1.0.0.3 - Beta 3

# Table of Contents

# Part I

# 1 The SqlRDD

## Welcome to X# SQL RDD 1.0.0.3 - Beta 3

Click here for the version history 40

## 1.1 Welcome to the SQLRDD

We are proud with the release of the SQLRDD

The X# SQL RDD was created to help fill the void that is created because SAP no longer develops the Advantage Database Server (ADS).
ADS is/was used by many of our clients because it offers a reliable and fast way to access data in a networked environment. Data stored in traditional DBF files and customized ADS specific data files. This data can be accessed both in Index/Sequential (Record oriented) mode and with SQL syntax in set-oriented mode.
As with all our development we try to avoid reinventing the wheel, so we did not want to add a SQL engine on top of the existing RDDs that we created to access DBF based data. Instead, we have chosen to build record-oriented data access on top of existing SQL engines, such as SQL Server, Oracle, MySql, Postgress etc.

We have done our best to ensure that this RDD will work with your existing code with as little modifications as possible.
Before you get started please read at least the contents of the first few chapters of this help file, since they describe the problems that you may encounter during integrating the SQLRDD in your application:


- Compatibility with the existing RDD's 30
- Basic Usage 19
- Indexes 21
- Version History 40


X# RDD's can be used in three different ways:
1. Using the (Clipper compatible) Db..() functions
2. Using the strongly typed VoDb..() versions of these functions
3. Using the DbServer class and its methods

In most of our sample code we have used method 1, but of course everything that can be done with the Db..() functions can also be done with the other two methods. Please remember that when you see a problem using the DbServer class, that the DbServer class may expect DBF specific behavior and may not always be handling the SQL environment well.

The RDD can operate in 2 modes:
- Query mode.
  This happens when the table name is a select statement, like "Select * from Customers"
  In Query mode the RDD will retrieve all the rows that are returned from the select statement.
  In Query mode the RDD is read only
- Table mode
  This happens when the table name is a simple name, like "Customers" or "Invoices"
  In Table mode you can control the # of rows that are returned from the server with properties such as MaxRecords, ServerFilter etc. See the chapters about the MetaData Providers 8 for more information.
  In Table mode the RDD is read/write

## 1.2 Database Independency

X# is compiling applications for .Net. And .Net already has a great data access layer in the form of Ado.Net. Ado.Net already provides several facilities to work with a common syntax with various databases. There are common classes for connections, commands and resultsets and each database vendor can create its own database specific implementation of these objects.

We have chosen to make the SQL RDD database agnostic. It should be able to work with any database for which an Ado.Net Database provider exists. The RDD works against the common layer that is supported by all DBMS-es.

The implementation of the Ado.Net database providers works great for many things, but unfortunately it lacks support to handle the differences in the syntax for Data Definition Language (DDL) statements, such as “CREATE TABLE” and “DROP INDEX” and also the syntax to retrieve a certain number of rows from a result set is different between providers. Some providers have a “SELECT TOP <n>” syntax, where others use a LIMIT clause “LIMIT <n>” to limit the number of rows in the result set. Also, the syntax to retrieve the value of the last inserted “autonumber” column differs, as well as the available datatypes for the database.

To solve that problem, we have introduced an abstract SqlDbProvider class in the SQLRDD (in the XSharp.RDD.SqlRDD.Providers namespace) that needs to be implemented, to work with a specific version of a SQL database. This class contains information about the assemblyname and class name of the DbProviderFactory class implemented for that database. The DbProviderFactory class is used to create DBMS specific connection, command, parameter classes etc. The SqlDbProvider class inside the SQLRDD also has properties that declare the syntax for the various DDL statements and has a method that can translate a DBF based column definition into the proper syntax for the target database.

The SQLRDD comes with implementations of the SqlDbProvider class for SqlServer, MySql, PostgreSQL ODBC, OleDB, Oracle and SQLite

Custom implementations can be created by inheriting from one of the built-in providers, or by directly inheriting from SqlDbProvider.

Please see the class documentation for the SQLRDD for the properties and methods in the SqlDbProvider class.

## 1.3 Record Oriented vs Set oriented.

Since the SQLRDD uses SQL databases, it is not really complicated to support retrieving data with a SQL statement. However, emulating Record Oriented data access is a bit more complicated. There are several decisions to be made when for example you would open a customers table with the SQL RDD like this:

```
USE Customers
```

Some of the things we need to consider:

- Retrieve all the rows from the database, or limit the # of rows (do we really want to retrieve a million rows when opening a table, when you're just interested in a particular customer)
- Retrieve all the columns or limit the # of columns to the ones you want to access.
- DBF files have the concept of a record number. SQL databases do not have that. We can emulate a record number, based on the ordinal position of a row in the result set, but that means that changing the order of the rows will result in different record numbers or a row after changing the order of the result set.
- In DBF files rows are not directly deleted but marked for deletion. You can also undo that by calling RECALL. In SQL environments deleted rows are gone immediately
- Column names in DBF files are limited to 10 characters only. SQL does not have that limitation
- Column names in DBF files may be reserved words in the SQL database. How to handle that?
- If you retrieve a character column from a DBF file, it will be always padded with spaces to its maximum length.
  Most SQL databases do not pad the column values but return "trimmed" values. How to handle that?
- Which indexes are available?
- Which index tags are available for each index and what are the Xbase expressions for these orders?

We have decided to add a Metadata 8 layer to the SQLRDD that allows you to control how the RDD handles these issues. The next chapter describes that layer.

## 1.4 MetaData

To allow you to emulate record-oriented data access as best as possible, we have created a mechanism (a so called MetadataProvider) that your application can use to communicate with the RDD and tell the RDD what to do for a particular table. Maybe you want to limit the # of rows or columns for one relatively small table, but not for a large table with transactions.
The SQLRDD contains 3 built-in mechanisms to provider this metadata (again in the XSharp.RDD.SqlRDD.Providers namespace)

- SqlMetadataProviderIni, which gets its information from an ini-file
- SqlMetadataProviderCallBack, which gets its information from a call back function in code.
- SqlMetadataProviderDatabase, which gets its information from tables inside the SQL database.

Further in this document we will describe the 3 providers and how they work.
And of course you can inherit from one of these 3 providers or create your own provider that implements the IMetadataProvider interface.

