Oracle® R Enterprise

User's Guide

Release 1.4

E39886-03

February 2014

ORACLE®

Oracle R Enterprise User's Guide, Release 1.4

E39886-03

Primary Author: David McDermid

Contributing Author: Margaret Taft

Contributors: Patrick Aboyoun, Dmitry Golovashkin, Mark Hornick, Sherry Lamonica, Kathy Taylor, Qin Wang, Lei Zhang

Contents

3 Preparing and Exploring Data in the Database

4 Building Models in Oracle R Enterprise

5 Predicting With R Models

6 Using Oracle R Enterprise Embedded R Execution

List of Examples

Preface

This book describes how to use Oracle R Enterprise.

Audience

This document is intended for anyone who uses Oracle R Enterprise. Use of Oracle R Enterprise requires knowledge of R and Oracle Database.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

The following documents are related to the Oracle Advanced Analytics option:

- *Oracle R Enterprise Installation and Administration Guide*
- *Oracle R Enterprise Release Notes*
- *Oracle Data Mining Concepts*
- *Oracle Data Mining User's Guide*

Oracle R Enterprise Online Resources

The following websites provide useful information for users of Oracle R Enterprise:

- The Oracle R Enterprise page on the Oracle Technology Network (OTN) provides downloads, the latest documentation, and information such as white papers, blogs, discussion forums, presentations, and tutorials. The website is at http://www.oracle.com/technetwork/database/options/advanced-analytics/r-enterprise/index.html
- The Oracle R Enterprise Discussion Forum (https://forums.oracle.com/forums/forum.jspa?forumID=1397) supports all

aspects of Oracle's R-related offerings, including: Oracle R Enterprise, Oracle R Connector for Hadoop (part of the Big Data Connectors), and Oracle R Distribution. Use the forum to ask questions and make comments about the software.

- The Oracle R Enterprise Blog (`https://blogs.oracle.com/R/`) discusses best practices, tips, and tricks for applying Oracle R Enterprise and Oracle R Connector for Hadoop in both traditional and Big Data environments.
- For information about R, see the R Project for Statistical Computing at `http://www.r-project.org`.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Changes in This Release for Oracle R Enterprise

Releases of Oracle R Enterprise often contain new features. The features in the current release and in some previous releases are described in the following topics:

Changes in Oracle R Enterprise 1.4

The following topics describe the changes in Oracle R Enterprise 1.4:

New Features in Oracle R Enterprise 1.4

The following changes are in Oracle R Enterprise 1.4:

- Additions and improvements to data preparation functions:
  - The new `factanal` function performs factor analysis on a `formula` or an `ore.frame` object that contains numeric columns.
  - Both signatures of the `princomp` function support the `scores`, `subset`, and `na.action` arguments.
  - The new `getXlevels` function creates a list of factor levels that can be used in the `xlev` argument of a `model.matrix` call that involves an `ore.frame` object.
- The new exploratory data analysis function `ore.esm` builds exponential smoothing models for time series data. The function builds a model using either the simple exponential smoothing method or the double exponential smoothing method. The function can preprocess the time series data with operations such as aggregation and the handling of missing values. See "Building Exponential Smoothing Models on Time Series Data" on page 3-30.
- Additions and improvements to the Oracle R Enterprise regression and neural network modeling functions:
  - The new `ore.glm` function provides methods for fitting generalized linear models, which include logistic regression, probit regression, and poisson regression. See "Building a Generalized Linear Model" on page 4-5.
  - The `ore.lm` and `ore.stepwise` functions are no longer limited to a total of 1,000 columns when deriving columns in the model formula.

- The `ore.lm` function now supports a `weights` argument for performing weighted least squares regression.
- The `anova` function can now perform analysis of variance on an `ore.lm` object.
- For the `ore.stepwise` function, the values for the `direction` argument have changed. The value `"both"` now prefers drops over adds. The new `direction` argument value `"alternate"` has the previous meaning of the `"both"` value.
- The `ore.neural` function has several new arguments.

- Additions and improvements to the Oracle Data Mining model algorithm functions:
  - The new `ore.odmAssocRules` function, which builds an Oracle Data Mining association model using the apriori algorithm. See "Building an Association Rules Model" on page 4-11.
  - The new `ore.odmNMF` function, which builds an Oracle Data Mining model for feature extraction using the Non-Negative Matrix Factorization (NMF) algorithm. See "Building a Non-Negative Matrix Factorization Model" on page 4-24.
  - The new `ore.odmOC` function, which builds an Oracle Data Mining model for clustering using the Orthogonal Partitioning Cluster (O-Cluster) algorithm. See "Building an Orthogonal Partitioning Cluster Model" on page 4-25.
- An additional global option for Oracle R Enterprise, `ore.parallel`. See "Oracle R Enterprise Global Options" on page 1-12.

Changes in Oracle R Enterprise 1.3

The following topics describe the changes in Oracle R Enterprise 1.3:

- New Features in Oracle R Enterprise 1.3
- Other Changes in Oracle R Enterprise 1.3

New Features in Oracle R Enterprise 1.3

The new features in Oracle R Enterprise 1.3 are the following:

- Predicting with R models using in-database data with the `OREpredict` package
- Ordering and indexing with `row.names<-`
- Predicting with Oracle Data Mining models using the `OREodm` package
- Saving and managing R objects in the database
- Date and time data types
- Sampling and partitioning
- Long names for columns
- Automatically connecting to an Oracle Database instance in embedded R scripts
- Building an R neural network using in-database data with the `ore.neural` function

Other Changes in Oracle R Enterprise 1.3

Other changes in this release are the following:

- Installation and administration information has moved from this manual to *Oracle R Enterprise Installation and Administration Guide*. New features related to installation and administration are described in that book.

