree is defined in Izenda Fusion. In other word, Izenda Fusion will retrieve and process data from data source, which was stored in Izenda System Database when user added connection.

The concerning for who is developing an adaptor component is not how Izenda build Query Tree, it is how a query tree is structured. Izenda Business Layer will call to adaptor component to retrieve data through SQL query, following that the adaptor component has to generate corresponding SQL syntax for specific RDBMS. In interface IDataSourceAdaptor, there is method that contain query tree definition and fusion context to generate the query. That method is [Query(stirng, QueryTreeNode, FusionContextData)](#_Query(stirng,_QueryTreeNode,_Fusion), it inputs connection string to data source and query tree structure in instance of query tree node and the fusion context. That method first first entry point of an adaptor component.

If you develop adaptor based on based adaptor component Izenda.BI.DataAdaptor.RDBM, you will not actually process on the query tree. The based adaptor will process it for you. However, I you develop adaptor component from draft, fully understanding of query tree and fusion context to be required.

The logical query tree in Relational Database look like below:

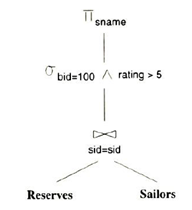


Figure 7: The logical query tree in Relation Database

In Izenda System, the query tree will look like below:

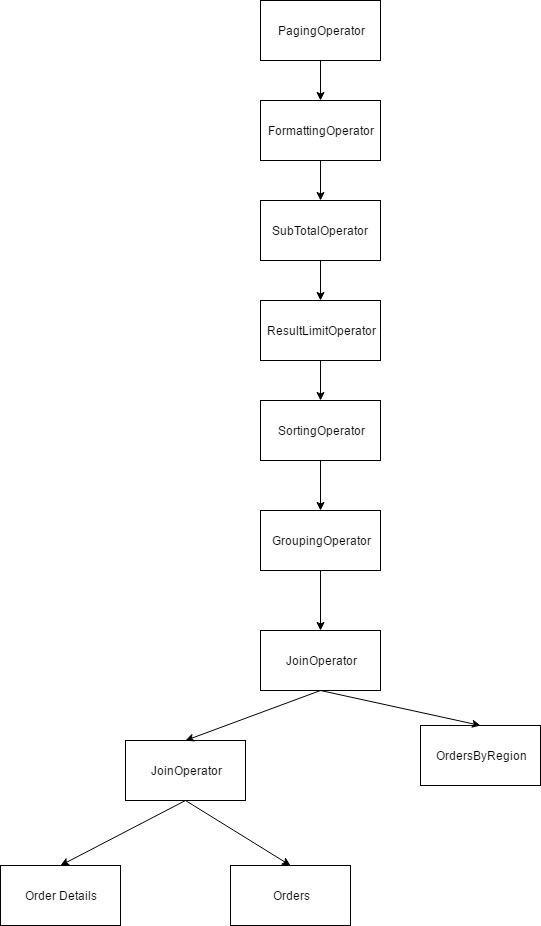


Figure 8: Example of Izenda Query Tree

The present of node type in the tree based on how the report definition was defined.

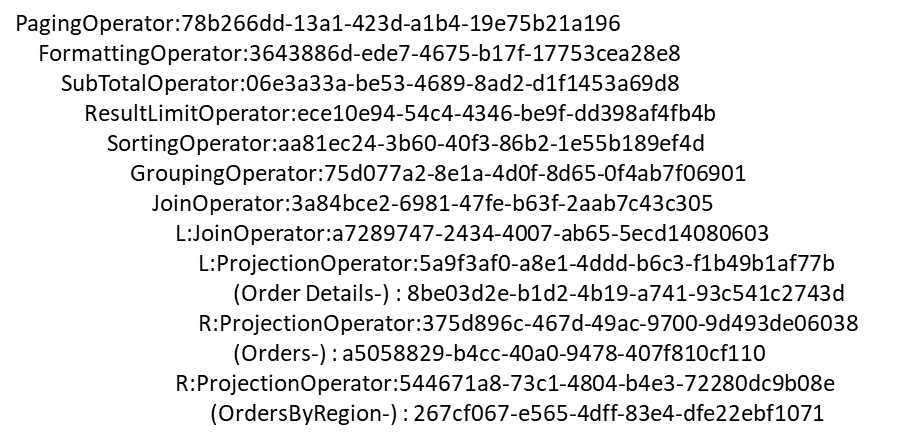
The container stack of query tree:

Figure 9: Example fo query tree stack

## Query Tree Node

|  |  |
| --- | --- |
| **Concept** | **Description** |
| Operand | This concept presents the operand operators in the queries. |
| Binary | This concept presents the binary operators in the queries like left and right operation. |
| Conditional | This concept presents the conditional operators in the queries like logic condition and filter condition. |
| ConvertNullToEmpty | This concept presents the convert operator from null data to empty as string in the queries. |
| Unary | This concept presents the unary operator in the queries at one time |
| Join | This concept presents the join operators between 2 or more objects in the queries. |
| Projection | This concept presents the projection operators in the queries. |
| Selection | This concept presents the selection operators in the queries. |
| Union | This concept presents the union operators between 2 or more queries together. |
| Cartesian | **This does not use in any driver for now.** |
| Grouping | This concept presents the group operators in the queries. This instance is to define which fields use to group, re-group or not. |
| Sorting | This concept presents the sorting operators in the queries. This instance is to define which fields use to sort, re-sort or not. |
| GrandTotal | This concept presents the grand total operator. This does not use. |
| SubTotal | This concept presents the sub total operator. |
| GroupPercentTotalField | **This does not use in any driver for now.** |
| ResultLimit | This concept presents the limit result from the returned data in the queries. Example: SELECT TOP X |
| SideTotal | This concept presents calculation for side total operator, indicate which field use to apply for side total operator. |
| PivotColumnLimit | This concept presents limit operators on the pivot column using in report. |
| Distinct | This concept presents the distinct operators in the queries,  indicate fields to distinct operator. |
| Paging | This concept presents the paging operators in the queries, contains information about page index, page size, whether last page or not. |
| Formatting | This concept presents the format information of fields font, size, style for field is applied(grand total field, subtotal field, side total field, side total grouping) |
| GroupPercentNormalField | This concept presents the group percentage operators on normal fields. |
| DefaultPercent | This concept presents the default percentage operators as default. |
| Running | This concept presents for running operators for the queries. Such as which field is pivot field, virtual field, related fields among report …. |
| GroupPercentSideTotal | This concept presents the group percent operators on side total field. |
| DatabaseFunctionToken | This concept presents token, alias for the functions, fields or any objects from database. |

Table 9: The list of operands are using Izenda fusion

## SQL Command Generators

### Operator Command Generator

|  |  |  |
| --- | --- | --- |
| **Concept** | **Operator Command Generator** | **Description** |
| Binary | BinaryOperatorCommandGenerator | This generates the query for binary operators, like left and right operator |
| Conditional | ConditionalOperator  \_CommandGenerator | This generates the query for conditional operators like logic or filter operator |
| ConvertNullToEmpty | ConvertNullToEmptyOperator  \_CommandGenerator | This generates the query for converting operator from null data to the empty as string. |
| Unary | UnaryOperatorCommandGenerator | This generates the query for the unary operator |
| Operand | OperandCommandGenerator | This generates the query for the operand operators |
| Join | JoinOperatorCommandGenerator | This generates the query for the join operators like left, right, cross join … |
| Projection | ProjectionOperator  \_CommandGenerator | This generates the query for projection operator, whether operator is the top projection or not |
| Selection | SelectionOperator  \_CommandGenerator | This generates the query for selection operator, like "SELECT \* FROM ({0}) X WHERE {1} |
|  | SelectFieldCommandGenerator | This generates the query for select fields in selection operator. |
| Grouping | GroupingOperator  \_CommandGenerator | This generates the query for group operators |
| Sorting | SortingOperator  \_CommandGenerator | This generates the query for sorting operators |
| SubTotal | SubTotalOperator  \_CommandGenerator | This generates the query for subtotal operator, whether there is sort node or not, has unary operator or not…. |
| ResultLimit | ResultLimitOperator  \_CommandGenerator | This generates the query for limit result from returned data like TOP {0} or Limit {0} |
| Distinct | DistinctOperator  \_CommandGenerator | This generates the query for distinct operator for all fields |
| DatabaseFunctionToken | DistinctToken  \_CommandGenerator | This generates the query for distinct operator with alias of fields. |
| Paging | PagingOperator  \_CommandGenerator | This generates the query for paging operator |

Table 10: Corresponding generator for each operator in Izenda Data Source Adaptor

### Filter Command Generator

These generators generate command, queries for filter operators.

|  |  |
| --- | --- |
| **Filter Command Generator** | **Description** |
| BlankFilterGenerator | This generates the query for blank condition, whether value is bank or not. |
| BooleanFilterGenerator | This generates the query for Boolean condition, whether value is true or false. |
| ComparisonFilterGenerator | This generates the query for comparison condition, whether value is less than, not less than, greater than, between or not between… |
| DateTimeFilterGenerator | This generates the query for date time condition, whether value is in period, not in period, equal a date or not… |
| EquivalenceFilterGenerator | This generates the query for equivalence condition, whether value is equal manual entry, equal selection, equal multiple values, equal with value in drop down list… |
| FieldComparisonFilterGenerator | This generates the query for field comparison condition, whether value is equal with input field or not, less than or greater than the input field. |
| FilterCommandGenerator | This is the base command generator for filter. |
| NotBlankFilterGenerator | This generates the query for not blank condition, whether value is not bank or blank. |
| NotNullFilterGenerator | This generates the query for null condition, whether value is null or not. |
| NullFilterGenerator | This generates the query for null condition, whether value is null or not. |
| StringFilterGenerator | This generates the query for string operators condition, whether value is end with, begin with, like or not like. |
| ThreeDotsFilterGenerator | This generates the query for default filter which will filter all data. |

Table 11: The list of filtering command generator

Note that all above filtering command generator is inherited form FilterGeneratorBase class, and that is identifier to indicate the class is filtering command generator.

### Calculated Command Generator

These generators generate command, queries for calculated operators.

|  |  |
| --- | --- |
| **Filter Command Generator** | **Description** |
| AddOperatorToken  \_CommandGenerator | This generates the command for add operator. |
| AggregateFunctionToken  \_CommandGenerator | This generates the command for aggregated function operators. |
| AndOperatorToken  \_CommandGenerator | This generates the command for and operator. string.Format("({0} {1} {2})", leftCommand, token.Expression, rightCommand); |
| AvgTokenCommandGenerator | This generates the command for average operator. |
| BetweenFormatToken  \_CommandGenerator | This generates the command for between operator. ({0} BETWEEN {1} AND {2}) |
| CaseWhenToken  \_CommandGenerator | This generates the command for case when operator. |
| CastTokenCommandGenerator | This generates the command for cast operator. cast({1} as {2}) |
| ConcatTokenCommandGenerator | This generates the command for concat operator. concat({0}) |
| ConvertTokenCommandGenerator | This generates the command for convert operator. convert({1}, {2}) |
| CountTokenCommandGenerator | This generates the command for count operator. |
| DatabaseFunctionToken  \_CommandGenerator | This generates the command to get function name, mapping function or full function name in database. |
| DateAddTokenCommandGenerator | This generates the command for dateadd operator. ("dateadd({1}, {2}, {3})", DatabaseFunction.DateAdd, GetDatePart(inputtedDatePart), interval, date) |
| DateDiffTokenCommandGenerator | This generates the command for datediff operator. ("datediff({1}, {2}, {3})", DatabaseFunction.DateDiff, GetDatePart(fullDatePart), fromDate, toDate) |
| DateFormatTokenCommandGenerator | This returns formats to apply formatting operators. |
| DateFunctionTokenCommandGenerator | This returns date parts which use in datetime operators, such as: year, yearabb, quarter, month… |
| DatePartTokenCommandGenerator | This returns datepart of datetime parameters. ("datepart({1}, {2})", DatabaseFunction.DatePart, GetDatePart(fulldatePart), date) |
| DivideOperatorToken  \_CommandGenerator | This generates the command for divide operator. |
| EqualOperatorToken  \_CommandGenerator | This generates the command for equal operator. |
| ExpressionCommandGenerator | This generates the command for expression such as: supported functions, row fields, parameters... |
| FieldTokenCommandGenerator | This returns field the command for divide operator. |
| FlowTokenCommandGenerator | This generates the command for flow operator. |
| FunctionTokenCommandGenerator | ??? |
| GetDateTokenCommandGenerator | This returns a getdate() function. |
| GreaterOperatorToken  \_CommandGenerator | This generates command for greater operator. |
| GreaterOrEqualOperatorToken  \_CommandGenerator | This generates the command for greater or equal operator. |
| IfTokenCommandGenerator | This generates the command for if the else statement. |
| IIfTokenCommandGenerator | This generates the command for case when {0} then {1} else {2} end statement. |
| IsNullTokenCommandGenerator | This generates the command to check value is null or not. IsNull({1}, {2}). |
| LenTokenCommandGenerator | This generates the command to return a length of input. len({1}). |
| LpadTokenCommandGenerator | This generate padding alignment from left side for data display like date time, numeric…etc. |
| MappingTokenCommandGenerator | ??? |
| MaxTokenCommandGenerator | This generates the command for max operator. |
| MinTokenCommandGenerator | This generates the command for min operator. |
| MultipleOperatorToken  \_CommandGenerator | This generates the command for multiple operator. |
| NotEqualOperatorToken  \_CommandGenerator | This generates the command for not equal operator. |
| OrOperatorTokenCommandGenerator | Generator OR operand in SQL command |
| RoundTokenCommandGenerator | This generates the command for round operator. ”Round({1},{2}),expression, length” |
| SmallerOperatorToken  \_CommandGenerator | This generates the command for smaller operator. |
| SmallerOrEqualOperatorToken  \_CommandGenerator | This generates the command for smaller or equal operator. |
| SubtractOperatorToken  \_CommandGenerator | This generates the command for subtract operator. |
| SumTokenCommandGenerator | This generates the command for sum operator. |
| SymbolOperatorToken  \_CommandGenerator | ??? |
| UserDefinedToken  \_CommandGenerator | This generates the command for the user defined function. ("{0}({1})", functionName, inputParams.ToString().TrimEnd(',')) |
| ValueTokenCommandGenerator | Generate token parameter identifier for a value |

Table 12: The list of expression command generator

## Izenda Raw Query

Izenda introduces a concept relates to the special SQL Syntax which is able to easily transform to standard SQL syntax of many RDBMSs. The purpose of creating based adaptor component is providing extensible component for developing the Data Source Adaptor for any RDBMS later. That based component is Izenda.BI.DataAdaptor.RDBMS.dll. That based component will generate Izenda Raw Query.

Currently, Izenda has already provided Data Source Adaptor for SQL Server, Oracle, MySQL, PostgreSQL and AzureSQL, all of them were developed based on Izenda.BI.DataAdaptor.RDBMS library.

*Note: If you plan to develop a data source adaptor in different architecture and do not inherits from based adaptor of Izenda, you can ignore this concept at all.*

There are three important syntax definition for Izenda Raw Query, which are naming identifier, customized function IZENDA\_CONCAT and DATETRUNCTE. Izenda generator will generate SQL statement contains that raw syntax and then we have to transform it to standard syntax that can be perform by specific RDBMS in Data Source Adaptor.

### Naming Identifier

Each RDBMS has its own specific naming identifier to escaping character set if it is not ANSI Code such as Unicode character or non-alphabet and non-numeric characters. According to SQL Standard 1999, it specifies double quote (“) to delimit identifiers but most of RDBMS does not support it as default.

For example, SQL Server and AzureSQL use open ([) and close bracket (]). MySQL uses back-tick quote (`). Oracle and PostgreSQL use double quote (“). Because of that, Izenda uses double open bracket ([[) and double closed bracket (]]) to specify naming identifier. That allows Izenda Raw Query is extensible on multiple RDBMSs and only requires simple converting logic to replace identifier in Izenda Raw Query by correct identifier of a RDBMS. Izenda Raw Query will become standard SQL and can be executable on specific RDBMS after transformation activities processed.

Izenda Raw Query with customized identifier will look like below:

SELECT [[CategoryID]] AS "categoryid\_15264036\_b36f\_",

[[CategoryName]] AS "categoryname\_32056250\_9bdf\_",

[[Description]] AS "description\_c9439929\_1cb9\_"

FROM [[dbo]].[[Categories]]

Normally, we will need a normalizer step to convert that naming identifier to specific one on concrete RDBMS. We will describe detail in another section later in this document.

For example, in case of MySQL, the query after process to replace naming identifier by back-tick (`) character will become:

SELECT `CategoryID` AS "categoryid\_15264036\_b36f\_",

`CategoryName` AS "categoryname\_32056250\_9bdf\_",

`Description` AS "description\_c9439929\_1cb9\_"

FROM `dbo`.`Categories`

### IZENDA\_CONCAT function

Izenda uses IZENDA\_CONCAT function to concatenate multiple value into one string value, this similar behavior will be different on each RDBMS then in order to process same logic across multiple database system we have to normally provide query logic in case by case for each one.

Example of IZENDA\_CONCAT in raw query:

SELECT IZENDA\_CONCAT([[FirstName]], ' ', [[LastName]]) as "FullName"

FROM [[dbo]].[[Customers]]

In case of MySQL, above raw query after process will become:

SELECT CONCAT(`FirstName`, ' ', `LastName`) as "FullName"

FROM `dbo`.`Customers`

It simply replaces IZENDA\_CONCAT by CONCAT function in MySQL, but for other RDBMS it will be more complex like Redshift database. We will come back to this concern later.

### DATETUNCATE function

To remove time value from date value, Izenda uses DATETRUNCATE function to expose extensibility chance to handle that logic across multiple RDBMS. It will be replaced by corresponding function of data database system.

Example using DATETRUNCATE in raw query:

SELECT DATETRUNCATE([[OrderDate]])

FROM [[dbo]].[[Order]]

# IDataSourceAdaptor Interface

The interface provides API to get data source information in a connection such as connection info as server address, database name and user credential, and the API for getting Data Source schema structure such as definition of table, view, store procedure and function.

It also provides the methods for reading result sets obtained by executing commands at a data source when executing single or multiple SQL query statement.

Izenda abstracts a Data Source by dividing into many units like query source, query source category, query source field, query source parameter…etc. Therefore, the implementation of this interface will provide the way Izenda splits up data source definition into its own concepts and store in Izenda system database for reporting and calculating.

According to the using purpose, we can group the methods in this interface into below groups:

* Crossing concern
* Get connection information
* Load data source schema definition
* Execute SQL commands and query data

## Crossing Concern Members

### LogManager Property

Get or set the instance of log manager to handle tracking information white running adapter logic. This log manager mostly set by Izenda application to use extensibility log of Izenda but you can replace it by your own custom logger if you want.

## Get connection information

### TestConnection(Guid, string)

ConnectionStatus TestConnection(Guid serverType, string connectionString);

The method to test connection string. It will return ConnectionStatus object, in case connection is OK the Status property is ConnectionDBStatus.Success (0 – zero value) otherwise it returns Status value is not equals 0.

Parameters:

* serverType – Guid: The ID of server type, which is provided by exported metadata in implementation class.
* connectionString - string: The connection string of data source.

### GetDatabaseServer(string)

string GetDatabaseServer(string connectionString);

Return database server name of connection string

Parameters:

* connectionString: The connection string to data source

### GetDatabaseName(string)

string GetDatabaseName(string connectionString);

Return the name of database in connection string.

Parameters:

* connectionString: The connection string to data source

### GetConnectionStringWithServerAndDatabaseName

string GetDatabaseName(string connectionString);

Return combined name of database server and database name of connection string.

Parameters:

* connectionString: The connection string to data source

## Load Data Source Schema Definition

### LoadDatabaseSchema(string)

string LoadDatabaseSchema(string connectionString);

Load schema definition of database in connection string, the implementation of this method has to provide ability to load all definition of table, view, function and store procedure.

The returned database schema in connection string will be presented in DBSource object with schema is stored as QuerySourceCategory and each category contains multiple QuerySource object presents for table, view, function or store procedure. The query source is also grouped into 4 groups type are Table, View, Function and Store Procedure which are defined in SQLQuerySourceType constant class.

See documentation of DBSource, QuerySourceCategory, QuerySource object for more detail about structure of database which is using in Izenda system.

Parameters:

* connectionString: The connection string to data source

### LoadFields(string, string, string, string, bool, List<QuerySourceParameter>, bool, int, ILog)

List<QuerySourceField> LoadFields(string connectionString, string type, string categoryName, string querySourceName, bool rollbackSP, List<QuerySourceParameter> parameters = null, bool ignoreError = true, int commandTimeout = 500, ILog log = null);

Load all query source fields of either table, view or store procedure of database in connection string. The query source field presents for column in table and view, and for column in result set of store procedure.

Parameters:

* connectionString: The connection string to the data source
* type: The SQL Query Source Type and the value can be Table, View or Store Procedure
* categoryName: The schema name of table, view or store procedure
* querySroucename: The either table name, view name or store procedure name.
* rollbackSP: Is using when loading fields for store procedure. If this value is true, it requests new transaction to rollback command to execute store procedure for getting all the fields of store procedure it requires executing procedure to receive the field list from result. Some procedure will change data after executed. Therefore, we have to rollback that change to make sure the data not corrupted. This parameter is only applied for loading field for store procedure.
* List<QuerySourceParameter>: The list of parameter for executing store procedure to get field set from result. This parameter is only applied when load field for store procedure.
* ignoreError: Indicate that system always continue to run although there is error when executing store procedure. This parameter is only applied when load field for store procedure.
* commandTimeout: The time out for executing store procedure. This parameter is only applied for loading field for store procedure.
* ILog: The Izenda logger instance to log tracking when loading field. You can use this log instance to log out your tracking information. The logging output will be available in Izenda log file.

### GetQuerySourceParameters(string)

List<QuerySourceParameter> GetQuerySourceParameters(string connectionString);

Get all parameters of Function and Store Procedure in database which current connection is pointing to. This method usually should query information from system table of database then getting parameter info.

Parameters:

* connectionString: The connection string to data source.

### GetQuerySourceParameters(List<QuerySourceParameter>, string, string, string, QuerySource)

List<QuerySourceParameter> GetQuerySourceParameters(List<QuerySourceParameter> parameters, string type, string categoryName, string querySourceName, QuerySource remoteQuerySource);

Use for loading parameters of function and Store Procedure in the case that connection string was saved in Izenda database. This method will find out the changed of function and store procedure in database and update parameter information, which will be stored into Izenda system database.

Parameters:

* List<QuerySourceParameter>: The list of existing query source parameter is stored in Izenda system database.
* type: The type query source, the value is either SQLQuerySourceType.Procedure or SQLQuerySourceType.Function.
* categoryName: The schema that function and store procedure is belong to.
* querySourceName: Is the function name or procedure name.
* remoteQuerySourceName: The query source which are stored in Izenda database system.

### LoadConnectionDetail(Connection, bool, string, ILog)

void LoadConnectionDetail(Connection connection, bool rollbackSP, string querySourceType = "", ILog log = null);

Load all detail of database in a connection includes table, view, function and store procedure as well as relationship in database. All information about schema, table and view, function, store procedure, and relationship will be updated back to Connection parameter object.

Parameters:

* connection: Is input connection, after completely to load connection detail all detailed information about connection will be store in this parameter object.
* rollbackSP: Is using when loading fields for store procedure. If this value is true, it requests new transaction to rollback command to execute store procedure for getting all the fields of store procedure it requires executing procedure to receive the field list from result. Some procedure will change data after executed. Therefore, we have to rollback that change to make sure the data not corrupted. This parameter is only applied for loading field for store procedure.
* querySourceType: The SQL Query Source Type to identify what query source type is loaded into connection detail. If this value is empty, system will load detail for all of query source type Table, View, Function and Store Procedure.
* log: The Izenda ILog instance to write tracking log into izenda log file.

### GetExtendedProperties(string)

dynamic GetExtendedProperties(string extendedProperty);

In some RDBMS, we are able to add more property into database object allow to customize the information, storing the data within the database itself. When we need to retrieve the information, you simply query it, then this method is used to query extended property value if any.

Parameters:

* extededProperty: The extended property name.

### GetSupportedDataSourceType(string)

List<QuerySourceType> GetSupportedDataSourceType();

Get all supported data source type of database server type in connection string. The value maybe is Table, View, Function and Store Procedure, but in some RDBMS like PostgreSQL it will not support Store Procedure, so you should return set of Table, View and Function.

The result is list of QuerySourceType object with key and value is SQLQuerySourceType value (Table, View, Function and Procedure).

### GetBaseDataTypes(string)

List<DatabaseDataType> GetBaseDataTypes();

Get all supported datatype of database server engine of current data source adapter.

The result is list of DatabaseDataType object with database type name, mapped to .NET type name and Izenda supported type name to handle data formatting, filtering on report.

### LoadCustomQuerySourceFields(string, string)

List<QuerySourceField> LoadCustomQuerySourceFields(string connectionString, string customQueryDefinition);

There is situation that Izenda system wants to load query source field from specific query statement, so this method will provide list of QuerySourceFiled of any custom query by open the connection and execute it then read query source field from data reader result.

Parameters:

* connectionString: The connection string to data source.
* customerQueryDefinition: The custom query statement for getting query source from result after executed.

## Execute SQL commands and query data

### Query(string, string, object, int)

IEnumerable<T> Query<T>(string connectionString, string query, object param = null, int queryTimeout = 60);

The method queries single SQL statement for getting data from data source. The implementation of this method should open connection then execute query with parameter list in method parameter, and then deserialize dataset to IEnumerable<T> and return as result set.

