InterSystems™
IRIS Data Platform

Using the InterSystems Spark Connector

Version 2019.1
2019-03-22

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using the InterSystems Spark Connector
InterSystems IRIS Data Platform   Version 2019.1   2019-03-22
Copyright © 2019 InterSystems Corporation
All rights reserved.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)

Tel:	+1-617-621-0700
Tel:	+44 (0) 844 854 2917
Email:	support@InterSystems.com

Table of Contents

About This Book

This book describes the InterSystems Spark Connector, an implementation of the Data Source API for Spark that allows the Spark data processing engine to make optimal use of the InterSystems IRIS Data Platform™ and its distributed data capabilities.

The following topics are discussed in this book:

- Introduction — introduces Apache Spark and the InterSystems Spark Connector, and describes how to configure the Spark Connector on your system.
- Spark Connector Data Source Options — provides a detailed description of all Spark Connector data source options.
- Using Spark Connector Extension Methods — demonstrates the Spark Connector method interface.
- Spark Connector Best Practices — describes ways to optimize Spark Connector hardware and software
- Spark Connector Internals — useful information not otherwise covered.
- Spark Connector Quick Reference — provides a quick reference to InterSystems-specific Scala extension methods provided by the Spark Connector API.

There is also a detailed Table of Contents.

1 Introduction

The InterSystems IRIS™ Spark Connector enables an InterSystems IRIS database to function as an Apache Spark data source. It implements a plug-compatible replacement for the standard Spark jdbc data source. This allows the results of a complex SQL query executed within the database to be retrieved by the Spark program as a Spark Dataset, and for a Dataset to be written back into the database as a SQL table.

1.1 Features

The Spark Connector has an intimate knowledge of —and tight integration with —the underlying database server that provides several advantages over the standard Spark jdbc data source:

- *Predicate Push Down*

  The Connector recognizes a richer set of operators than the standard jdbc data source, allowing more operations to be 'pushed down' into the underlying database for execution.

- *Shard Aware*

  An InterSystems IRIS database can be *sharded*, meaning that tables that are transparently partitioned across multiple servers running on different computers. The Connector allocates compute tasks so that each Spark executor is co-located with the server from which it draws its data. This not only reduces the movement of data across the network, but, more importantly, allows the Spark program's performance to scale linearly with the number of shards, and so size of the data set, on which it operates.

- *Implicit Parallelism*

  The Connector exploits the server's innate ability to automatically parallelize certain queries and so allow large result sets to be returned quickly to the Spark driver program through multiple concurrent network connections. By contrast, the standard jdbc data source requires the user to explicitly specify how the result set is to be partitioned, which in practice is often very difficult to do well.

1.2 Data Source Provider Class Names

Spark data sources are accessed through provider class names. The standard Spark jdbc data source provider class is named org.apache.spark.sql.execution.datasources.jdbc.JdbcRelationProvider, and can be used by specifying it in a call to **format()** as demonstrated in the following example:

```
var df = spark.read
   .format("org.apache.spark.sql.execution.datasources.jdbc.JdbcRelationProvider")
   .option("dbtable","mytable").load()
```

Since the full class name is very awkward to use, it is normally specified with the short alias jdbc:

```
var df = spark.read.format("jdbc").option("dbtable","mytable").load()
```

The InterSystems Spark Connector data source is referenced in exactly the same way, using the full provider class name com.intersystems.spark or the short alias iris:

```
var df = spark.read.format("iris").option("dbtable","mytable").load()
```

Important: The terms jdbc and iris (lower case, in the same typography as other class names) are used frequently in this book, and always refer specifically to the data source provider class names, never to Java JDBC or InterSystems IRIS.

1.3 Requirements and Configuration

1.3.1 Requirements

The Spark Connector requires the following:

- *InterSystems IRIS* — for more information on configuration, see "Spark Connector Best Practices" later in this book.
- *Spark 2.0+* — the Connector has been primarily tested against Spark version 2.1.1 under JVM 1.8 u144, using the Spark Standalone cluster manager.
- *Java 1.8* — InterSystems IRIS does not support earlier versions. Apache Spark does not currently support JVM 1.9.

1.3.2 Optional Configuration Settings

The Connector recognizes a number of configuration settings that parameterize its operation. These are parsed from the Apache Spark SparkConfconfiguration structure at startup and may be specified by:

- the file spark-defaults.conf associated with the Spark cluster.
- values for the `--conf` option passed on the command line.
- arguments to the **SparkConf()** constructor or its **set()** member functions, called from within the driver application itself.

The *url*, *user*, and *password* options specify connection string values for a read or write. The default values are automatically defined using information from the default InterSystems IRIS master instance specified in the SparkConf configuration. Connection options can be explicitly specified in a read or write operation (see "Connection Options") to override the defaults.

- spark.iris.master.url

  A string of the form `"IRIS://`*host*`:`*port*`/`*namespace*`"` that specifies the address of the default Spark master server to connect to if none is.

- spark.iris.master.user

  The user account with which to connect to the master server.

- spark.iris.master.password

The password for the user account with which to connect to the master server.

Default values are also assigned to the following settings:

- spark.iris.master.expires

  A positive integral number of seconds after which a connection to an instance is judged to have expired and is automatically closed and garbage collected. Default = 60.

- spark.iris.worker.host

  An optional string of the form `pattern => rewrite` where:

  – `pattern` is a regular expression (which may contain parenthesized groups)

  – `rewrite` is a replacement string (which may contain $ references to those groups)

    If specified, this setting describes how to convert the host name of an InterSystems IRIS server into the host name of the preferred Spark worker that should handle requests to read and write dataset partitions to it.

    The most common cluster configuration is to arrange that a Spark worker runs on each machine that hosts an InterSystems IRIS server, for then records need not travel across the network. It can happen, however, that the host name by which the master server knows its shard server differs from the host name by which the Spark master knows its worker, even though they are running on the very same machine. Their host names could be aliased by a DNS server, for example, or could be running in separate Docker containers.

    This setting offers a means of defining at install time a function that maps between the two host names.

For more information, see the Apache Spark documentation on "Spark Configuration" and the org.apache.spark.SparkConf class.

2 Spark Connector Data Source Options

The Spark Connector iris data source and the standard Spark jdbc data source both use the generic option interface, which sets options by passing option keys and values to the **option()** or **options()** methods. The iris data source supports the same set of options as the Spark jdbc data source, and also includes several options implemented specifically for the InterSystems IRIS™ SQL dialect.