Changes in Oracle R Enterprise 1.1

The new features in Oracle R Enterprise 1.1 are the following:

- Support for additional operation systems:
  - Oracle R Distribution and Oracle R Enterprise are now supported IBM AIX 5.3 and higher and on 10 and higher for both 64-bit SPARC and 64-bit x386 (Intel) processors.
  - The Oracle R Enterprise Server now runs on 64-bit and 32-bit Windows operating systems.
- Improved mathematics libraries in R:
  - You can now use the improved Oracle R Distribution with support for dynamically picking up either the Intel Math Kernel Library (MKL) or the AMD Core Math Library (ACML) with Oracle R Enterprise.
  - On Solaris, Oracle R Distribution dynamically links with Oracle SUN performance library for high speed BLAS and LAPACK operations.
- Support for Oracle Wallet enables R scripts to no longer need to have database authentication credentials in clear text. Oracle R Enterprise is integrated with Oracle Wallet for that purpose.
- Improved installation scripts provide more prerequisite checks and detailed error messages. Error messages provide specific instructions on remedial actions.

1

Introducing Oracle R Enterprise

This chapter introduces Oracle R Enterprise. The chapter contains the following topics:

See Also: *Oracle R Enterprise Installation and Administration Guide*

About Oracle R Enterprise

Oracle R Enterprise is a component of the Oracle Advanced Analytics Option of Oracle Database Enterprise Edition. Oracle R Enterprise is comprehensive, database-centric environment for end-to-end analytical processes in R, with immediate deployment to production environments. It is a set of R packages and Oracle Database features that enable an R user to operate on database-resident data without using SQL and to execute R scripts in one or more embedded R engines that run on the database server.

Using Oracle R Enterprise from your local R session, you have easy access to data in an Oracle Database instance. You can create and use R objects that specify data in database tables. Oracle R Enterprise has overloaded functions that translate R operations into SQL that executes in the database. The database consolidates the SQL and can use the query optimization, parallel processing, and scalability features of the database when it executes the SQL statements. The database returns the results as R objects.

Embedded R execution provides some of the most significant advantages of using Oracle R Enterprise. Using embedded R execution, you can store and run R scripts in the database through either an R interface or a SQL interface or both. You can use the results of R scripts in SQL-enabled tools for structured data, R objects, and images.

See Also: "Advantages of Oracle R Enterprise" on page 1-2

Advantages of Oracle R Enterprise

Using Oracle R Enterprise to prepare and analyze data in an Oracle Database instance has many advantages for an R user. With Oracle R Enterprise, you can do the following:

- **Operate on Database-Resident Data Without Using SQL.** Oracle R Enterprise has overloaded open source R methods and functions that transparently convert standard R syntax into SQL. These methods and functions are in packages that implement the Oracle R Enterprise **transparency layer**. With these functions and methods, you can create R objects that access, analyze, and manipulate data that resides in the database. The database can automatically optimize the SQL to improve the efficiency of the query.
- **Eliminate Data Movement.** By keeping the data in the database, you eliminate the time involved in transferring the data to your desktop computer and the need to store the data locally. You also eliminate the need to manage the locally stored data, which includes tasks such as distributing the data files to the appropriate locations, synchronizing the data with changes that are made in the production database, and so on.
- **Keep Data Secure.** By keeping the data in the database, you have the security, scalability, reliability, and backup features of Oracle Database for managing the data.
- **Use the Power of the Database.** By operating directly on database-resident data, you can use the memory and processing power of the database and avoid the memory constraints of your client R session.
- **Use Current Data.** As data is refreshed in the database, you have immediate access to current data.
- **Prepare Data in the Database.** Using the transparency layer functions, prepare large database-resident data sets for predictive analysis through operations such as ordering, aggregating, filtering, recoding, and the use of comprehensive sampling techniques without having to write SQL code.
- **Save R Objects in the Database.** You can save R objects in an Oracle Database instance as persistent database objects that are available to others. You can store R and Oracle R Enterprise objects in an Oracle R Enterprise datastore, which is managed by the Oracle database.
- **Build Models in the Database.** You can build models in the database and store and manage them in an Oracle R Enterprise datastore. You can use functions in packages that you download from CRAN (Comprehensive R Archive Network) to build models that require large amounts of memory and that use techniques such as ensemble modeling.
- **Score Data in the Database.** You can include your R models in scripts to score database-resident data. You can perform tasks such as the following:
  - Go from model building to scoring in one step because you can use the same R code for scoring. You do not need to translate the scoring logic as required by some standalone analytic servers.
  - Schedule scripts to be run automatically to perform tasks such as bulk scoring.
  - Score data in the context of a transaction.
  - Perform online what-if scoring.

  – Optionally convert a model to SQL, which Oracle Database does automatically for you. You can then deploy the resulting SQL for low-latency scoring tasks.

- **Execute R Scripts in the Database.** Using Oracle R Enterprise **embedded R execution** functionality, you can create, store, and execute R scripts in the database. When the script executes, Oracle Database starts, controls, and manages one or more R engines that can run in parallel on the database server. By executing scripts on the database server, you can take advantage of scalability and performance of the server.