Note that most query from Izenda system passes to this method is “raw” query; it requires transformation into specific SQL syntax that your custom RDBMS Data Source supports. There is the guideline to implement normalizer activity to transform “raw” query to specific SQL syntax, so prefer to section ABC to get the idea of how to normalize and transform SQL query. For example, Izenda always generates string concatenation as IZEDAN\_CONCAT function, actually there is no RDBMS has that function, so the query statement transformation step will have an activity to convert IZENDA\_CONCAT to supported function by that RDBMS, for an instance if RDBMS is MySQL it will replace IZENDA\_CONCAT by CONCAT method because MySQL supports CONCAT function naturally.

Parameters:

* connectionString: The connection string to data source.
* query: The query statement, the query in this parameter always is single statement. Some RDBMS support multiple query statements separate by semicolon character (;).
* param: The object present for instance of IDictionary<string, object> as list of parameter will be used in SQL query statement.
* queryTimeout: The query timeout when executing query.

### Query(string, string, string, string, string)

IEnumerable<T> Query<T>(string connectionString, string categoryName, string querySource, string queurySourceField, string sortOrder)

The method to query data result from specific query source (specified table, view or procedure) with result column set are defined by querySourceField parameter. The implementation of this method will open new connection then query with select and order by columns in querySourceField parameter.

Parameters:

* connectionString: The connection string to data source.
* categoryName: The schema name of query source to query data.
* querySource: The name of table, view or procedure.
* querySourceField: The column name will be selected in SELECT clause of query statement.
* sortOrder: The sort order in ORDER BY clause in query, it should be either SortType.ASC or SortType.DES.

### Query(stirng, QueryTreeNode, FusionContextData)

List<IDictionary<string, object>> Query(string connectionString, QueryTreeNode operand, FusionContextData context);

On Izenda Report Designer UI, end-user is able to combine multiple reporting conditions on UI to create expected report. Izenda will abstract that combination into operand operator and fusion context. By that way SQL query for each operand and fusion context will be generated and execute for reporting data in each combination scenario.

Parameters:

* connectionString: The connection string to data source.
* operand: Is one of supported operand by Izenda system (see concept section).
* context: The query and calculation context (Fusion Context) was built from combination scenario on report UI.

### CanQuery(QueryTreeNode, FusionContextData, string)

DatabaseQueryable CanQuery(QueryTreeNode operand, FusionContextData context, string connectionString);

The method to check precondition before execute query for specific operand, if this method returns DatabaseQueryable.None, the query method for that operand will be ignored. Because some of operation maybe is not supported by your RDBMS. In that case, you should check the availability of that operand. For example, some RDBMS supports few operands in newer version but your system is running on an older version so you can reject query for that operand by return DatabaseQueryable.Node in this method.

Parameters:

* operand: Is one of supported operand by Izenda system (see concept section).
* context: The fusion context was built from combination scenario on report designer logic from UI.
* connectionString: The connection string to data source.

### Execute(string, string, List<dynamic>)

int Execute(string connectionString, string command, List<dynamic> objectParams = null);

Normally, Izenda is only getting data from your data source and display on report but in some scenario, it wants to modify your data source like add and delete temporary table for tracking something. That are reason this method is persist here. In this method, you will handle normalizer step to transform Izenda Raw Query to supported syntax on your RDBMS then open new connection and execute input command.

The returned number indicates the number of rows effected when executing command.

Parameters:

* connectionString: The connection string to your data source.
* command: The SQL command will be executed. Mostly this command will modify something in your data source.
* objectParams: The parameters of command to be executed.

### GetFirstValueInFilteredValue(string, string)

object GetFirstValueInFilteredValue(string connectionString, string filteredValue);

The method returns first value in filtered value, Izenda builds filtered value as json structure contains information about filtering query with data source name, field name (ex: table and field name) and in this method Data Source Adapter will provide the first value of filtering query. Because each RDBMS has different syntax for select limited number of record, this method is place to indicate how your Data Source Adapter select limited rows by specific SQL syntax that it is supported.

For instance, SQL Server use SELECT TOP to query top rows while MySQL use LIMIT keyword for limitation result range.

Parameters:

* connectionString: The connection string to data source.
* filteredValue: The filtering context was built by Izenda system, this parameter value can be deserialized to FilteredValueExtendedPropery class.

### QueryMultiple(string, string, object, int, Action<SqlMapper.GridReader>)

IEnumerable<T> QueryMultiple<T>(string connectionString, string query, object param = null, int queryTimeout = 60, Action<SqlMapper.GridReader> action = null)

This method supports query multiple query statements, which was split by semicolon character (;) in input query script. Currently Izenda usually uses this method to query two SQL query statement, first one for selecting report result and second one for query data paging information. The second one query result is passed as GridReader object of Dapper ORM framework and Izenda will use that reader object to get out paging information.

Note that if your RDBMS is not supported multiple query, you no need to implement this method. However, if it is, you should implement this method to improve query performance.

Inside implementation for this, you have to handle query normalizer step to transform query to supported syntax on your RDBMS then execute the query and return result.

Izenda is using Dapper ORM framework to work with database then this method absolutely stuck with usage of GridReader object which is defined in that framework. In guideline to create custom Data Source Adapter, the implementation by using Dapper will be described in detail.

Parameters:

* connectionString: The connection string to data source.
* query: The SQL query statement, usually is multiple query statements
* param: The object as IDictionary<stirng, object> presents for list of parameter is using in query.
* queryTimeout: The query timeout, default is 60 second.
* action: Callback action to pass back GridReader object to Izenda system, it will be used to collect paging information of result set.

# Specific Configuration for RDBMS in Adaptor Component

## Data Types

Because each RDBMS has its own data types definition, to works across multiple database system Izenda needs mapping database data type to supported data type that Izenda uses on data source reporting. That ensures Izenda has consistency operations when handle reporting logic such data formatting, data filtering and data calculation on any data source type without depend on specific database system.

Generally, Izenda group supported data type into group of Date Time, Numeric, Money, Boolean, Large Object (LOB), Text and few others data type. The data types group is defined in constant class Izenda.BI.Framework. Constants.IzendaDataType.

We have to provide data type mapping configuration for concreate RDBMS in implementation of Data Source Adaptor. To make it easy and consistency, Izenda provide class DatabaseSupportDataType to indicate data type mapping, in your custom adaptor you will define a class inherits DatabaseSupportDataType and add the mapping configure by AddDatabaseDataType method.

Example for Redshift database, its numeric types include integers, decimals, and floating-point numbers then all of that data types will be grouped into Izenda numeric group as below:

AddDatabaseDataType("numeric", IzendaDataType.NumericType, true, "System.Decimal", true /\*default mapping\*/);

// integer

AddDatabaseDataType("smallint", IzendaDataType.NumericType, true, "System.Int16");

AddDatabaseDataType("int2", IzendaDataType.NumericType, true, "System.Int16");

AddDatabaseDataType("integer", IzendaDataType.NumericType, true, "System.Int32");

AddDatabaseDataType("int", IzendaDataType.NumericType, true, "System.Int32");

AddDatabaseDataType("int4", IzendaDataType.NumericType, true, "System.Int32");

AddDatabaseDataType("int8", IzendaDataType.NumericType, true, "System.Int64");

AddDatabaseDataType("bigint", IzendaDataType.NumericType, true, "System.Int64");

// floating point number

AddDatabaseDataType("real", IzendaDataType.NumericType, true, "System.Single");

AddDatabaseDataType("float4", IzendaDataType.NumericType, true, "System.Single");

AddDatabaseDataType("float", IzendaDataType.NumericType, true, "System.Double");

AddDatabaseDataType("float8", IzendaDataType.NumericType, true, "System.Double");

AddDatabaseDataType("double precision", IzendaDataType.NumericType, true, "System.Double");

AddDatabaseDataType("decimal", IzendaDataType.NumericType, true, "System.Decimal");

In the guideline to create a custom Data Source Adapter, you will see how to implement this mapping configuration in detail.

That indicates when your reporting Data Source contains Query Source data type (column) is one of type in above grouping configuration (ex: int2 or float). Izenda will treat that query source as numeric and provide all features relate to numeric reporting for it. For example, number formatting for numeric display, number comparing for numeric filter operator like less than, greater than, equal…etc.

### DatabaseSupportDataType class

You will use this class as based class to configure data types mapping for your RDBMS, this section describes functionality of methods inside it.

This class manages data types mapping between your database system data types and Izenda supported data types, it stores your mapping configuration then provide to Izenda system for handling data manipulation on reporting data source.

You also override its method to provide better behavior to access data type mapping configuration.

#### AddDatabaseDateType(string, string, bool, string, bool, string)

protected void AddDatabaseDataType(string dataType, string izendaDataType, bool allowDistinct, string netFrameworkType, bool defaultMapping = false, string defaultDatabaseTypeLength = "")

This method will be used in your derived class to initialize list of supported data types that your RDBMS data types can be mapped to Izenda data type group.

Parameters:

dataType: The data type name in your RDBMS.

izendaDataType: The data type group that Izenda supports.

allowDistinct: Whether allow to duplicate data type in mapping configuration or not. If this parameter is true, you cannot add a mapping any more it means we only one time to register for that data type, if you call AddDatabaseDataType with that data type again Izenda will throw exception.

netFrameworkType: The corresponding type in .NET framework.

defaultMapping: Indicate Izenda system use this type for default type in data type group.

defaultDatabaseTypeLenght: The value to set default max length for a data type. For example, in Redshift the TEXT type will have 260 bytes length.

#### GetIzendaDataType(string)

public virtual string GetIzendaDataType(string fieldDataType)

Return Izenda data type group name by database system data type name.

Parameters:

fieldDataType: The database system data type name.

#### GetNetFrameworkType(string)

public virtual string GetNetFrameworkType(string fieldDataType)

Return .net framework data type corresponding with database system type.

Parameters:

fieldDataType: The database system data type name.

#### GetDatabaseDataType(string)

public virtual string GetDatabaseDataType(string izendaType)

Return first database data type name corresponding with Izenda data type group name.

Parameters:

izendaType: The Izenda data type group name.

#### GetDatabaseDataTypeMapping(string)

public DatabaseDataType GetDatabaseDataTypeMapping(string izendaType)

Return first corresponding mapping object by Izenda data type group name.

Parameters:

izendaType: The name of Izenda data type group name.

#### GetBaseDataTypes()

public virtual List<DatabaseDataType> GetBaseDataTypes()

Get all mapping object of current database system.

#### IsSupportDataType(string)

public virtual bool IsSupportDataType(string fieldDataType)

Check whether database data type is supported or not.

Parameters:

* fieldDataType: The name of database type name.

## Database Functions

The table #2 below lists the collection of aggregate functions that Izenda uses. If your RDBMS does not have a function in the list, you have to provide SQL query with similar function. Normally, if you do not have different architecture approach to implement adaptor component, you just only inherit the corresponding generator to re-use corresponding functions in your database system or write new custom function syntax in override generator to make Izenda works properly on the similar query logic.

All using functions are declared in DatabaseFunction class (in component Izenda.BI.DataAdaptor, namespace Constants), you should override some functions to indicate correct function in your database system if it is not same in below table.

|  |  |  |  |
| --- | --- | --- | --- |
| .No | Property Name | Func Name | Description |
| 1 | DateAdd | dateadd | Increments a date or time stamp value by a specified interval |
| 2 | DateDiff | datediff | Returns the difference between the data parts of two date or time expresses |
| 3 | DatePart | datepart | Extract date part values from an expression |
| 4 | Convert | convert | Converts an expression of one data type to another |
| 5 | Cast | cast | Converts an expression of one data type to another |
| 6 | Len | len | Returns the length of the specific string |
| 7 | GetDate | getdate | Return the current date and time |
| 8 | IIF | iif | Return one of two values, depending on whether the Boolean expression evaluates to true or false |
| 9 | IsNull | isnull | Returns an alternative value when an expression is NULL |
| 10 | Round | round | Return a number rounded to a certain number or decimal places |
| 12 | Distinct | distinct | Return a one-column table that contains the distinct values from the specified column, that means the duplicated values are removed and only unique values are returned |
| 13 | Case | case | The Case clause statement, it is using for building if and case-when token command |
| 14 | When | when | The When clause statement, it is using for building if and case-when token command |
| 15 | Then | then | The Then clause statement, it is using for building if and case-when token command |
| 16 | Else | else | The Else clause statement, it is using for building if and case-when token command |
| 17 | End | end | The End clause statement, it is using for building if token command |

Table 13: The functions are using in Izenda

## Database Functions Format

Each RDBMS has its own syntax to call function and syntax to check conditions, that is reason Izenda has configuration extension to indicate format string for using function in a specific database system.

As the convention to build Izenda raw query, the format for database column field will be inside notation of [[ ]] and the value will be in { }. The format string follows convention of string format in .Net, the place holder is counted from zero base index.

Table below lists all functions format that are using in Izenda system, if your RDBMS have different syntax for similar function, you have to override them by declare a derived class inherits from DatabaseFunctionFormat in namespace Constants of component Izenda.BI.DataAdaptor. RDBMS, then override corresponding property or method to provide new custom syntax.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | Format Name | Value Format | Field Format | Description |
| 1 | DateDiffMonthFormat | DATEDIFF(month, {0}, {1}) |  | Return the different of month part between two date values |
| 2 | DateDiffYearFormat | DATEDIFF(year, {0}, {1}) |  | Return the different of year part between two date values |
| 3 | DateDiffDayFormat | DATEDIFF(day, {0}, {1}) |  | Return the different of day part between two date values |
| 4 | BetweenFormat | {0} BETWEEN {1} AND {2} | [[{0}]] BETWEEN {1} AND {2} | Format for between clause |
| 5 | RightOpenIntervalFormat | {0} >= {1} AND {0} < {2} | [[{0}]] >= {1} AND [[{0}]] < {2} | Check value in a range |
| 6 | NotBetweenFormat | {0} NOT BETWEEN {1} AND {2} | [[{0}]] NOT BETWEEN {1} AND {2} | Format for not between clause |
| 7 | CastToDateFormat | CAST({0} AS DATE) | CAST([[{0}]] AS DATE) | Format to cast value or field to DATE data type |
| 8 | CastToTimeFormat | CAST({0} AS TIME(0)) | CAST([[{0}]] AS TIME(0)) | Format to cast value or field to TIME data type |
| 9 | CheckBlankFormat | 1 = 0 |  | Format to check value whether is blank or not |
| 10 | CheckBlankFormatText |  | DATALENGTH([[{0}]]) = 0 | Format to check if the text field is blank or not |
| 11 | CheckNotBlankFormat |  | ([[{0}]] IS NULL OR DATALENGTH([[{0}]]) <> 0) | Format to check a field whether is not blank or not |
| 12 | CheckNotBlankFormatText |  | ([[{0}]] IS NULL OR DATALENGTH([[{0}]]) <> 0) | Format to check a text field whether is not blank or not |
| 13 | CheckNullFormat |  | [[{0}]] IS NULL | Format to check if a field is null |
| 14 | CheckNullFormatText |  | [[{0}]] IS NULL | Format to check if a text field is null |
| 15 | CheckNotNullFormat |  | [[{0}]] IS NOT NULL | Format to check if a field is not null |
| 16 | CheckNotNullFormatText |  | [[{0}]] IS NOT NULL | Format to check if a text field is not null |
| 17 | CheckTrue |  | [[{0}]] = 1 | Format to check is a field is true (Boolean value) |
| 18 | CheckFalse |  | [[{0}]] != 1 | Format to check is a field is false (Boolean value) |

Table 14: The function formats are using in Izenda

## Database Constants

The other configuration for specific database system is identifies the datepart and timepart names and abbreviations that are accepted as arguments in most of date time functions. The table below list all datepart identifiers are configured as default in Izenda system, it was declaring as property DateParts in class Izenda.BI.DataAdaptor.RDBMS.Constants.DatabaseConstants as a dictionary has key is identifier name and value is actually name of datepart or timepart in database system.

|  |  |  |  |
| --- | --- | --- | --- |
| # | Identifier Name | Default Value | Description |
| 1 | DatePart.Year | Year | Year part in date time value |
| 2 | DatePart.Quarter | Quarter | The Quarter part in date time value |
| 3 | DatePart.Month | Month | The month part in date time value |
| 4 | DatePart.DayOfYear | DayOfYear | The day in year of date time value |
| 5 | DatePart.Date | Date | The entire date part in date time value |
| 6 | DatePart.Day | Day | The day in month part in date time value |
| 7 | DatePart.Week | Week | The week in year part in date time value |
| 8 | DatePart.WeekDay | WeekDay | The day in week part in date time value |
| 9 | DatePart.Hour | Hour | Hour part in date time value |
| 10 | DatePart.Minute | Minute | Minute part in date time value |
| 11 | DatePart.Second | Second | Second part in date time value |
| 12 | DatePart.Millisecond | Millisecond | Millisecond part in date time value |

Table 15: The list of datepart and timepart identifier in DatabaseConstants.DateParts property

Note that the Identifier Name on table above is defined in constant class Izenda.BI.Framework.Constants.DatePart

In based DatabaseConstanst class contains configuration to converting a date time string value to date value, it is DateFucntions property – a dictionary type. By default it only have one configure with the key is DatePart.Date and value is format CONVERT(date, {0}).

When developing new data adaptor, if targeted database system has different datepart and timepart identifier name, you have to override class DatabaseConstants class and provide new identifier name for each different one.

## Select Field Command Format

# Query Normalizer

Izenda RDBMS Adaptor component generates query in raw format with some concrete convention syntax, normally that convention is not supported by RDMBS so to make the raw query can be executable on a specific database system it must be transformed to standard syntax, which is able to run completely that RDBMS. Ideally, we use query normalizer activity to implement converting logic to transform Izenda raw query to standard query of a RDMBS.

Prefer to Izenda Raw Query concept section to get more detail about Izenda Raw Query.

Izenda uses sequence workflow with multiple activities to execute transformation from raw query to standard syntax on RDBMS. Each sequence workflow is presented by instance of SequenceWorkflow<TActivity, TContext>, and an activity (TActivity) is implementation of IActivity<TContext> and contains Order property to indicates priority of activity, Execute method contains query converting logic.

Izenda system will automatically load all TActivity inherits interface IActivity<TContext> from library has name is starting with “Izenda” in application domain directory and sort activities ascending by Order value. When Execute method of sequence workflow is called, it will execute one by one activity.

Detail for definition and usage of query normalizer activity and sequence workflow will be described detail in other section in this guideline.

According to Izenda Raw Query concept, you have to provide below query normalizer activities:

* Escape column, table, schema name
* Replace IZENDA\_CONCAT
* Replace DATETRUNCATE

Except above required activities, you can freely add more activity to process query string to modify Izenda Raw Query to handle your custom query logic. Query normalizer activity also is extension point to custom query syntax on your database system.

# Preparation for New Data Source Adaptor

This guideline shows how to create a data source adaptor and its dependency component from scratch and entire step by step will focus to create adaptor for Redshift database.

All existing RDBMS of Izenda system are developed based on component Izenda.BI.SystemRepository.RDBMS.dll, so this new adaptor also based on that component and follows architecture that Izenda is using to develop adaptor for RDBMS.

As beginning, create a folder named Redshift contain sub folder libs and an empty solution file RedshiftDataAdaptor.sln as below:

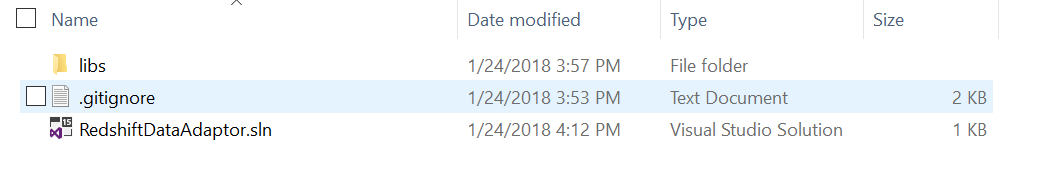


Figure 16: Initial project folder

Inside libs folder, create two sub-folders Dapper and Izenda to contain dependency libraries.

## Download dependency components

Download Izenda back-end package from <https://downloads.izenda.com/latest/API.zip> and extract the zip file to temp location on your machine (API folder).

From API folder that you have downloaded and extracted, browse to bin sub-folder and copy Dapper.dll to ~\Redshift\libs\Dapper (target path you have created above).

*Note that Dapper.dll above was modified for Izenda system only, other Dapper library from other source (ex: nugget package) is not supported.*

Copy below Izenda dependency libraries from API/bin to ~\Redshift\libs\Izenda:

* Izenda.BI.DataAdaptor.RDBMS.dll
* Izenda.BI.DataSourceAdapter.dll
* Izenda.BI.Framework.dll
* Izenda.BI.Logging.dll
* Izenda.BI.QueryNormalizer.Utility.dll
* Izenda.BI.RDBMS.dll
* Izenda.BI.Resource.dll
* Izenda.BI.Utility.dll

Following suggestion architecture for developing new Redshift data source adaptor, you will develop 2 components:

* Izenda.BI.QueryNormalizer.Redshift: The component provides logic to transform Izenda Raw Query to standard query for Redshift database system. It will use sequence workflow and activity pattern that are provided in library Izenda.BI.Framework.dll.
* Izenda.BI.DataAdaptor.RDBMS.Redshift: The adaptor component, contain all logic to make sure Izenda system works properly with Redshift database.

# Create Query Normalizer Component

This section describes how to create Query Normalizer Component for Redshift database, but for other database system, you can follow the idea to analysis and step by step to implement corresponding component.

## General Purpose

This section shows how to create and implement a Query Normalizer component, which will be used for Redshift data source adaptor. This component is developed based on sequence workflow and activity step for converting Izenda Raw Query to standard query of Redshift. Mostly, it contains definition of transformation activities, which are required to normalize query for Redshift database includes:

* RedshiftEscapseTableNameActivity: This activity converts column, table and schema name escaping notation of Izenda system ( [[ and ]] ) to notation is supported by Redshift.
* RedshiftQueryNormalizerActivity: The base class defines a workflow activity. It will be inherited from IActivity<QueryNormalizerConctex>. Other activity classes will inherit this class and will be loaded automatically by sequence workflow, which is provided in component Izenda.BI.Framework.dll.
* RedshiftReplaceConcatFunctionActivity: The step replaces IZENDA\_CONCAT by corresponding concatenation logic in Redshift.
* RedshiftReplaceDateTruncateFunctionActivity: This will replace DATETRUNCATE by corresponding truncate method in Redshift.
* Other activity if any.

## Analysis to implement Query Normalizer Activity

Izenda normally requires minimum 3 query normalizer activities to convert Izenda Raw Query to standard query for target database system. In case of Redshift database, we also have to provide suitable query string processor to adapt that requirement.

* For column, table and schema name escaping: The activity execution will replace notation [[ and ]] in Izenda Raw Query by double quote (“). Basically, Redshift supports to use double quote (“) to handle naming identifier, it followed SQL:1999 standard.
* For IZENDA\_CONATE replacement: Although Redshift support CONCAT function but it only supports two parameters in case Izenda system requires more, that is reason we must implement new logic to process concatenation multiple values, replacement for Izenda Raw function IZENDA\_CONCAT.
* For DATETRUNCATE replacement: Because Redshift supports DATE\_TRUNC function so it will be used to replace Izenda Raw function DATETRUNCATE.

## Create Izenda.BI.QueryNormalizer.Redshift Library

Izenda back-end run on .NET Framework 4.0 so this library must be targeted to .Net Framework 4.0.

As the naming convention is applied in Izenda system, this library also be named beginning with Izenda.BI, it is required to reserve for Sequence Workflow look up query normalizer activities automatically in library from Izenda application domain directory (usually is putted in API\bin folder).

Open RedshiftDataAdaptor.sln in Visual Studio IDE, then operate to add new project has target framework is 4.0 and project template is Class Library (.Net Framework):

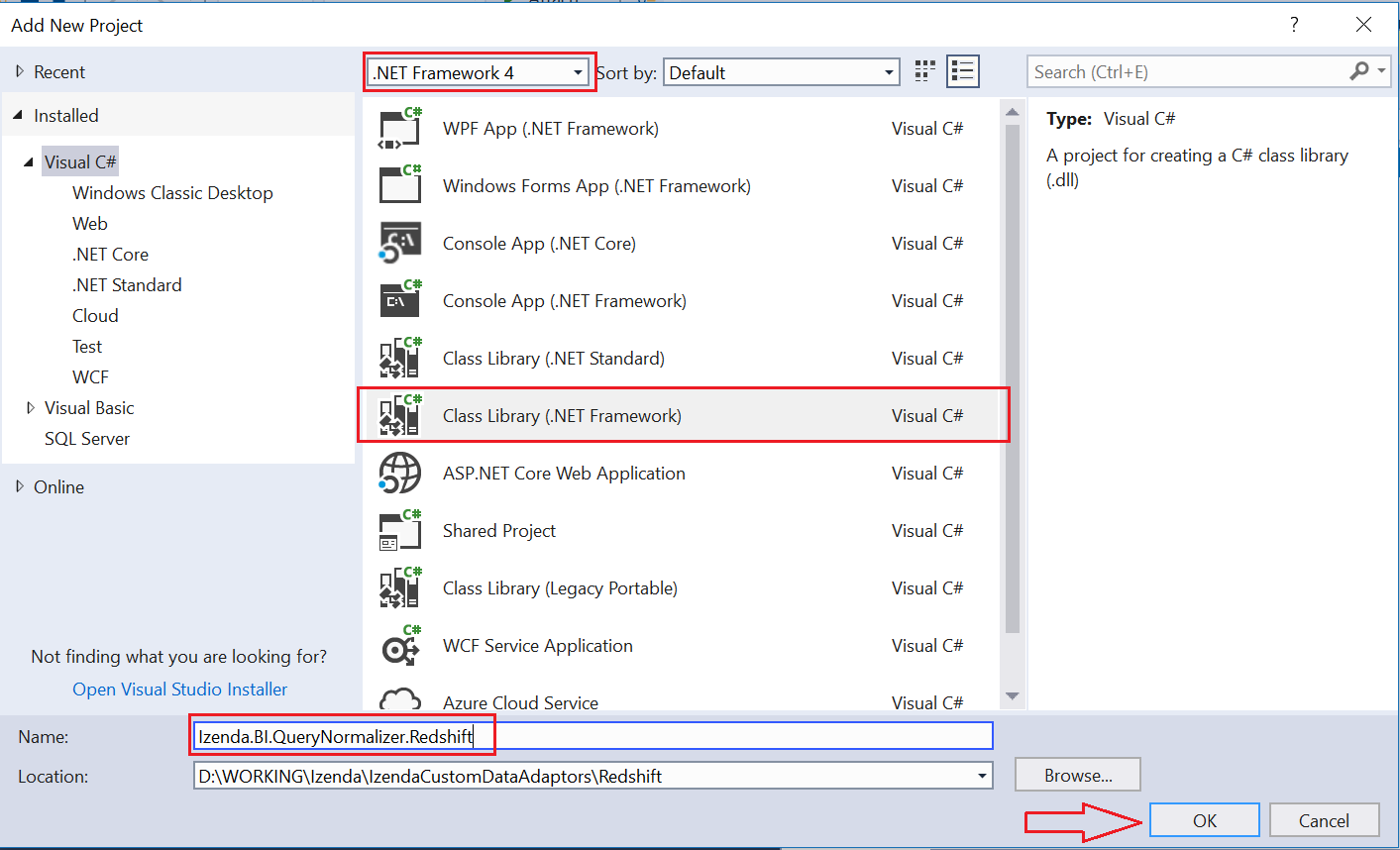


Figure 17: Create new Query Normalizer Library for Redshift

Click OK to create project.