For performance reasons the built in providers read values only once for a table, as long as the connection is open. The tables are stored in a cache in the SqlMetadataProviderAbstract class.

Per connection 15 you can setup the MetadataProvider. If you do not assign a MetadataProvider to a connection, then the RDD will assume that you use the SqlMetadataProviderIni, unless you create the connection with a callback event handler. In that case the SqlMetadataProviderCallBack will be used.

### 1.4.1 IniFile

SqlMetadataProviderIni gets its information from an INI file that must be located in the application folder.
This provider is the default provider for the SQLRDD, except when you open a connection with a callback event handler. In that case the RDD will assume that you're using callbacks to provide the metadata.

The default INI file name is SqlRdd.Ini, but of course you can choose your own name. If the ini file does not exist then the RDD will fall back to default values.
The provider checks for an optional section with Default values, a section per Table that gets opened and a section per index and tag that are listed in the Table section and index section.
The provider looks for the following entries in the [Defaults] and table sections. These values will be used as default for tables where the value is missing.
Optional columns fall back to the settings of the connection.

| Name | Type | Default | Mandatory | Description |
|---|---|---|---|---|
| AllowUpdates | Logic | Yes | No | Are updates allowed? |
| ColumnList | String | No | No | A comma separated list of names of the columns that should be read. Defaults to all columns |
| CompareMemo | Logic | Yes | No | Should memo columns be included when generating where clause for updates or deletes? |
| DeletedColumn | String | Yes | No | The name of the column that emulates the deleted flag |
| Indexes | String | No | No | A comma separated list of index names. For each index the provider will retrieve he tlist of tags |
| KeyColumns | String | No | No | A comma separated list of names of the columns that should be used as key columns for where clauses for update/ delete statements. Defaults to all columns, except when CompareMemo is set to false. |
| LegacyFieldTypes | Logic | Yes | No | Should field types be mapped to the VO compatible field types (CDLMN) or can the RDD also return newer (FoxPro) types such as Currency (Y) and DateTime (T) |
| LongFieldNames | Logic | Yes | No | Should long field names be used |
| MaxRecords | Numeric | Yes | No | The maximum number of rows to return |
| MaxRecnoAsRecCount | Logic | No | No | Should RecCount return the maximum record number or the physical # of rows in the table |

| Name | Type | Default | Mandatory | Description |
|---|---|---|---|---|
| RealName | String | No | No | This real table name of the table on the server. |
| RecnoColumn | String | Yes | No | The name of the column that emulates the recordnumber column |
| SeekReturnsSubset | Logic | Yes | No | Should a Seek operation return a subset of the rows matching the key, or should it return all rows and position the cursor on the first row that matches the key. |
| ServerFilter | String | No | No | An additional filter that can be used to filter rows. This must be in SQL syntax. |
| TrimTrailingSpaces | Logic | Yes | No | Should trailing spaces for string columns be trimmed when |
| UpdatableColumns | String | No | No | A comma separated list of names of the columns that may be updated. Defaults to all columns |
| UpdateAllColumns | Logic | Yes | No | Should all columns be included (written) in update statements, or only the changed columns |

Index sections are named [Index:IndexName]
They contain one entry

| Name | Type | Mandatory | Description |
|---|---|---|---|
| Tags | String | Yes | Comma separated list of tag names |

Tag sections are named [Tag:IndexName:TagName]
They may contain the following entries

| Name | Type | Mandatory | Description |
|---|---|---|---|
| Expression | String | Yes | Index expression in Xbase format |
| Condition | String | No | Index condition in Xbase format |
| Unique | Logic | No | Should the entries in the index be unique |

### 1.4.2 CallBack

SqlMetadataProviderCallBack gets its information from a callback event handler in the application.
The calls into the callback receive a SqlRddEventArgs object that contains the event reason, the table/index name and a default value, and they must return a value in that object.
When the connection is opened the RDD calls the Callback for the table called "Defaults" to retrieve some default values. Later when a table is opened the RDD calls the event handle for the table with the name from the USE statement.

| SqlRDDEventReason | ResultType | Default | Mandatory | Description |
|---|---|---|---|---|
| AllowUpdates | Logic | Yes | No | Are updates allowed? |
| ColumnList | String | No | No | A comma separated list of names of the columns that should be read. Defaults to all columns |
| CompareMemo | Logic | Yes | No | Should memo columns be included when generating where clause for updates or deletes? |
| DeletedColumn | String | Yes | No | The name of the column that emulates the deleted flag |
| Indexes | String | No | No | A comma separated list of index names. For each index the provider will retrieve he tlist of tags |
| KeyColumns | String | No | No | A comma separated list of names of the columns that should be used as key columns for where clauses for update/ delete statements. Defaults to all columns, except when CompareMemo is set to false. |
| LegacyFieldTypes | Logic | Yes | No | Should field types be mapped to the VO compatible field types (CDLMN) or can the RDD also return newer types such as Currency (Y) and DateTime (T) |
| LongFieldNames | Logic | Yes | No | Should long field names be used |
| MaxRecords | Numeric | Yes | No | The maximum number of rows to return |
| MaxRecnoAsRecCount | Logic | No | No | Should RecCount return the maximum record number or the physical # of rows in the table |
| RealName | String | No | No | This real table name of the table on the server. |
| RecnoColumn | String | Yes | No | The name of the column that emulates the recordnumber column |

| SqlRDDEventReason | Result Type | Default | Mandatory | Description |
|---|---|---|---|---|
| SeekReturnsSubset | Logic | Yes | No | Should a Seek operation return a subset of the rows matching the key, or should it return all rows and position the cursor on the first row that matches the key. |
| ServerFilter | String | No | No | An additional filter that can be used to filter rows. This must be in SQL syntax. |
| TrimTrailingSpaces | Logic | Yes | No | Should trailing spaces for string columns be trimmed when |
| UpdatableColumns | String | No | No | A comma separated list of names of the columns that may be updated. Defaults to all columns |
| UpdateAllColumns | Logic | Yes | No | Should all columns be included (written) in update statements, or only the changed columns |

If a table has one or more Indexes defined then the callback is called with the index name prefixed with "Index:"

| SqlRDDEventReason | Type | Mandatory | Description |
|---|---|---|---|
| Tags | String | Yes | Comma separated list of tag names |