The following sections provide detailed information on all supported iris data source options:

- Using Spark Connector Generic Options — provides a brief overview of generic data source option support.
- Query Options — *dbtable* and *fetchsize* are standard read options. The iris data source also supports the *query* key, which is an optional synonym for *dbtable*.
- Standard Save Options — *mode*, *batchsize*, and *isolationlevel* are standard write options.
- InterSystems IRIS Save Options — *description*, *publicrowid*, *shard*, and *autobalance* are write options supported only by iris.
- Connection Options — *url*, *user*, and *password* are standard options that explicitly specify connection string values for a read or write.
- Partition Tuning Options — *partitionColumn*, *lowerBound*, *upperBound*, *numPartitions*, and *mfpi* are standard options that specify how a query will be partitioned.

In all of the following examples, *spark* is an instance of SparkSession, and *df* is an instance of DataFrame.

Note: In this book, the terms jdbc and iris (lower case, in the same typography as other class names) are always data source provider class names, not references to Java JDBC or InterSystems IRIS. See "Data Source Provider Class Names" for details.

2.1 Using Spark Connector Generic Options

Both the iris data source and the jdbc data source provide a generic option interface to set data source options. For example, the following code uses the generic Spark jdbc data source with the InterSystems JDBC driver to load a table from the InterSystems IRIS database:

```
val df = spark.read.format("jdbc")
                   .option("driver","com.intersystems.jdbc.IRISDriver")
                   .option("dbtable","tablename")
                   .option("url","IRIS://localhost:51773/USER")
                   .option("user","_system")
                   .option("password","SYS")
                   .load()
```

Most options supported by the jdbc data source can also be used in exactly the same way with the iris data source . However, the same table loading code can be much simpler when using the Spark Connector iris data source because most of the corresponding iris options have defaults (see “Default Configuration Settings” for details). For example:

```
import com.intersystems.spark._

val df = spark.read.format("iris")
                   .option("dbtable","tablename")
                   .load()
```

The most important difference in this example is that the **format()** call specifies the Spark Connector’s iris data source provider class, rather than the generic jdbc provider class. We omit the *driver* option because the Spark Connector always uses the built-in InterSystems JDBC driver. The *url*, *user*, and *password* options can also be omitted because iris can use default values obtained from Spark configuration settings (see “Connection Options”).

2.2 Query Options

The Spark jdbc format and the iris format can both use *dbtable* and *fetchsize* when loading a table or performing a query.

dbtable (or query)

The Spark jdbc format and the iris format both use *dbtable* to specify a table name or SQL query. Any string that would be valid in a FROM clause can be used as a *dbtable* value. The *dbtable* key is also a valid write option (see “Standard Save Options”).

The iris *query* key is just a synonym for *dbtable*. This optional key can improve code readability by distinguishing a SQL query from a table name. For example:

```
var df = spark.read.format("iris")
                   .option("dbtable","mytable")   // Load a table
                   .load()

var df = spark.read.format("iris")
                   .option("query","SELECT * FROM mytable")   // Perform a query
                   .load()
```

The Spark Connector also offers several alternative query methods that allow this option to be set as a method parameter (see “Using Query and Save Extension Methods”).

fetchsize

The Spark jdbc format and the iris format both use *fetchsize* to specify the number of rows to fetch per server round trip. Defaults to 1000. This option is applied when loading a table or performing a query, and will be ignored by write operations.

```
var df = spark.read.format("iris").option("dbtable","mytable")
                   .option("fetchsize",500)
                   .load()
```

2.3 Standard Save Options

The Spark jdbc format and the iris format can both use *dbtable*, *mode*, *batchsize*, and *isolationlevel* when saving a table.

dbtable

The Spark jdbc format and the iris format both use *dbtable* to specify a table name when saving. This key is also a valid read option (see "Query Options").

```
spark.write.format("iris")
           .option("dbtable","mytable")   // Save a table
           .save()
```

This option can also be specified as a parameter of the DataFrameReader.**iris()** extension method (see "Using Extension Methods to Set Data Source Options").

mode

Describes how to behave if the target table already exists. Can be one of `OVERWRITE`, `APPEND`, `IGNORE`, or `ERROR`, and defaults to `ERROR`. These settings correspond to the values defined in the SaveMode enum (see org.apache.spark.sql.SaveMode for more information). This option is applied when saving a table, and will be ignored by read operations.

```
df.write.format("iris")
        .option("mode","OVERWRITE")   //overwrite any existing table
        .save()
```

This option can also be set by the standard DataFrameWriter.**mode()** method.

batchsize

Specifies an integer number of rows to insert per server round trip. Defaults to `1000`. This option is applied when saving a table, and will be ignored by read operations.

```
df.write.format("iris")
        .option("batchsize",500)
        .save()
```

isolationlevel

Sets the JDBC transaction isolation level. Can `READ_UNCOMMITTED` (the default) or `NONE`. These values correspond to the transaction isolation levels defined in java.sql.Connection. This option is applied when saving a table, and will be ignored by read operations.

```
df.write.format("iris")
        .option("isolationlevel","READ_COMMITTED")
        .save()
```

2.4 InterSystems IRIS Save Options

The *description*, *publicrowid*, *shard*, and *autobalance* options are supported only by the iris data source, and are used to set certain options unique to InterSystems IRIS. These options are applied when saving a table, and will be ignored by read operations.

description

Specifies an optional description to document the newly created table. Will be ignored when appending to a table that already exists. Defaults to `""`.

```
df.write.format("iris")
        .option("description","This is a table of no importance whatsoever.")
        .save()
```

See "CREATE TABLE" in the *InterSystems SQL Reference* for more information on this option.

This option can also be set with the DataFrameReader.**description()** extension method (see "Setting Save Options with Extension Methods").

publicrowid

Specifies whether the master row ID column for the newly created table is to be made publicly visible. Will be ignored when appending to a table that already exists. Defaults to `"false"`.

```
df.write.format("iris")
        .option("publicrowid","false")
        .save()
```

See "CREATE TABLE" in the *InterSystems SQL Reference* for more information on this option.

This option can also be set with the DataFrameReader.**publicRowID()** extension method (see "Setting Save Options with Extension Methods").

shard

Determines sharding for the newly created table. Will be ignored when appending to a table that already exists. The value can be either a boolean indicating whether the newly created table is to be sharded, or a sequence of field name strings (possibly empty) to be used as the user defined shard key.