After created project completely, add project dependency to Izenda.BI.Framework.dll and Izenda.BI.QueryNormalizer.Utility.dll from folder ~Redshift\libs\Izenda:

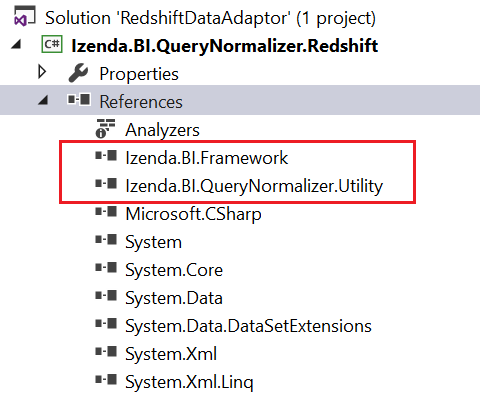


Figure 18: Add Izenda dependency libraries into Redshift Query Normalizer project

### Create abstracted class RedshiftQueryNormalizerActivity

This is abstract class presents for an activity. It inherits from IActivity<QueryNormalizerContext>, and will be used as parent class for every activity of Redshift Query Normalizer.

Add RedshiftQueryNormalizerActivity.cs into project then declare a class as below:

### Add class RedshiftEscapseTableNameActivity

using Izenda.BI.Framework.Components.SequenceWorkflows;

using Izenda.BI.Framework.Models.Contexts;

namespace Izenda.BI.QueryNormalizer.Redshift

{

public abstract class RedshiftQueryNormalizerActivity

: IActivity<QueryNormalizerContext>

{

public abstract int Order { get; }

public abstract void Execute(QueryNormalizerContext context);

}

}

Implementation of query normalizer activity that replaces Izenda Raw identifier for column, table and schema name as [[ and ]] by double quote (“) notation.

Add RedshiftEscapseTableNameActivity.cs file as below:

using Izenda.BI.Framework.Models.Contexts;

namespace Izenda.BI.QueryNormalizer.Redshift

{

public class RedshiftEscapseTableNameActivity : RedshiftQueryNormalizerActivity

{

/// <summary>

/// The activity order

/// </summary>

public override int Order

{

get

{

return 20;

}

}

/// <summary>

/// Execute the activity

/// </summary>

/// <param name="context">The context</param>

public override void Execute(QueryNormalizerContext context)

{

var sql = context.Query;

sql = sql.Replace("[[", @"""").Replace("]]", @"""");

context.Query = sql;

}

}

}

The Order property indicates priority of activity that will be processed by Sequence Workflow, the activity has smaller number will be processed first, and otherwise bigger order will be processed later. This activity processes for naming identifier is considered as less important than other activities, so its Order value is 20 indicates that you are able to declare 19 remaining activities to be processed before this one.

The processing logic in this activity in implementation of Execute method is simply that replaces double open and close bracket ([[ and ]]) by double quote (“).

For example, Izenda Raw query:

SELECT [[CategoryID]] AS "categoryid\_15264036\_b36f\_",[CategoryName]] AS "categoryname\_32056250\_9bdf\_"

FROM [dbo]].[Categories]]

above raw query after is processed by this normalizer activity will become:

SELECT "CategoryID" AS "categoryid\_15264036\_b36f\_","CategoryName" AS "categoryname\_32056250\_9bdf\_"

FROM "dbo"."Categories"

### Add class RedshiftReplaceConcatFunctionActivity

Izenda Raw Query contains IZENDA\_CONCAT function, which absolutely is not supported in Redshift database, we have to replace that custom function by custom query logic to handle concatenate multiple values. This normalizer activity will take that responsibility.

On Izenda.BI.QueryNormalizer.Redshift project, add RedshiftReplaceConcatFunctionActivity.cs and modify class’s content like below:

using System.Text;

using Izenda.BI.Framework.Models.Contexts;

using Izenda.BI.QueryNormalizer.Utility;

namespace Izenda.BI.QueryNormalizer.Redshift

{

public class RedshiftReplaceConcatFunctionActivity

: RedshiftQueryNormalizerActivity

{

public override int Order => 10;

public override void Execute(QueryNormalizerContext context)

{

var sql = context.Query;

int index = -1;

var concat = "IZENDA\_CONCAT";

index = sql.IndexOf(concat);

while (index >= 0)

{

var openIndex = sql.IndexOf("(", index);

var closeIndex = ConcatFunctionUtil.FindMatchCloseIndex(sql, openIndex + 1);

var fieldValues = sql.Substring(openIndex + 1, closeIndex - 1 - openIndex);

var fields = ConcatFunctionUtil.GetConcatParams(fieldValues);

var builder = new StringBuilder();

for (int i = 0; i < fields.Count; i++)

{

builder.Append($"CAST({fields[i].Trim()} AS TEXT)");

if (i < fields.Count - 1)

{

builder.Append(" || ");

}

}

var replacedContent = builder.ToString();

var originalContent = sql.Substring(index, closeIndex - index + 1);

sql = sql.Replace(originalContent, replacedContent);

index = sql.IndexOf(concat);

}

context.Query = sql;

}

}

}

This activity has higher priority than RedshiftEscapseTableNameActivity, so it’s Order value is 10 less than 20 of naming identifier escaping.

The concatenate processing logic is finding all parameters value of IZENDA\_CONCAT function and concatenate them by concatenation operator (||) in Redshift.

### Add class RedshiftReplaceDateTruncateFunctionActivity

Izenda generates customized function DATETRUNCATE for removing time part from date value. Redshift does not support it by default, so we will implement a normalizer activity to convert that function to the function is supported in Redshift. On other hand, we will replace DATETRUCATE by DATE\_TRUNC function in Redshift.

In current project, add RedshiftReplaceDateTruncateFunctionActivity.cs and modify class like below:

using Izenda.BI.Framework.Models.Contexts;

namespace Izenda.BI.QueryNormalizer.Redshift

{

public class RedShiftReplaceDateTruncateFunctionActivity

: RedshiftQueryNormalizerActivity

{

public override int Order => 12;

public override void Execute(QueryNormalizerContext context)

{

var sql = context.Query;

sql = sql.Replace(@"DATETRUNCATE", @"DATE\_TRUNC");

context.Query = sql;

}

}

}

The Order of this activity is 12, that means it will be processed before RedshiftEscapseTableNameActivity but after RedshiftReplaceConcatFunctionActivity.

The replacement logic simply replaces DATETRUNCATE by DATE\_TRUNC.

### Write your custom Query Normalizer Activity

If you want to write a new customized query normalizer activity to custom your query in a manner way, you can add new class that overrides abstracted class RedshiftQueryNormalizerActivity then indicates processing order as well as string processing logic in Execute method.

A customized activity is considered as extension place to customize the query, which will be executed by you Data Source Adaptor.

# Create Data Source Adaptor for Redshift

A Data Source Adaptor component is implementation of IDataSoruceAdaptor interface to adapt data loading and data calculation between Izenda logic layouts with a database system, in this tutorial the database system is Redshift. Other RDBMS will have different approach for developing an adaptor component but you can keep logical thinking and analysis solution like this guideline while it is describing specific for Redshift.

As mentioned before, if you aim to develop new adaptor component with architecture other than based on Izenda.BI.DataSourceAdaptor.RDBMS of Izenda, you can ignore implantation description from here. However, this guideline includes analysis and notice can be helpful for you when developing an adaptor in different architecture and solution approach.

Implementation of IDataSourceAdaptor is very wide abstraction scope so it will decouple into multiple logical dependency units. That are Connection handler, Schema Loader, database system configuration and SQL Command Generator. The SQL generators include Operator Command Generator, Expression Command Generator and Filter Generator. [Overview](#_Izenda_Data_Source) section has already described this concept. Through the guideline, we are going to create RedshiftConnection, RedshiftSchemaLoader, Redshift database configuration and SQL command generator for operand, function and report filtering.

The full implementation of Redshift Data Source Adaptor is available at <https://github.com/Izenda7Series/IzendaCustomDataAdaptors/tree/redshiftdataadapter>

## Identify Data Provider

Each database system will provide its own data provider to woks with .Net Framework. It is library that we can use in .NET framework application. Table below describes Data Provider library for some RDBMSs:

|  |  |  |  |
| --- | --- | --- | --- |
| .No | Database | Data Provider Library | Description |
| 1 | SQL Server | System.Data.SqlClient | Namespace inside System.Data library in .NET Framework |
| 2 | Oracle | Oracle.ManagedDataAccess.dll | Oracle Data Provider for .NET, Managed Driver |
| 3 | PostgreSQL | Npgsql | ADO.NET Data Provider for PostgreSQL |
| 4 | MySQL | MySql.Data | Connector/Net, fully-managed ADO.NET driver for MySQL. |
| 5 | IBM - DB2 | IBM.Data.DB2.dll | IBM Data Server Driver |

Table 19: Data Provider for Relational Database System

Most RDBMS’s vendor will provide Data Provider or Driver to support connecting to a database, executing commands and retrieving result in .NET Application. To develop Izenda Data Source Adaptor for specific RDBMS, you have to identify which Data Provider will be used first.

On other hand, if your targeted RDBMS supports to work with an ODBC Driver you also use OdbcConnection which build-in in .NET Framework to communicate with your database server. We also provide an example using OdbcConnection to build Data Source Adaptor on Github repository at <https://github.com/Izenda7Series/IzendaCustomDataAdaptors/tree/odbcdataadapter>. Note that ODBC Driver will have many limitations that leads you to issues later; be careful when choosing Data Provider.

In the case of Redshift database, we are able to use ADO.NET Data Prover for PostgreSQL or Redshift ODBC Driver. Until this document has been written, the ODBC Driver for Redshift has limitation issue when executing multiple SQL commands with multiple parameters (multiple SQL commands is split by semi colon (;)), for that reason we decide to use Data Provider for PostgreSQL in Izenda Data Source Adaptor for Redshift (Basically, Amazon Redshift was based on PostgreSQL 8.0.2). Although that, we also provides example using ODBC Driver in Redshift database, it will be described later.

## Dapper Usage

All Data Access Layer components in Izenda system use Dapper ORM Framework to process querying and executing SQL command, the adaptor for Redshift also use that library to process communicate between database and Izenda business layer.

That does not mean you are required to use Dapper for executing SQL command but IDataSourceAdaptor is depends on some type which is defined in Dapper library, that requires adaptor project always references Dapper library.

Dapper works well with almost RDBMSs but for avoiding unexpected issues happens on your data source adaptor we suggest having fully investigation to be sure that your RDBMS Data Provider is able to work perfectly with Dapper. If they not compatibility together, you have to ignore usage of QueryMutiple method of DataSourceAdator by override GetPagingResult method and implement your customized query executing, the example for that is provided in ODBC adaptor example and will be mentioned later.

For Redshift database, we use Npgsql Data Provider and there are no issues when working with Dapper library.

## Create new Adaptor Project

Like SQL Query Normalizer component, the adaptor component also has to target to .NET Framework 4.0 and follow naming convention for library in entire Izenda system.

Open RedshiftDataAdaptor.sln in Visual Studio IDE, then add new project Izenda.BI.DataAdaptor.RDBMS.Redshift like below:

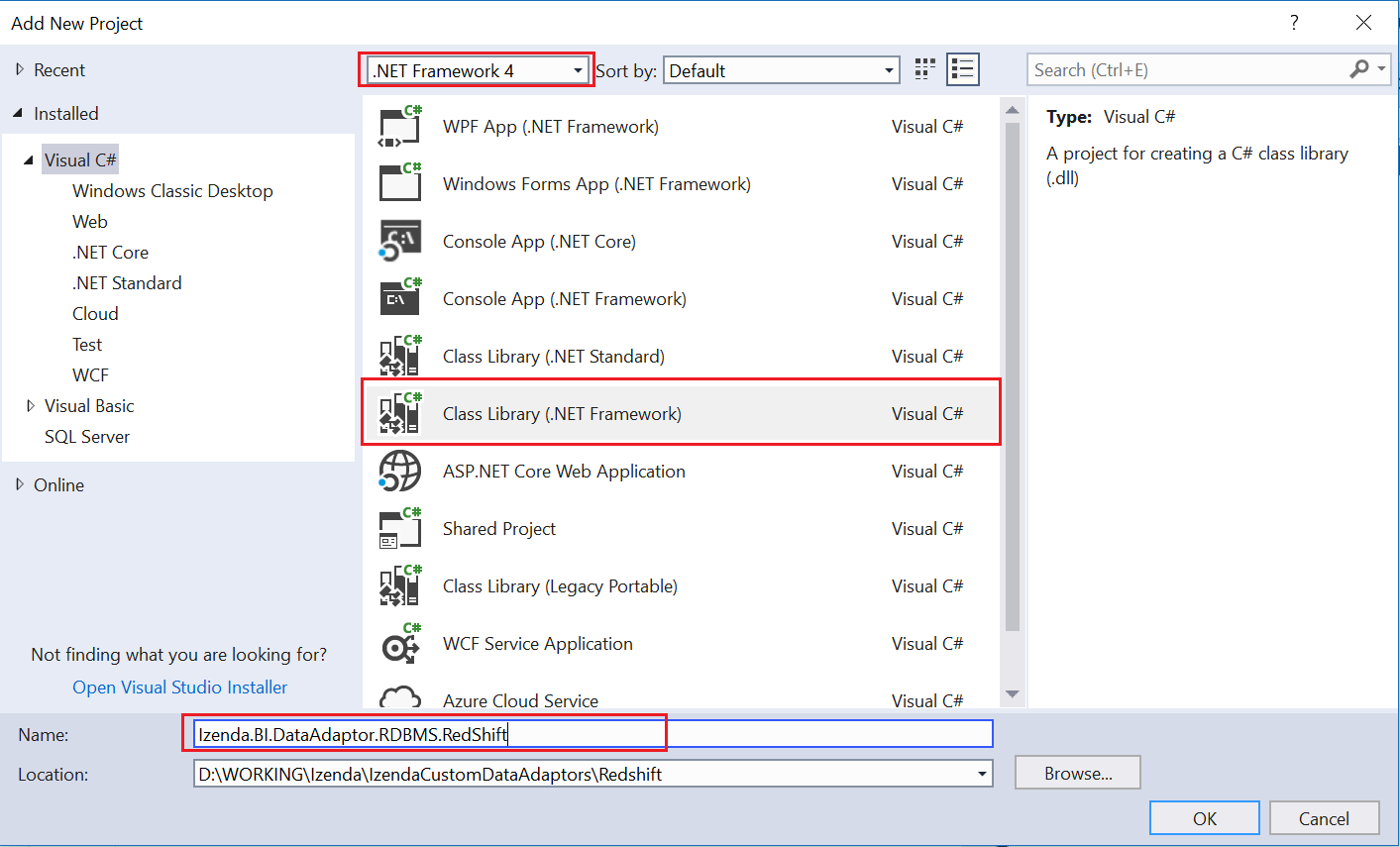


Figure 10: Create new Redshift Data Source Adaptor

Click OK button to create empty project.

## Add Project Dependency

Add library dependency from ~\Redshift\libs\Izenda and ~\Redshift\libs\Dapper:

* Izenda.BI.DataAdaptor.RDBMS.dll
* Izenda.BI.DataSourceAdapter.dll
* Izenda.BI.Framework.dll
* Izenda.BI.Logging.dll
* Izenda.BI.RDBMS.dll
* Izenda.BI.Resource.dll

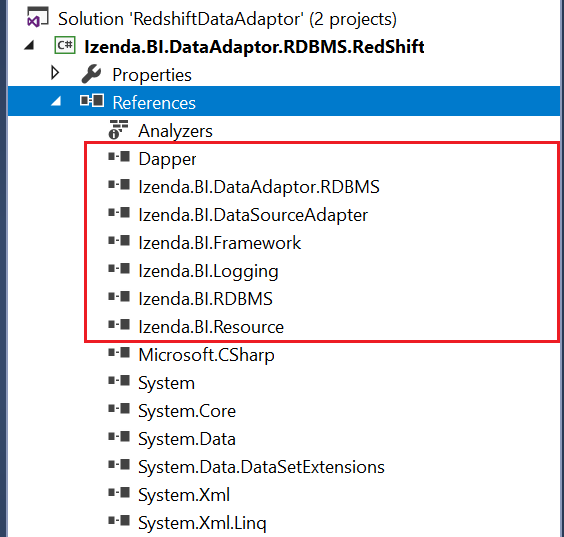


Figure 11: Izenda library dependency in adaptor component

Add project dependency Izenda.BI.QueryNormalizer.Redshift:

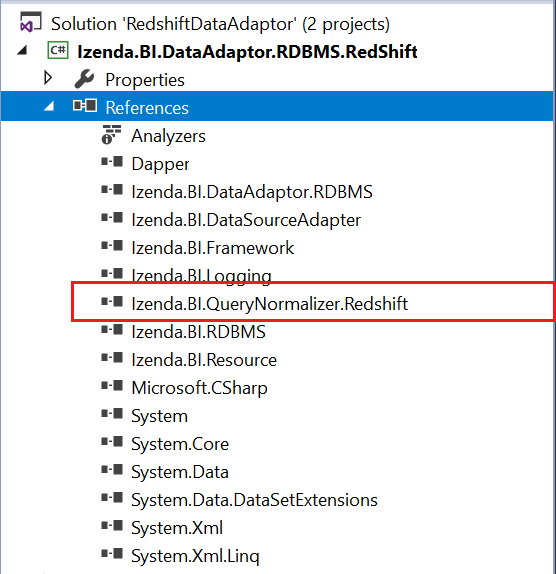


Figure 12: Add Query Normalizer project dependency

Add ADO.NET Data Provider for PostgreSQL from nugget package by open Tools/Nuget Package Manager/Package Manager Console and type PM> Install-Package Npgsql -Version 2.2.7

Note that, we only need Npgsql version 2.2.7 works on .NET Framework 4.0, the latest version is higher but it works for newer .NET version.

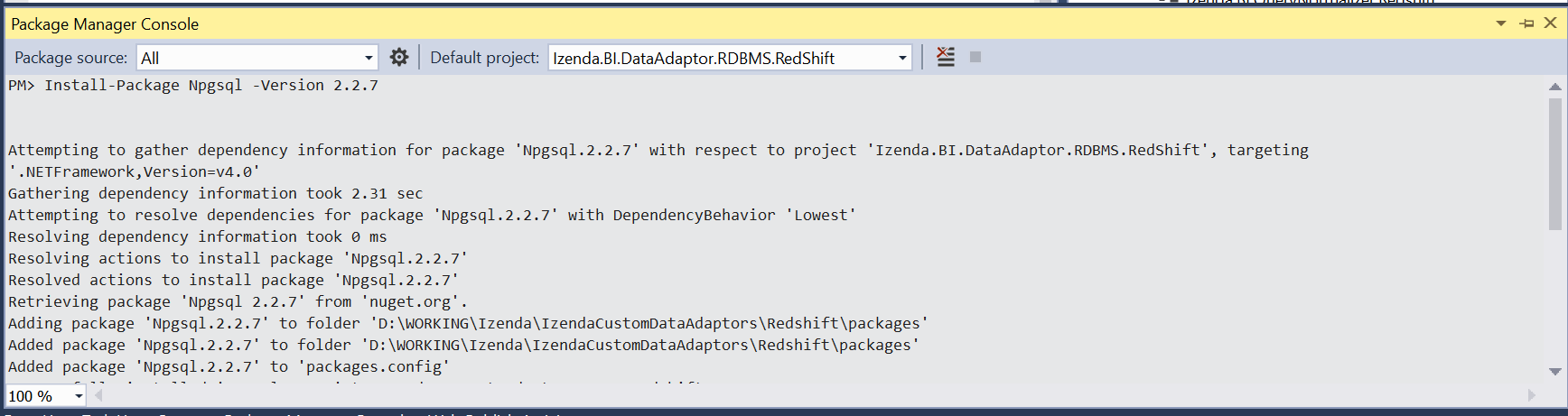


Figure 13: Using Nuget Package Manager Console to add dependency to Npgsql 2.2.7

The completed dependency of adaptor project will like below:

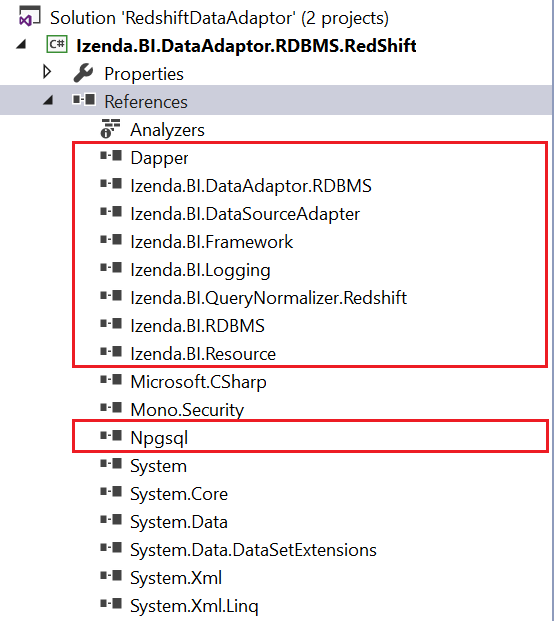


Figure 14: Library dependency of adaptor project

## Configure Data Types and Build-in Functions

### Data Types

Create folder Constants in project then add the class RedshiftSupportDataType.cs, modify that class to inherit from DatabaseSupportDataType in Izenda.BI.DataAdaptor.RDBMS.Constants.

Adding the mapping between data type of Redshift, Izenda and .NET type:

Date Time Data Types:

private void RegisterDateTime()

{

AddDatabaseDataType("date", IzendaDataType.DatetimeType, true, "System.DateTime", true/\*default mapping\*/);

// timestamp is alias for [timestamp with time zone]

AddDatabaseDataType("timestamp", IzendaDataType.DatetimeType, true, "System.DateTime");

AddDatabaseDataType("timestamp with time zone", IzendaDataType.DatetimeType, true, "System.DateTime");

// timestamptz is alias for [timestamp without time zone]

AddDatabaseDataType("timestamptz", IzendaDataType.DatetimeType, true, "System.DateTime");

AddDatabaseDataType("timestamp without time zone", IzendaDataType.DatetimeType, true, "System.DateTime");

}

Numeric Data Types:

private void RegisterNumeric()

{

AddDatabaseDataType("numeric", IzendaDataType.NumericType, true, "System.Decimal", true /\*default mapping\*/);

// integer

AddDatabaseDataType("smallint", IzendaDataType.NumericType, true, "System.Int16");

AddDatabaseDataType("int2", IzendaDataType.NumericType, true, "System.Int16");

AddDatabaseDataType("integer", IzendaDataType.NumericType, true, "System.Int32");

AddDatabaseDataType("int", IzendaDataType.NumericType, true, "System.Int32");

AddDatabaseDataType("int4", IzendaDataType.NumericType, true, "System.Int32");

AddDatabaseDataType("int8", IzendaDataType.NumericType, true, "System.Int64");

AddDatabaseDataType("bigint", IzendaDataType.NumericType, true, "System.Int64");

// floating point number

AddDatabaseDataType("real", IzendaDataType.NumericType, true, "System.Single");

AddDatabaseDataType("float4", IzendaDataType.NumericType, true, "System.Single");

AddDatabaseDataType("float", IzendaDataType.NumericType, true, "System.Double");

AddDatabaseDataType("float8", IzendaDataType.NumericType, true, "System.Double");

AddDatabaseDataType("double precision", IzendaDataType.NumericType, true, "System.Double");

AddDatabaseDataType("decimal", IzendaDataType.NumericType, true, "System.Decimal");

}

Boolean Data Types:

private void RegisterBoolean()

{

// bool is alias for boolean

AddDatabaseDataType("bool", IzendaDataType.BooleanType, true, "System.Boolean");

AddDatabaseDataType("boolean", IzendaDataType.BooleanType, true, "System.Boolean", true/\*default mapping\*/);

}

Text Data Types:

private void RegisterText()

{

// 4k range

AddDatabaseDataType("char", IzendaDataType.TextType, true, "System.String");

AddDatabaseDataType("character", IzendaDataType.TextType, true, "System.String");

AddDatabaseDataType("nchar", IzendaDataType.TextType, true, "System.String");

AddDatabaseDataType("national character", IzendaDataType.TextType, true, "System.String");

// 64k range

AddDatabaseDataType("varchar", IzendaDataType.TextType, true, "System.String", true/\*default mapping\*/);

AddDatabaseDataType("character varying", IzendaDataType.TextType, true, "System.String");

AddDatabaseDataType("nvarchar", IzendaDataType.TextType, true, "System.String");

AddDatabaseDataType("national character varying", IzendaDataType.TextType, true, "System.String");

// 265 bytes

AddDatabaseDataType("bpchar", IzendaDataType.TextType, true, "System.String");

AddDatabaseDataType("text", IzendaDataType.TextType, true, "System.String");

}

Combine above data types mapping to register in constructor method:

public class RedShiftSupportDataType : DatabaseSupportDataType

{

public RedShiftSupportDataType()

{

RegisterNumeric();

RegisterDateTime();

RegisterBoolean();

RegisterText();

}

…

}

The full implementation of RedhisfhtSupportDataType class at [Provide Link].