  With the embedded R execution functionality, you can do the following:

  – Develop and test R scripts interactively and make the scripts available for use by SQL applications

  – Use CRAN and other packages in R scripts on the database server

  – Operationalize entire R scripts in production applications and eliminate porting R code; avoid reinventing code to integrate R results into existing applications

  – Seamlessly leverage Oracle Database as a high performance computing (HPC) environment for R scripts, providing data parallelism and resource management

  – Use the processing and memory resources of Oracle Database and the increased efficiency of read/write operations between the database and the embedded R execution R engines

  – Use the parallel processing capabilities of the database for data-parallel or task-parallel operations

  – Perform parallel simulations

  – Generate XML and PNG images that can be used by R or SQL applications

- **Integrate with the Oracle Technology Stack.** You can take advantage of all aspects of the Oracle technology stack to integrate your data analysis within a larger framework for business intelligence or scientific inquiry. For example, you can integrate the results of your Oracle R Enterprise analysis into Oracle Business Intelligence Enterprise Edition (OBIEE).

Get Online Help for Oracle R Enterprise Classes, Functions, and Methods

The Oracle R Enterprise client packages contain the R components that you use to interact with data in an Oracle database. For a list and brief descriptions of the client packages, see *Oracle R Enterprise Installation and Administration Guide*.

To get help on Oracle R Enterprise classes, functions, and methods, use R functions such as `help` and `showMethods`. If the name of a class or function has an `ore` prefix, you can supply the name to the `help` function. To get help on an overloaded method of an open-source R function, supply the name of the method and the name of the `ore` class.

Example 1–1 has several examples of getting information on Oracle R Enterprise classes, functions, and methods. In the listing following the example some code has been modified to display only a portion of the results and the output of some of the functions is not shown.

Example 1–1 Getting Help on Oracle R Enterprise Classes, Functions, and Methods

```
# List the contents of the OREbase package.
ls("package:OREbase")
```

```
# Get help for the OREbase package.
help("OREbase")

# Get help for the ore virtual class.
help("ore-class")

# Show the subclasses of the ore virtual class.
showClass("ore")

# Get help on the ore.vector class.
help("ore.vector")

# Show the arguments for the aggregate method.
showMethods("aggregate")

# Get help on the aggregate method for an ore.vector object.
help("aggregate,ore.vector-method")

# Get help on the ore.frame class.
help(ore.frame)

# Show the signatures for the merge method.
showMethods("merge")

# Get help on the merge method for an ore.frame object.
help("merge,ore.frame,ore.frame-method")

showMethods("scale")

# Get help on the scale method for an ore.number object.
help("scale,ore.number-method")

# Get help on the ore.connect function.
help("ore.connect")
```

Listing for Example 1–1

```
R> options(width = 80)
# List the contents of the OREbase package.
R> head(ls("package:OREbase"), 12)
 [1] "%in%"          "Arith"         "Compare"       "I"
 [5] "Logic"         "Math"          "NCOL"          "NROW"
 [9] "Summary"       "as.data.frame" "as.env"        "as.factor"
R>
R># Get help for the OREbase package.
R> help("OREbase")      # Output not shown.
R>
R> # Get help for the ore virtual class.
R> help("ore-class")    # Output not shown.
R>
R># Show the subclasses of the ore virtual class.
R> showClass("ore")
Virtual Class "ore" [package "OREbase"]

No Slots, prototype of class "ore.vector"

Known Subclasses:
Class "ore.vector", directly
Class "ore.frame", directly
Class "ore.matrix", directly
```

```
Class "ore.number", by class "ore.vector", distance 2
Class "ore.character", by class "ore.vector", distance 2
Class "ore.factor", by class "ore.vector", distance 2
Class "ore.date", by class "ore.vector", distance 2
Class "ore.datetime", by class "ore.vector", distance 2
Class "ore.difftime", by class "ore.vector", distance 2
Class "ore.logical", by class "ore.vector", distance 3
Class "ore.integer", by class "ore.vector", distance 3
Class "ore.numeric", by class "ore.vector", distance 3
Class "ore.tblmatrix", by class "ore.matrix", distance 2
Class "ore.vecmatrix", by class "ore.matrix", distance 2
R>
R># Get help on the ore.vector class.
R> help("ore.vector")    # Output not shown.
R>
R># Show the arguments for the aggregate method.
R> showMethods("aggregate")
Function: aggregate (package stats)
x="ANY"
x="ore.vector"

# Get help on the aggregate method for an ore.vector object.
R> help("aggregate,ore.vector-method")      # Output not shown.

# Get help on the ore.frame class.
R> help(ore.frame)       # Output not shown.

# Show the signatures for the merge method.
R> showMethods("merge")
Function: merge (package base)
x="ANY", y="ANY"
x="data.frame", y="ore.frame"
x="ore.frame", y="data.frame"
x="ore.frame", y="ore.frame

# Get help on the merge method for an ore.frame object.
R> help("merge,ore.frame,ore.frame-method")  # Output not shown.

R> showMethods("scale")
Function: scale (package base)
x="ANY"
x="ore.frame"
x="ore.number"
x="ore.tblmatrix"
x="ore.vecmatrix"

# Get help on the scale method for an ore.number object.
R> help("scale,ore.number-method")    # Output not shown.

# Get help on the ore.connect function.
R> help("ore.connect")                # Output not shown.
```

From an R session, you can view the Oracle R Enterprise documentation in HTML or PDF formats by invoking the `OREShowDoc` function, as shown in Example 1–2. The function starts a browser that displays the Oracle documentation library for this release.

Example 1–2   Viewing Oracle R Enterprise Documentation

```
OREShowDoc()
```

See Also:

- *Oracle R Enterprise Installation and Administration Guide* for information on installing the Oracle R Enterprise client packages

About Transparently Using R on Oracle Database Data

Oracle R Enterprise has overloaded open source R methods and functions that you can use to operate directly on data in an Oracle Database instance. The methods and functions are in packages that implement a transparency layer that translates R functions into SQL.