- A boolean value of `"true"` indicates that the records of the table are to be distributed across the cluster using a system assigned sharding key. Boolean `"false"` (the default) indicates that records should be stored locally on the shard master itself.

  ```
  df.write.format("iris")
          .option("shard","false") // store locally
          .save()
  ```

- A field name or sequence of names indicates that the records of the table are to be distributed across the cluster using the given sequence of fields to compute the shard key:

  ```
  df.write.format("iris")
          .option("shard","columnA")
          .save()

  df.write.format("iris")
          .option("shard",Seq("columnA","columnX"))
          .save()
  ```

- If the sequence is empty then the table will be sharded on the system assigned key:

  ```
  df.write.format("iris")
          .option("shard","")
          .save()
  ```

This option can also be set with the DataFrameReader.**shard()** extension method (see "Setting Save Options with Extension Methods").

autobalance

When writing a dataset to a table that is sharded on a system assigned shard key, an *autobalance* value of `"true"` (the default) specifies that the inserted records are to be evenly distributed amongst the available shards. Value `"false"` specifies that every dataset partition should attempt to write its records directly into the shard on which its Spark executor also runs.

```
df.write.format("iris")
        .option("autobalance","false") // disable autobalancing)
        .save()
```

In a properly configured cluster, a Spark slave runs on each shard server. Writing a dataset with the *autobalance* option disabled can be faster, since records no longer need to travel across the network to reach their destination shard. However, it now becomes the Spark application's responsibility to ensure that roughly the same number of records is written to each shard.

Will be ignored when writing to a table that is not sharded, or when writing to a table sharded on a user defined shard key.

This option can also be set with the DataFrameReader.**autobalance()** extension method (see "Setting Save Options with Extension Methods").

2.5 Connection Options

The *url*, *user*, and *password* options explicitly specify connection string values for a read or write. If these options are not set, they will be automatically defined using information from the default InterSystems IRIS master instance specified in the SparkConf configuration (see "Default Configuration Settings"). When set, these options override the default for the duration of the current read or write operation.

url, user, and password

The value of *url* is a string of the form `"IRIS://[host]:[port]/[namespace]"` that names the master InterSystems IRIS instance with which the data is to be loaded or saved. The values of *user* and *password* are strings containing the account and password required to make the connection to *url*.

For example, the following call specifies the connection required to read a table named `mytable` in the `USER` namespace:

```
var df = spark.read.format("iris").option("dbtable","mytable")
              .options(Map("url" -> "IRIS://localhost:51773/USER",
                           "user" -> "_system",
                           "password" -> "SYS"))
              .load()
```

For more information on constructing a URL string, see "Defining a JDBC Connection URL" in *Using InterSystems IRIS with JDBC*.

These options can also be set with the DataFrameReader.**address()** extension method (see "Setting Connection Options with the address() Method").

Note: The standard Spark jdbc format also offers the *driver* option, which specifies the class name of the JDBC driver to use. The Spark Connector iris format ignores this option (if specified) because it always uses the InterSystems JDBC driver, which is embedded within the Connector itself.

2.6 Partition Tuning Options

These options determine how a read operation will be partitioned. When reading data, the Connector attempts to negotiate with the server as to how best to partition the resulting Dataset. Depending on how the cluster is configured, each partition can potentially run in parallel within its own Spark executor and establish its own independent connection into the shard from which it draws its data.

Partitioning requests can be either implicit or explicit:

- Implicit Partitioning — *mfpi* specifies the maximum number of partitions to create, and the Connector automatically sets other partitioning parameters.
- Explicit Partitioning — *partitionColumn*, *lowerBound*, *upperBound*, and *numPartitions* provide the Connector with detailed instructions for partitioning.

The Spark Connector also offers several extension methods that allow these options to be set as method parameters (see "Setting Partitioning Options with Query Method Parameters").

2.6.1 Implicit Partitioning

The *mfpi* option (*m*aximum number of *p*artitions *p*er *i*nstance) specifies an upper limit to the number of partitions that the server will create per queried instance in any implicit query factorization. The default is `"1"`.

This option is applied when reading, and will be ignored by write operations. It is also ignored if a value has been specified for *numPartitions*, which takes precedence over *mfpi* (see "Explicit partitioning").

Implicit partitioning with "mfpi"

Using the *mfpi* option, you could execute a query with the following statement:

```
var df = spark.read.format("iris").option("dbtable","SELECT * FROM mytable")
                   .option("mfpi", 2)
                   .load()
```

2.6.2 Explicit Partitioning

The following options provide an explicit description of how to partition the queries sent to each distinct instance:

- *partitionColumn* — String name of integer valued column in the result set to be used for partitioning.
- *lowerBound* — Long minimum value of *partitionColumn*.
- *upperBound* — Long maximum value of *partitionColumn*.
- *numPartitions* — Int number of partitions per instance to create.

These options correspond exactly to the similarly named arguments for the Apache Spark DataFrameReader.**jdbc()** method. See the documentation on that method for more information.

Explicit partitioning options take precedence over the *mfpi* option (described in the previous section). They are applied when reading, and will be ignored by write operations.

Explicit partitioning with partitionColumn, lowerBound, upperBound, and numPartitions

Using these options, you could execute a query by calling:

```
var df = spark.read.format("iris").option("dbtable","SELECT * FROM mytable")
                   .option("partitionColumn", "columnA")
                   .option("lowerBound", 0)
                   .option("upperBound", 10000)
                   .option("numPartitions", 2)
                   .load()
```

3 Using Spark Connector Extension Methods

This chapter discusses a set of InterSystems-specific Scala extension methods provided by the Spark Connector. These methods extend the generic Spark interface with implicit Scala classes and types that provide more convenience and better type safety.

The following topics are discussed:

- Using Query and Save Extension Methods — describes extension methods that simplify reading and writing a DataFrame, Dataset, or RDD.
- Using Extension Methods to Set Data Source Options — describes extension methods and method parameters that provide alternate ways to set data source options.

All of the examples in this chapter assume that *spark* is an instance of SparkSession and *df* is an instance of DataFrame.

Note: In this book, the terms jdbc and iris (lower case, in the same typography as other class names) are always data source provider class names, not references to Java JDBC or InterSystems IRIS™. See "Data Source Provider Class Names" for details.

3.1 Using Query and Save Extension Methods

The following topics are discussed:

- Using the iris() Save Method — describes an extension method that simplifies saving a DataFrame.
- Using the iris() and dataset[T]() Query Methods — describes extension methods that return query results as a DataFrame or Dataset.
- Using the rdd[T]() Query Method — describes an extension method that returns query results as an RDD.