### Database Functions

The table #2 list all functions that are used in Izenda system but some of functions maybe is not available on your targeted database system or function naming can be different, in that case you have to provide configuration to indicate new adapter component use new name or syntax of corresponding function.

Comparing in Redshift database, we see the DatePart and IsNull property in DatabaseFunction class have to be overridden to provide in new name and new syntax. The DataPart must be date\_part and the IsNull function will be replaced by NVL function.

In Constants folder, add class RedshiftDatabaseFunction.cs and override property DatePart and IsNull as below:

using Izenda.BI.DataAdaptor.RDBMS.Constants;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.Constants

{

public class RedshiftDatabaseFunction : DatabaseFunction

{

public override string DatePart

{

get

{

return "date\_part";

}

}

public override string IsNull

{

get

{

return "nvl";

}

}

}

}

Logically, this configuration will be used to generate corresponding SQL syntax for expression command will be described in later section.

### Database Function Format

According the function formats listed in table #3, Izenda system has predefined function formats, which are using to call database function. If on your database system, the function has different format that will require we have to configure that changes to make Izenda system understands function format syntax on your database system.

Most of formats in DatabaseFunctionFormat server for date time format and filtering date time value on Izenda report, those configurations take affected when we provide filtering command generator on adaptor component, combine with feature that data adaptor loads filtering command generator by .NET refection, all of them will be overridden so that makes format configuration affect automatically.

Investigation and comparing the function format syntax in based adaptor and Redshift, the format property CheckBlankFormatText, CheckNotBlankFormat and CheckNotBlankFormatText will be override to provide new syntax.

In Constants folder, Add RedshiftDatabaseFunctionFormat.cs and implement overriding like below:

using Izenda.BI.DataAdaptor.RDBMS.Constants;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.Constants

{

public class RedshiftDatabaseFunctionFormat : DatabaseFunctionFormat

{

public override string CheckBlankFormatText

{

get

{

return @"[[{0}]] = ''";

}

}

public override string CheckNotBlankFormat

{

get

{

return "([[{0}]] IS NULL OR [[{0}]] <> '')";

}

}

public override string CheckNotBlankFormatText

{

get

{

return "([[{0}]] IS NULL OR [[{0}]] <> '')";

}

}

}

}

Those properties are used in JoinOperatorCommandGenerator, so to make this configuration effect that operator command generator have to be overridden and it will be described detail in later section.

### Database Constants

Because of datepart and timepart in Redshift database are supported similar with default configuration in DatabaseConstants class in Izenda.BI.Framework.Constants namespace, so we no need to override to change the configuration for this database.

The CONVERT method also work well to convert date time value in Redshift then.

## Implement connection handler class

This class manages connection to the Redshift database over ADO.NET Data Provider for PostgreSQL (Npgsql library), and take responsibility for retrieving information about connection.

ReshiftConnection class will be using to decouple abstraction from implementation of IDataSourceAdaptor interface so that it can vary independently on connection functionality.

Add RedshiftConnection.cs and modify to inherit from IConnection in Izenda.BI.Framwork.

The table below describes functionality for each method inside IConnection interface:

|  |  |  |
| --- | --- | --- |
| # | Method | Description |
| 1 | GetDatabaseName | Get the name of database which is connected by current connection string |
| 2 | GetDatabaseServer | Return the name of database server which is connected by current connection string |
| 3 | GetUserName | Get user name in user credential info for connecting to the database |
| 4 | OpenConnection | Use Data Provider to open new connection for current connection string |
| 5 | TestConnection | Actually open the connection and verify whether connection is connected OK or not |

Figure 15: Describe method of IConnection interface

The completed implementation of RedshiftConnection class as below:

using Izenda.BI.Framework;

using System;

using System.Collections.Generic;

using Izenda.BI.Framework.Models;

using System.Data;

using Izenda.BI.Framework.Constants;

using Npgsql;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift

{

public class RedShiftConnection : IConnection

{

public string GetDatabaseName(string connectionString)

{

var builder = new NpgsqlConnectionStringBuilder(connectionString);

return builder.Database;

}

public string GetDatabaseServer(string connectionString)

{

var builder = new NpgsqlConnectionStringBuilder(connectionString);

return builder.Host;

}

public string GetUserName(string connectionString)

{

var builder = new NpgsqlConnectionStringBuilder(connectionString);

return builder.UserName;

}

public IDbConnection OpenConnection(string connectionString)

{

var connection = new NpgsqlConnection(connectionString);

connection.Open();

return connection;

}

public ConnectionStatus TestConnection(Guid serverType, string connectionString)

{

var result = new ConnectionStatus

{

Status = ConnectDBStatus.Success,

ConnectionString = connectionString

};

try

{

var databaseName = GetDatabaseName(connectionString);

if (string.IsNullOrEmpty(databaseName))

{

result.Status = ConnectDBStatus.DatabaseNotValid;

return result;

}

using (var connection = OpenConnection(connectionString))

{

result.Status = ConnectDBStatus.Success;

return result;

}

}

catch (NpgsqlException ex)

{

var invalidServerMessage = new List<string> { "08000", "08003", "08006", "08001", "08004", "08007", "08P01" }; //These numbers is status code return by database relate to server name or network issue.

var invalidDatabase = "3D000"; //Error code return when database is not valid

var loginFail = new List<string> { "28000", "28P01" }; //Error code return when cannot loggin database

if (invalidServerMessage.Contains(ex.Code))

{

result.Status = ConnectDBStatus.ServerNotValid;

}

else if (ex.Code == invalidDatabase)

{

result.Status = ConnectDBStatus.DatabaseNotValid;

}

else if (loginFail.Contains(ex.Code))

{

result.Status = ConnectDBStatus.LoginFail;

}

else

{

result.Status = ConnectDBStatus.Fail;

}

}

catch

{

result.Status = ConnectDBStatus.Fail;

}

return result;

}

}

}

## Implement schema loader class

The class takes responsibility for discovering database schema structure as schema name, table, view, column, function and store procedure parameter, relationship then convert to Izenda abstraction entry such as DB Source, Query Source Category, Query Source, Query Source Field, Query Source Parameter and Relationship.

Schema loader class will inherit from ISchemaLoader interface in Izenda.BI.DataAdaptor.RDBMS library. It will be composite component inside implementation of IDataSourceAdaptor, the based data source adaptor class DataSourceAdaptor in Izenda.BI.DataAdaptor.RDBMS component as well as implementation of specific database system. By split schema loading logic into ISchemaLoader, we can re-use many coding from DataSourceAdaptor then reduce much effort to develop new adaptor component.

Schema loader is important logic and requires carefully analysis and develop, it explores your data source schema and structure then transform to Izenda logic concept so if that progress is not perform correctly it can make Izenda system works incorrectly with many multiple major issues.

Each RDBMS stores database structure in different manner, to implement schema loader you have to study and have fully understand about it. With Amazon Redshift, the collection of existing schema is stored in *pg\_namespace* table. All tables and views are stored in *information\_schema.tables* and distinct between them by *table\_type* column value *BASE TABLE* and *VIEW*. All the columns of table and view are stored in *information\_schema.columns*. The information of function is stored in *pg\_proc* and *information\_schema.routine*. Redshift does not support store procedure by default. For parameters, Redshift store them in table *information\_schema.parameters*. About relationship information, they are stored in *information\_schema.key\_column\_usage*, *information\_schema.constraint\_column\_usage*, *information\_schema.referential\_constraints* and *pg\_constrain* as well. We will use those tables to explore database structure in Redshift for schema loader implementation.

### ISchemaLoader interface and implementation

This section will describe detail functionality of methods in schema loader interface. And how to implement it in inherited class.

For Redshift database, schema loader is named as RedshiftSchemaLoader, add RedshiftSchemaLoader.cs into project and modify it inherit from ISchemaLoader interface.

using Izenda.BI.DataAdaptor.SQL.SchemaLoader;

using System;

using System.Collections.Generic;

using Izenda.BI.Framework.Models;

using Izenda.BI.Logging;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift

{

public class RedshiftSchemaLoader : ISchemaLoader

{

public List<QuerySourceParameter> GetQuerySourceParameters(string connectionString)

{

throw new NotImplementedException();

}

public List<QuerySourceParameter> GetQuerySourceParameters(string connectionString, string specificSchema, string specificName)

{

throw new NotImplementedException();

}

public List<QuerySourceField> LoadCustomQuerySourceFields(string connectionString, string customQueryDefinition)

{

throw new NotImplementedException();

}

public List<QuerySourceField> LoadFields(string connectionString, string type, string categoryName, string querySourceName, bool rollbackSP, List<QuerySourceParameter> parameters = null, bool ignoreError = true, int commandTimeout = 500, ILog log = null)

{

throw new NotImplementedException();

}

public List<QuerySourceField> LoadQuerySourceFields(string connectionString)

{

throw new NotImplementedException();

}

public List<Relationship> LoadRelationships(string connectionString, List<string> schemas = null)

{

throw new NotImplementedException();

}

public DBSource LoadSchema(string connectionString)

{

throw new NotImplementedException();

}

}

}

### GetQuerySourceParameters(string)

List<QuerySourceParameter> GetQuerySourceParameters(string connectionString);

Get all parameters of all functions and store procedures in database in connection string.

Parameters:

* connectionString: The connection string to data source.

The implementation for this method is call to other overload of GetQuerySourceParameters method.

public List<QuerySourceParameter> GetQuerySourceParameters(string connectionString)

{

return GetQuerySourceParameters(connectionString, string.Empty, string.Empty);

}

### GetQuerySourceParameters(string, string, string)

Get all parameters of specific function or store procedure in specific schema.

public List<QuerySourceParameter> GetQuerySourceParameters(string connectionString, string specificSchema, string specificName)

Parameters:

* connectionString: the connection string to data source.
* specificSchema: The schema name to filter parameter list.
* specificName: The name of function or store procedure to get all parameters.

Implementation:

For Redshift database, all parameters are stored in *information\_schema.paramters*, in this method we query parameter info from that table then convert to Izenda Query Source Parameter.

public List<QuerySourceParameter> GetQuerySourceParameters(string connectionString, string specificSchema, string specificName)

{

var dataTypeAdaptor = new RedshiftSupportDataType();

using (var conn = new NpgsqlConnection(connectionString))

{

var sql = @"SELECT specific\_schema, specific\_name, parameter\_mode, is\_result

,parameter\_name, data\_type, ordinal\_position

FROM information\_schema.parameters";

if (!string.IsNullOrEmpty(specificSchema) && !string.IsNullOrEmpty(specificName))

{

sql += " WHERE specific\_schema = @SPECIFIC\_SCHEMA AND specific\_name=@SPECIFIC\_NAME ";

}

var parametes =

conn.Query<dynamic>(sql, new { SPECIFIC\_SCHEMA = specificSchema, SPECIFIC\_NAME = specificName })

.Select(s => new QuerySourceParameter

{

Name = s.parameter\_name,

QuerySourceName = s.specific\_name,

Category = s.specific\_schema,

DataType = s.data\_type,

IzendaDataType = dataTypeAdaptor.GetIzendaDataType(s.data\_type),

InputMode = s.parameter\_mode.Equals("IN", StringComparison.OrdinalIgnoreCase),

Result = s.is\_result.Equals("YES", StringComparison.OrdinalIgnoreCase),

Position = s.ordinal\_position,

Value = DBNull.Value,

AllowDistinct = dataTypeAdaptor.GetAllowDistinct(s.data\_type)

}).ToList();

return parametes;

}

}

On other RDBMS, you have to find out how it manages function and store procedure parameters and provide corresponding query to get out parameter information.

### LoadCustomQuerySourceFields(string, string)

List<QuerySourceField> LoadCustomQuerySourceFields(string connectionString, string customQueryDefinition);

Get all columns from custom query.

Parameters:

* connectionString: The connection string to the data source.
* customQueryDefinition: The custom query.

Implementation: Execute custom query and get column list from data reader result then convert to Izenda Query Source Field.

public List<QuerySourceField> LoadCustomQuerySourceFields(string connectionString, string customQueryDefinition

{

var result = new List<QuerySourceField>();

var dataTypeAdaptor = new RedShiftSupportDataType();

using (var conn = new NpgsqlConnection(connectionString))

{

conn.Open();

NpgsqlCommand command = new NpgsqlCommand(customQueryDefinition, conn);

command.CommandType = CommandType.Text;

NpgsqlDataReader reader = command.ExecuteReader(CommandBehavior.SchemaOnly);

DataTable schema = reader.GetSchemaTable();

for (int i = 0; i < schema.Rows.Count; i++)

{

string dataType = reader.GetDataTypeName(i);

result.Add(

new QuerySourceField

{

Name = schema.Rows[i]["ColumnName"].ToString() ?? "",

DataType = dataType,

IzendaDataType = dataTypeAdaptor.GetIzendaDataType(dataType),

AllowDistinct = dataTypeAdaptor.GetAllowDistinct(dataType),

ExtendedProperties = "",

Position = Convert.ToInt32(schema.Rows[0]["ColumnOrdinal"].ToString())

}

);

}

}

return result;

}

### LoadQuerySourceField(string)

List<QuerySourceField> LoadQuerySourceFields(string connectionString);

Get all columns of all tables and views in database.

Parameters:

* connectionString: The connection string to the data source.

Implementation: Call the method LoadFieldFromTable which is defined in section of LoadField method.

public List<QuerySourceField> LoadQuerySourceFields(string connectionString)

{

return LoadFieldsFromTable(connectionString);

}

### LoadSchema(string)

DBSource LoadSchema(string connectionString);

The method loads entire data source structure includes schema, table, view, store procedure and functions then transform to Izenda abstraction concept. All database schema structure will be stored into DBSource object, and logically that object will be stored into Izenda system database when a connection string is saved.

Parameters:

* connectionString: The connection string to the data source.

Implementation of this method follows the idea that it gets all database schemas as list of Query Source Category, in each Query Source Category (schema) it gets all tables, views, store procedures and functions as list of Query Source which is belong the current schema.

Implementation in RedshiftSchemaLoader:

Get all schema: This gets all schemas of database in current connection from *pg\_namespace* tables and exclude built-in schemas.

protected IList<QuerySourceCategory> GetSchemas(IDbConnection conn, string excludeSchemas)

{

string sql = string.Format(@"SELECT nspname FROM pg\_namespace WHERE nspname NOT IN({0})", excludeSchemas);

var querySourceCategories = conn.Query<dynamic>(sql)

.Select(s => new QuerySourceCategory { Name = s.nspname })

.ToList();

return querySourceCategories;

}

Get all tables and views: This get all tables or views from *information\_schema.tables* for each schema of database in current connection.

protected IList<QuerySource> GetTables(IDbConnection conn, string schemaName, string tableType)

{

string sql = @" SELECT table\_name,table\_type

FROM information\_schema.tables

WHERE table\_schema=@schema AND table\_type=@type";

var sqlQuerySourceType = "UNKNOW";

if (SQL\_TYPE\_TABLE.Equals(tableType, StringComparison.OrdinalIgnoreCase))

{

sqlQuerySourceType = SQLQuerySourceType.Table;

}

else if (SQL\_TYPE\_VIEW.Equals(tableType, StringComparison.OrdinalIgnoreCase))

{

sqlQuerySourceType = SQLQuerySourceType.View;

}

var querySources = conn.Query<dynamic>(sql, new { schema = schemaName, type = tableType })

.Select(s => new QuerySource

{

Name = s.table\_name,

Type = sqlQuerySourceType

})

.ToList();

return querySources;

}

Get all functions: This get all functions from *information\_schema.routines* of each schema of database in current connection.

protected IList<QuerySource> GetFunctions(IDbConnection conn, string schemaName)

{

string sql = string.Format(LOAD\_SCHEMA\_FUNCTION, 0);

var querySources = conn.Query<dynamic>(sql, new { schema = schemaName, type = SQL\_TYPE\_FUNCTION });

var result = new List<dynamic>();

foreach (var querySource in querySources)

{

result.Add(querySource);

}

return result.Select(s => new QuerySource

{

Name = s.routine\_name,

Type = SQLQuerySourceType.Function

}).GroupBy(x => x.Name).Select(x => x.First()).ToList();

}

Get all store procedure: This get all store procedures from *information\_schema.routines* of each schema of database in current connection. Although Redshift does not support procedure but this is using to demonstrate store procedure loading, which has similar logic on other RDBMS.

protected IList<QuerySource> GetProcedures(IDbConnection conn, string schemaName)

{

string sql = string.Format(LOAD\_SCHEMA\_FUNCTION, 1);

var querySources = conn.Query<dynamic>(sql, new { schema = schemaName, type = SQL\_TYPE\_FUNCTION });

var result = querySources.Select(s => new QuerySource

{

Name = s.routine\_name,

Type = SQLQuerySourceType.Procedure

}).GroupBy(x => x.Name);

if (result.Where(x => x.Count() > 1).Count() > 0)

{

var modelErrors = new ModelErrors();

modelErrors.AddError(string.Empty, Messages.DuplicateFunctionName);

throw new IzendaModelException(modelErrors);

}

return result.Select(x => x.First()).ToList();

}

Combine getting schemas, tables, views, functions and store procedures above we have the full implementation of LoadSchema method in RedshiftSchemaLoader class:

public DBSource LoadSchema(string connectionString)

{

using (var conn = new NpgsqlConnection(connectionString))

{

conn.Open();

var querySourceCategories = GetSchemas(conn, EXCLUDE\_SCHEMAS);

foreach (var category in querySourceCategories)

{

category.QuerySources = new List<QuerySource>();

// Load Tables

category.QuerySources.AddRange(GetTables(conn, category.Name, SQL\_TYPE\_TABLE));

// Load Views

category.QuerySources.AddRange(GetTables(conn, category.Name, SQL\_TYPE\_VIEW));

// Load Functions

category.QuerySources.AddRange(GetFunctions(conn, category.Name));

// Load store procedure

category.QuerySources.AddRange(GetProcedures(conn, category.Name));

// Sort by name

category.QuerySources = category.QuerySources.OrderBy(s => s.Name).ToList();

}

return new DBSource

{

QuerySources = querySourceCategories.ToList()

};

}

}

### LoadFields(string, string, string, string, bool, List<QuerySourceParamter>, bool, int, ILog)

This method will load all columns (query source fields) of a table, view or store procedure in specific schema of current connection and map them to Query Source Field objects. In case of loading query source field for store procedure it requires to execute that procedure, that case makes the last 5 parameters to be used.

This method is exactly same with similar one in IDataSourceAdaptor interface. It is declared here to provide better design for breaking dependency as well as serving single responsibility in implementation. It will using in multiple locations on Izenda Business Layer and based class DataSourceAdaptor in Izenda.BI.DataAdaptor.RDBMS component.

List<QuerySourceField> LoadFields(string connectionString, string type, string categoryName, string querySourceName, bool rollbackSP, List<QuerySourceParameter> parameters = null, bool ignoreError = true, int commandTimeout = 500, ILog log = null);

Parameters:

* connectionString: The connection string to the data source
* type: The SQL Query Source Type and the value can be Table, View or Store Procedure
* categoryName: The schema name of table, view or store procedure
* querySroucename: The either table name, view name or store procedure name.
* rollbackSP: Is using when loading fields for store procedure. If this value is true, it requests new transaction to rollback command to execute store procedure for getting all the fields of store procedure it requires executing procedure to receive the field list from result. Some procedure will change data after executed. Therefore, we have to rollback that change to make sure the data not corrupted. This parameter is only applied for loading field for store procedure.
* List<QuerySourceParameter>: The list of parameter for executing store procedure to get field set from result. This parameter is only applied when load field for store procedure.
* ignoreError: Indicate that system always continue to run although there is error when executing store procedure. This parameter is only applied when load field for store procedure.
* commandTimeout: The time out for executing store procedure. This parameter is only applied when loading field for store procedure.
* ILog: The Izenda logger instance to log tracking when loading field. You can use this log instance to log out your tracking information and that info will be appending into izenda log file.

Implementation in RedshiftSchemaLoader:

Get all columns of table or view: This implementation will get all columns of a table or view from *information\_schema.columns* then convert to Izenda Query Source Field object.

private List<QuerySourceField> LoadFieldsFromTable(string connectionString, string schemaName = "", string tableName = "")

{

var result = new List<QuerySourceField>();

var dataTypeAdaptor = new RedshiftSupportDataType();

string sql = @"SELECT DISTINCT c.column\_name, c.data\_type, c.ordinal\_position, c.table\_schema, c.table\_name

FROM information\_schema.Columns c";

if (!string.IsNullOrWhiteSpace(tableName) && !string.IsNullOrWhiteSpace(schemaName))

{

sql += @" WHERE c.table\_schema = @TableSchema AND c.table\_name = @TableName ";

}

using (var conn = new NpgsqlConnection(connectionString))

{

conn.Open();

result = conn.Query<dynamic>(sql, new { TableSchema = schemaName, TableName = tableName })

.Select(s => new QuerySourceField

{

Name = s.column\_name,

CategoryName = s.table\_schema ?? "",

QuerySourceName = s.table\_name ?? "",

DataType = s.data\_type,

IzendaDataType = dataTypeAdaptor.GetIzendaDataType(s.data\_type),

AllowDistinct = dataTypeAdaptor.GetAllowDistinct(s.data\_type),

Position = s.ordinal\_position,

ExtendedProperties = ""

})

.Where(x => !string.IsNullOrEmpty(x.IzendaDataType))

.ToList();

}

return result;

}

Get all columns in store procedure: This implementation is not necessary for Redshift. In other RDBMS, we need to execute store procedure then get the columns from schema of data reader result, the implementation detail for this will be described in section how to create ODBC Data Adaptor.

### LoadRelationship(string, List<string>)

List<Relationship> LoadRelationships(string connectionString, List<string> schemas = null);

Load all relationships of database in connection string.

Parameters:

* connectionString: The connection string to data source.
* schemas: The list of schema name to filter constraints.

Implementation:

Combine information from *information\_schema.key\_coulumn\_usage*, *information\_schema.constraint\_column\_usage* and *information\_schema.referenctial\_constraints* we get all necessary information about relationship then convert to Relationship object.

public List<Relationship> LoadRelationships(string connectionString, List<string> schemas = null)

{

using (var conn = new NpgsqlConnection(connectionString))

{

string sql = @"SELECT kcu.constraint\_name as fk\_name,

kcu.constraint\_schema as jionschema,

kcu.table\_name as jiontable,

kcu.column\_name as jioncolumn,

ccu.constraint\_schema as foreignschema,

ccu.table\_name as foreigntable,

ccu.column\_name as foreigncolumn

FROM information\_schema.key\_column\_usage as kcu

INNER JOIN information\_schema.constraint\_column\_usage as ccu on ccu.constraint\_name = kcu.constraint\_name

INNER JOIN information\_schema.referential\_constraints as rc on rc.constraint\_name = kcu.constraint\_name

if (schemas != null && schemas.Count() > 0)

{

sql += $" WHERE kcu.constraint\_schema IN ({schemas.Aggregate((s1, s2) => $"'{s1}', '{s2}'")})";

}

var relationships = conn.Query<dynamic>(sql)

.Select(r => new Relationship

{

JoinQuerySourceName = r.jionschema + '.' + r.jiontable,

ForeignQuerySourceName = r.foreignschema + '.' + r.foreigntable,

JoinFieldName = r.jioncolumn,

ForeignFieldName = r.foreigncolumn

})

.ToList();

return relationships;

}

}

## Implement Data Source Adaptor class

### Declare adaptor class and exported configuration

In adaptor project add RedshiftDataAdaptor.cs and modify the class inherits from DataSourceAdaptor in Izenda.BI.DataAdaptor.RDBMS namespace.

Logically, Izenda system automatically loads all Data Source Adaptor from libraries, which are available in application domain directory. To register the implementation of IDataSourceAdator interface is adaptor lets declare export contract and metadata on top of adaptor class like below:

[Export(typeof(IDataSourceAdaptor))]

[ExportMetadata("ServerType", "E285BFD1-F8D5-4BEB-A345-B3D2EF5A3DE8" + "|AWSRS|[AWSRS] Redshift")]

public class RedshiftDataSourceAdaptor : DataSourceAdaptor

Note that Export and ExportMetadata keyword only are available when you add reference System.ComponentModel.Composition into adaptor project.

Izenda uses MEF to handle exporting an adaptor, the type of interface IDataSourceAdator is provided as exported contract and the metadata ServerType is provided to distinguish between multiple existing data adaptor available in application domain directory. The metadata basically contains two part, first part E285BFD1-F8D5-4BEB-A345-B3D2EF5A3DE8 is GUID identifier of adaptor and second part "|AWSRS|[AWSRS] Redshift is display name of adaptor.