The Oracle R Enterprise transparency layer packages and the limitations of converting R into SQL are described in the following topics:

- "About the Transparency Layer"
- Transparency Layer Support for R Data Types and Classes

See Also: Chapter 2, "Getting Started with Oracle R Enterprise"

About the Transparency Layer

The Oracle R Enterprise transparency layer is implemented by the `OREbase`, `OREgraphics`, and `OREstats` packages. These Oracle R Enterprise packages contain overloaded methods of functions in the open source R `base`, `graphics`, and `stats` packages, respectively. The Oracle R Enterprise packages also contain Oracle R Enterprise versions of some of the open source R functions.

With the methods and functions in these packages, you can create R objects that specify data in an Oracle Database instance. When you execute an R expression that uses such an object, the method or function transparently generates a SQL query and sends it to the database. The database then executes the query and returns the results of the operation as an R object.

A database table or view is represented by an `ore.frame` object, which is a subclass of `data.frame`. Other Oracle R Enterprise classes inherit from corresponding R classes, such as `ore.vector` and `vector`. Oracle R Enterprise maps Oracle Database data types to Oracle R Enterprise classes, such as `NUMBER` to `ore.integer`. For more information on Oracle R Enterprise data types and object mappings, see "Transparency Layer Support for R Data Types and Classes" on page 1-7.

Example 1–3 illustrates the translation of an R function invocation into SQL. It uses the overloaded Oracle R Enterprise `aggregate` function to get the mean of the petal lengths from the `IRIS_TABLE` object from Example 1–3.

Example 1–3 Finding the Mean of the Petal Lengths by Species in R

```
aggplen = aggregate(IRIS_TABLE$Petal.Length,
                    by = list(species = IRIS_TABLE$Species),
                    FUN = mean)
aggplen
```

Listing for Example 1–3

```
R> aggplen = aggregate(IRIS_TABLE$Petal.Length,
                       by = list(species = IRIS_TABLE$Species),
                       FUN = mean)
R> aggplen
              species     x
setosa         setosa 1.462
```

```
versicolor versicolor 4.260
virginica   virginica 5.552
```

Example 1–4 shows the SQL equivalent of the `aggregate` function in Example 1–3.

Example 1–4 SQL Equivalent of Example 1–3

```
SELECT "Species", AVG("Petal.Length")
FROM IRIS_TABLE
GROUP BY "Species"
ORDER BY "Species";

Species     AVG("PETAL.LENGTH")
----------- -------------------
setosa       1.4620000000000002
versicolor   4.26
virginica    5.552
```

You can use the transparency layer methods and functions to prepare database-resident data for analysis. You can then use functions in other Oracle R Enterprise packages to build and fit models and use them to score data. For large data sets, you can do the modeling and scoring using R engines embedded in Oracle Database.

See Also:

- "Transparency Layer Support for R Data Types and Classes" for information on the correspondences between R, Oracle R Enterprise, and SQL data types and objects
- Chapter 2, "Getting Started with Oracle R Enterprise"

Transparency Layer Support for R Data Types and Classes

Oracle R Enterprise transparency layer has classes and data types that map R data types to Oracle Database data types. Those classes and data types are described in the following topics:

- About Oracle R Enterprise Data Types and Classes
- About the ore.frame Class
- Support for R Naming Conventions
- About Coercing R and Oracle R Enterprise Class Types

About Oracle R Enterprise Data Types and Classes

Oracle R Enterprise has data types that map R data types to SQL data types. In an R session, when you create database objects from R objects or you create R objects from database data, Oracle R Enterprise translates R data types to SQL data types and the reverse where possible. See Table 1–1 for a list of data type mappings.

Oracle R Enterprise creates objects that are instances of Oracle R Enterprise classes. Oracle R Enterprise overloads many standard R functions so that they use Oracle R Enterprise classes and data types.

R language constructs and syntax are supported for objects that are mapped to Oracle Database objects. For information on the R operators and functions that are supported by Oracle R Enterprise, see Appendix A.

Table 1–1 lists the mappings between R, Oracle R Enterprise, and SQL data types.

Table 1–1 Mappings Between R, Oracle R Enterprise, and SQL Data Types

R Data Type	Oracle R Enterprise Data Type	SQL Data Type
character mode `vector`	`ore.character`	`VARCHAR2` `INTERVAL YEAR TO MONTH`
integer mode `vector`	`ore.integer`	`NUMBER`
logical mode `vector`	`ore.logical`	The `NUMBER` 0 for `FALSE` and 1 for `TRUE`

numeric mode `vector`	`ore.number`	`BINARY_DOUBLE`
		`BINARY_FLOAT`
		`FLOAT`
		`NUMBER`

`Date`	`ore.date`	`DATE`

`POSIXct`
`POSIXlt`
	`ore.datetime`	`TIMESTAMP`
		`TIMESTAMP WITH TIME ZONE`
		`TIMESTAMP WITH LOCAL TIME ZONE`

`difftime`	`ore.difftime`	`INTERVAL DAY TO SECOND`

	Not supported	`LONG`
		`LONG RAW`
		`RAW`
		User defined data types
		Reference data types

Note: Objects of type `ore.datetime` do not support a time zone setting, instead they use the system time zone `Sys.timezone` if it is available or GMT if `Sys.timezone` is not available.

About the ore.frame Class

An `ore.frame` object represents a relational query for an Oracle Database instance. It is the Oracle R Enterprise equivalent of a `data.frame`. Typically, you get `ore.frame` objects that are proxies for database tables. You can then add new columns, or make other changes, to the `ore.frame` proxy object. Any such change does not affect the underlying table. If you then request data from the source table of the `ore.frame` object, the transparency layer function generates a SQL query that has the additional columns in the select list, but the table is not changed.