3.1.1 Using the iris() Save Method

The following example demonstrates a standard way to save an existing DataFrame *df* to a table named *Owls*:

```
df.write.format("iris").option("dbtable","Owls").save()
```

The Spark connector provides the DataFrameWriter.**iris()** extension method, which performs the same task in a much simpler manner. The following code is functionally identical to the previous example:

```
df.write.iris("Owls")
```

In this example, the *dbtable* option is replaced by an argument, the format is automatically set to iris, and the call to **save()** is made internally.

3.1.2 Using the iris() and dataset[T]() Query Methods

The following example demonstrates a standard way to save an existing DataFrame *df* to a table named *Owls*, and then query the database to read the table back into new DataFrame *df2*:

```
val df2 = spark.read.format("iris").option("dbtable","SELECT * FROM Owls").load()
```

The Spark connector provides the DataFrameWriter.**iris()** method to perform the same task in a much simpler manner. The following code is functionally identical to the previous example:

```
val df2 = spark.read.iris("SELECT * FROM Owls")
```

In this example, the *dbtable* option is replaced by an argument, the format is automatically set to iris, and the call to **load()** is made internally.

Similarly, the SparkSession.**dataset[T]()** executes a query or loads a table and returns results as a Dataset of the specified type.

3.1.3 Using the rdd[T]() Query Method

SparkContext.**rdd[T]()** executes a query or loads a table, formats each row of the result set with the provided function, and returns an RDD[T] containing the formatted rows.

Defining a Type Format[T] function

The SparkContext.**rdd[T]()** method executes a query on the cluster and returns an RDD where each element has been formatted as an instance of the specified type T. Formatting is performed by a user-defined function of type Format[T]. For example, the following code specifies a function **pair**, which extracts a pair of strings from the first two columns of the current row of a result set:

```
val pair: Format[(String,String)]  =  r => (r.getString(1),r.getString(2))
```

This function constructs an `RDD[(String,String)]` from the results of any query of the cluster that includes at least two strings per record.

Using SparkContext.rdd[T]()

The following example calls **rdd[T]()** using the previously defined Format[T] function:

```
val asPair: Format[(Int,Double)] = r => (r.getInt(1),r.getDouble(2))
val newRDD = spark.sparkContext
                  .rdd("myTable",1,asPair)
```

Note: Format functions will be invoked for each and every record requested by the client, and therefore should normally restrict themselves to calling only pure (that is, non side effecting) member functions of the result set (**getInt**, **getDouble**, **getDate** and the like).

3.2 Using Extension Methods to Set Data Source Options

In addition to generic option support (see "Spark Connector Data Source Options"), the Spark Connector also provides a number of methods that allow many of these options to be specified as method arguments. These methods can simplify code and improve type safety by providing alternatives to the generic option interface.

The standard way to set data source options is with the generic option interface (see "Using Spark Connector Generic Options"), but most data source options can also be set with Spark Connector extension methods or method parameters.

- Setting the dbtable Query and Save Options — setting *dbtable* with query methods and the **iris()** save method.
- Extension Methods for InterSystems IRIS Save Options — setting the *description*, *publicrowid*, *shard*, and *autobalance* options.
- Setting Connection Options with the address() Method — setting the *url*, *user*, and *password* options
- Setting Partitioning Options with Query Method Parameters — setting the *mpfi*, *partitionColumn*, *lowerBound*, *upperBound*, and *numPartitions* with optional query method parameters.

Note: This chapter does not discuss these options in detail. For complete information on data source option settings, see the descriptions in the previous chapter ("Spark Connector Data Source Options").

3.2.1 Setting the dbtable Query and Save Options

The *dbtable* option is specified as the first argument of all query and save extension methods (see "Using Query and Save Extension Methods" earlier in this chapter), as demonstrated in the following examples

Setting the dbtable query option with query methods

All of the query extension methods (DataFrameReader.**iris()**, SparkContext.**rdd[T]()**, and SparkSession.**dataset[T]()**) accept the *dbtable* (alias *query*) option as the first argument. For example, the following code fragments specify *dbtable* as either a table name or an SQL query, and use the DataFrameReader.**iris()** method to set the format, set the *dbtable* option, and call **load()**:

```
  var df = spark.read.iris("mytable")   // Load a table

  var df = spark.read.iris("SELECT * FROM mytable")   // Perform a query
```

rather than:

```
var df = spark.read.format("iris").option("dbtable","mytable").load()   // Load a table

var df = spark.read.format("iris").option("query","SELECT * FROM mytable").load()   // Perform
 a query
```

See "Using the iris() and dataset[T]() Query Methods" and "Using the rdd[T]() Query Method" earlier in this chapter for detailed information about the query methods. For more on the *dbtable* query option, see the dbtable entry under Standard Save Options in the previous chapter.

Setting the dbtable save option with DataFrameWriter.iris()

When saving with the DataFrameWriter.**iris()** extension method, the *dbtable* option is specified as the method parameter. For example, the following code uses DataFrameWriter.**iris()** to set the format, set the *dbtable* option, and call **save()**:

```
spark.write.iris("mytable")
```

rather than:

```
spark.write.format("iris")
           .option("dbtable","mytable")   // Save a table
           .save()
```

See "Using the iris() Save Method" earlier in this chapter for detailed information about this extension method. For more on the *dbtable* save option, see the dbtable (or query) entry under Query Options in the previous chapter.

3.2.2 Extension Methods for InterSystems IRIS Save Options

The save options (*description*, *publicrowid*, *shard*, and *autobalance*) can also be set with DataFrameWriter extension methods. See "InterSystems IRIS Save Options" in the previous chapter for detailed information on these options.

description()

The DataFrameWriter.**description()** extension method is an alternate way to set the *description* option. For example, you can use the following code:

```
spark.write.description("This is a table.")
```

rather than:

```
df.write.format("iris").option("description","This is a table.").save()
```

See description in the previous chapter for more information on this option.

publicRowID()

The DataFrameWriter.**publicRowID()** extension method is an alternate way to set the *publicrowid* option. For example, you can use the following code:

```
spark.write.publicRowID("false")
```

rather than:

```
df.write.format("iris").option("publicrowid","false").save()
```

See publicrowid in the previous chapter for more information on this option.

shard()

The DataFrameWriter.**shard()** extension method is an alternate way to set the *shard* option. For example, you can use the following code:

```
spark.write.shard("columnA","columnX")
```

rather than:

```
df.write.format("iris") .option("shard",Seq("columnA","columnX")).save()
```

Notice that this method call automatically interprets the arguments as a Seq.