To retrieve tag info the callback is called with the tagname prefixed with "Tag:<indexname>:"

| SqlRDDEventReason | Type | Mandatory | Description |
|---|---|---|---|
| Expression | String | Yes | Index expression in Xbase format |
| Condition | String | No | Index condition in Xbase format |
| Unique | Logic | No | Should the entries in the index be unique |

### 1.4.3 Database

SqlMetadataProviderDatabase gets the table information from special tables in the SQL database.
When you enable this provider, then the tables will be automatically added to the database, when they do not exist.
The tables are called "xs_tableinfo" and "xs_indexinfo". More tables may follow later
The definition of these tables are (for SQL Server)

```
CREATE TABLE [dbo].[xs_tableinfo](
	[TableName] [nvarchar](50) NULL,
	[RealName] [nvarchar](255) NULL,
	[LongFieldNames] [bit] NULL,
	[AllowUpdates] [bit] NULL,
	[MaxRecords] [numeric](10, 0) NULL,
	[MaxRecnoAsReccount] [bit] NULL,
	[RecnoColumn] [nvarchar](50) NULL,
	[DeletedColumn] [nvarchar](50) NULL,
	[TrimTrailingSpaces] [bit] NULL,
	[CompareMemo] [bit] NULL,
	[Indexes] [nvarchar](250) NULL,
	[ColumnList] [nvarchar](255) NULL,
	[KeyColumns] [nvarchar](255) NULL,
	[UpdatableColumns] [nvarchar](255) NULL,
	[ServerFilter] [nvarchar](255) NULL,
	[SeekReturnsSubset] [bit] NULL,
	[UpdateAllColumns] [bit] NULL
)

CREATE TABLE [dbo].[xs_indexinfo](
	[TableName] [nvarchar](50) NULL,
	[IndexName] [nvarchar](50) NULL,
	[Ordinal] [int] NULL,
	[TagName] [nvarchar](50) NULL,
	[Expression] [nvarchar](250) NULL,
	[Condition] [nvarchar](250) NULL,
	[Unique] [bit] NULL
)
```

The length of the string columns can be shorter or longer. You have to make sure that the values fit in the columns.
The tags are read in the order defined with the Ordinal column.
When opening the database the RDD will read default values from a row in the xs_tableinfo table where the tablename is "Defaults".

### 1.4.4 Default Values

The default values for the metadata are set at the Connection Level and can be overridden by the provider.

#### Table defaults

| Name | Default Value |
|---|---|
| AllowUpdates | TRUE |
| ColumnList | "*" |
| CompareMemo | TRUE |
| DeletedColumn | "" |
| Indexes | "" |
| KeyColumns | "*" |
| LegacyFieldTypes | TRUE |
| LongFieldNames | FALSE |
| MaxRecords | 1000 |
| MaxRecnoAsRecCount | FALSE |
| RealName | "" |
| RecnoColumn | "" |
| SeekReturnsSubset | FALSE |
| ServerFilter | "" |
| UpdatableColumns | "*" |
| UpdateAllColumns | FALSE |

#### Index Defaults

| Name | Default Value |
|---|---|
| Expression | "" |
| Condition | "" |
| Unique | FALSE |

## 1.5 Connections

When you work with connections inside ADS you will work with handles for connections, commands and tables.
To make it easy to adjust your code we have also added similar functions.
However, since the SQLRDD supports working with different database vendors, you also need to tell the RDD which vendor you want to use.
Internally in the RDD however we work with objects.
Because we support different Ado.Net data providers you will also have to tell the RDD which provider to use
That may look like this:

```
// Tell the RDD to use the SQLServer SqlDbProvider class
SqlDbSetProvider("SQLSERVER")

// Open a connection to the local Northwind sample database for
SqlServer
// This connection will be the 'default' connection
// store the handle as hConn1
var hConn1 := SqlDbOpenConnection("Server=(local);Initial
catalog=Northwind;Trusted_Connection=True;")

// Open a remote connection to server SRV001, database MyDb and
name it Conn2
// store the handle as hConn2
var hConn2 := SqlDbOpenConnection("Conn2","Server=SRV001;Initial
catalog=MyDb;Trusted_Connection=True;"

// Open the DbServer object on the default connection
var oServer1 := DbServer{"Customer"}

// Pass the connection name when opening a table in connection 2
var oServer2 := DbServer{"Conn2::Orders"}

// do some work with both servers
.
.
.
// Close both servers
oServer1:Close()
oServer2:Close()

// Close both connections
SqlDbCloseConnection(hConn1)
SqlDbCloseConnection(hConn2)
```

If you want to access the SqlDbConnection object behind the connection you can retrieve it with the SqlDbGetConnection function
like this

```
var oConnection := SqlDbGetConnection(hConn1)
```

You can then call any method on this object to execute commands, open tables etc.
You can call SqlDbGetConnection with the handle or the name of the connection.

### 1.5.1 Connection Strings

The connection strings for the various DBMS's are all slightly different. We recommand that yo check the documentation for the DotNet provider for these.
The SQLRDD has added support for the following extra elements in the connection string. Each of the settings can be enabled/disabled with a True/False value.

| Keyword (case insensitive) | Description | Default | Can be overridden by metadata per table |
|---|---|---|---|
| AllowUpdates | Should updates for tables be allowed by default | TRUE | Yes |
| CompareMemo | Should memo columns be included when generating where clause for updates or deletes? | TRUE | Yes |
| DeletedColumn | Default column name for the Deleted Column | "" | Yes |
| LegacyFieldTypes | Should only legacy field types be returned (CDLMN) or should also the FoxPro field types such as I (Integer), T (DateTime) and Y (currency) be used. | True | No |
| LongFieldNames | Should the column names be truncated to 10 characters or are longer column names allowed | True | Yes |
| Maxrecords | The maximum number of rows to return | 1000 | Yes |
| MaxRecnoAsRecCount | Should RecCount return the maximum record number or the physical # of rows in the table | False | Yes |
| RecnoColumn | Default column name for the Recno Column | "" | Yes |
| SeekReturnsSubset | Should a Seek operation return a subset of the rows matching the key, or should it return all rows and position the cursor on the first row that matches the key. | False | Yes |
| TrimTrailingSpaces | Should trailing spaces be trimmed when values are read ? | True | Yes |
| UseNulls | Should null values in the database be returned as DbNull.Value or as blank values? | True | No |