In R, the elements of a `data.frame` have an explicit order. You can specify elements by using integer indexing. In contrast, relational database tables do not define any order of rows and therefore cannot be directly mapped to R data structures.

If a table has a primary key, which is a set of one or more columns that form a distinct tuple within a row, you can produce ordered results by performing a sort using an `ORDER BY` clause in a `SELECT` statement. However, ordering relational data can be expensive and is often unnecessary for transparency layer operations. For example, ordering is not required to compute summary statistics when invoking the `summary` function on an `ore.frame`.

Oracle R Enterprise has both ordered and unordered ore.frame objects. For information on creating and using these objects, see "Creating Ordered and Unordered ore.frame Objects" on page 2-7.

Example 1–5 creates a data.frame with columns that contain different data types and displays the structure of the data.frame. The example then invokes the ore.push function to create a temporary table in the database that contains a copy of the data of the data.frame. The ore.push invocation also generates an ore.frame object that is a proxy for the table. The example displays the classes of the ore.frame object and of the columns in the data.frame and the ore.frame objects.

Example 1–5 Classes of a data.frame and a Corresponding ore.frame

```
df <- data.frame(a="abc",
                 b=1.456,
                 c=TRUE,
                 d=as.integer(1),
                 e=Sys.Date(),
                 f=as.difftime(c("0:3:20", "11:23:15")))
ore.push(df)
class(of)
class(df$a
class(of$a)
class(df$b)
class(of$b)
class(df$c)
class(of$c)
class(df$d)
class(of$d)
class(df$e)
class(of$e)
class(df$f)
class(of$f)
```

Listing for Example 1–5

```
R> df <- data.frame(a="abc",
+                   b=1.456,
+                   c=TRUE,
+                   d=as.integer(1),
+                   e=Sys.Date(),
+                   f=as.difftime(c("0:3:20", "11:23:15")))
R> ore.push(df)
R> class(of)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> class(df$a)
[1] "factor"
R> class(of$a)
[1] "ore.factor"
attr(,"package")
[1] "OREbase"
R> class(df$b)
[1] "numeric"
R> class(of$b)
[1] "ore.numeric"
attr(,"package")
[1] "OREbase"
R> class(df$c)
[1] "logical"
```

```
R> class(of$c)
[1] "ore.logical"
attr(,"package")
[1] "OREbase"
R> class(df$d)
[1] "integer"
R> class(of$d)
[1] "ore.integer"
attr(,"package")
[1] "OREbase"
R> class(df$e)
[1] "Date"
R> class(of$e)
[1] "ore.date"
attr(,"package")
[1] "OREbase"
R> class(df$f)
[1] "difftime"
R> class(of$f)
[1] "ore.difftime"
attr(,"package")
[1] "OREbase"
```

See Also: "Moving Data to and from the Database" on page 2-12 for information on ore.create

Support for R Naming Conventions

Oracle R Enterprise uses R naming conventions for ore.frame columns instead of the more restrictive Oracle Database naming conventions. The column names of an ore.frame can be longer than 30 bytes, can contain double quotes, and can be non-unique.

About Coercing R and Oracle R Enterprise Class Types

The generic as.ore function coerces in-memory R objects to ore objects. The more specific functions, such as as.ore.character, coerce objects to specific types. The ore.push function implicitly coerces R class types to ore class types and the ore.pull function coerces ore class types to R class types. For information on those functions, see "Moving Data to and from the Database" on page 2-12.

Example 1–6 illustrates coercing R objects to ore objects. creates an R integer object and then uses the generic method as.ore to coerce it to an ore object, which is an ore.integer. The example coerces the R object to various other ore class types. For an example of using as.factor in embedded R execution function, see Example 6–9, "Using the ore.groupApply Function" on page 6-15.

Example 1–6 Coercing R and Oracle R Enterprise Class Types

```
x <- 1:10
class(x)
X <- as.ore(x)
class(X)
Xn <- as.ore.numeric(x)
class(Xn)
Xc <- as.ore.character(x)
class(Xc)
Xc
Xf <- as.ore.factor(x)
Xf
```

Listing for Example 1–6

```
R> x <- 1:10
R> class(x)
[1] "integer"
R> X <- as.ore(x)
R> class(X)
[1] "ore.integer"
attr(,"package")
[1] "OREbase"
R> Xn <- as.ore.numeric(x)
R> class(Xn)
[1] "ore.numeric"
attr(,"package")
[1] "OREbase"
R> Xc <- as.ore.character(x)
R> class(Xc)
[1] "ore.character"
attr(,"package")
[1] "OREbase"
R> Xc
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
R> Xf <- as.ore.factor(x)
R> Xf
 [1] 1  2  3  4  5  6  7  8  9  10
Levels: 1 10 2 3 4 5 6 7 8 9
```

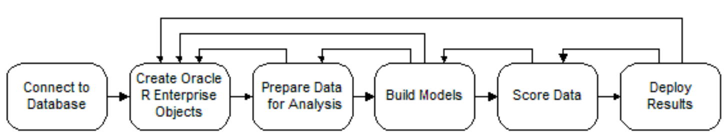
Typical Operations in Using Oracle R Enterprise

In using Oracle R Enterprise, the following is a typical progression of operations:

1. In an R session, connect to a schema in an Oracle Database instance.
2. Attach the schema and synchronize with the schema objects, which generates Oracle R Enterprise proxy objects for database tables.
3. Prepare the data for analysis and possibly perform exploratory data analysis and data visualization.
4. Build models using functions in the `OREmodels` or `OREdm` packages.
5. Score data using the models either in your local R session or by using embedded R execution.
6. Deploy the results of the analysis to end users.