See shard in the previous chapter for more information on this option.

autobalance()

The DataFrameWriter.**autobalance()** extension method is an alternate way to set the *autobalance* option. For example, you can use the following code:

```
spark.write.autobalance("false")
```

rather than:

```
df.write.format("iris").option("autobalance","false").save()
```

See autobalance in the previous chapter for more information on this option.

3.2.3 Setting Connection Options with the address() Method

The *url*, *user*, and *password* options explicitly specify connection string values for a query or save (see "Connection Options" in the previous chapter for detailed information and examples). These connection options can also be set as parameters of the DataFrameReader.**address()** and DataFrameWriter.**address()** extension methods, as demonstrated in the following examples.

Querying with DataFrameReader.address()

The following code uses the DataFrameReader.**address()** method to set the *url*, *user*, and *password* options, and then uses the DataFrameReader.**iris()** method to set the format, set the *dbtable* option, and call **load()**:

```
var df = spark.read.address("IRIS://localhost:51773/USER", "_system", "SYS").iris("mytable")
```

rather than:

```
var df = spark.read.format("iris")
                   .option("dbtable","mytable")
                   .option("url", "IRIS://localhost:51773/USER")
                   .option("user", "_system")
                   .option"password", "SYS")
                   .load()
```

Saving with DataFrameWriter.address()

The following code uses the DataFrameWriter.**address()** method to set the *url*, *user*, and *password* options, and then uses the DataFrameWriter.**iris()** method to set the format, set the *dbtable* option, and call **save()**:

```
df.write.address("IRIS://localhost:51773/USER", "_system", "SYS").iris("mytable")
```

rather than:

```
df.write.format("iris")
        .option("dbtable","mytable")
        .option("url", "IRIS://localhost:51773/USER")
        .option("user", "_system")
        .option"password", "SYS")
        .save()
```

3.2.4 Setting Partitioning Options with Query Method Parameters

All of the query extension methods (DataFrameReader.**iris()**, SparkContext.**rdd[T]()**, and SparkSession.**dataset[T]()**) accept optional arguments that specify how to partition a query. See "Using Query and Save Extension Methods" earlier in this chapter for a discussion of query methods. The following examples use the **iris()** method, but all three methods accept the same set of arguments.

Implicit partitioning with the mfpi parameter

The *mfpi* option can be specified as a argument when reading with any of the query extension methods. For example, the following call to DataFrameReader.**iris()** specifies the first argument as the value for *dbtable* and the second argument as the value for *mfpi*:

```
var df = spark.read.iris("SELECT * FROM table",2)
```

This call is exactly equivalent to the following example:

```
var df = spark.read.format("iris")
                   .option("dbtable","SELECT * FROM mytable")
                   .option("mfpi", 2)
                   .load()
```

See “Implicit Partitioning” in the previous chapter for more information on this option.

Explicit partitioning with the partitionColumn, lowerBound, upperBound, and numPartitions parameters

These options can be specified as arguments when reading with any of the query extension methods. For example, the following call to DataFrameReader.**iris()** specifies the first argument as the value for *dbtable* and the remaining arguments as the values for *partitionColumn*, *lowerBound*, *upperBound*, and *numPartitions*:

```
var df = spark.read.iris("SELECT * FROM mytable","column",0,10000,2)
```

This call is exactly equivalent to the following example:

```
var df = spark.read.format("iris")
                   .option("dbtable","SELECT * FROM mytable")
                   .option("partitionColumn", "columnA")
                   .option("lowerBound", 0)
                   .option("upperBound", 10000)
                   .option("numPartitions", 2)
                   .load()
```

See “Explicit Partitioning” in the previous chapter for more information on these options.

4

Spark Connector Best Practices

This chapter describes the preferred way to set up a Spark cluster on the InterSystems IRIS Data Platform™. The following topics are discussed:

- Configuration — describes and explains the preferred Spark cluster topology.
- Queries — provides examples that demonstrate how to optimize query performance.
- Loading Data — discusses how to optimize loading of large datasets into sharded clusters.

Note: **Pre-installed Spark in Containers**

The InterSystems Cloud Manager (ICM) provides you with a simple, intuitive way to provision cloud infrastructure and deploy services on it. See the "Using ICM" chapter of the *InterSystems Cloud Manager Guide* for instructions on how to configure and deploy containers in a topology similar to the one described in this chapter. The "Deploy and Manage Services" section provides specific instructions for using images that include pre-installed instances of Spark.

4.1 Configuration

4.1.1 Intended topology

1. The InterSystems IRIS™ shard master **I**0 and Spark master **S**0 both run on host machine **H**0.
2. A Spark slave **s**0 also runs on host machine **H**0.
3. Furthermore, for each additional distinct host machine **H**1 ... **H**n hosting an InterSystems IRIS shard server instance **I**1 ... **I**n , exactly one Spark slave **s***i* also runs on **H***i*
4. Optionally, for each additional distinct host machine **H**1 ... **H**n hosting an InterSystems IRIS shard server instance **I**1 ... **I**n , exactly one HDFS data node **d***i* also runs on **H***i*
5. The Spark cluster runs in what the Spark documentation refers to as *Standalone Mode*.

Host	IRIS	Spark	HDFS
H0	**I**0	**S**0	**D**0
		s0	**d**0
H1	**I**1	**s**1	**d***i*
...	...	...	...
Hn	**I**n	**s**n	**d**n

4.1.2 Rationale

While other topologies are certainly possible and will work (in the sense that Spark programs will save and restore data correctly and compute correct results), this specific topology will generally perform best because:

1. Each Spark slave is collocated with exactly one InterSystems IRIS shard server, and the connector is able to exploit this fact by requesting that a dataset partition requiring data from shard server **I***i* be scheduled for execution on host machine **H***i*. In other words, the process is executed on the same machine as the data on which it depends. Thus movement of data across the network connecting the machines **H***i* is greatly reduced.
2. Each Spark slave creates Spark worker processes that are multithreaded. The Spark master is aware of the number of processing cores available to each worker, and distributes tasks amongst them accordingly (subject to 1. above). Thus there is no advantage in running more than one Spark slave on each host machine.
3. Not all tables are sharded. Un-sharded tables are stored on the InterSystems IRIS shard master **I**0. (The partitions of) datasets reading or writing such tables are best scheduled for execution on a Spark slave that is collocated with this instance, hence there should also be a Spark slave **s**0 running on **H**0 to service these requests.
4. When loading large data sets into a cluster, there are significant advantages to installing HDFS on the nodes of the cluster (see below).