These settings will be stored as properties in the SqlDbConnection
An example connection string for SqlServer to the Northwind Database on the server SRV001 could be

```
Server=SRV001;Initial
catalog=Northwind;Trusted_Connection=True;TrimTrailingSpaces=False;UseNulls=False;L
egacyFieldTypes=False;
```

Each of the MetaData Providers has a mechanism to override the values from the connection object
For the IniMetadataProvider that is the section [Defaults]
For the DatabaseMetadataProvider that is the entry for the table with the name "defaults"
For the CallbackMetdataProvider the RDD will callback with the tablename "defaults"

## 1.6 Basic Usage

To change a program that was using DBF files to using the SQLRDD you need to do the following steps

1. Choose the database that you want to use. If this is one of the databases that are standard supported by the X# SQL RDD than you're fine. Otherwise you will have to create a SqlDbProvider subclass for that database, and fill it with the information needed by the X# SQLRDD. Please contact us if you need help with that.
   ***If you have created your own provider***, then you need to register that provider with the RDD by calling `SqlDbRegisterProvider()` and passing the name that you want to use and the type (System.Type) of the class. For example if you have created a provider for SQLite, you could call

```
    SqlDbRegisterProvider("SQLITE",
Typeof(MyNamespace.SqliteDbProvider))
```

2. In the code you then need to tell the RDD which provider you want to use. For example

```
// The builtin SQL Server provider
SqlDbSetProvider("SQLSERVER")
// or the provider that was registered in the previous paragraph
SqlDbSetProvider("SQLITE")
```

3. Set the Default RDD to "SQLRDD"

```
RddSetDefault("SQLRDD")
```

4. Then you open a connection to the database:

```
var handle := SqlDbOpenConnection(ConnectionString)   // This opens
a default connection
// to open a second connection you need to specify a name (the
first connection defaults to DEFAULT
var handle2 := SqlDbOpenConnection("ALTERNATE", ConnectionString2)
```

5. By default the RDD expects you to use the Ini Metdata Provider. If you want to specify another provider you can do that like this.

```
// Get the connection object
var conn   := SqlDbGetConnection(handle)
// Set the MetadataProvider for the connection
conn:MetadataProvider := DatabaseMetaDataProvider{conn}
```

6. Then you can open tables as normal

```
// This will open the Customer table and will use the metadata
provider to ask for more information
var oSrv := DbServer{"Customers"}
// You can also open a table from the secondary connection by
// prefixing the table name with the connection name followed by a
double colon
var oSrv2 := DbServer{"ALTERNATE::Employees"}
// You can also open a query. This will be readonly
var oSrv2 := DbServer{"Select * from Orders where ShippedDate is
NULL"}
```

7. When you're done with the database you can (should) close the connection(s)

```
// Note: Closing the connection will fail if there are still open
areas for the connection
SqlDbCloseConnection(handle)
SqlDbCloseConnection(handle2)
```

## 1.7 Indexes

For a DBF table you can create indexes. These can be stored as files with a single sort order (NTX) or as files with multiple sort orders (CDX). The individual orders inside a CDX are often referred to as TAGS.
Indexes consist of an index expression, which can be an individual column name ( “CustomerNo”) but also as a combination of several fields in an Xbase expression (“CustomerNo+DTOS(OrderDate)”).
They can also contain a For condition that makes sure that only records where that condition evaluates to TRUE become part of the index.
These index expressions do not always translate directly into SQL.
That is why the RDD parses the index expressions into individual elements and tries to translate them into SQL expressions.

### 1.7.1 Index getting started

Of course every serious application will use indexes. Therefore we also support Index creation with the SQLRDD.
This is an area where you may find some differences between SQLRDD and the standard RDDs.
When you analyze the use of indexes in your application you will see that indexes are used for two different purposes:

- Displaying your data in a particular order
- Locating records using the Seek functionality

The standard RDD files achieve this by reading all the records in a table and storing a list of record numbers and key value pairs in a structured way in a file on the local disk, together with the index expression that created the key values.
Since the key values are calculated on the local workstation any VO expression (including user defined functions) may be used to calculate the key values.

## Problems

The SQLRDD also implements indexes for both purposes, but has to handle some problems:

- It is not feasible to store the record number-key value pairs on the local hard disk. Indexing should be handled by the server
- Record numbers in general are not available in a SQL environment.
- For that reason we can not use User Defined Functions in an index key expression, since these would not be available for the server
- Most of the built-in functions of Clipper, such as SUBSTR(), LEFT(), RIGHT(), UPPER(), STR(), DTOS() are available on the server but have a different name or different parameters.
- In a Client/Server environment you usually try to reduce the number of records that get send to the workstation. So it makes sense to use the Seek() functionality to limit the number of rows that get returned to the workstation

## Solution

We have come up with the following solution:

- When creating an index file we split the work between the client and the server:
  - The Client stores the index and order(tag) name on the local hard disk, together with the index expression
  - The Client asks the Server to create an index using the field(s) included in the index
- When selection an order the client will generate an order by clause that gets sent to the server so the rows get returned in the proper order
- When seeking a record, the client will generate a where clause, using the index expression, so the proper record(s) get(s) returned.

During this process the RDD has to parse the index expression, and to translate the VO functions used in the expression to functions supported by the server. Since this may be server specific, the RDD will call back into the client application for help with this translation process.

### 1.7.2 Index Files or indexes through a callback

The RDD can work with both physical index files (SDX files) but can also get the index information through callback methods from the running application.
In both situations the only information really used from the files or callback is:

- the index/tag name(s)
- the index expression
- the Unique flag
- the Descend flag.

No other information is needed. The physical sorting is done by the server. The RDD uses the information mentioned to generate a column list for the Create Index statement that it sends to the server, and also to generate an order by clause and where clause when seeking information.

The default behaviour of the RDD is to retrieve index information from the Metadataprovider. You can also open a physical SDX file. This is an INI file that contains the index expressions.