Figure 1–1 illustrates these steps and typical reiterations of them.

Figure 1–1 Typical Oracle R Enterprise Workflow



Chapter 2, "Getting Started with Oracle R Enterprise" describes the following operations:

- Connecting to a database.
- Creating Oracle R Enterprise proxy objects for database tables.

- Moving data from a data.frame in your local R session to a database table, represented by an ore.frame proxy object, and the reverse.

Chapter 3, "Preparing and Exploring Data in the Database" describes preparing data for analysis and exploring data. Preparing and exploring data may include operations such as the following:

- Selecting data from a data set or table.
- Cleaning the data by filtering out unneeded information.
- Ordering the data.
- Intermediate aggregations of data.
- Time-series analysis.
- Recoding or formatting of data.
- Exploratory data analysis.

Chapter 4, "Building Models in Oracle R Enterprise" describes building models, including Oracle Data Mining models, using functions in the OREmodels and OREdm packages.

Chapter 5, "Predicting With R Models" describes using the ore.predict function on Oracle R Enterprise models.

Chapter 6, "Using Oracle R Enterprise Embedded R Execution" describes how to create and execute R scripts in one or more R engines running in the database, and how to save those scripts in the Oracle Database script repository.

Oracle R Enterprise Global Options

Oracle R Enterprise has global options that affect various functions. Table 1–2 lists the Oracle R Enterprise global options and descriptions of them.

Table 1–2 Oracle R Enterprise Global Options

Global	Description
ore.na.extract	A logical value used during logical subscripting of an ore.frame or ore.vector object. When TRUE, rows or elements with an NA logical subscript produce rows or elements with NA values, which mimics how R treats missing value logical subscripting of data.frame and vector objects. When FALSE, an NA logical subscript is interpreted as a FALSE value, resulting in the removal of the corresponding row or element. The default value is FALSE.
ore.parallel	A preferred degree of parallelism to use in embedded R execution. One of the following: ■ A positive integer greater than or equal to 2 for a specific degree of parallelism ■ FALSE or 1 for no parallelism ■ TRUE for the default parallelism of the data argument ■ NULL for the database default for the operation The default value is NULL.
ore.sep	A character string that specifies the separator to use between multiple column row names of an ore.frame. The default value is |.

Table 1–2 (Cont.) Oracle R Enterprise Global Options

Global	Description
ore.trace	A logical value that indicates whether iterative Oracle R Enterprise functions should print output at each iteration. The default value is FALSE.
ore.warn.order	A logical value that specifies whether Oracle R Enterprise displays a warning message when an ore.frame that lacks row names or an ore.vector that lacks element names is used in a function that requires ordering. The default value is TRUE.

See Also:

- "Global Options Related to Ordering" on page 2-8 for information on using ore.sep and ore.warn.order
- "Support for Parallel Execution" on page 6-3

Oracle R Enterprise Examples

Oracle R Enterprise includes several example scripts that demonstrate the use of Oracle R Enterprise functions. This section contains the following topics:

- Listing the Oracle R Enterprise Examples
- Running an Oracle R Enterprise Example Script

See Also:

- "Oracle R Enterprise Online Resources" on page xi

Listing the Oracle R Enterprise Examples

You can display a list of the Oracle R Enterprise example scripts with the demo function as shown in Example 1–7.

Example 1–7 Using demo to List Oracle R Enterprise Examples

```
demo(package = "ORE")
```

Listing for Example 1–7

```
R> demo(package = "ORE")

Demos in package 'ORE':

aggregate      Aggregation
analysis       Basic analysis & data processing operations
basic          Basic connectivity to database
binning        Binning logic
columnfns      Column functions
cor            Correlation matrix
crosstab       Frequency cross tabulations
datastore      DataStore operations
datetime       Date/Time operations
derived        Handling of derived columns
distributions  Distribution, density, and quantile functions
do_eval        Embedded R processing
esm            Exponential smoothing method
freqanalysis   Frequency cross tabulations
glm            Generalized Linear Models
```

```
graphics       Demonstrates visual analysis
group_apply    Embedded R processing by group
hypothesis     Hyphothesis testing functions
matrix         Matrix related operations
nulls          Handling of NULL in SQL vs. NA in R
odm_ai         Oracle Data Mining: attribute importance
odm_ar         Oracle Data Mining: association rules
odm_dt         Oracle Data Mining: decision trees
odm_glm        Oracle Data Mining: generalized linear models
odm_kmeans     Oracle Data Mining: enhanced k-means clustering
odm_nb         Oracle Data Mining: naive Bayes classification
odm_nmf        Oracle Data Mining: non-negative matrix factorization
odm_svm        Oracle Data Mining: support vector machines
push_pull      RDBMS <-> R data transfer
rank           Attributed-based ranking of observations
reg            Ordinary least squares linear regression
row_apply      Embedded R processing by row chunks
sampling       Random row sampling and partitioning of an ore.frame
sql_like       Mapping of R to SQL commands
stepwise       Stepwise OLS linear regression
summary        Summary functionality
table_apply    Embedded R processing of entire table
```

Running an Oracle R Enterprise Example Script

You can run an Oracle R Enterprise example script with the `demo` function. Most of the examples use the `iris` data set that is in the `datasets` package that is included in the R distribution.

To run an example script, start R, load the `ORE` packages with `library(ORE)`, connect to the database, and then use the `demo` function.

Example 1–8 runs the `basic.R` example script. In the listing that follows the example, only the first several lines of the output of the script are shown. The script creates an in-memory database object, `IRIS_TABLE`, which is an `ore.frame` object. The script then demonstrates that the `iris` `data.frame` and the `IRIS_TABLE` `ore.frame` have the same structure and contain the same data.