The first implementation of RedshiftDataAdaptor class looks like below:

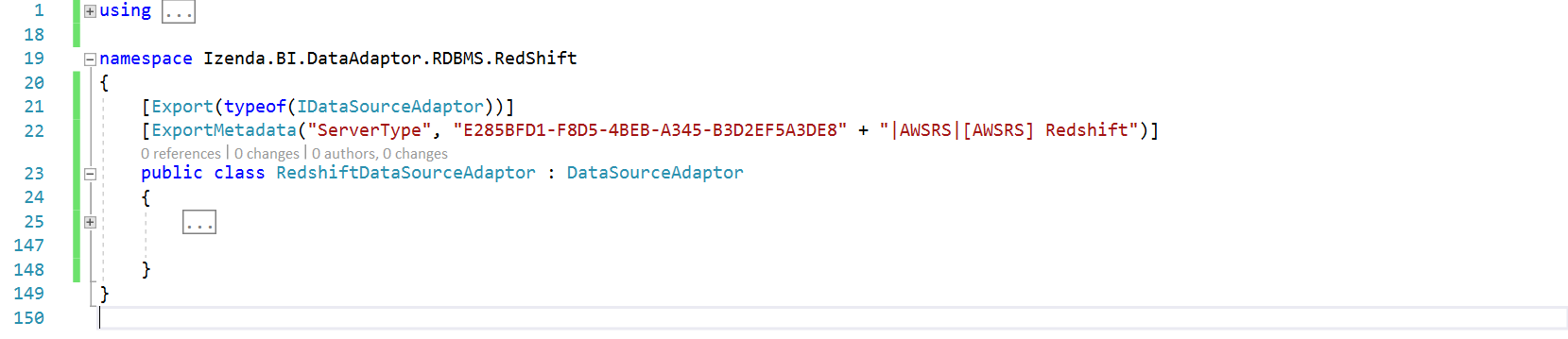


Figure 16: The first figure of implementation in RedshiftDataAdaptor class

We no need to implement all methods in IDataSourceAdaptor interface. we will re-use source code implementation from DataSourceAdaptor - the based adaptor class for any RDBMS. Sub sections after this will describes the way to override some methods form DataSourceAdapter class to provide necessary implementation to make data adaptor work properly with Izenda business layer. Most of other RDBMS also is able to work when applying the list of overriding method in other next sections.

To avoid confusion when following sample code, the adaptor will use below namespaces at all:

* Dapper
* Izenda.BI.DataAdaptor.RDBMS.CommandGenerators
* Izenda.BI.DataAdaptor.SQL.SchemaLoader
* Izenda.BI.Framework
* Izenda.BI.Framework.Components.SequenceWorkflows
* Izenda.BI.Framework.Models.Contexts
* Izenda.BI.Redshift
* Izenda.BI.QueryNormalizer.Redshift
* System.Collections.Generic
* System.ComponentModel.Composition
* Izenda.BI.Framework.Models
* Izenda.BI.DataAdaptor.RDBMS.Redshift.Constants
* Izenda.BI.Framework.Constants
* Izenda.BI.DataAdaptor.RDBMS.Redshift.CommandGenerators
* Npgsql
* Izenda.BI.Framework.Exceptions
* Izenda.BI.Resource
* System
* System.Text.RegularExpressions
* System.Linq

From that namespace, you are able to figure out which dependency components and technology will be used to implement the Redshift data adaptor.

### Override SchemaLoader and Connection Property

Override SchemaLoader and Connection property, return corresponding RedshiftSchemaLoader and RedshiftConnection we have created above.

public override ISchemaLoader SchemaLoader => new RedshiftSchemaLoader();

public override IConnection Connection => new RedshiftConnection();

### Override NormalizeQuery(string) Method

This method is used to normalize query to standard query of Redshift, it will convert Izenda Raw Query to standard query syntax of Redshift database.

Declare sequence workflow property as one-time initialization property:

ISequenceWorkflow<RedshiftQueryNormalizerActivity, QueryNormalizerContext> queryNormalizer;

protected ISequenceWorkflow<RedshiftQueryNormalizerActivity, QueryNormalizerContext> QueryNormalizer

{

get

{

if (queryNormalizer == null)

{

queryNormalizer = new SequenceWorkflow<RedshiftQueryNormalizerActivity, QueryNormalizerContext>();

}

return queryNormalizer;

}

}

Override method NormalizeQuery(string) as below:

protected override string NormalizeQuery(string query)

{

var normalizerContext = new QueryNormalizerContext { Query = query };

QueryNormalizer.Execute(normalizerContext);

return normalizerContext.Query;

}

### Override GetConnectionStringWithServerAndDatabaseName(string) Method

This method simply returns concatenation of database server and database name in specific format.

public override string GetConnectionStringWithServerAndDatabaseName(string connectionString)

{

var builder = new NpgsqlConnectionStringBuilder(connectionString);

if (builder.IntegratedSecurity)

{

return connectionString;

}

return string.Format("Server={0}; Database={1};", builder.Host, builder.Database);

}

### Override GetFirstValueInFilteredQuery Property

Property provides SQL syntax format to get first value in data source from database, first holder reserves for field list (columns) and second one is reserved for query source name (table or view).

public override string GetFirstValueInFilteredQuery

{

get

{

return @"SELECT ""{0}"" FROM {1} LIMIT 1";

}

}

### Override GetBaseDataTypes() Method

Return list of supported data types in database system from RedshiftSupportDataType class, which was configured before.

public override List<DatabaseDataType> GetBaseDataTypes()

{

var dataTypeAdaptor = new RedshiftSupportDataType();

return dataTypeAdaptor.GetBaseDataTypes();

}

### Override Query(string, string, object, int) Method

The method executes query on input parameter, the input query into this method is Izenda Raw Query, that requires transformation from Izenda Raw Query to standard query for Redshift before open connection then execute SQL statements.

public override IEnumerable<T> Query<T>(string connectionString, string query, object param = null, int queryTimeout = 60)

{

var standardQuery = NormalizeQuery(query);

using (var connection = OpenConnection(connectionString))

{

try

{

return connection.Query<T>(standardQuery, param, commandTimeout: queryTimeout);

}

catch (NpgsqlException ex)

{

Log($"QUERY: {standardQuery}", LogType.Error);

Log("Query error: " + ex.ToString(), LogType.Error);

var incorrectSyntax = new List<string> { "42000", "42601" };

if (incorrectSyntax.Contains(ex.Code))

{

//Hide detail incorect syntax message. This may contain sensitive information

throw new IzendaException(Messages.FusionQueryInCorrectSyntax);

}

throw new FusionException(Messages.FusionCanNotQueryData);

}

}

}

Note that the statement connection.Query is provided by Dapper library, if you do not use Dapper you have to implement this manually in sequence of normalize query, open connection, execute query and convert result set to IEnumerable<T> type.

### Override QueryMutiple(string, string, object, int, Action<SqlMapper.GridReader>) Method

This method supports to execute multiple query statements, which are spit by semicolon character (;). Izenda Business Layer usually calls this method to query reporting data and handle pagination at same time by SQL syntax to improve better performance.

Redshift supports multiple query so overriding this method is best choice.

In the RDDMS that does not support multiple query, you no need to override this method but have to modify call hierarchy to avoid calling this method by override method GetPaggingResult(string, string, FusionContextData), the overriding this method will be described in example of using ODBC Driver for Redshift in this document.

This method directly depends on Dapper Framework because of using type SqlMapper.GridReader on parameter, that leads adaptor component always depends on Dapper although in case you do not use that ORM Framework. If you do not use Dapper at all, lets avoid call this method by override GetPaggingResult method which is mentioned before, that means you no need to override this method.

Similar to Query method, this method also receive Izenda Raw Query in query parameter, to make it works properly on Redshift we must convert it to the query syntax that is supported by Redshift database by execute query normalizer step before open connection then execute query.

Note that the type is using on parameter SqlMapper.GridReader and the execution command connection.QueryMultiple are provided by Dapper Framework.

public override IEnumerable<T> QueryMultiple<T>(string connectionString, string query, object param = null, int queryTimeout = 60, Action<SqlMapper.GridReader> action = null)

{

var normalizerContext = new QueryNormalizerContext { Query = query };

QueryNormalizer.Execute(normalizerContext);

query = normalizerContext.Query;

using (var connection = OpenConnection(connectionString))

{

try

{

using (var data = connection.QueryMultiple(query, param, commandTimeout: queryTimeout))

{

var result = data.Read<T>();

action?.Invoke(data);

return result;

}

}

catch (NpgsqlException ex)

{

Log($"QUERY: {query}", LogType.Error);

Log("Query error: " + ex.ToString(), LogType.Error);

var incorrectSyntax = new List<string> { "42000", "42601" };

if (incorrectSyntax.Contains(ex.Code))

{

//Hide detail incorect syntax message. This may contain sensitive information

throw new IzendaException(Messages.FusionQueryInCorrectSyntax);

}

throw new FusionException(Messages.FusionCanNotQueryData);

}

}

}

### Override QueryTreeCommandGenerator Propery

Izenda system organizes query operators in the hierarchical tree called Query Tree, from query tree data source adaptor will generate SQL statements corresponding with targeted database system.

The based data adaptor class DataSourceAdaptor has already provided mechanism to generate SQL query statements from Query Tree. On the RedshiftDataSourceAdaptor class. we will indicate specific Query Tree Command Generator for Redshift.

Logically, whole command generators in data adaptor for RDBMS is following Visitor design pattern, based on logic case of fusion context the visitor class will visit over each operator then generator corresponding SQL syntax for the operand that has an operator presents for.

In adaptor project, create folder CommandGenerators then add RedshiftQueryTreeCommandGeneratorVisitor.cs and RedshiftQueryTreeCommandGenerator.cs.

The project structure until here:

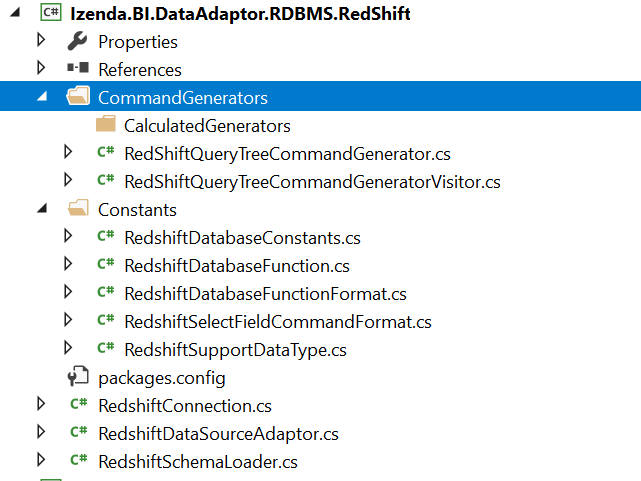


Figure 17: Structure of adaptor project at time create query tree command generator

Implementation of RedshiftQueryTreeCommandGenerator class:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.Framework.Components.QueryExpressionTree;

using Izenda.BI.Framework.Models;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedShiftQueryTreeCommandGenerator : QueryTreeCommandGenerator

{

protected override string GenerateCommand(QueryTreeNode operand, FusionContextData context)

{

var visitor = new RedShiftQueryTreeCommandGeneratorVisitor();

visitor.ContextData = context;

operand.Accept(visitor);

return visitor.NodeData[operand.Id];

}

protected override void ApplyAdvancedSetting(FusionContextData context)

{

}

}

}

The operand parameter in GenerateCommand method will be a specific operator, the visitor (RedshiftQueryTreeCommandGeneratorVisitor) based on the context will generate corresponding SQL statement for that operand.

The ApplyAdvancedSetting method must be overriding to clear base implementation that is not supported in Redshift, it generates SQL command to set, begin and commit transaction but that syntax does not work on Redshift database.

Implemenation of RedshiftQueryTreeCommandGeneratorVisitor class:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedShiftQueryTreeCommandGeneratorVisitor

: QueryTreeCommandGeneratorVisitor

{

//Will override operator command generator properties here

}

}

In next section, we will describe the way to identify what operator generator must be overridden and then declare overriding in this visitor class. Generally, this is placeholder to indicate which operator generator has to re-write for giving correct SQL syntax on targeted database system.

In RedshiftDataSourceAdaptor, override QueryTreeCommandGenerator property to indicate new generator for Redshift database.

public override QueryTreeCommandGenerator QueryTreeCommandGenerator

{

get

{

return new RedshiftQueryTreeCommandGenerator();

}

}

The call hierarchy in base adaptor DataSourceAdaptor class will use this generator to generate SQL command instead of default QueryTreeCommandGenerator.

## Implement Custom Operator Command Generators

### Operator Command Generators Are Required to Override

In order to generate operator SQL statement and look up for other command generators inside current adaptor library, the overriding for classes in table below are required. In other word, based adaptor DataSourceAdaptor uses .NET Reflection to find out dependency generators that are ExpressionCommandGenerator, FilterCommandGenerator and NullToEmptyConverter as well.

|  |  |  |
| --- | --- | --- |
| # | Operator Command Generator | Description |
| 1 | ConvertNullToEmptyOperator  \_CommandGenerator | Require overriding for .NET Reflection looks up generator to convert null to empty syntax. |
| 2 | GroupingOperatorCommandGenerator | Require overriding for reflection find out ExpressionCommandGenerator in adaptor library. |
| 3 | JoinOperatorCommandGenerator | Require overriding for reflection find out ExpressionCommandGenerator in adaptor library. |
| 4 | OperandCommandGenerator | Require overriding to provide overridden for SelectFieldCommandFormat |
| 5 | PagingOperatorCommandGenerator | Require overriding for reflection find out ExpressionCommandGenerator in adaptor library, and customize SQL paging syntax. |
| 6 | ProjectionOperatorCommandGenerator | Require overriding for reflection find out ExpressionCommandGenerator in adaptor library. |
| 7 | SelectFieldCommandGenerator | Require overriding for reflection find out ExpressionCommandGenerator in adaptor library. |
| 8 | SelectionOperatorCommandGenerator | Require overriding for reflection find out FilterCommandGenerator in adaptor library. |

Table 20: List of operator command generators have to be overridden on custom data adaptor

### RedhisftConvertNullToEmptyOperatorCommandGenerator

In CommandGenerators folder, add RedshiftConvertNullToEmptyOperatorCommandGenerator.cs and modify to inherit from ConvertNullToEmptyOperatorCommandGenerator class.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedshiftConvertNullToEmptyOperatorCommandGenerator : ConvertNullToEmptyOperatorCommandGenerator

{

public RedshiftConvertNullToEmptyOperatorCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override SelectFieldCommandGenerator SelectFieldCommandGenerator

{

get

{

return new RedShiftSelectFieldCommandGenerator(visitor);

}

}

}

}

This class depends on RedshiftSelectFieldCommandGenerator. We will create it later.

### RedshiftGroupingOperatorCommandGenerator

In CommandGenerators folder, add RedshiftGroupingOperatorCommandGenerator.cs and modify to inherit from GroupingOperatorCommandGenerator class.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedshiftGroupingOperatorCommandGenerator : GroupingOperatorCommandGenerator

{

public RedshiftGroupingOperatorCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override string DefaultGroupBy

{

get

{

return string.Empty;

}

}

}

}

### RedshiftJoinOperatorCommandGenerator

In CommandGenerators folder, add RedshiftJoinOperatorCommandGenerator.cs and modify to inherit from JoinOperatorCommandGenerator class.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.DataAdaptor.RDBMS.Constants;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedshiftJoinOperatorCommandGenerator : JoinOperatorCommandGenerator

{

public RedshiftJoinOperatorCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

protected override SelectFieldCommandFormat SelectFieldCommandFormat

{

get

{

if (selectFieldCommandFormat == null)

{

selectFieldCommandFormat = new RedShiftSelectFieldCommandFormat();

}

return selectFieldCommandFormat;

}

}

}

}

### RedshiftOperandCommandGenerator

In CommandGenerators folder, add RedshiftOperandCommandGenerator.cs and modify to inherit from OperandCommandGenerator class.

This generator depends on expression command generator RedshiftDateAddTokenCommandGenerator and RedshiftCastTokenCommandGenerator, those generator class will be implemented in next section.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.DataAdaptor.RDBMS.Constants;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedshiftOperandCommandGenerator : OperandCommandGenerator

{

private SelectFieldCommandFormat selectFieldCommandFormat;

protected override SelectFieldCommandFormat SelectFieldCommandFormat

{

get

{

if (selectFieldCommandFormat == null)

{

selectFieldCommandFormat = new RedshiftSelectFieldCommandFormat();

}

return selectFieldCommandFormat;

}

}

public override CastTokenCommandGenerator CastTokenCommandGenerator

{

get

{

return new RedshiftCastTokenCommandGenerator(null);

}

}

public RedshiftOperandCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

}

}

### RedshiftPagingOperatorCommandGenerator

In CommandGenerators folder, add RedshiftPagingOperatorCommandGenerator.cs and modify to inherit from PagingOperatorCommandGenerator class.

Overriding of this method has two purpose that supports to find out ExpressionCommandGenerator inside adaptor component and provide SQL paging syntax. In Redshift, it uses LIMIT and OFFSET keyword to handle that.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.Framework.Components.QueryExpressionTree;

using Izenda.BI.Framework.Components.QueryExpressionTree.Operator;

using Izenda.BI.Framework.Constants;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedshiftPagingOperatorCommandGenerator : PagingOperatorCommandGenerator

{

public RedshiftPagingOperatorCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override string GenerateCommand(QueryTreeNode treeNode, string childCommand)

{

var pagingOperator = (PagingOperator)treeNode;

// Modify sortingOperator and selectOperator

childCommand = childCommand.Replace(PlaceHolder.PagingField, "");

var query = @"SELECT \* FROM({0}) x LIMIT {1} OFFSET {2}";

var paging = visitor.ContextData.Paging;

int offset = (paging.PageIndex - 1) \* paging.PageSize;

return string.Format(query, childCommand, paging.PageSize, offset);

}

}

}

### RedshiftProjectionOperatorCommandGenerator

In CommandGenerators folder, add RedshiftProjectionOperatorCommandGenerator.cs and modify to inherit from ProjectionOperatorCommandGenerator class.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedshiftProjectionOperatorCommandGenerator : ProjectionOperatorCommandGenerator

{

public RedshiftProjectionOperatorCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

}

}

Overriding this method only has one purpose that indicates .NET Reflection find ExpressionCommandGenerator and NullToEmptyConverter in adaptor library.

### RedshiftSelectFieldCommandGenerator

In CommandGenerators folder, add RedshiftSelectFieldCommandGenerator.cs and modify to inherit from SelectFieldCommandGenerator class.

Overriding this class to indicate library and namespace follows convention to look up ExpressionCommandGenerator and NullToEmptyConverter generator.

The base adaptor looks up NullToEmptyConverter in namespace Izenda.BI.DataAdaptor.RDBMS.Redshift.CommandGenerators.ConvertNullToEmptyGenerator.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedshiftSelectFieldCommandGenerator : SelectFieldCommandGenerator

{

public RedshiftSelectFieldCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

protected override string SpecificNamespaceForConvertNullToEmpty

{

get

{

return string.Format(ConvertNullToEmptyNamespace, ".RedShift", ".RedShift", this.GetType().Assembly.GetName().Name);

}

}

}

}

In folder CommandGenerators, create sub-folder ConvertNullToEmptyGenerator.

Add RedshiftNullToEmptyConvertor.cs and modify its class inherit NullToEmptyConvertor:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.ConvertNullToEmptyGenerator;

using Izenda.BI.Framework.Models;

using System.Text;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.ConvertNullToEmptyGenerator

{

public class RedshiftNullToEmptyConvertor : NullToEmptyConvertor

{

public override StringBuilder GenerateSelectCommand(StringBuilder stringBuilder, string fieldAlias, QuerySourceField querySourceField = null)

{

if (querySourceField != null)

{

return stringBuilder.AppendFormat(@"NVL([[{0}]],'') AS ""{1}"",", querySourceField.Name, fieldAlias);

}

return stringBuilder.AppendFormat(@"NVL([[{0}]],'') AS ""{1}"",", fieldAlias, fieldAlias);

}

public override StringBuilder GenerateSelectCommandFormat(StringBuilder stringBuilder, string fieldAlias, QuerySourceField querySourceField = null)

{

if (querySourceField != null)

{

return stringBuilder.AppendFormat(@"NVL({0},'') AS ""{1}"",", querySourceField.Name, fieldAlias);

}

return stringBuilder.AppendFormat(@"NVL({0},'') AS ""{1}"",", fieldAlias, fieldAlias);

}

}

}

### RedshiftSelectionOperatorCommandGenerator

In CommandGenerators folder, add RedshiftSelectionOperatorCommandGenerator.cs and modify to inherit from SelectionOperatorCommandGenerator class.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedshiftSelectionOperatorCommandGenerator : SelectionOperatorCommandGenerator

{

public RedshiftSelectionOperatorCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

}

}

Overriding this intends for reflection purpose such as load correct FilterCommandGenerator.

### Overriding Other Operator Command Generators

Most Operator Command Generator in based adaptor component Izenda.BI.DataAdaptor.RDBMS was developed to adapt many relational database system as much as possible that means the generated SQL syntax can run on may RDBMS, but some of them can be different on your targeted RDBMS that case required you manually override the generator and provide correct SQL syntax on your database.

Your responsibility is comparing between SQL syntax is generated on based adaptor and standard syntax that your database system supports, if they are different you must override generator to generate correct SQL syntax.

One example for overriding generator and provide new SQL syntax is ResultLimitOperator, Izenda based adaptors generate SQL as SELECT TOP clause but we want to using LIMIT clause in Redshift to handle that logic so that require to override ResultLimitOperatorCommandGenerator.

In CommandGenerators folder, add RedshiftResultLimitOperatorCommandGenerator.cs and modify to inherit from ResultLimitOperatorCommandGenerator class then implement GenerateCommand method to provide new SQL syntax.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.Framework.Components.QueryExpressionTree;

using Izenda.BI.Framework.Components.QueryExpressionTree.Operator;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators

{

public class RedshiftResultLimitOperatorCommandGenerator : ResultLimitOperatorCommandGenerator

{

public RedshiftResultLimitOperatorCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override string GenerateCommand(QueryTreeNode treeNode, string childCommand)

{

var resultLimitOperator = treeNode as ResultLimitOperator;

var query = @"SELECT \* FROM ({0}) X LIMIT {1}";

var resultLimit = string.Format(query, childCommand, resultLimitOperator.Limit);

visitor.ContextData.TempData["resultLimit"] = resultLimit;

return resultLimit;

}

}

}

### Register Operator Command Generator in Visitor Class

To indicate adaptor uses new generator class we have to register them into visitor class. That simply override corresponding property for each Operator Command Generator and return instance of new generator.

Open RedshiftQueryTreeCommandGenerator.cs and add overridden properties like below:

public class RedshiftQueryTreeCommandGeneratorVisitor : QueryTreeCommandGeneratorVisitor

{

public override PagingOperatorCommandGenerator PagingOperatorCommandGenerator

{

get

{

return new RedshiftPagingOperatorCommandGenerator(this);

}

}

public override OperandCommandGenerator OperandCommandGenerator

{

get

{

return new RedshiftOperandCommandGenerator(this);

}

}

public override ProjectionOperatorCommandGenerator ProjectionOperatorCommandGenerator

{

get

{

return new RedshiftProjectionOperatorCommandGenerator(this);

}

}

public override SelectionOperatorCommandGenerator SelectionOperatorCommandGenerator

{

get

{

return new RedshiftSelectionOperatorCommandGenerator(this);

}

}

public override GroupingOperatorCommandGenerator GroupingOperatorCommandGenerator

{

get

{

return new RedshiftGroupingOperatorCommandGenerator(this);

}

}

public override ResultLimitOperatorCommandGenerator ResultLimitOperatorCommandGenerator

{

get

{

return new RedshiftResultLimitOperatorCommandGenerator(this);

}

}

public override JoinOperatorCommandGenerator JoinOperatorCommandGenerator

{

get

{

return new RedshiftJoinOperatorCommandGenerator(this);

}

}

public override ConvertNullToEmptyOperatorCommandGenerator ConvertNullToEmptyOperatorCommandGenerator

{

get

{

return new RedshiftConvertNullToEmptyOperatorCommandGenerator(this);

}

}

}

## Implement Expression Command Generator

### Analysis Overriding Expression Command Generator

This type of generator will generate SQL statement for calculated function such as AVG, SUM, ROUND...etc. It also generates SQL for function, which are list in DatabaseFunction class.

All Expression Command Generator will inherit IExpressionCommandGenerator that is identifier of this generator type.

Generally, the calculated function and token are similar on most of database system. The base adaptor has already generated SQL to adapt most of RDBMSs except some calculation functions relate to date time format because it frequently is different between database systems. In Redshift database, we have DatePart and IsNull function that uses new syntax in RedshiftDataFunction. Cause of that, the generator for that function will overridden to provide the correct syntax and function name, and then we have to override the DateDartTokenCommandGenerator and IsNullTokenCommandGenerator in this adaptor.

Other generators requires to override are CastTokenCommandGenerator and ConvertTokenCommandGenerator because it depends on supported data types on each database system, it is RedshiftSupportDatType class.

The overriding of DatabaseConstants will configure for datepart and timepart identifier for date time value. For Redshift we no need to override the related generators, but if you are developing an adaptor works with other database, this configuration maybe change then you have to override generators inherits from DateFunctionTokenCommandGenerator are DatePartTokenCommandGenerator, DateAddCommandGenerator and DateDiffFunctionCommandGenerator.

The generator LpadTokenCommandGenerator is using Izenda Raw method IZENDA\_CONCAT to combine padding character to align date time value (for example, the day 1st will be formatted as 01st) but Redshift does not support CONCAT function with multiple parameters more than 2. Reshift contains better function is LPAD. That seems better to override this generator to replace complex logic with IZENDA\_CONCAT by using LPAD function.

We also override MappingTokenCommandGenerator to indicate correct adaptor identifier (in this adaptor, it is E285BFD1-F8D5-4BEB-A345-B3D2EF5A3DE8).

Summary of above analyses, it requires to override Expression Command Generators in table below:

|  |  |  |
| --- | --- | --- |
| # | Generator Name | Overriding Purpose |
| 1 | DateDartTokenCommandGenerator | Require to use new DatePart function: date\_part |
| 2 | IsNullTokenCommandGenerator | Require to use new IsNull function: NVL |
| 2 | CastTokenCommandGenerator | Require to use data types configuration in RedshiftSupportDatType |
| 3 | ConvertTokenCommandGenerator | Require to use data types configuration in RedshiftSupportDatType |
| 4 | LpadTokenCommandGenerator | Override to replace complex logic of using IZENDA\_CONCATE by more simple one LPAD function |
| 5 | MappingTokenCommandGenerator | Provide adaptor identifier GUID of Redshift data adaptor |

Table 21: List of expression command generator must be overridden in Redshift data adaptor

Following next sections, we will implement one by one above expression command generator and its consumer visitor class.