### 1.7.3 Index support

## What is currently supported ?

The following syntax of the **INDEX ON** command (and functions) is supported

```
INDEX ON <uKeyValue> [TAG <xcOrder>] [TO <xcIndexFile>]
 [UNIQUE] [ASCENDING | DESCENDING] [ADDITIVE]
```

## What is NOT supported ?

The following elements of the INDEX ON command are **NOT** supported

```
[WHILE <lCondition>]
[FOR <lCondition>]
[EVAL <cbEval> [EVERY <nInterval>]
[USECURRENT] [CUSTOM] [NOOPTIMIZE]
```

## Unique Indexes

The SQLRDD handles the UNIQUE property of an index differently than the standard RDDs. It tries to create an Unique Index on the server, which will fail if two rows in the table have the same index key value(s).

### 1.7.4 Index expressions

Index expression should be valid Xbase  expressions, but these expressions are rarely valid SQL expressions. We therefore need to translate the index expression to its SQL counterpart.
The SQLRDD has a built-in index expression parser that parses your index expression. What it does is the following:

- First it parses the index expression and tries to locate the field names, function, operators and constants that you have used in your expression.
- For every field name it checks if the field name is valid for the workarea. If you have used a name that doesn't exist an error will be generated.
- For every function in the VO key expression, the RDD calls back into the program's SqlDbProvider Callback method, allowing you to replace the function with the equivalent for your server/provider. The function names will have the following layout: UPPER(%1%) or STR(%1%,%2%, %3%) etc.
- The RDD also calls back to find the string concatenator ('+' sign). Some providers use another operator such as '||'
- After this a corresponding SQL Index expression is built by putting the index key elements back in place.
- The SQL index key expression is then used to build:
  o an OrderBy clause to open the data
  o a Where clause when you are Seeking in the workarea
  o Where clauses when you are setting Scopes
- If your index expression is a simple expression (without functions) the SQLRDD will also build an index on the server.
- When building where clauses the SQLRDD uses the SUBSTR() function if you are seeking on partial strings, or setting scopes on partial key values

### 1.7.5 Index Expression with Descend()

Index expressions containing the **DESCEND()** function are supported only when:

- There is no nesting of functions, in other words the DESCEND() function is not nested within other functions.
- You can even reverse the order of indexes containing DESCEND using the OrderDescend() function/method

**This is supported:**

```
INDEX on CUSTOMERID+DESCEND(DTOS(ORDERDATE)) TO Order1
INDEX ON AU_LNAME + DESCEND(AU_FNAME) TO AU_NAME
INDEX ON DESCEND(CUSTOMERID) + DESCEND(DTOS(ORDERDATE)) TO Orders2
INDEX ON DESCEND(CUSTOMERID + DTOS(ORDERDATE)) TO Orders3
```

***This is NOT supported:***

```
INDEX ON CUSTOMERID+DTOS(DESCEND(ORDERDATE)) TO Order1  // Descend
is nested in DTOS()
INDEX ON STR(DESCEND(ORDERID),10,0) TO Order2        // Descend is
nested in STR()
```

### 1.7.6 Index expression examples

## Example 1

```
USE CUSTOMER
INDEX ON LASTNAME TO LASTNAME
SET SCOPETOP TO 'H'
SET SCOPEBOTTOM TO 'H'
GO TOP
? CUSTOMER->LASTNAME
```

Because the index expression is simple the following index will be created:

```
CREATE INDEX CUSTOMER_LASTNAME ON CUSTOMER(LASTNAME)
```

The syntax of the CREATE INDEX command can be set through the SqlDbProvider (sub) class

When applying the Scopes the RDD will call back into your application, to get the replacement for the **`SUBSTR(%1%, %2%, %3%)`** function for your server. Assuming you are returning **`SUBSTRING(%1%,%2%,%3%)`** this will lead to the following SQL statement:

```
SELECT * FROM CUSTOMER
   WHERE SUBSTRING(LASTNAME,1,1) >= 'H'
   AND SUBSTRING(LASTNAME,1,1) <= 'H'
   ORDER BY LASTNAME
```

## Example 2

```
USE CUSTOMER
INDEX ON UPPER(LASTNAME+FIRSTNAME) TO NAME
SEEK "HULST"
? CUSTOMER->LASTNAME
```

When parsing the index key, the RDD will call back into your application to get the repacement for **`UPPER(%1%).`** Lets assume you are returning **`UPPER(%1%)`** unmodified. When applying the Seek the RDD will again call back into your application, to get the replacement for the **`SUBSTR(%1%, %2%, %3%)`** function for your server. Assuming you are returning **`SUBSTRING(%1%,%2%,%3%)`** this will lead to the following SQL statement:

```
SELECT * FROM CUSTOMER
WHERE SUBSTRING(UPPER(LASTNAME+FIRSTNAME),1,5) = 'HULST'
ORDER BY UPPER(LASTNAME+FIRSTNAME)
```

If you would be running on a case-insensitive server you could force the RDD to create an index by replacing the **`UPPER(%1%)`** function in your SqlDbProvider method with **`%1%.`** In that case the following SQL statements would be generated

```
CREATE INDEX CUSTOMER_NAME ON CUSTOMER (LASTNAME, FIRSTNAME)
SELECT * FROM CUSTOMER
WHERE SUBSTRING(LASTNAME+FIRSTNAME,1,5) = 'HULST'
ORDER BY LASTNAME+FIRSTNAME
```

## Example 3

```
USE INVOICE
INDEX ON STR(INVOICE_NO,10,0) TO INVOICE_NUM
SEEK STR(100,10,0)
? INVOICE->INVOICE_NO
```

When parsing the index key, the RDD will call back into your application to get the repacement for **`STR(%1%,%2%,%3%).`** Lets assume you are returning **`STR(%1%,%2%,%3%)`** unmodified. This will lead to the following SQL statement:

```
SELECT * FROM INVOICE
WHERE STR(INVOICE_NO,10,0) = '       100'
ORDER BY INVOICE_NO
```

## Note

For a DBF workarea you could achieve the same results (assuming the INVOICE_NO field has a size of 10 and 0 decimals) by using the index expression **`STR(INVOICE_NO)`** or **`STR(INVOICE_NO,10)`**. This is because VO will read the Field Size and Decimal information from the DBF when indexing and will generate as string of 10 characters and without decimals. You should not rely on this when indexing a SQL environment. One way to get around that without changing your sourcecode would be to replace the **`STR(%1%,%2%)`** function with **`STR(%1%, %2%, 0),`** but a similar replacement for the field size will be much more difficult.