4.1.3 Hardware

While more cores, RAM, and disk storage are obviously merrier than less, one generally has a choice between a few powerful host machines, or more, less powerful hosts. A sweet spot exists, however, when the largest tables in the database fit comfortably within the collective RAM of all the shard servers in the cluster, for then the cluster effectively behaves as a scalable cache. This is especially true for large scale analytics of predominantly static or infrequently changing 'data lakes'.

Special consideration should be given to the machine **H**0 that hosts the InterSystems IRIS shard master. Complex queries, particularly those with aggregates, ORDER BY, TOP, or DISTINCT clauses and the like, generally require a non-trivial reduction of the intermediate results returned by the individual shard servers, and this computation is staged in temporary tables on the shard master. For a query that returns a large result set that must be sorted, for example, the reduction step on the shard master (the sort) can easily dominate the rest of the computation.

4.2 Queries

When designing an application to run in this environment, one frequently has a choice as to where the aggregation will take place. Consider the following Spark queries:

Example 1

```
        > val q1 = Spark.read.iris("SELECT max(i) FROM A")
        res1: org.apache.Spark.sql.DataFrame = [max(i): int]
```

Example 2

```
        > val q2 = Spark.read.iris("A").selectExpr("MAX(i)")
        res2: org.apache.Spark.sql.DataFrame = [max(i): int]
        > Spark.read.iris("A").createOrReplaceTempTable("A")
```

Example 3

```
        > val q3 = Spark.sql("SELECT max(i) FROM B")
        res3: org.apache.Spark.sql.DataFrame = [max(i): int]
```

where *A* is a sharded table with a numeric column named *i*. All three queries compute the same single value, namely the maximum value of column *i*, but they each accomplish this task in rather different ways:

Example 2 fetches the entire column *i* of table *A* (though, thanks to the connector's ability to automatically prune columns, only column *i* and not all of *A*) into the Spark slaves, each of which then computes the maximum value of its portion of the data. These local maximum are then sent to the Spark master, which picks the single greatest value amongst them and returns it to the driver program in a DataFrame.

To see what's going on, it can be helpful to take a look at the output of the DataFrame's **explain** operator:

```
> q2.explain
== Physical Plan ==
*HashAggregate(keys=[], functions=[max(i#106)])
+- Exchange SinglePartition
   +- *HashAggregate(keys=[], functions=[partial_max(i#106)])
      +- *Scan Relation(SELECT * FROM (A)) [i#106] ReadSchema: struct<i:int>
```

Notice:

- the `Scan`, which corresponds to the parallelized fetch of the data in each of the slaves,
- the inner `HashAggregate`, which corresponds to the work that each slave will perform in parallel in locating its local maximum, and
- the `Exchange` and subsequent `HashAggregate`, which corresponds to the transmission of the local maxima back to the master and subsequent selection of the global maximum.

Example 3 demonstrates the use of Spark's own built-in SQL interpreter. For more complex queries, especially those that are federated across multiple databases or other disparate sources of data, this can be provide a powerful and flexible way of treating the various sources like a single giant database.

In the case, however, beneath the covers exactly the same work is performed as in Example 2, as can be seen with the **explain** operator.

Example 1 on the other hand, demonstrates the power of using Spark with InterSystems IRIS: now the entire query is performed within the InterSystems IRIS database, and is thus amenable to the InterSystems IRIS query optimization, the use if indexes, and so on:

```
> q1.explain
== Physical Plan ==
*Scan Relation(select max(i) from B) [Aggregate_1#215] ReadSchema: struct<Aggregate_1:int>
```

4.3 Loading Data

Spark is particularly useful for loading large datasets into sharded InterSystems IRIS clusters.

For best performance and maximum parallelism, you will want to install HDFS on the nodes of the cluster too. In this scenario, each host **H***i* should also acts as a data node **d***i*.

Example: loading a large CSV file into a sharded table.

```
~>  hadoop fs -put x.csv /x.csv
  # Copy file "X.csv" into HDFS as /x.csv
```

And now from within Spark:

```
scala> val df = Spark.read.options( ... )
                          .csv("hdfs:///host:port/x.csv")  // Read local partitions
scala> df.write.shard(true).options( ... )
                           .autobalance(false).iris("X")   // Write to local shards
```

In this example, each Spark slave reads its own locally available portion of the original file, which was split and distributed to the nodes of the cluster automatically by HDFS in step 1, and writes the parsed records to its own co-located shard: thus no data moves across the network once the source file has been moved into HDFS.

5 Spark Connector Internals

This chapter covers datatype mapping, predicate pushdown operators, and other useful information not otherwise available.

- SQL/Spark Datatype Mapping — a table of SQL datatypes and their corresponding Spark datatypes.
- Predicate Pushdown Operators — a list of Spark operators that the Spark Connector recognizes as having direct counterparts within the underlying database.
- Logging — the Spark connector, like the Spark system itself, uses Log4J to log events of interest.
- Known Issues — problems to be aware of.

5.1 SQL/Spark Datatype Mapping

Internally, the Connector uses the InterSystems JDBC driver to read and write values to and from servers. This constrains the data types that can be serialized in and out of database tables via Spark. The JDBC driver exposes the JDBC data types in the following tables as available projections for InterSystems IRIS™ data types, and converts them to and from the listed Spark Catalyst types (members of the org.apache.spark.sql.types package).

This mapping between Spark Catalyst and JDBC data types differs subtly from that used by the standard Spark jdbc data source, as noted in the following sections.

JDBC / Spark Value Type Conversions

The following JDBC value types are exposed, and are converted directly to and from the listed Spark Catalyst types:

JDBC Value Type	Spark Type
BIT	BooleanType
TINYINT	ByteType
BIGINT	LongType
INTEGER	IntegerType
SMALLINT	ShortType
DOUBLE	DoubleType
DOUBLE	FloatType

Note:
- Value types are all qualified as being `NOT NULL` when saved to a table.
- Spark Catalyst distinguishes between the different sizes of integer TINYINT, SMALLINT, INTEGER, BIGINT, whereas the Spark jdbc data source does not.