### Add generator and visitor class

In CommandGenerators folder, add new folder CalculatedGenerators.

In CalculatedGenerators add RedshiftExpressionCommandGenerator.cs below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.CalculatedGenerators

{

public class RedshiftExpressionCommandGenerator : ExpressionCommandGenerator

{

public override ExpressionCommandGeneratorVisitor ExpressionCommandGeneratorVisitor

{

get

{

return new RedshiftExpressionCommandGeneratorVisitor();

}

}

}

}

Add RedshiftExpressionCommandGeneratorVisitor.cs in same CalculatedGenerators folder:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.CalculatedGenerators

{

public class RedshiftExpressionCommandGeneratorVisitor : ExpressionCommandGeneratorVisitor

{

//Override Epxression Command Generators here

}

}

The based adaptor will automatically look up RedshiftExpressionCommandGenerator in current library then construct to use it. Izenda uses visitor design pattern to implement this generator class. We will override property corresponding with each expression in visitor class to provide new correct SQL syntax if necessary.

### Override DateDartTokenCommandGenerator

In CalucatedGenerators folder, add RedshiftDateDartTokenCommandGenerator.cs and modify class inherits from DateDartTokenCommandGenerator.

As analysis above, overriding this generator to take effect of date\_part function in Redshift instead datepart function in based adaptor.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.DataAdaptor.RDBMS.Constants;

using Izenda.BI.DataAdaptor.RDBMS.RedShift.Constants;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.CalculatedGenerators

{

public class RedshiftDateDartTokenCommandGenerator : DatePartTokenCommandGenerator

{

public RedshiftDateDartTokenCommandGenerator(ExpressionCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override DatabaseFunction DatabaseFunction

{

get

{

return new RedshiftDatabaseFunction();

}

}

public override DatabaseConstants DatabaseConstants

{

get

{

return new RedshiftDatabaseConstants();

}

}

}

}

### Override IsNullTokenCommandGenerator

In CalculatedGenerator folder, add RedshiftIsNullTokenCommandGenerator.cs and lets it inherit from IsNullTokenCommandGenerator:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.DataAdaptor.RDBMS.Constants;

using Izenda.BI.DataAdaptor.RDBMS.RedShift.Constants;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.CalculatedGenerators

{

public class RedshiftIsNullTokenCommandGenerator : IsNullTokenCommandGenerator

{

public RedshiftIsNullTokenCommandGenerator(ExpressionCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override DatabaseFunction DatabaseFunction

{

get

{

return new RedshiftDatabaseFunction();

}

}

}

}

This class override DatabaseFunction property to take effect of new IsNull function in Redshift, the NVL function.

### Override CastTokenCommandGenerator

In CalculatedGenerator folder, add RedshiftCastTokenCommandGenerator.cs and modify to it inherits CastTokenCommandGenerator:

Override DatabaseSupportDataType property to use new data type configuration in RedshiftSupportDataType class.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.DataAdaptor.RDBMS.Constants;

using Izenda.BI.DataAdaptor.RDBMS.RedShift.Constants;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.CalculatedGenerators

{

public class RedshiftCastTokenCommandGenerator : CastTokenCommandGenerator

{

public RedshiftCastTokenCommandGenerator(ExpressionCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override DatabaseSupportDataType DatabaseSupportDataType

{

get

{

return new RedShiftSupportDataType();

}

}

}

}

### Override ConvertTokenCommandGenerator

In CalculatedGenerator folder, add RedshiftConvertTokenCommandGenerator.cs and modify to it inherits ConvertTokenCommandGenerator:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.DataAdaptor.RDBMS.Constants;

using Izenda.BI.DataAdaptor.RDBMS.RedShift.Constants;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.CalculatedGenerators

{

public class RedshiftConvertTokenCommandGenerator : ConvertTokenCommandGenerator

{

public RedshiftConvertTokenCommandGenerator(ExpressionCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override DatabaseSupportDataType DatabaseSupportDataType

{

get

{

return new RedShiftSupportDataType();

}

}

}

}

Override this generator has same purpose with CastTokenCommandGenerator.

### Override LpadTokenCommandGenerator

In CalculatedGenerator folder, add RedshiftLpadTokenCommandGenerator.cs and modify to it inherits LpadTokenCommandGenerator

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.Framework.Components.ExpressionEvaluations;

using Izenda.BI.Framework.Components.ExpressionEvaluations.Functions;

using System;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.CalculatedGenerators

{

public class RedshiftLpadTokenCommandGenerator : LpadTokenCommandGenerator

{

public RedshiftLpadTokenCommandGenerator(ExpressionCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override string GenerateSelfCommand(Token token)

{

var functionToken = token as DatabaseFunctionToken;

var subTrees = functionToken.SubTrees;

string expression = visitor.NodeData[(subTrees["1"] as Token).TokenId];

var length = visitor.NodeData[(subTrees["2"] as Token).TokenId];

var paddingCharactor = visitor.NodeData[(subTrees["3"] as Token).TokenId];

var lengthValue = Convert.ToInt32(length);

return string.Format("LPAD({0},{1},{2})", expression, length, paddingCharactor);

}

}

}

Override this generator to replace using of IZENDA\_CONCAT by LPAD function.

### Override MappingTokenCommandGenerator

In CalculatedGenerator folder, add RedshiftMappingTokenCommandGenerator.cs and modify to it inherits MappingTokenCommandGenerator:

Override ServerTypeId to indicate correct Data Source Adaptor identifier.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.CalculatedGenerators

{

public class RedshiftMappingTokenCommandGenerator : MappingTokenCommandGenerator

{

public RedshiftMappingTokenCommandGenerator(ExpressionCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override string ServerTypeId

{

get

{

return "E285BFD1-F8D5-4BEB-A345-B3D2EF5A3DE8";

}

}

}

}

### Register Overriding Expression Command Generator

To indicate which Expression Command Generator class we have to register overriding that generator in Visitor class. Open RedshiftExpressionCommandGeneratorVisitor.cs and declare overriding property for each above generator.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

using Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.CalculatedGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.Redshift.CommandGenerators.CalculatedGenerators

{

public class RedshiftExpressionCommandGeneratorVisitor : ExpressionCommandGeneratorVisitor

{

public override ConvertTokenCommandGenerator ConvertTokenCommandGenerator

{

get

{

return new RedshiftConvertTokenCommandGenerator(this);

}

}

public override CastTokenCommandGenerator CastTokenCommandGenerator

{

get

{

return new RedshiftCastTokenCommandGenerator(this);

}

}

public override DatePartTokenCommandGenerator DatePartTokenCommandGenerator

{

get

{

return new RedshiftDatePartTokenCommandGenerator(this);

}

}

public override IsNullTokenCommandGenerator IsNullTokenCommandGenerator

{

get

{

return new RedshiftIsNullTokenCommandGenerator(this);

}

}

public override MappingTokenCommandGenerator MappingTokenCommandGenerator

{

get

{

return new RedshiftMappingTokenCommandGenerator(this);

}

}

public override LpadTokenCommandGenerator LpadTokenCommandGenerator

{

get

{

return new RedshiftLpadTokenCommandGenerator(this);

}

}

}

}

## Implement Filter Generator

### Overview of Filter Generators

The generator inherits FilterGeneratorBase class will generate SQL statement for filtering on Izenda report. Izenda system automatically look up all generators by reflection.

Logically, this type of generator will generator SQL syntax after WHERE clause.

We will override all generators to provide full functionality relate to report filtering because that feature is predefined on Izenda UI. If one of generator in table below is missing, corresponding function on Izenda Report will not work.

|  |  |  |
| --- | --- | --- |
| # | Generator Name | Description |
| 1 | BlankFilterGenerator | Check the field whether is blank or not |
| 2 | BooleanFilterGenerator | Compare a field with Boolean value, true or false |
| 3 | ComparisionFilterGenerator | Generate comparison operand between two fields, the operand includes >, >=, <, <= and != |
| 4 | DateTimeFilterGenerator | Generator filter statement for date time value |
| 5 | EquivalenceFilterGenerator | Check equivalence between two field |
| 6 | FieldComparisionFilterGenerator | Similar to ComparisionFilterGenerator |
| 7 | StringFilterGenerator | Generator string filter in LIKE clause with wildcard character like \_, % and combination of [, ], ! and ^ |
| 8 | NotBlankFilterGenerator | Check the field whether is not blank or blank |
| 9 | ThreeDotsFilterGenerator | General filter with specific condition, which is defined by end-user |

Table 22: List of build in filter generator in Izenda system

The other purpose to override all filter generator is intending to load correct DatabaeFunctionFormat configuration, which is RedshiftDatabaeFunctionFormat in this adaptor component.

### Implement RedshiftFilterCommandGenerator class

Izenda also look up and create instance of this class to use by using .NET Reflection, that means we no need to declare using statement at all. It will inherit FilterCommandGenerator class.

In CommandGenerators folder, create sub-folder FilerGenerators and then add RedshiftFilterCommandGenerator.cs:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

using Izenda.BI.Framework.Models;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftFilterCommandGenerator : FilterCommandGenerator

{

public RedshiftFilterCommandGenerator(FusionContextData contextData) : base(contextData)

{

}

protected override string DatabasePrefix

{

get

{

return "RedShift";

}

}

}

}

It needs to override DatabasePrefix to indicate correct namespace for looking all filter generators. The full namespace in this adaptor will be Izenda.BI.DataAdaptor.RDBMS.Redshift.CommandGenerators.FilterGenerators. Implementation of FilterCommandGenerator define and using that convention then that makes this naming convention is required by default.

### Override BlankFilterGenerator

In FilerGenerators folder, Add RedshiftBlankFilterGenerator.cs like below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftBlankFilterGenerator : BlankFilterGenerator

{

//This is intended for reflection purpose such as load correct DatabaseFunctionFormat

}

}

### Override BooleanFilterGenerator

In FilerGenerators folder, Add RedshiftBooleanFilterGenerator.cs like below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftBooleanFiilterGenerator : BooleanFiilterGenerator

{

//This is intended for reflection purpose such as load correct DatabaseFunctionFormat

}

}

### Override ComparisionFilterGenerator

In FilerGenerators folder, Add RedshiftComparisionFilterGenerator.cs like below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftComparisionFilterGenerator : ComparisionFilterGenerator

{

//This is intended for reflection purpose such as load correct DatabaseFunctionFormat

}

}

### Override DateTimeFilterGenerator

In FilerGenerators folder, Add RedshiftDateTimeFilterGenerator.cs like below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftDateTimeFilterGenerator : DateTimeFilterGenerator

{

protected override string DatabasePrefix

{

get

{

return "Redshift";

}

}

}

}

This generator create SQL statement for date time value. It requires to extract many part value of date time value such as year, quarter, month, week, day and time. That leads it depends on list of other generators in table below. By default, we also declare overriding for all of this generator to provide full functionality relate to handle date time value, which are predefined in Izenda System.

In the base class DateTimeFilterGenerator, it looks up all below generators in the namespace Izenda.BI.DataAdaptor.RDBMS.Redshift.CommandGenerators.FilterGenerators.DateTimeGenerators. So this is required convention when implement adaptor component.

|  |  |  |
| --- | --- | --- |
| # | Generator Name | Description |
| 1 | RedshiftCalendarMonth | Calendar month filter |
| 2 | ReshiftCalendarQuarter | Calendar quarter filter |
| 3 | ReshiftCalendarWeek | Calendar week filter |
| 4 | ReshiftCalendarYear | Calendar year filter |
| 5 | ReshiftDay | Filter by day of date time |
| 6 | ReshiftFiscalQuarter | Filter by fiscal quarter of date time |
| 7 | ReshiftFiscalYear | Filter by fiscal year of date time |
| 8 | ReshiftLongTermPeriod | Filter by period of time |
| 9 | ReshiftSecond | Filter by second value of date time |

Table 23: List of date time filtering generator

Declare overriding for all above date time filtering generator is same. It follows convention that begin with Redshift and following by base class name. For example, class RedshiftCalendarMonth will inherit base class CalendarMonth. That rule will come across all other class on table.

In FilterGenerator folder, add sub-folder DateTimeGenerator and then add RedshiftCalendarMonth.cs:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators.DateTimeGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators.DateTimeGenerator

{

public class RedshiftCalendarMonth : CalendarMonth

{

//This is intended for reflection purpose such as load correct DatabaseFunctionFormat

}

}

We only describe for only RedshiftCalendarMonth class, but you can do similar for remaining date time filtering generators. The result look like below:

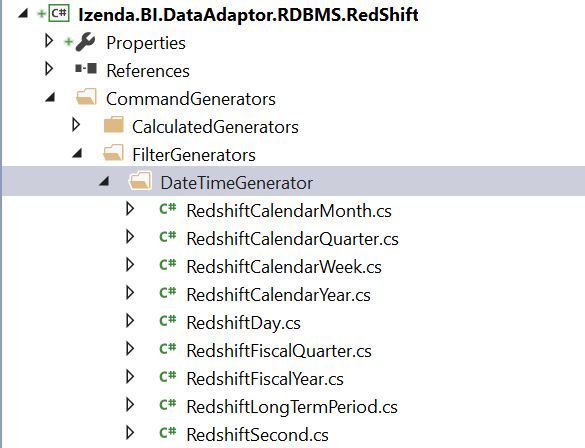


Figure 18: All date time filtering generator in adaptor project

### Override EquivalenceFilterGenerator

In FilerGenerators folder, Add RedshiftEquivalenceFilterGenerator.cs like below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftEquivalenceFilterGenerator : EquivalenceFilterGenerator

{

//This is intended for reflection purpose such as load correct DatabaseFunctionFormat

}

}

### Override FieldComparisonFilterGenerator

In FilerGenerators folder, Add RedshiftFieldComparisionFilterGenerator.cs like below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftFieldComparisonFilterGenerator : FieldComparisonFilterGenerator

{

//This is intended for reflection purpose such as load correct DatabaseFunctionFormat

}

}

### Override StringFilterGenerator

In FilerGenerators folder, Add RedshiftStringFilterGenerator.cs like below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftStringFilterGenerator : StringFilterGenerator

{

//This is intended for reflection purpose such as load correct DatabaseFunctionFormat

}

}

### Override NotBlankFilterGenerator

In FilerGenerators folder, Add RedshiftNotBlankFilterGenerator.cs like below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftNotBlankFilterGenerator : NotBlankFilterGenerator

{

//This is intended for reflection purpose such as load correct DatabaseFunctionFormat

}

}

### Override ThreeDotsFilterGenerator

In FilerGenerators folder, Add RedshiftThreeDotsFilterGenerator.cs like below:

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators.FilterGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.RedShift.CommandGenerators.FilterGenerators

{

public class RedshiftThreeDotsFilterGenerator : ThreeDotsFilterGenerator

{

//This is intended for reflection purpose such as load correct DatabaseFunctionFormat

}

}

# Create Redshift Data Source Adaptor Using ODBC Driver

## Introduction and Analysis

Generally, create Data Source Adaptor using ODBC Drive will be similar to create an adaptor in previous section, except that instead of using specific Data Provider library we will use ODBC Drive as Data Provider. That comes to using OdbcConnection class in System.Data.Odbc of .NET Framework. OdbcConnection class will take responsibility to open connection and explore database schema structure as well.

This section only describes the different when analyzing and developing an adaptor using ODBC Driver. Other steps to do and analysis will be similar when applying for other RDBMS. That means you also create logical units includes Data Adaptor, Connection, Schema Loader, Operator Command Generator, Expression Command Generator, Filter Generator and Specific Configuration for database data types and SQL functions.

According on feature of ODBC that it uses question mark (?) is placeholder for parameters. While Izenda base adaptor implement following named parameters (@parameterName) it will come to few issues:

* Parameter placeholder generated by base adaptor is not suitable on ODBC.
* When using named parameters, it allows re-using parameter name in multiple place in SQL query. That mean you can pass one parameter value but can use it multiple times. However, with question mark of ODBC, you always have to pass number of parameters equals the time using parameter placeholder in SQL query. However, the base adaptor was not implementation like that. It uses named parameter.

Above are two major issues that requires we have to resolve on an ODBC Adaptor.

The thirst issue is coming with using of Dapper when execute query on OdbcConnection object. Because Dapper automatically maps parameter data type, then in some case due the limitation of implementation of ODBC Driver it detects wrong data type to build parameter object before execute SQL statement. That leads failed query error. The SQL Server ODBC Driver faced this issue when we write this document. To resolve that issue you can avoid using Dapper and manually building parameter list and executing SQL statement directly on OdbcConnection object.

This guideline introduces how to implement Izenda Data Source Adaptor for Redshift database. We will avoid three issues know issues one by one. Izenda based adaptor component uses NameFormater class to allow overriding generator avoid first issue relate the named parameter. The second issue of multiple parameter, we will redirect call hierarchy to avoid calling of QueryMultiple in IDataSourceAdaptor interface and override LpadTokenCommandGenerator to reject using of multiple parameter inside them. The thirst issue does not happen on Redshift ODBC Driver.

The full example source is available on Github repository at <https://github.com/Izenda7Series/IzendaCustomDataAdaptors/tree/odbcdataadapter>. In that example we split source code into two projects to demonstrate what are specific for ODBC implementation and what for specific Redshift database. However, in your custom adaptor we suggest to merge all that code into only one adaptor project to make it simple to maintenance and troubleshooting as well.

## Install and Configure Amazon Redshift ODBC Driver

Following instruction in the link <https://docs.aws.amazon.com/redshift/latest/mgmt/install-odbc-driver-windows.html> to download and install Redshift ODBC Driver.

The way to get connection string to Redshift database by using ODBC here <https://docs.aws.amazon.com/redshift/latest/mgmt/configure-odbc-connection.html>

To verify installing of ODBC on Windows, open ODBC Data Source Administrator and check that it is available on Driver tab:

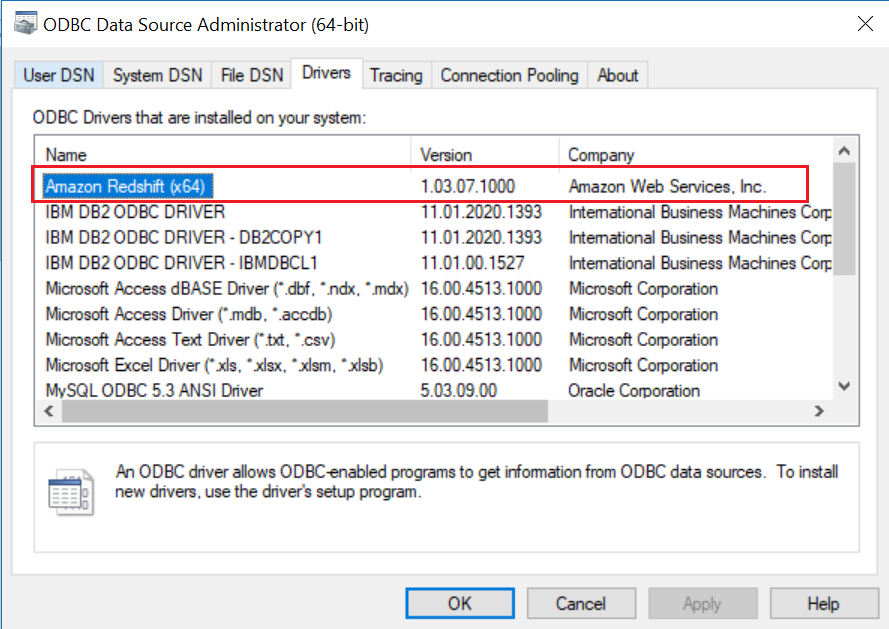


Figure 19: The installed Redshift ODBC Driver on Windows

## Implement Adaptor with OdbcConnection

### Overriding NameFormatter in Command Generator

This requires for avoiding the first issue mentioned above. By provide new instance of NameFormater and declare overriding using of it in command generator, we can replace named parameter in generated SQL by question mark (?) in ODBC syntax.

In CommandGenerators folder, create sub-folder Formater and then add ODBCNameFormater.cs:

Based adaptor uses the method FormatParater to get parameter sign normally combines @ and parameter name. For ODBC it just is question mark, so we return question mark character only.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.ODBC.CommandGenerators

{

public class ODBCNameFormatter : NameFormatter

{

public override string FormatParameter(string parameterName)

{

return "?";

}

}

}

There are three place in based adaptor component that NameFormater is using for build parameter name that are in below table:

|  |  |  |
| --- | --- | --- |
| # | Generator Name | Description |
| 1 | ValueTokenCommandGenerator | It is an Expression Command Generator, inside CalculatedGenerators folder. The using of this generator will be avoiding in use if we override LpadTokenCommandGenerator. Actually, in Redshift adaptor will always override that generator instead of this one. That will provide better performance as well as logical processing. We also recommend you override this generator in your custom adaptor component. |
| 2 | JoinOperatorCommandGenerator | It is an Operator Command Generator inside CommandGenerators folder. The generator that we always have to override in a new adaptor. |
| 3 | FilterCommandGenerator | The main generator for other Filter Command Generator |

Figure 20: The place require overriding NameFormater for providing correct placeholder in SQL query

Overriding NameFormater is similar on all used generator. We just describe only one for JoinOperatorCommandGenerator as example.

using Izenda.BI.DataAdaptor.RDBMS.CommandGenerators;

namespace Izenda.BI.DataAdaptor.RDBMS.ODBC.CommandGenerators

{

public class ODBCJoinOperatorCommandGenerator : JoinOperatorCommandGenerator

{

public ODBCJoinOperatorCommandGenerator(QueryTreeCommandGeneratorVisitor visitor) : base(visitor)

{

}

public override NameFormatter NameFormatter => new ODBCNameFormatter();

}

}

That simple return new instance of ODBCNameFormatter, which we created before.

### Redshift Connection class in ODBC

Instead of using Data Provider, which is provided by RDBMS’s vendor, we will use ODBC Driver to connect to database. For that reason, all database query relate to connection in an ODBC adaptor will be handled by OdbcConnection class in .NET Framework.

Below is implementation of ODBC Connection class, which inherit IConnection in Izenda.BI.Framework:

using Izenda.BI.Framework;

using System;

using Izenda.BI.Framework.Models;

using System.Data;

using Izenda.BI.Framework.Constants;

using System.Data.Odbc;

namespace Izenda.Synergy.DataAdaptor.RDBMS.ODBC

{

public class ODBCConnection : IConnection

{

public string GetDatabaseName(string connectionString)

{

var builder = new OdbcConnectionStringBuilder(connectionString);

return builder["database"].ToString();

}

public string GetDatabaseServer(string connectionString)

{

var builder = new OdbcConnectionStringBuilder(connectionString);

return builder["server"].ToString();

}

public string GetUserName(string connectionString)

{

var builder = new OdbcConnectionStringBuilder(connectionString);

return builder["uid"].ToString();

}

public virtual IDbConnection OpenConnection(string connectionString)

{

var connection = new OdbcConnection(connectionString);

connection.Open();

return connection;

}

public virtual ConnectionStatus TestConnection(Guid serverType, string connectionString)

{

var result = new ConnectionStatus

{

Status = ConnectDBStatus.Success,

ConnectionString = connectionString

};

try

{

using (var connection = OpenConnection(connectionString))

{

result.Status = ConnectDBStatus.Success;

return result;

}

}

catch (OdbcException ex)

{

result.Status = ConnectDBStatus.Fail;

result.ErrorMessage = ex.Message;

}

return result;

}

}

}

It uses OdbcConnectionStringBuilder to parse database name, server name and user name, and uses OdbcConnection to open connection to database.

### RedhisftSchemaLoader class in ODBC

OdbcConnection class provides GetSchema method to query database structure, then schema loader will use that feature to explore fields in database then convert to Izenda abstraction classes.

The idea to implement ISchemaLoader class is similar with description in section [Implement schema loader class](#_Implement_schema_loader), except we do not query database structure directly by query statement.

The full implementation of this class is at <https://github.com/Izenda7Series/IzendaCustomDataAdaptors/blob/odbcdataadapter/ODBC/Izenda.Synergy.DataAdaptor.RDBMS.ODBC/ODBCSchemaLoader.cs>.

### RedshiftDataAdaptor class in ODBC

This class implementation is similar in section [Implement Data Source Adaptor class](#_Implement_Data_Source).

Because the second issue relates to multiple parameter in SQL query in ODBC, we have to modify implementation in this class to avoid that issue. Generally idea is using flag IsSupportedMultipleQuery to identify the adaptor whether is supported handle multiple query or not. If it is not, we will call customize paging method with separated queries one for getting data and one for getting paging information.

Overriding GetPagingResult in adaptor class to avoid call hierarchy to QueryMutiple method in Data Source Adaptor class:

protected override IEnumerable<dynamic> GetPagingResult(string connectionString, string query, FusionContextData context)

{

if (IsSupportedMultipleQuery)

{

return base.GetPagingResult(connectionString, query, context);

}

// If not supported mutiple query, execute two query separately

var result = Query<dynamic>(connectionString, query,

context.Parameters, context.PerformanceSetting.QueryTimeoutValue);

var countTotalQuery = CountTotalCommand(context);

var totalRows = Query<int>(

connectionString,

countTotalQuery,

context.Parameters,

context.PerformanceSetting.QueryTimeoutValue).AsList()[0];

context.Paging.Total = context.RowLimit > 0 ? Math.Min(context.RowLimit, totalRows) : totalRows;

return result;

}

In the based adaptor implementation, this method only execute query one time by call QueryMutiple method. In here to resolve multiple parameters or avoid processing to duplicate input parameters for SQL query, we query database two time. The first time is getting data and second to get paging info.