## Example 4

```
USE INVOICE
INDEX ON STR(CUSTOMERID,10,0) + DTOS(INV_DATE) TO INV_DATE
SEEK STR(100,0)+"20010331"
```

When parsing the index key, the RDD will be smart enough to discover that you are only calling the STR() function and the DTOS() function to concatenate the two columns. So it will generate an index in this case:

```
    CREATE INDEX INVOICE_INVOICE_DATE ON INVOICE (CUSTOMERID,
INVOICE_DATE)
```

But when doing the seek, the RDD will call back into your application to get the repacement for **`STR(%1%,%2%,%3%)`** and **`DTOS(%1%).`** Lets assume you are returning **`STR(%1%,%2%,%3%)`** and **`Convert(Char,%1%,112)`**
This will lead to the following SQL statement:

```
SELECT * FROM INVOICE
WHERE STR(CUSTOMERID,10,0)+Convert(Char,INVOICE_DATE,112) = '
  10020010331'
```

## 1.8 Compatibility with existing RDDs

The primary targets when designing the SQLRDD was:

- **To be as much compatible as possible with the existing RDDs, especially the DBFNTX and DBFCDX RDD.**
- **To allow you to work with ANY Ado.Net dataprovider**

We have the feeling that we have succeeded in doing so, but the compatibility is not 100%.

The most important difference (we feel) is:

**The SQLRDD (for performance reasons) does not automatically read the first row from the result set.**
**You must call one of the navigational commands / functions or DbServer methods such as Go Top, Go Bottom or Go to, Append or Seek to execute the SQL Query and retrieve rows from the back end.**

Using the SQLRDD means using XSharp default workarea based data access.
That means that you can operate the RDD 4 different ways:

- Using the Data Access commands
- Using the DB..() functions
- Using the VODB..() functions
- Using the DbServer Class

Of course there are also some differences between the existing RDDs and the SQLRDD


- Connections 15
- Index support 22
- Null_Date 36
- Table level validations 36
- Record Numbers and Reccount 33
- Deleted Records 34

## 1.9 Workarea Operations

The following table lists what is happening 'under the hood' when you call a specific RDD operation.
We are using the DB..() function syntax here, but of course this also applies to the corresponding commands, VODB..() function and DbServer methods:

| Operation | Description |
|---|---|
| DbUseArea() | The structure of the Table is read and the workarea information for the table is initialized. The table is NOT opened. Also some of the properties of the workarea are through the Metadata provider |
| DbGoTop() | If the table is not open, it will be opened. If the table is open, and a where clause resulting from a Seek is active, the table will be closed and reopened without this where clause |
| DbGoBottom() | If the table is not open, it will be opened. |
| DbSkip() | If the table is not open, it will be opened. |
| Recno() | If you have assigned a RecnoColumn, the value of this column will be returned, else the relative position in the current cursor |
| DbGoTo() | If you have assigned a RecnoColumn, the row with this value will be located, else the cursor will be moved to the relative position. If the value does not exist or if the rownumber is larger than the # of rows in the current result set then the resultset will be positioned at EOF |
| DbCreate() | A "Drop Table" SQL statement will be executed, followed by a Create Table SQL statement. The syntax of these statements can be set through the SqlDbProvider. |
| DbDelete() | If there is a DeletedColumn, then this column will be set to TRUE.<br>Otherwise the row will be deleted in the local cursor, and that change will be sent to the server when you move the record pointer to another row. |
| DbReCall() | If there is a DeletedColumn, then this column will be set to FALSE.<br>Otherwise the row will be restored in the local cursor. Once the deleted row is deleted on the server then you cannot undo the delete. |
| Deleted() | Will return TRUE if you have just deleted a row and have not moved the record pointer yet. In all other cases will return FALSE. |
| DbAppend() | Will immediately append a blank row, fill all the columns of this row with the appropriate blank values and write the new row to the local buffer. The row will be sent to the server if you move the record pointer to another row. |
| FieldGet() | If the table is not open, it will be opened. |
| FieldPut() | If the cursor allows updates the new value will be written to the server. |

| Operation | Description |
|---|---|
| RLock() and FLock() | Will always return TRUE. Locking is handled by the server. |
| DbUnLock() | Will always return TRUE. |
| LastRec() and RecCount() | Return the number of rows in the current open cursor. This may be different from the real number of rows on the server |
| DbPack() | Will not do anything |
| DbZap() | Will execute a "Delete All From " SQL statement. The syntax of these statements can be set through the SqlDbProvider. |
| DbCreateIndex()/ OrdCreate() | A SDX file will be created that contains the index name and expression, and when possible an index is created on the server as well. The syntax of these statements can be set through the SqlDbProvider. |
| DbSetIndex() | The index information in the SDX file will be read, and the first tag in this file will be activated. A Current open cursor will be closed.<br>If the index contains functions, then the function will be translated using the GetFunction method in the Database provider |
| DbSetOrder() / OrdSetFocus() | If the order is available it will be activated. A Current open cursor will be closed.<br>New data will be selected when an operation is done that requires new data.<br>Please NOTE that THE CURRENT RECORD POSITION WILL NOT BE SAVED |
| OrdDestroy() | The order will be removed and a Current open cursor will be closed. |
| DbSeek() | A where clause will be built based on the seek value and the current index. When Soft seek is used all rows greater then or equal to the seek value will be selected, else only the row(s) that match the seek value. Please not that this selection will remain active until you do a GO TOP or start another operation that closed the current cursor. |

## 1.10 Record Numbers and Reccount

One of the areas where the DBF world distinguishes itself from the SQL world is the concept of Record Numbers.
In the SQL RDD we try to emulate the **record number**. There are 2 ways to do this:

1. By creating a special column in the table that holds a unique numeric identifier. Preferable an identity column or a column with a sequence attached. You can define that by setting the value for RecnoColumn in the metadata provider[8].
2. If such a column does not exist then the relative position in the current result set is used to emulate a record number

**Reccount** is defined by the SQLRDD as the number of rows in the table.

### Some remarks about working with a Recno Column

Emulating a record number works fine, but has the disadvantage that (after deleting rows in the database) you may have record numbers that are larger than the # of rows in the table. Assume you start with records 1 .. 10 and delete the first 2 records. Then the RecCount for the table will be 8, but the largest record number will be 10 and the smallest record number 3.