JDBC / Spark Object Type Conversions

The following JDBC object types are exposed, and are represented by the listed Spark Catalyst types. Bidirectional conversion is not supported for LONGVARCHAR, GUID, LONGVARBINARY, and TIME because these JDBC types do not correspond to unique Spark Catalyst types:

JDBC to Spark Catalyst projections		Spark Catalyst to JDBC projections	
JDBC Object Type	**Spark Type**	**Spark Type**	**JDBC Object Type**
VARCHAR, GUID, and LONGVARCHAR	StringType	StringType	VARCHAR
VARBINARY and LONGVARBINARY	BinaryType	BinaryType	VARBINARY
NUMERIC(p,s)	DecimalType(p,s)	DecimalType(p,s)	NUMERIC(p,s)
DATE	DateType	DateType	DATE
TIMESTAMP and TIME	TimestampType	TimestampType	TIMESTAMP

Note:
- In NUMERIC and DecimalType the positive integers *p* and *s* respectively denote the precision and scale of the numeric representation.
- There is no Spark SQL encoder currently available for type java.sql.Time, so JDBC TIME is represented as Spark TimestampType.
- Spark Catalyst (unlike the Spark jdbc data source) recognizes the UNIQUEIDENTIFIER data type, which is not widely supported by all JDBC vendors
- JDBC type GUID is the JDBC representation of Intersystems SQL datatype UNIQUEIDENTIFIER.

See "InterSystems SQL Data Types" for details on how values are represented within InterSystems IRIS, and "DDL Data Types Exposed by InterSystems ODBC / JDBC" for specific information on how they are projected to JDBC.

5.2 Predicate Pushdown Operators

The Spark Connector currently recognizes the following Spark operators as having direct counterparts within the underlying database (see "Overview of Predicates" in the *SQL Reference*):

EqualTo	LessThanOrEqual	StringStartsWith	IsNull	And
LessThan	GreaterThanOrEqual	StringEndsWith	IsNotNull	Not
GreaterThan	StringContains	In	Or	

5.3 Logging

The connector logs various events of interest using the same infrastructure as the Spark system itself uses, namely Log4J.

The content, format, and destination of the system as a whole is configured by the file ${SPARK_HOME}/conf/log4j-defaults. The connector is implemented in classes that reside in a package named com.intersystems.spark and so can easily be configured by specifying keys of the form:

```
log4j.logger.com.intersystems.spark                      = INFO
log4j.logger.com.intersystems.spark.core                 = DEBUG
log4j.logger.com.intersystems.spark.core.IRISDataSource = ALL
...
```

5.4 Known Issues

We hope to address the following issues in a subsequent release:

- Pruning Columns with Synthetic Names
- Java 9 Compatibility
- Handling of TINYINT
- JDBC Isolation Levels

5.4.1 Pruning Columns with Synthetic Names

Consider the following query Spark session:

```
scala> spark.read.iris("select a, min(a) from A")
```

where A is some table that presumably has a column named `a`.

Notice that no alias is provided for the selection expression `min(a)`. The server synthesizes names for such columns, and in this case might describe the schema for the resulting dataframe as having two columns, named 'a' and 'Aggregate_2' respectively.

No actual field named 'Aggregate_2' exists in the table however, so an attempt to reference it in an enclosing selection would fail:

```
scala> val df = spark.read.iris("select Aggregate_2 from (select a, min(a) from A)")
> SQL ERROR: No such field SQLUSER.A.Aggregate_2 ....
```

This is to be expected in a standard SQL implementation.

The connector uses just such enclosing projection expressions as these however when attempting to prune the columns of a dataframe to those that are actually referenced in subsequent code:

```
scala> spark.read.iris("select a, min(a) from A").select("a")...
```

internally generates the query `select a from(select a, min(a) from A)` to be executed on the server in order to minimize the motion of data into the spark cluster.

As a result, the connector cannot efficiently prune columns with synthetic names and instead resorts to fetching the entire result set:

```
scala> spark.read.iris("select a, min(a) from A").select("Aggregate_2")
```

internally generates the query `select * from(select a, min(a) from A)`.

For this reason, you should consider modifying the original query by attaching aliases to columns that would otherwise receive server synthesized names.

We hope to address this issue in a subsequent release.

5.4.2 Java 9 Compatibility

Java 9, and the JVM 1.9 on which it runs, became available for general release in September 2017. Neither Apache Spark nor the InterSystems Spark Connector currently run on this version of the JVM. We hope to address this issue in a subsequent release.

5.4.3 Handling of TINYINT

The mapping between Spark Catalyst and JDBC datatypes (see "SQL/Spark Datatype Mapping" earlier in this chapter) differs subtly from that used by the Spark jdbc data source. The Connector achieves this mapping by automatically installing its own subclass of class org.apache.spark.sql.jdbc.JdbcDialect but this also has the side effect of changing the mapping used by Spark JDBC itself.

By and large this is a good thing, but one problem that has been identified recently is that due to a bug in Spark 2.1.1, which neglects to implement a low level reader function for the ByteType, attempting to read an InterSystems IRIS table with a column of type TINYINT using the Spark jdbc data source will fail once the Connector has been loaded.

For now, it is probably best to avoid reading and writing DataFrames using the Spark jdbc data source directly once the Connector has been loaded. We hope to address this issue in a subsequent release.

5.4.4 JDBC Isolation Levels

The InterSystems IRIS server does not currently support the writing of a dataset to a SQL table using JDBC isolation levels other than `NONE` and `READ_UNCOMITTED`. We hope to address this issue in a subsequent release.

6 Spark Connector Quick Reference

This chapter is a quick reference to the InterSystems Spark Connector API methods, which define a custom Scala interface for the Spark Connector. It extends the generic Spark interface with a set of implicit Scala classes and types that provide more convenience and better type safety.

Note: This is not a definitive reference for this API. It is intended as a "cheat sheet" containing only short descriptions of members and parameters, plus links to detailed documentation and examples.

For the most complete and up-to-date information, see the Spark Connector online documentation.

The Spark Connector API provides the following extension methods to Spark Scala classes:

- DataFrameReader Extension Methods:
  - **iris()** — executes a query or loads a table, tunes partitioning, and returns results in a DataFrame.
  - **address()** — specifies the connection details of the cluster to read from
- DataFrameWriter Extension Methods:
  - **iris()** — saves a DataFrame to the given table on the cluster.
  - **address()** — specifies the connection details of the cluster to write to.
  - **description**() — specifies an arbitrary description for the newly created table.
  - **publicRowID()** — specifies whether the master RowID field of the newly created table should be publicly visible.
  - **shard()** — specifies the shard key for the newly created table.
  - **autobalance()** — specifies how to distribute records saved to a table using a system assigned shard key.
- SparkSession and SparkContext Extension Methods:
  - **rdd[T]()** — executes a query or loads a table, tunes partitioning, formats each row of the result set with a specified function, and returns an RDD containing the formatted data.
  - **dataset[T]()** — executes a query or loads a table, tunes partitioning, and returns results in a Dataset with elements of type T.
- ml.Pipeline Extension Method:
  - **iscSave()** — saves a model to an InterSystems IRIS PMML definition class.