The reason that Izenda designed to use multiple query is improving performance when querying data. To avoid issue mentioned before, we simply ignore using that feature. If you want to bring it back to you data adaptor, you have to resolve the issue manually by find out the way to duplicate parameter in potential case that it happens.

The implementation of QueryMultiple method now will look like below:

If you indicate the database adaptor does not support multiple query, it will throw exception. Otherwise, it will execute like other adaptors.

public override IEnumerable<T> QueryMultiple<T>(string connectionString, string query, object param = null, int queryTimeout = 60, Action<SqlMapper.GridReader> action = null)

{

if (IsSupportedMultipleQuery)

{

query = NormalizeQuery(query);

using (var connection = OpenConnection(connectionString))

{

try

{

using (var result = connection.QueryMultiple(query, param, commandTimeout: queryTimeout))

{

var returnResult = result.Read<T>();

action?.Invoke(result);

return returnResult;

}

}

catch (OdbcException ex)

{

Log($"Query error: {ex.ToString()}. {Environment.NewLine}Query: {query}", LogType.Error);

throw new FusionException($"{Messages.FusionCanNotQueryData}{Environment.NewLine}Error Detail: {ex.Message}");

}

}

}

throw new NotSupportedException("Not support multiple query");

}

Refer full implementation of an ODBC adaptor at <https://github.com/Izenda7Series/IzendaCustomDataAdaptors/blob/odbcdataadapter/ODBC/Izenda.Synergy.DataAdaptor.RDBMS.ODBC/ODBCDataSourceAdaptor.cs>

And for Redshift ODBC at <https://github.com/Izenda7Series/IzendaCustomDataAdaptors/blob/odbcdataadapter/ODBC/Izenda.Synergy.DataAdaptor.RDBMS.ODBC.Redshift/ODBCRedshiftDataSourceAdaptor.cs>

# Troubleshooting When Developing New Data Source Adaptor

## Debugging the Data Source Adaptor

The way to debug your custom adaptor is attach process in Visual Studio. To make it simple, we recommend you run Izenda in deployment mode 3.

Firstly, build your adaptor component in debug mode then copy output libraries to bin folder of application domain, which you can attach process into Visual Studio debugger.

Run Izenda deployment mode 3 server without debug mode, and then attach your adaptor project to IIS Express process, which is running Izenda Back-end server.

Image below demonstrates Izenda MVC Starter Kit Mode 3 process is running in IIS Express. On adaptor project solution, open Attach to Process window select iisexpress.exe project then click Attach button to attach debugging.

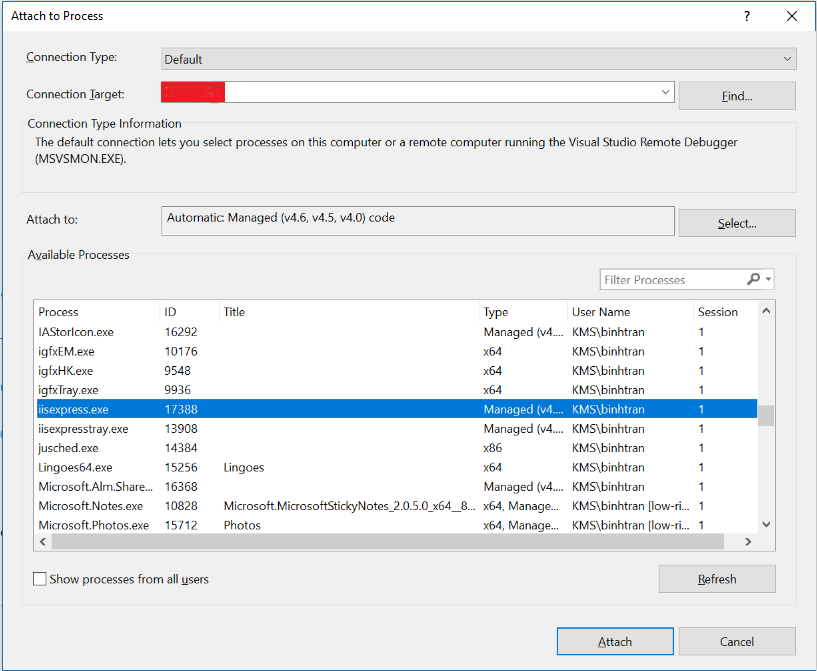


Figure 21: Attach source code of adaptor project into iisexpress.exe process to debug

The MVC Starter Kit in deployment mode 3 is available at <https://github.com/Izenda7Series/Mvc5StarterKit>.

In addition, its documentation at:

<https://www.izenda.com/docs/dev/code_mvc5starterkit.html>

<https://www.izenda.com/docs/install/doc_mvc_setup_guide.html#downloads-and-materials>

## Investigate Issue by Analysis Izenda Log

While developing adaptor component, you are able to face problem relates to syntax of generated SQL statement, it is better to log out the query to Izenda log file.

Normally, Izenda server logs out the Izenda Raw Query in the log file. For example:

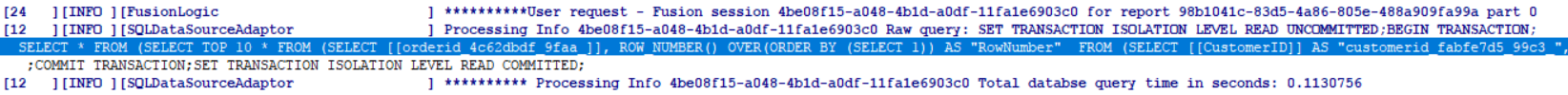


Figure 22: Izenda back-end logs out Izenda Raw Query in log file

You should log out your standard query after doing normalizer activities to transform raw query from Izenda Raw Query. In many cases, by comparing between them you can find out where the issue comes from when generating SQL syntax.

# Testing New Data Adaptor From Izenda UI

On the UI, user can add connection strings and try some main features (connection string, data model, report, dashboard, …) to test the custom driver is configured correctly.

## Connection String

When the custom driver for a new server type is configured successfully, it will be automatically listed to the Data Server Type dropdown in the Connection String page.

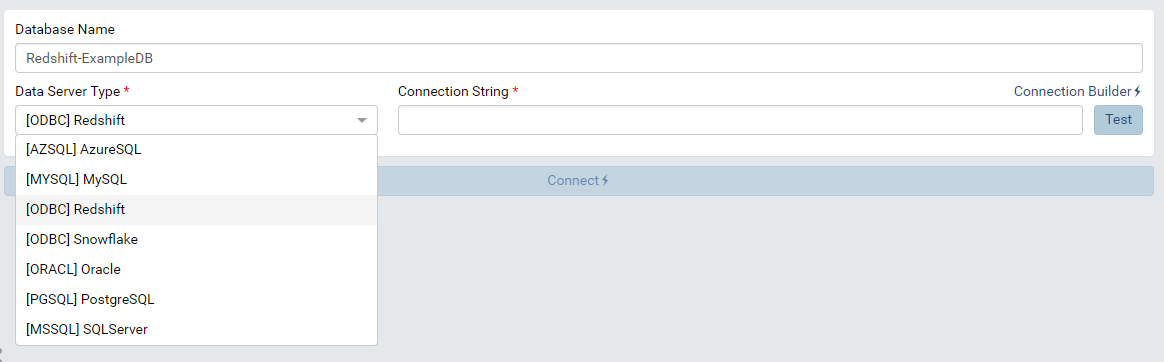


Figure 23: New server types are listed in Data Server Type dropdown

By selecting the new data server type, administrator can add this type of connection string to the system. Please see [Connection String Administrator Guide](http://localhost:1234/ui/doc_connection_string.html) for more details about adding connection string.

The image below show that a Redshift connection string is successfully add to Izenda system through custom driver.

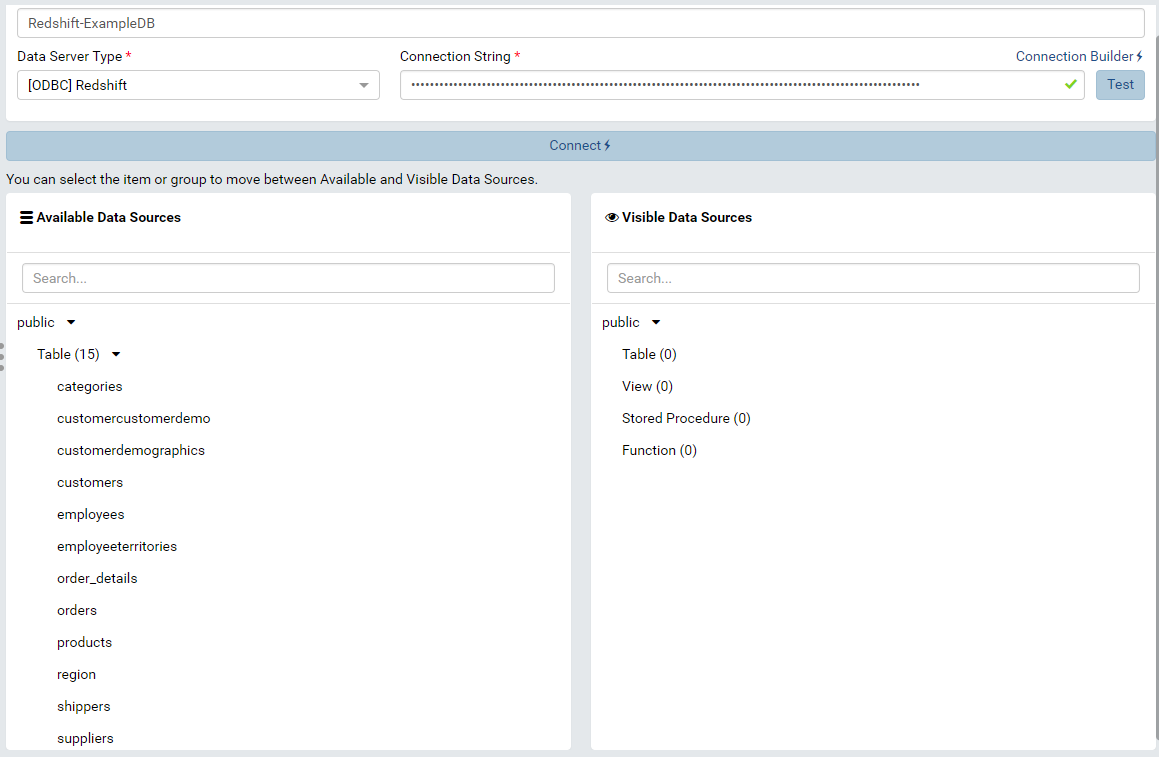


Figure 24: A Redshift connection string is connected successfully through Custom Driver

## Data Model

After adding connection string and choosing visible data sources, these data sources will be loaded in the specific tab of Data Model.

Administrator should check Tables, Views, Store Procedures, Functions and Relationship to make sure that the schema is loaded properly. Please see [Data Model/Tables, Views and Stored Procedures](https://www.izenda.com/docs/ui/doc_data_model_tables,_views_and_stored_procedures.html), [Data Model/Functions](https://www.izenda.com/docs/ui/doc_data_model_functions.html) and [Data Model/Relationships](https://www.izenda.com/docs/ui/doc_data_model_relationships_and_schema.html) for more details.

For example, the tables and relationships are loaded with all their fields in Tables tab in the below image.

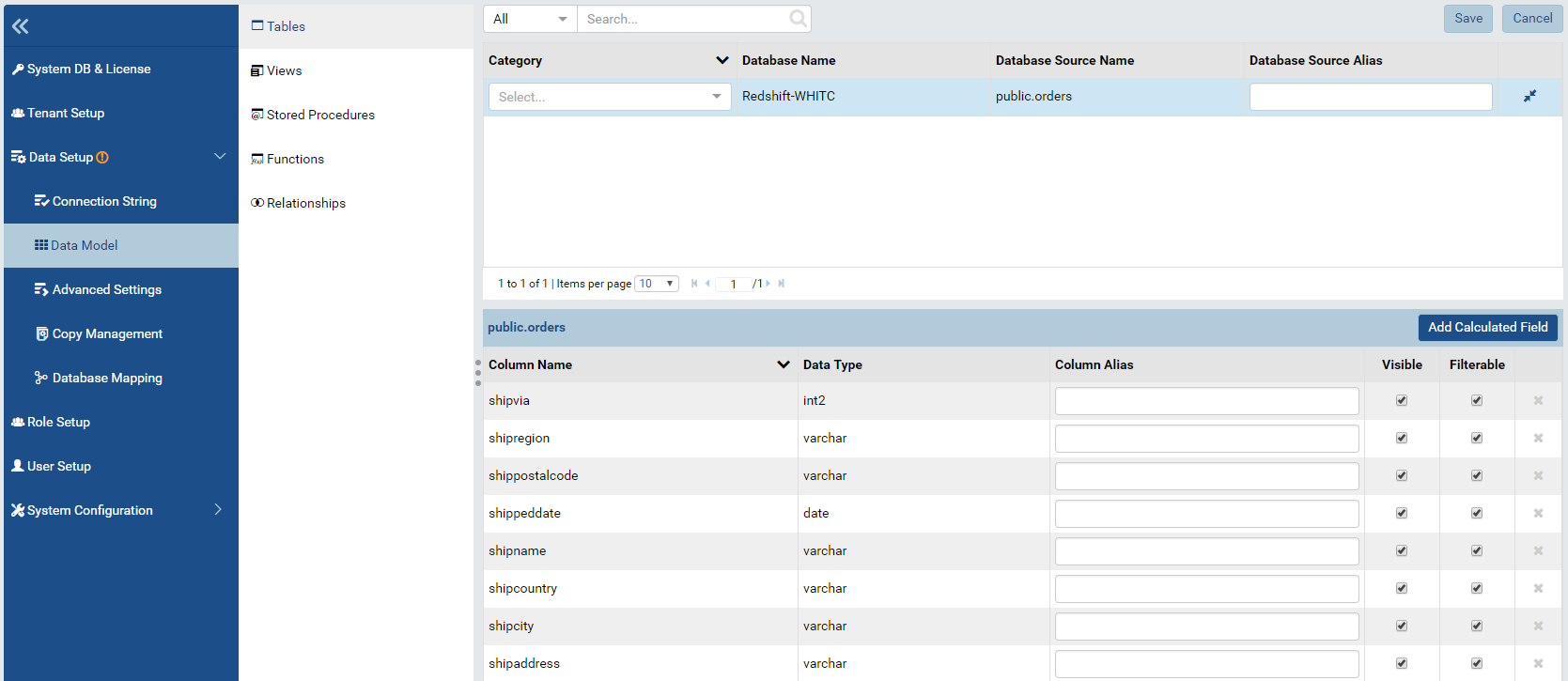


Figure 25: The selected table is loaded in the Data Model – Tables

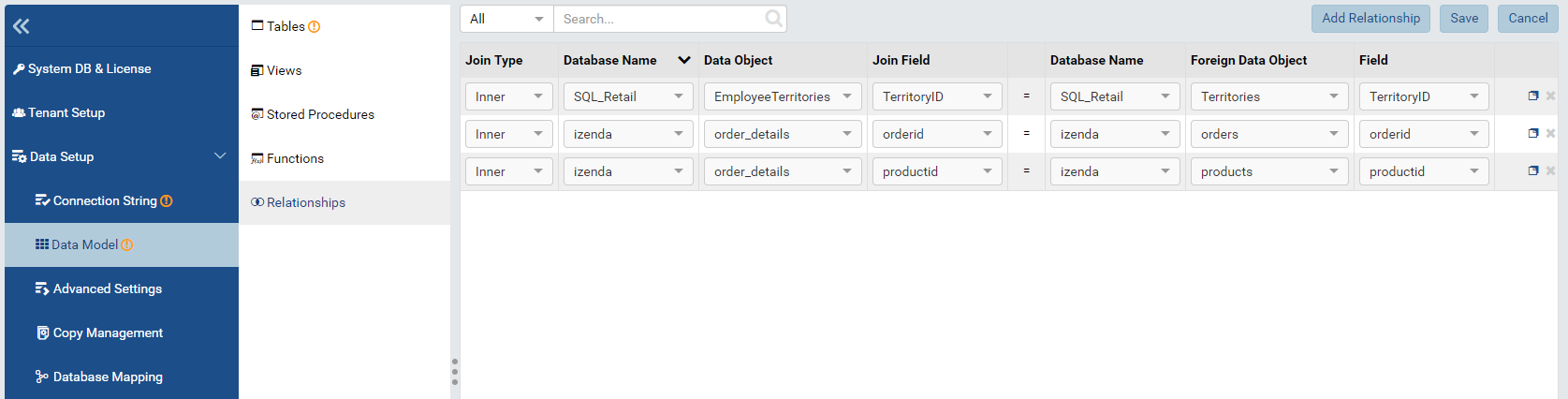


Figure 26: The relationships are loaded in Data Model - Relationships

Administrator can also Add custom view for the database that is connected by Custom driver. Please see [Custom View Setup Guide](https://www.izenda.com/docs/ui/doc_data_model_custom_view.html) for more details.

For instance, the custom view below to get all fields of order\_details in Redshift database.

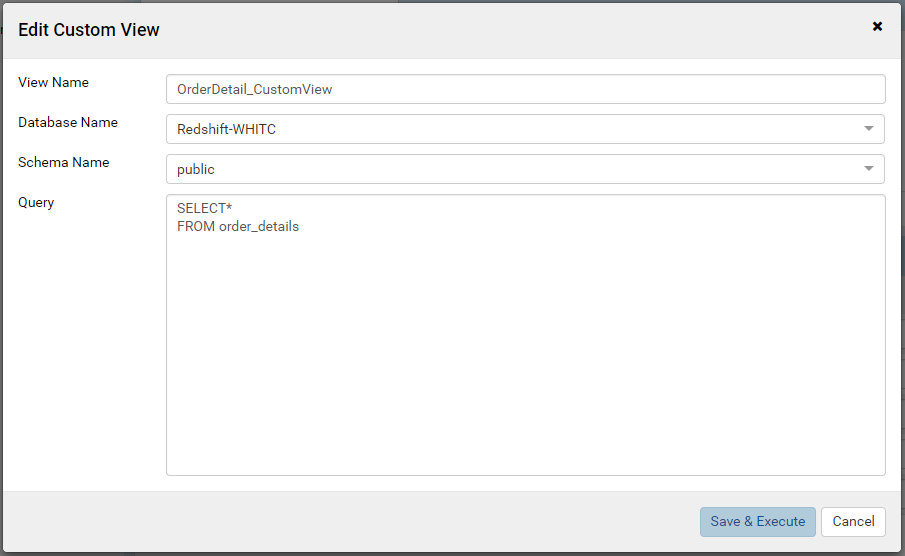


Figure 27: Add a custom view in Redshift

After pressing “Save & Execute” button, the query will be executed to get schema for the view. Then all fields will be shown in the slave section.

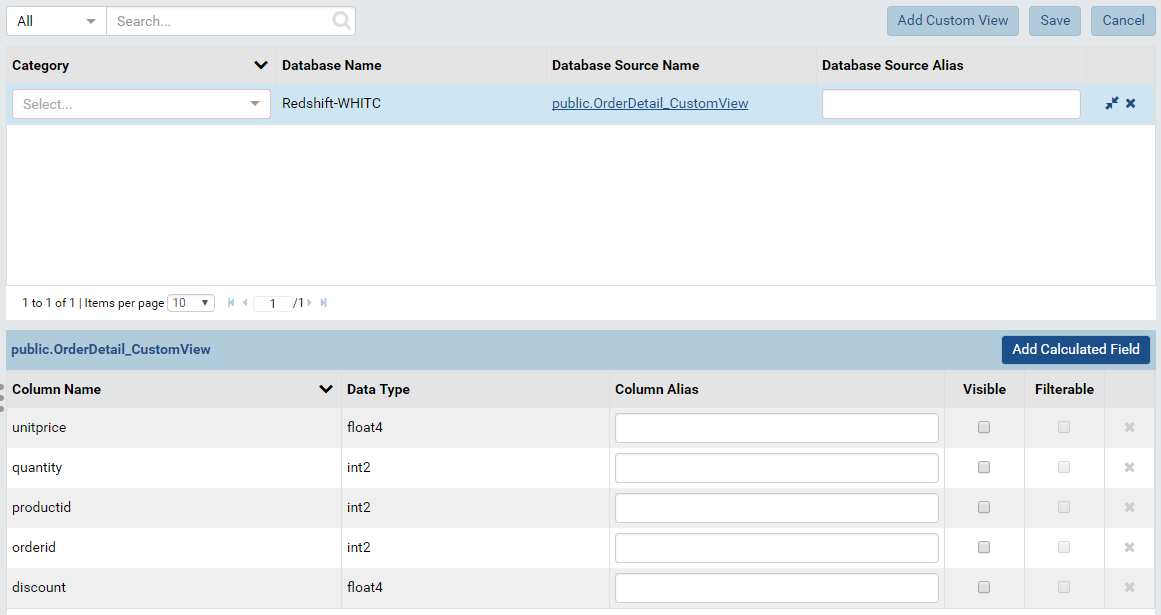


Figure 28: All fields of a custom view are shown in the slave section

## Report

All features relating to report should work correctly with the database connected by custom driver like all another types of database. Administrator can try some key feature as the following cases below:

### Case 1: Global report contains tenant field

Administrator can try sharing a global report that contains tenant field for a tenant and checking whether it retrieve data correctly at tenant level.

1. At System level, set up a database mapping for a tenant (WHITC in this example)

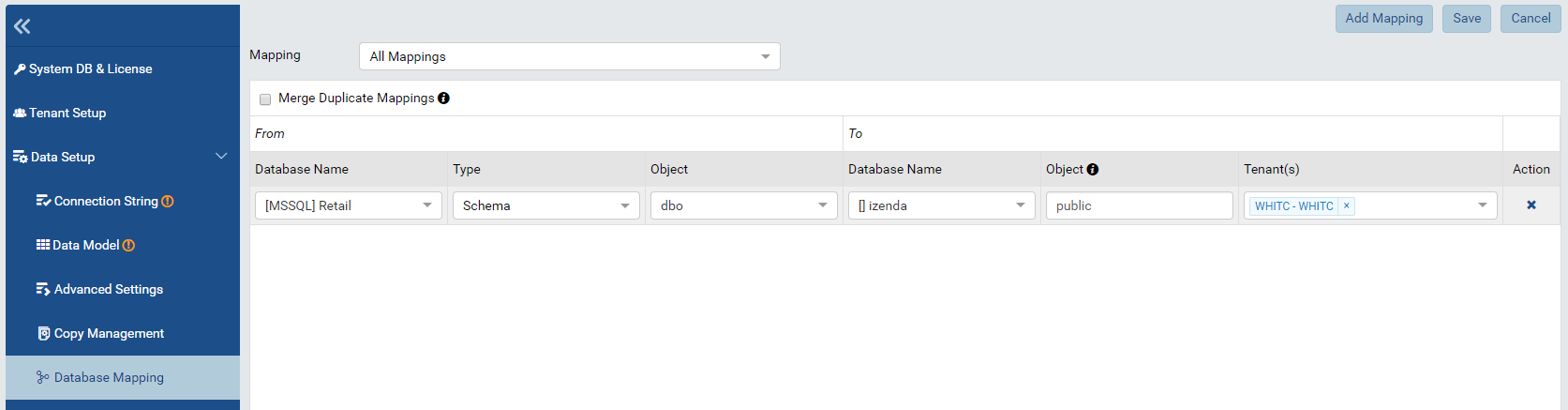


Figure 29: Database Mapping for tenant WHITC

1. Setting up tenant field. Please see [Advanced Setting - Security Settings](https://www.izenda.com/docs/ui/doc_advanced_settings.html#update-security-settings) for more details.