In a DBF environment you would never be able to Goto(10) for a table with 8 rows. In this scenario you could do that.
In a similar way you cannot do a Goto(1) in the SQL table, because that record is deleted from the database. If you do that, then the RDD will position itself on the Phantom record.

When you append records in a DBF then the Recno automatically becomes the value of what previously was RecCount +1.
In a SQL environment the new record is not written to the database, so we do not know what the new value of Recno is until the record has been written.
The RDD will assign the current maximum value of the Recno Column +1 to the record number of the newly appended record, and after the record is written to the server then the RDD will fetch the real value of the record number. In theory it is possible that 2 users are appending a record to the same SQL table at the same time. They will initially both get the same record number. However after the record was written to the SQL table then both result sets will have a unique number for that column.
Retrieving the new record number is either done by executing the code that is defined in the provider for GetIdentity.
When the provider does not support that, then a query is executed to retrieve the maximum value of the Recno Column.

## 1.11 Deleted Records

Another area where the DBF world distinguishes itself from the SQL world is the way how records can be deleted and recalled.
In the SQL RDD we try to emulate the **deleted flag.** That can be done in 2 ways

1. By creating a special column in the table that holds the deleted flag. This should contain a numeric value or a logic (bit) value. You can define that by setting the value for DeletedColumn in the metadata provider. For logical columns TRUE means deleted and Null/FALSE means not deleted. For numeric columns Null/0 means NOT deleted. All other numeric values indicate that the record is deleted.
2. If you do not define a deleted column, then we will physically delete the row from the SQL table as soon as the Delete() operation is committed.

The advantage of 1) is that you can later also restore deleted records.

## 1.12 Names and reserved words

One of the areas that may cause problems when converting a DBF app to SQL is the area of reserved words.
In a SQL environment a large number of words are treated as reserved words, and may not be used as column names, table names or index names.

The SQL RDD tries to solve this by surrounding the column names and table names with the characters that the server. Each DbProviderFactory provides a CommandBuilder class that has a QuoteIdentifier() method. We have exposed that method in the SqlDbProvider class.

Also some providers are case sensitive. We have added a CaseSync() method to the provider that you can use to synchronize the case of the identifiers

## 1.13 Validations and Integrity

If the back end that your application talks to supports validation and integrity rules, such as primary and foreign keys and mandatory fields, the SQLRDD application will of course follow those rules.

One problem may be here, that when you insert a record in the table (using Append Blank or DbAppend()), the RDD will fill all the updatable columns of that record with blank values (for compatibility with DBF applications).

It is up to you to make sure that you fill the new record with valid information.

If the validation and integrity rules of your server don't allow you to add a blank row, you must be sure you will update this new row properly, or else a runtime error will be generated.

Another area where you may encounter problems is when deleting or updating records. If you break the integrity rules of your server by deleting or updating a record, this operation will fail.

### NULL_DATE and DateTime.MinValue

Most SQL databases do not know the concept of an empty Date. The SQLRDD therefore converts the values NULL_DATE and DateTime.Minvalue into a NULL.
When your database does not allow NULL in Date or DateTime columns, then you will have to make sure in your application that you write a different value.

## 1.14 Transactions

If you want to work with transactions, then you can retrieve the connection object and call BeginTrans(), CommitTrans() or RollBackTrans() on the connection.

```
// Tell the RDD to use the SQLServer SqlDbProvider class
// store the handle as hConn1
SqlDbSetProvider("SQLSERVER")
// Open a connection to the local Northwind sample database for
SqlServer
// This connection will be the 'default' connection
// store the handle as hConn1
var hConn1 := SqlDbOpenConnection("Server=(local);Initial
catalog=Northwind;Trusted_Connection=True;")
var oConn1 := SqlDbGetConnection()
oConn1:BeginTrans()
// open a server
var oServer := DbServer{"Customer"}
// write to the server
oServer:FieldPut(#LastName, "Jones")
// commit and close
oServer:Commit()
oServer:Close()
// Rollback the changes
oConn:RollBackTrans()
// Close the connection
SqlDbCloseConnection(hConn1)
```

## 1.15 Converting from DBF to SQL

To convert DBF data to SQL you should do this:

Select a provider and open a connection, just like you can see in the Basic Usage 19 topic

- Open the DBF file with the DBFCDX or DBFNTX RDD
- Use DbCopy() or dbServer:Copy() to copy the file to the SQL database, by setting the target RDD to SQLRDD.
  This will create the table and copy the data.
- Then create the indexes.
- You will then have to decide which method to use if give the "metadata" to the RDD.
  Depending on that you should add information to the ini file, callback or database tables.

## 1.16 Error Handling

The SQLRDD uses the standard X# structured error handling. Every time an error occurs in the RDD, a VO Error object is generated with at least the following properties filled:

| | |
|---|---|
| Subsystem | SQLRDD |
| GenCode | EG_OPEN, EG_READ, EG_WRITE, EG_CLOSE |
| FuncSym | The RDD method that caused the failure |
| SubError | A number between 2000 and 2046 indicating an error code (see list below) |
| Description | A textual description of the error |

If you have setup a local or global error handler you can catch this error and decide what to do.

If you are using the RDD with the VO DbServer class you should note the following:

- If you are using the DbServer object **without a client** (not on a DataWindow, DataBrowser, DataListView), the DbServer class will catch the error, and store it in its ErrInfo property. It will also create a Status hyper label and finally will call your **local or global error handler**.

- If you are using the DbServer object **with a client** object, the DbServer class will also fill the ErrInfo and Status properties, but will not call the local error handler. In stead it will call the **Error method** of the first client attached. Unfortunately **the default implementation of the Error method on the DataWindow and DataBrowser classes is empty**. So you will get no error messages ! We advise to create a useful Error method in your DataWindow subclass.