Example 1–8 Running the basic.R Example Script

```
demo("basic", package = "ORE")
```

Listing for Example 1–8

```
R> demo("basic", package = "ORE")

        demo(basic)
        ---- ~~~~~

Type  <Return>   to start :

R> #
R> #     O R A C L E  R  E N T E R P R I S E  S A M P L E  L I B R A R Y
R> #
R> #     Name: basic.R
R> #     Description: Demonstrates basic connectivity to database
R> #
R> #
R> #
R>
R> ## Set page width
```

```
R> options(width = 80)

R> # Push the built-in iris data frame to the database
R> IRIS_TABLE <- ore.push(iris)

R> # Display the class of IRIS_TABLE
R> class(IRIS_TABLE)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"

R> # Basic commands
R>
R> # Number of rows
R> nrow(iris)
[1] 150

R> nrow(IRIS_TABLE)
[1] 150

R> # Column names of the data frame
R> names(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

R> names(IRIS_TABLE)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

# The rest of the output is not shown.
```

See Also:

- Chapter 2, "Getting Started with Oracle R Enterprise" for more information on using basic Oracle R Enterprise functions

2 Getting Started with Oracle R Enterprise

This chapter describes how to start using Oracle R Enterprise by connecting to an Oracle Database instance and creating Oracle R Enterprise objects and storing them in the database.

This chapter discusses these topics:

- Connecting to an Oracle Database Instance
- Creating and Managing R Objects in Oracle Database

Connecting to an Oracle Database Instance

To use Oracle R Enterprise, you first connect to an Oracle Database instance as described in the following topics:

- About Connecting to the Database
- Using the ore.connect and ore.disconnect Functions

About Connecting to the Database

Oracle R Enterprise client components connect an R session to an Oracle Database instance and the Oracle R Enterprise server components. The connection makes the data in a database schema available to the R user. It also makes the processing power, memory, and storage capacities of the database server available to the R session.

This section has the following topics:

- About Using the ore.connect Function
- About Using the ore.disconnect Function

About Using the ore.connect Function

To begin using Oracle R Enterprise, you first connect to a schema in an Oracle Database instance with the `ore.connect` function. Only one Oracle R Enterprise connection can exist at a time during an R session. If an R session is already connected to the database, then invoking `ore.connect` terminates the active connection before opening a new connection. Before attempting to connect, you can discover whether an active connection exists by using the `is.ore.connected` function.

You explicitly end a connection with the `ore.disconnect` function. If you do not invoke `ore.disconnect`, then the connection is automatically terminated when the R session ends. For more information on `ore.disconnect`, see "About Using the ore.disconnect Function" on page 2-2.

With the `type` argument of `ore.connect`, you specify the type of connection, either ORACLE or HIVE. A HIVE type of connection connects to Hive tables in a Hadoop cluster. An ORACLE type of connection connects to a schema in an Oracle Database instance. The default value of `type` is `"ORACLE"`.

If the connection type is HIVE, then `ore.connect` ignores all other arguments. For information on Oracle R Connector for Hadoop and Hive, see *Oracle Big Data Connectors User's Guide*. The HIVE option applies only if you are using Oracle R Advanced Analytics for Hadoop (ORAAH) in conjunction with a Hadoop cluster. ORAAH is part of the Oracle Big Data Connectors option to the Big Data Appliance.

If the connection type is ORACLE, then you do the following:

- Use the logical `all` argument to specify whether Oracle R Enterprise automatically creates an `ore.frame` object for each table to which the user has access in the schema and makes those `ore.frame` objects visible in the current R session. The `ore.frame` objects contain metadata about the tables. The default value of the `all` argument is `FALSE`.

  If `all = TRUE`, then Oracle R Enterprise implicitly invokes the `ore.sync` and `ore.attach` functions. If `all = FALSE`, then the user must explicitly invoke `ore.sync` to create `ore.frame` objects. To access these objects by name, the user must invoke `ore.attach` to include the names in the search path. For information on those functions, see "Creating R Objects for In-Database Data" on page 2-4.

- Use either the `conn_string` argument, or various combinations of the `user`, `sid`, `host`, `password`, `port`, `service_name`, and `conn_string` arguments to specify information that identifies the connection.

  To avoid using a clear-text password, you can specify an Oracle Wallet password with the `conn_string` argument. No other arguments are needed. By specifying an Oracle Wallet password, you can avoid embedding a database user password in application code, batch jobs, or scripts. For information on using an Oracle Wallet, see *Oracle Database Security Guide*.

  With the other connection identifier arguments, you specify a database user name, host name, and password, and either a system identifier (SID) or service name, and, optionally, a TCP port, or you specify a database user name, password, and a `conn_string` argument.

  The default value of the `port` argument is 1521, the default value of `host` is `"localhost"`, which specifies the local host, and the default value of `conn_string` is NULL. You specify the local host when your R session is running on the same computer as the Oracle Database instance to which you want to connect.

  See Also:

  - "Using the ore.connect and ore.disconnect Functions" on page 2-3 for examples of using the various connection identifiers
  - "Creating R Objects for In-Database Data" on page 2-4

About Using the ore.disconnect Function

To explicitly end the connection between an R session and the Oracle Database instance, invoke the `ore.disconnect` function. Oracle R Enterprise implicitly invokes `ore.disconnect` if you do either of the following:

- Quit the R session.
- Invoke `ore.connect` while an Oracle R Enterprise connection is already active.

When you disconnect the active connection, Oracle R Enterprise discards all Oracle R Enterprise objects that you have not explicitly saved in an Oracle R Enterprise datastore. For information on saving objects, see "Saving and Managing R Objects in the Database" on page 2-15.