6.1 Spark Connector Method Reference

Each entry in this section provides a brief method overview including signatures and parameter descriptions. Most entries also include links to examples and more detailed information.

6.1.1 DataFrameReader Extension Methods

iris()

DataFrameReader.**iris()** executes a query on the cluster or loads the specified table, tunes partitioning, and returns results in a DataFrame. See "Using the iris() Read and Write Methods" for more information and examples.

```
def iris(text: String,mfpi: N = 1): DataFrame
```

```
def iris(text: String,column: String,lo: Long,hi: Long,partitions: N): DataFrame
```

- *text* — text of a query to be executed, or name of a table to load.
- *column*, *lo*, *hi*, *partitions*— allow you to explicitly specify partitioning parameters.
- *mfpi* — allows the server to determine partitioning parameters within certain limits.

See "Partition Tuning Options" for detailed information and examples concerning partitioning parameters (*column*, *lo*, *hi*, *partitions*, and *mfpi*).

address()

DataFrameReader.**address()** specifies the connection details of the cluster to read from. Overrides the default instance specified in the Spark configuration for the duration of this read operation.

```
def address(url: String,user: String = "",password: String = ""): DataFrameReader
```

- *url*, *user*, *password* — String values used to define a JDBC connection to the master server.

See "Connection Options" for more information and examples.

6.1.2 DataFrameWriter Extension Methods

iris()

DataFrameWriter.**iris()** saves a DataFrame to the specified table on the cluster. See "Using the iris() Read and Write Methods" for detailed information and examples.

```
def iris(table: String): Unit
```

- *table* — a String containing the name of the table to write to.

See "CREATE TABLE" in the *InterSystems SQL Reference* for more information on table creation options.

address()

DataFrameWriter.**address()** specifies the connection details of the cluster to write to. Overrides the default instance specified in the Spark configuration for the duration of this write operation.

```
def address(url: String,user: String = "",password: String = ""): DataFrameWriter[T]
```

- *url*, *user*, *password* — String values used to define a JDBC connection to the master server.

See “Connection Options” for more information and examples.

description()

DataFrameWriter.**description()** specifies a description to document the newly created table.

```
def description(value: String): DataFrameWriter[T]
```

- *value* — a String containing an arbitrary description for the table.

See “InterSystems IRIS Save Options” for more information and examples.

publicRowID()

DataFrameWriter.**publicRowID()** specifies whether the master RowID field of the newly created table should be publicly visible.

```
def publicRowID(value: Boolean): DataFrameWriter[T]
```

- *value* — a Boolean indicating whether the table should be publicly visible.

See “InterSystems IRIS Save Options” for more information and examples.

shard()

DataFrameWriter.**shard()** specifies the user defined shard key for the newly created table, or specifies whether the newly created table is to be sharded. Has no effect if the table already exists and the save mode is anything other than `OVERWRITE`.

```
def shard(fields: String*): DataFrameWriter[T]
def shard(value: Boolean): DataFrameWriter[T]
```

- *fields* — A String sequence of field names (possibly empty) to be used as the user defined shard key. If the sequence is empty then the table will be sharded on the system assigned key.
- *value* — Boolean value indicating whether the newly created table is to be sharded

See “InterSystems IRIS Save Options” for more information and examples.

autobalance()

Specifies whether or not inserted records should be evenly distributed among the available shards of the cluster when saving to a table that is sharded on a system assigned shard key. Has no effect if the table is not sharded, or is sharded using a custom shard key.

```
def autobalance(value: Boolean): DataFrameWriter[T]
```

- *value* — Boolean `true` to evenly distribute records amongst the available shards of the cluster, or `false` to save records into shards that are 'closest' to where the partitions of the dataset reside.

See “InterSystems IRIS Save Options” for more information and examples.

6.1.3 SparkSession and SparkContext Extension Methods

dataset[T]()

SparkSession.**dataset[T]()** executes a query on the cluster or loads the specified table, optionally tunes partitioning, and returns results in a Dataset[T]s. See “Using the dataset[T]() and rdd[T]() Methods” for more information and examples.

```
def dataset[T](text: String, mfpi: N = 1)(implicit arg0: Encoder[T]): Dataset[T]

def dataset[T](text: String, column: String, lo: Long, hi: Long, partitions: N)(implicit arg0:
 Encoder[T]): Dataset[T]
```

- *text* — text of a query to be executed, or name of a table to load.
- *column*, *lo*, *hi*, *partitions*— allow you to explicitly specify partitioning parameters.
- *mfpi* — allows the server to determine partitioning parameters within certain limits.

See "Partition Tuning Options" for detailed information and examples concerning partitioning options.

rdd[T]()

SparkContext.**rdd[T]()** executes a query on the cluster or loads the specified table, optionally tunes partitioning, and computes a suitably partitioned RDD whose elements are formatted by the provided formatting function (an instance of Format[T]). See "Using the dataset[T]() and rdd[T]() Methods" for more information and examples.

```
def rdd[T](text: String, mfpi: N, format: Format[T])(implicit arg0: ClassTag[T]): RDD[T]

def rdd[T](text: String, column: String, lo: Long, hi: Long, partitions: N, format:
Format[T])(implicit arg0: ClassTag[T]): RDD[T]
```

- *text* — text of a query to be executed, or name of a table to load.
- *column*, *lo*, *hi*, *partitions*— allow you to explicitly specify partitioning parameters.
- *mfpi* — allows the server to determine partitioning parameters within certain limits.
- *format* — an instance of Format[T] containing the function used to encode each row of the result set.

See "Partition Tuning Options" for concerning partitioning parameters (*column*, *lo*, *hi*, *partitions*, and *mfpi*).

See "Format[T]" for more information on creating and using an instance of Format[T].

6.1.4 ml.PipelineModel Extension Method

iscSave()

ml.PipelineModel.**iscSave()** saves the PMML definition for the given model as a Class Definition on the master instance identified by *address* for subsequent execution on the cluster. See "Using PMML Models with InterSystems Products" for more information and examples.

```
def iscSave(klass: String,schema: StructType,address: Address = Address()): Unit
```

- *klass* — name of the class definition to create. If this class already exists it will be overwritten.
- *schema* — The schema of the model's source Dataset, on which the PMML file's data dictionary will be based.
- *address* — The master instance to which the class will be written.