## 1.17 Version History

### Version 1.0.0.3 May 2025

#### Third Beta

- We have moved some common code from the various MetaData providers in the abstract parent class.
- The DbProvider class now has a virtual **TrueLiteral** and **FalseLiteral** property that can be overridden in a subclass.
- We have added a setting **SeekReturnsSubset**, which controls the behavior for Seek() operations. When **TRUE** (the default) then a Seek operation returns just the row(s) that match the Seek key. This is compatible with the previous behavior. When **FALSE**, then all rows are returned and the cursor is positions on the first row that matches the key. You can then also Skip -1 to reach the rows before the row that matches the key
- We have added the setting **MaxRecnoAsRecCount** which controls how the RecCount in the RDD is calculated. When this setting is **TRUE** then the RecCount will be calculated from the maximum value of the Recno column. That could result in a RecCount that is larger than the physical # of rows in the table. This setting is only used for tables that have a RecnoColumn.
  When set to FALSE (the default) then the RDD does a select count(*) on the table. When you have declared a server side filter, then that filter is included on the Select count(*) statement.
- The RDD is no longer creating a xs_license table in the database
- When a table has a recno column, then the recno is always added as last column to order by clause, to make sure that records with duplicate keys are shown in recno order
- When a table has a recno column, then switching the index order will move the record pointer to the row that contains that record number. Otherwise the record pointer will be on the first row in the table in the selected order
- Seek NIL will move the record pointer to the first record in the index to be compatible with other RDDs

### Version 1.0.0.1 - March 2024

#### Second beta. Changes are:

- We Implemented Pack() and Zap()
- Several changes in the providers
  - o All providers must implement CaseSync()
  - o MySql: GetRowCount and SelectTopStatement
  - o PostgreSQL: changed the class name. GetIdentity and GetRowCount are not returning an empty string
  - o SQLite: new (also included as example)
- Fixed a problem with case sensitivity for identifiers
- Implemented several OrderInfo calls for current OrderNumber, OrderCount, OrderKeyNo, OrderKeyCount etc
- IMetadataProvider now has 2 new methods (CreateTable and CreateIndex) that are called when a table is created or an index is created.
- The implementation of these methods by the Database Metadataprovider is this:

  - Creating a table with DbCreate() will now also add the table to the xs_tableinfo table
  - Creating an order with the Database Metadata provider now also adds the index and tag to the xs_infoinfo table and the index name to the xs_tableinfo table
- The Ini metadata provider does NOT implement CreateTable and CreateIndex. Please let us know if you would like to add that too.
- If you have declared a default RecnoColumn and/or a default DeletedColumn, then these columns will be automatically added when you create a table with DbCreate()
- Special columns, such as the RecnoColumn and DeletedColumn are not included in the array that is returned from DbStruct().
- When you open a connection we will now add 2 special tables to the database (when they do not exist) in which we store license information and the current connected users. For now there is no limitation here, but in the release version of the SqlRDD we will use this information to restrict the # of concurrent users based on the license.
- Generated OrderBy clauses will now include the recnocolumn (when relevant). This makes sure that duplicate keys are ordered in record number order, just like what RDD indexes do.
- The record number and/or deleted columns will be moved to the end of the column list
- If you do not specify KeyColumns for your table, but the table has a RecnoColumn, then we will use that column to generate update and/or delete statements.
- The where condition for 2 equal top and bottom scopes is now compressed to a single comparison
- Fixed a problem with writing Null values to DateTime columns
- Fixed a problem when DbCopy() was copying deleted rows to the SQL table.
- Some Ado.Net providers are returning System.Decimal values for columns without decimals. These are now converted to Integer values.
- The SqlDbConnection class now reads several properties from the Ado.Net datasource and stores these in the DataSourceProperties collection

## Version 1.0.0.0 - February 2024

- **First Beta**

## 1.18 Callbacks

Sometimes you may want to monitor the SQL commands that the RDD sends to the provider.
You can do that by registering an Event Handler that will be called for each statement sent to the server.
That event handler receives the sending object and an argument of type SqlRddEventArgs that describes what the RDD is doing or which information it wants:

That may look like this:

```
function TestTransaction()
    // Tell the RDD to use the SQLServer SqlDbProvider class
    // store the handle as hConn1
    SqlDbSetProvider("SQLSERVER")
    RddSetDefault("SQLRDD")
    // Open a connection to the local Northwind sample database for SqlServer
    // This connection will be the 'default' connection
    // store the handle as hConn1
    var hConn1 := SqlDbOpenConnection("Server=(local);Initial
catalog=Northwind;Trusted_Connection=True;")
    var oConn1 := SqlDbGetConnection(hConn1)
    // register an event handler
    oConn1:CallBack += @@EventHandler

    // open a server
    var oServer := DbServer{"Customers"}
    oServer:GoTop()
    // write to the server
    oServer:FieldPut(#ContactName, "Jones")
    // commit and close
    oServer:Commit()
    oServer:Close()
    SqlDbCloseConnection(hConn1)
    return true

FUNCTION EventHandler(oSender AS Object, e AS
XSharp.RDD.SqlRDD.SqlRddEventArgs) AS OBJECT
   ? "Event", e:Table, e:Reason:ToString(), e:Value
   RETURN e:Value
```

This event handler will show the following for this example:

```
1)  Event Customers CommandText select * from [Customers] where 0=1
2)  Event Customers CommandText select [CustomerID], [CompanyName],
[ContactName], [ContactTitle],
          [Address], [City], [Region], [PostalCode], [Country], [Phone],
[Fax] from [Customers] where 1=0
3)  Event Customers WhereClause
4)  Event Customers CommandText select top 1000 [CustomerID], [CompanyName],
[ContactName], [ContactTitle],
          [Address], [City], [Region], [PostalCode], [Country], [Phone],
[Fax] from [Customers]
5)  Event Customers CommandText update [Customers] set [ContactName] = @p1
```

```
where [CustomerID] = @o1;
          select @@ROWCOUNT
```

You can see that the event handler is called first a couple of times:

1) To get the column list,
2) To select the full column information for all columns in the Customers Table.
3) To query for a whereclause
4) The Select statement that gets the data
4) The update statement to update the table.
   In this case the where clause contains just the CustomerId clause, because that was specified in the metadata provider for the table. And only the ContactName column is updated because UpdateAllColumns is set to FALSE by the metadata provider

# Index

## - # -



## - : -



## - A -



## - B -



## - C -



## - D -



## - E -



## - F -



## - G -



## - H -



## - I -



## - L -



## - N -



## - O -

# - P -



# - R -



# - S -



# - T -



# - U -



# - V -



# - W -

Back Cover