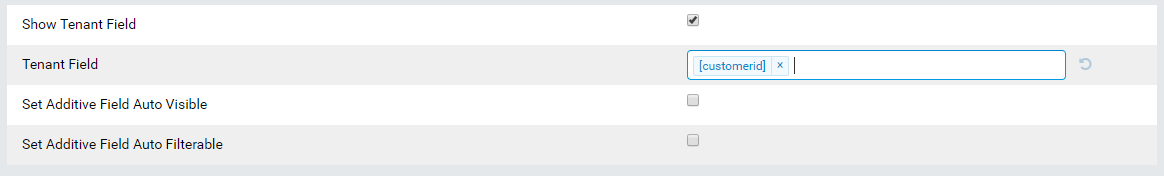


Figure 30: Setting field “customerid” as tenant field

1. Create a global report and add sharing.

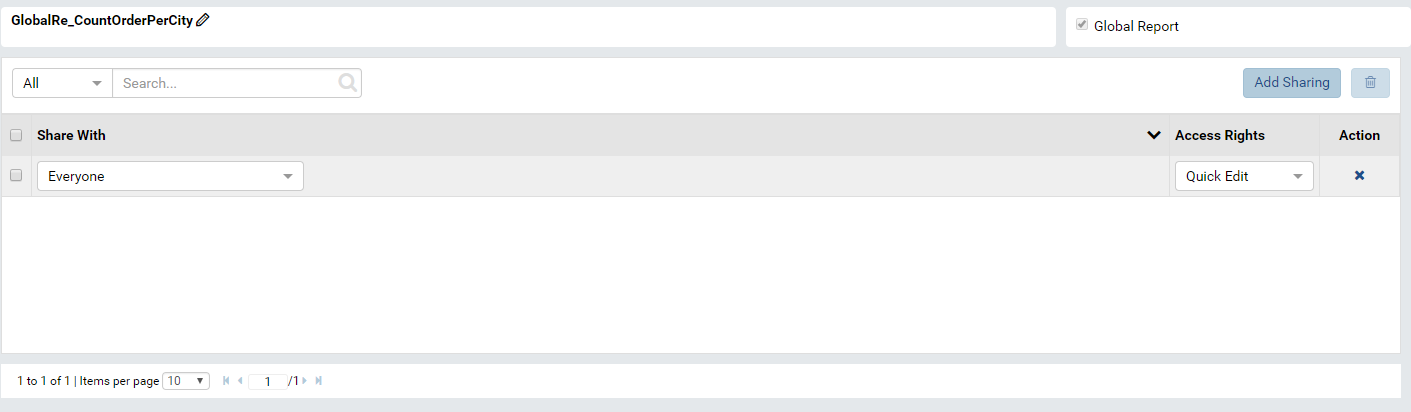


Figure 31: Add sharing for global report

1. View the below report at both system level and tenant level to check if the date is retrieve correctly.

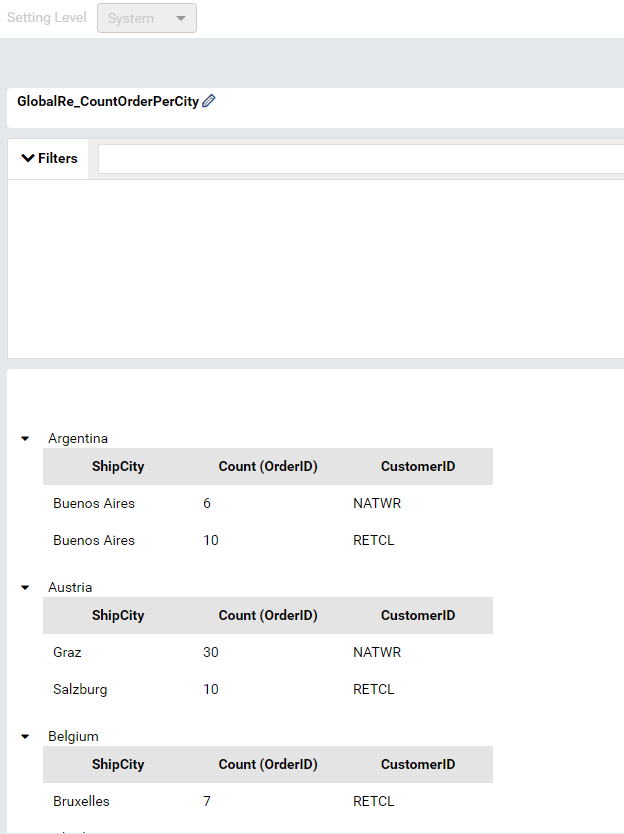


Figure 32: The Global report show all data at System level

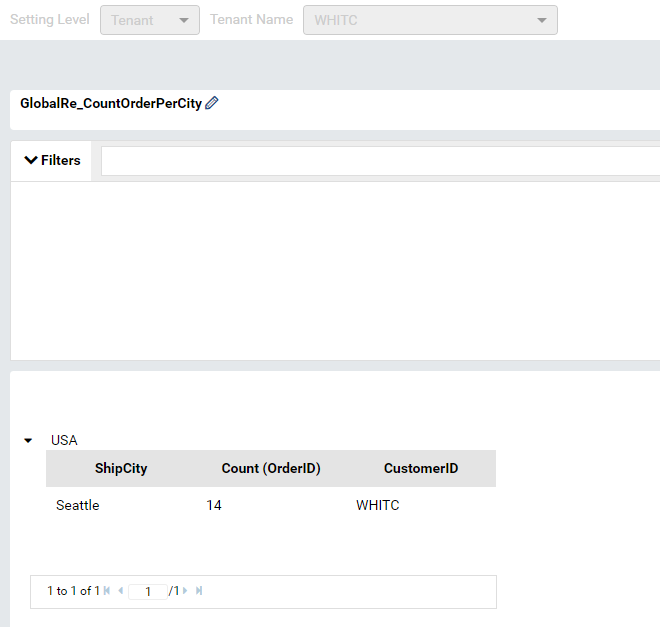


Figure 33: The Global report retrieve data for tenant WHITC

### Case 2: Report with filters

The images below show the report with filter that queries data from Redshift. Please see [Report Designer/Fields – Filter](https://www.izenda.com/docs/ui/doc_report_designer_fields.html#add-a-report-filter) for more details.

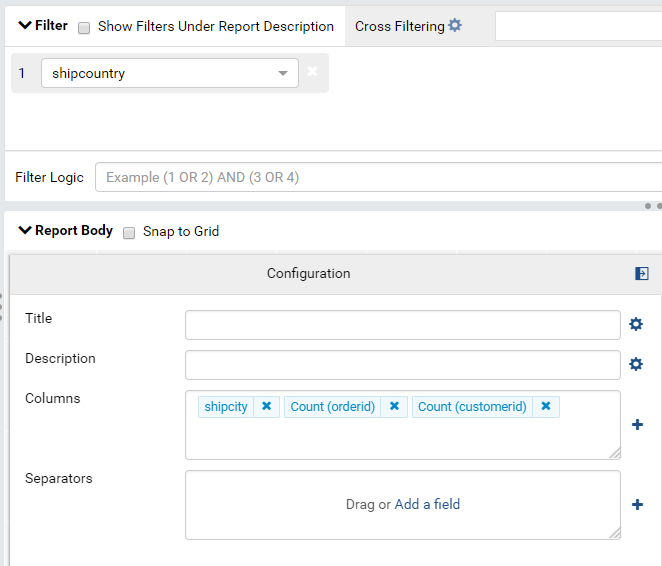


Figure 34: Report Designer backside of report part contains filter

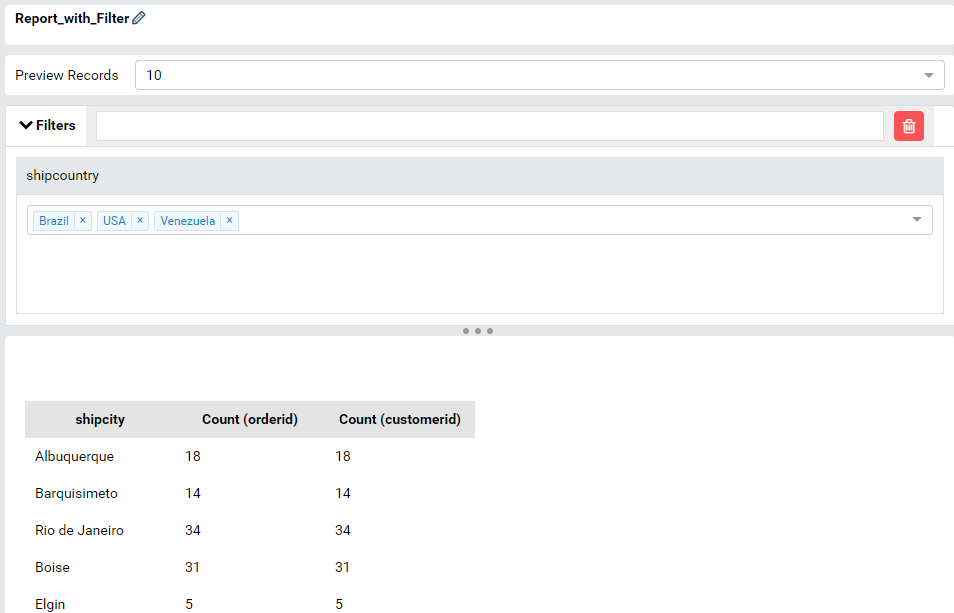


Figure 35: The report viewer contains filter

### Case 3: Report with grouping, aggregated function and sub total

Administrator can also try with the report containing aggregated function and subtotal following several steps below:

* Create a simple report with a couple of fields. For example, creating a report with 2 columns (shipcity and sum(freight)) and a separator (shipcountry)

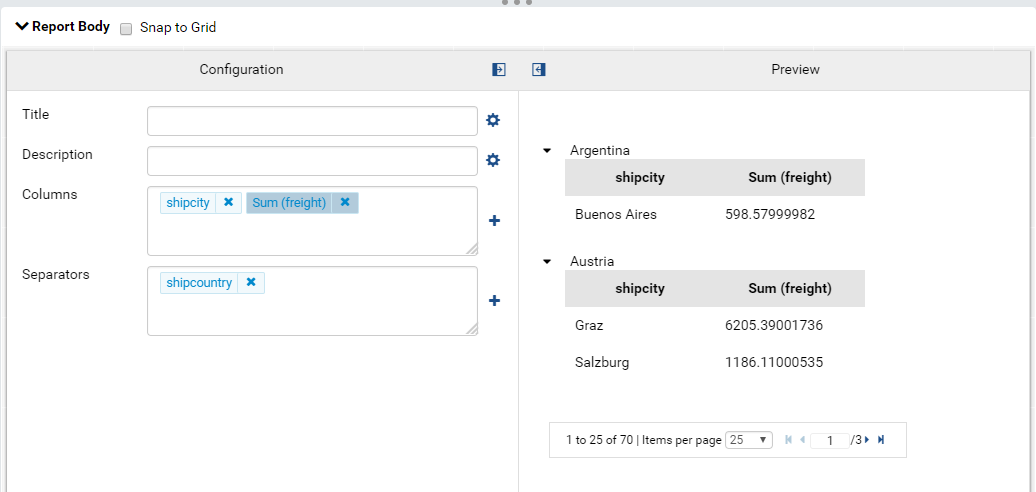


Figure 36: A grid report part contains 2 columns and a separator

* Add a Subtotal for a field. For example, add sum subtotal for sum(freight)

Please see [Sub Total](https://www.izenda.com/docs/ui/doc_grand_total_sub_total.html#sub-total) for more details.

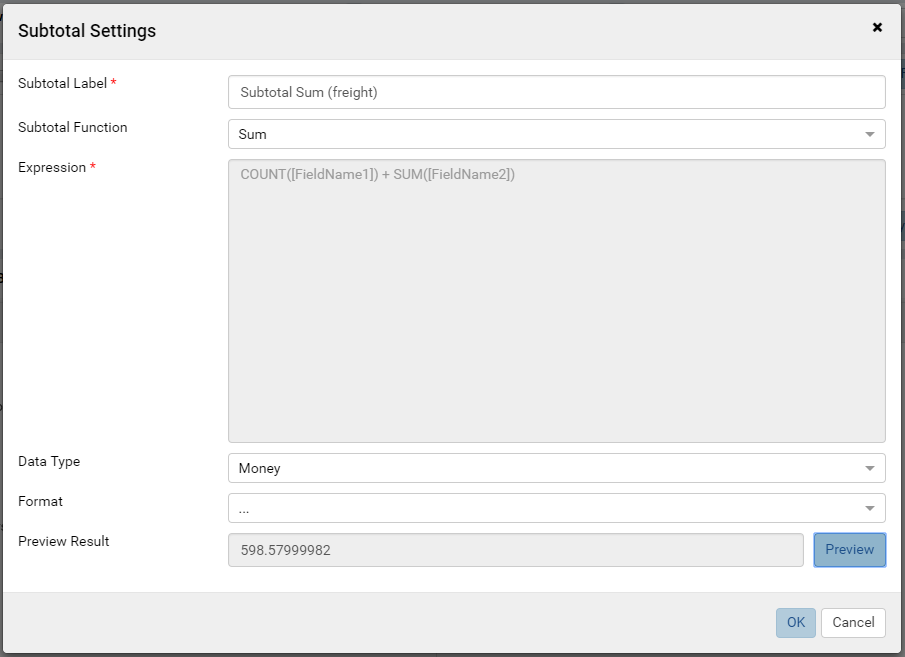


Figure 37: Add sum subtotal for sum(freight)

* Then save and view the report to ensure that all data is retrieved and calculated correctly.

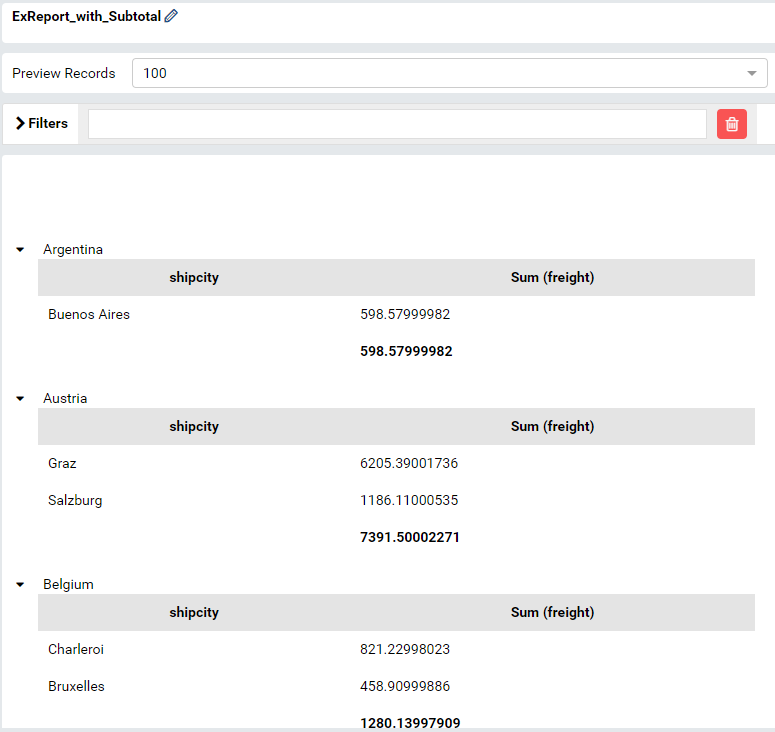


Figure 38: The report with aggregated and subtotal in viewer

### Case 4: Report with Calculated Field

The report retrieving data from database through Custom Driver should also work correctly with the [Calculated Field](https://www.izenda.com/docs/ui/doc_report_designer_fields.html#add-calculated-field).

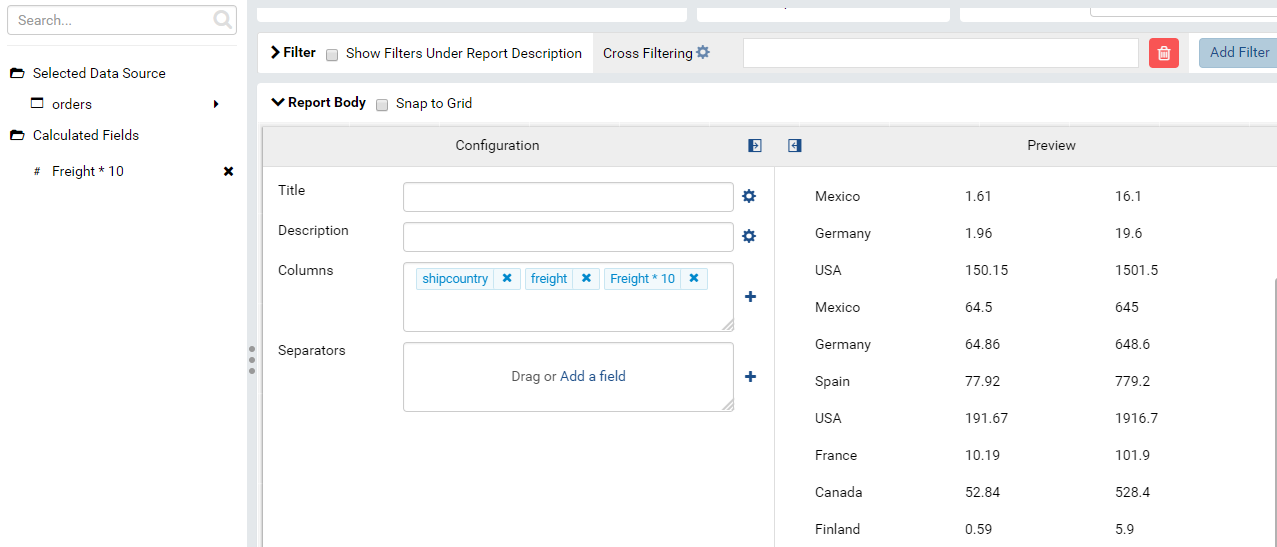


Figure 39: Report contains Calculated Field

### Case 5: Drilldown chart report part

The drilldown chart report part should work well when querying data from database that connected through Custom Driver.

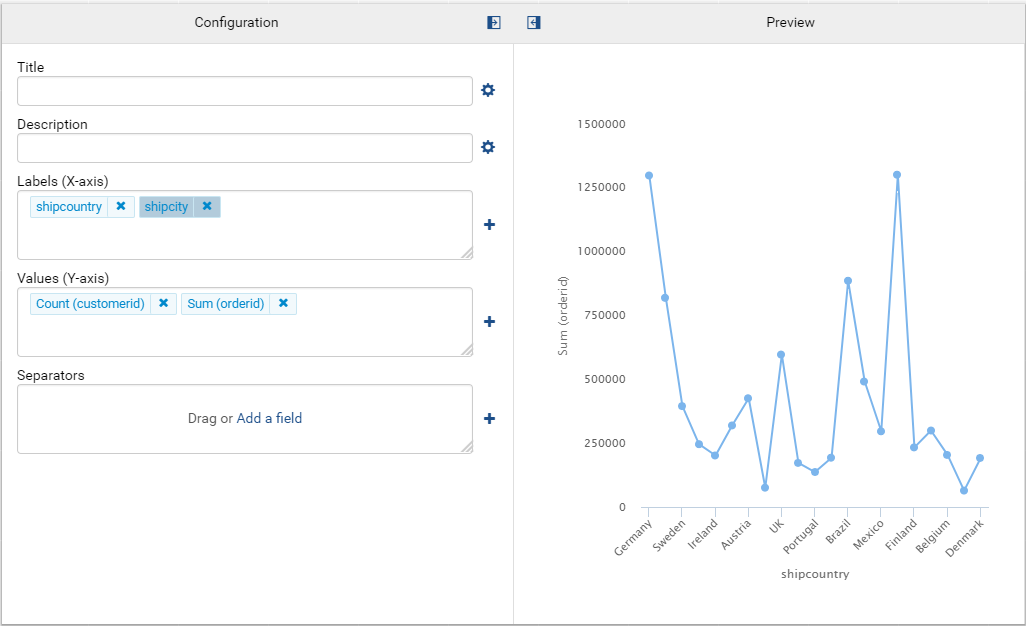


Figure 40: A drilldown chart querying data from Redshift database

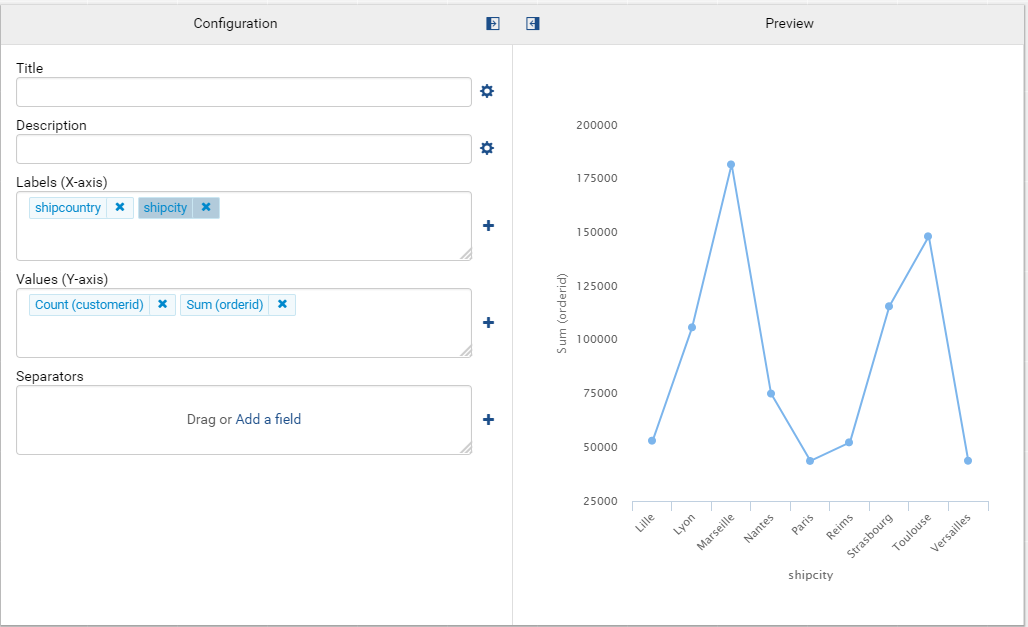


Figure 41: A chart querying data from Redshift after drill down on a Country

### Case 6: Exporting report

Exporting the chart report above in a couple of type, such as: PDF, CSV, … to make sure exporting feature work well with Custom Driver.

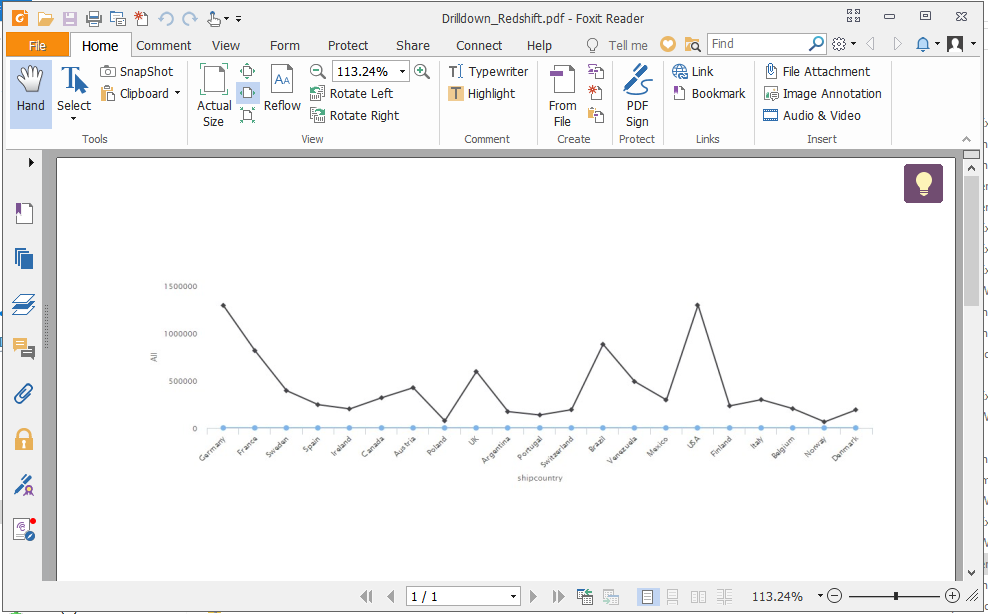


Figure 42: The PDF exporting file

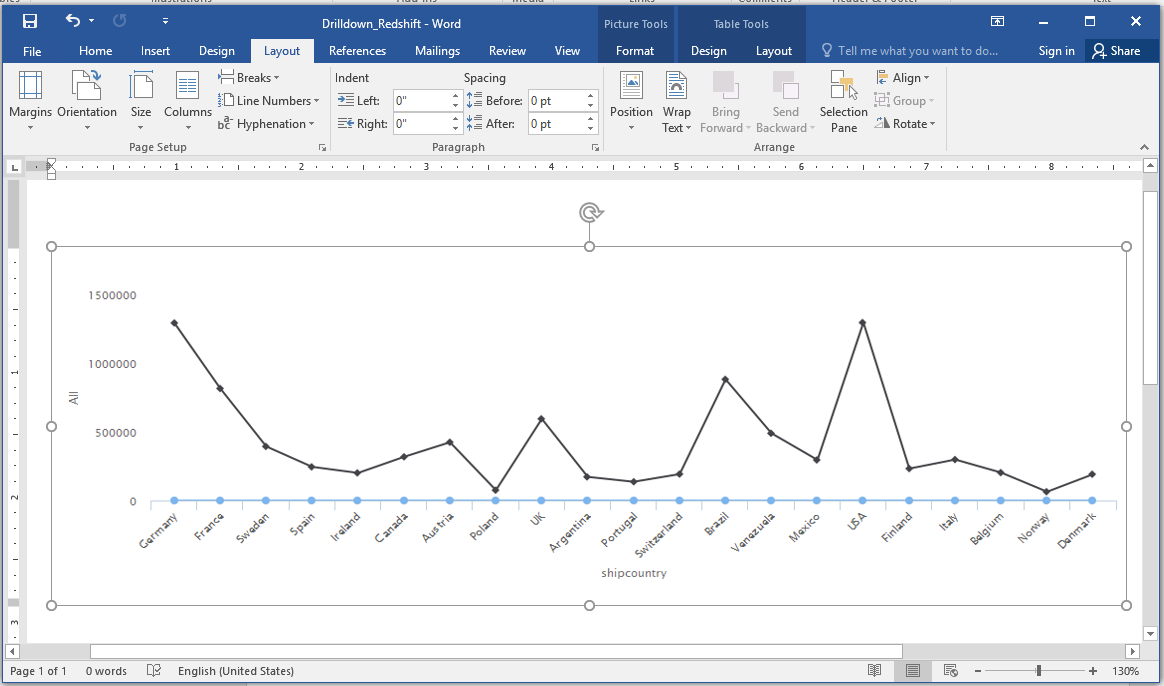


Figure 43: The Word exporting file

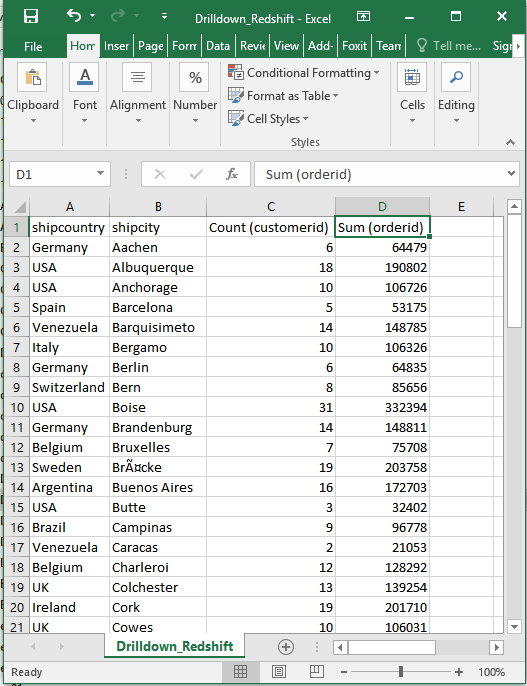


Figure 44: The CSV exporting file