Using the ore.connect and ore.disconnect Functions

The examples in this section demonstrate the various ways of specifying a connection.

Example 2–1 first determines whether a connection exists. It then uses values that the `ore.connect` function accepts as the `user`, `sid`, `host`, and `password` arguments. The example ends the connection and then connects to the same database instance as a different user.

Example 2–1 Using ore.connect and Specifying a SID

```
if (!is.ore.connected())
  ore.connect("rquser", "sales", "sales-server", "rquserStrongPassword")
ore.disconnect()
ore.connect("diffuser", "sales", "sales-server", "diffuserStrongPassword")
```

Example 2–2 demonstrates using a service name rather than a SID. It also specifies connecting to the local host.

Example 2–2 Using ore.connect and Specifying a Service Name

```
ore.connect("rquser", host = "localhost", password = "rquserStrongPassword",
            service_name = "sales.example.com")
```

Example 2–3 uses the `conn_string` argument to specify an easy connect string that identifies the connection.

Example 2–3 Using ore.connect and Specifying an Easy Connect String

```
ore.connect(user = "rquser", password = "rquserStrongPassword",
            conn_string = "sales-server:1521:sales
             (ADDRESS=(PROTOCOL=tcp) (HOST=sales-server) (PORT=1521))
             (CONNECT_DATA=(SERVICE_NAME=sales.example.com)))")
```

Example 2–4 uses the `conn_string` argument to specify a full connection string that identifies the connection.

Example 2–4 Using ore.connect and Specifying a Full Connection String

```
ore.connect(user = "rquser", password = "rquserStrongPassword",
            conn_string = "DESCRIPTION=
             (ADDRESS=(PROTOCOL=tcp) (HOST=sales-server) (PORT=1521))
             (CONNECT_DATA=(SERVICE_NAME=myserver.example.com)))")
```

Example 2–5 uses an empty connection string to connect to the local host. It then explicitly ends the connection.

Example 2–5 Using ore.connect and Specifying an Empty Connection String

```
ore.connect(user = "rquser", password = "rquserStrongPassword", conn_string = "")
ore.disconnect()
```

Creating and Managing R Objects in Oracle Database

With transparency layer functions you can connect to an Oracle Database instance and interact with data structures in a database schema. You can move data to and from the database and create database tables. You can also save R objects in the database. The Oracle R Enterprise functions that perform these actions are described in the following topics.

- Creating R Objects for In-Database Data
- Moving Data to and from the Database
- Creating and Deleting Database Tables
- Saving and Managing R Objects in the Database

Creating R Objects for In-Database Data

Using Oracle R Enterprise, you can create R proxy objects in your R session from database-resident data as described in the following topics.

- About Creating R Objects for Database Objects
- Using the ore.sync Function
- Using the ore.get Function
- Using the ore.attach Function

About Creating R Objects for Database Objects

When you invoke `ore.connect` in an R session, Oracle R Enterprise creates a connection to a schema in an Oracle Database instance. To gain access to the data in the database tables in the schema, you use the `ore.sync` function. That function creates an `ore.frame` object that is a proxy for a table in a schema. You can use the `ore.attach` function to add an R environment that represents a schema to the R search path. For information on connecting to the database, see "Connecting to an Oracle Database Instance" on page 2-1.

When you use the `ore.sync` function to create an `ore.frame` object as a proxy for a database table, the name of the `ore.frame` proxy object is the same as the name of the database object. Each `ore.frame` proxy object contains metadata about the corresponding database object.

You can use the proxy `ore.frame` object to select data from the table. When you execute an R operation that selects data from the table, the operation returns the current data from the database object. However, if some application has added a column to the table, or has otherwise changed the metadata of the database object, the `ore.frame` proxy object does not reflect such a change until you again invoke `ore.sync` for the database object.

If you invoke the `ore.sync` function with no tables specified, and if the value of the `all` argument was `FALSE` in the `ore.connect` function call that established the connection to the Oracle database instance, then the `ore.sync` function creates a proxy object for each table in the schema specified by `ore.connect`. You can use the `table` argument to specify the tables for which you want to create `ore.frame` proxy objects.

> **Tip:** To conserve memory resources and save time, you should only add proxies for the tables that you want to use in your R session.

With the `schema` argument, you can specify the schema for which you want to create an R environment and proxy objects. Only one environment for a given database schema can exist at a time. With the `use.keys` argument, you can specify whether you want to use primary keys in the table to order the `ore.frame` object.

> **Tip:** Ordering is expensive in the database. Because most operations in R do not need ordering, you should generally set `use.keys` to `FALSE` unless you need ordering for sampling data or some other purpose. For more information on ordering, see "Creating Ordered and Unordered ore.frame Objects" on page 2-7.

You can use the `ore.ls` function to list the `ore.frame` proxy objects that correspond to database tables in the environment for a schema. You can use the `ore.exists` function to find out if an `ore.frame` proxy object for a database table exists in an R environment. The function returns `TRUE` if the proxy object exists or `FALSE` if it does not. You can remove an `ore.frame` proxy object from an R environment with the `ore.rm` function.

Using the ore.sync Function

Example 2–6 demonstrates the use of the `ore.sync` function. The example first invokes `ore.sync` and specifies three tables in the current schema, which creates an R environment for the rquser schema and creates proxy `ore.frame` objects for the specified tables in that schema. The example lists the `ore.frame` proxy objects in the current environment. The example next invokes `ore.sync` again and creates an R environment for the SH schema and proxy objects in that environment for the specified tables in that schema. The example invokes the `ore.exists` function to find out if the specified table exists in the current environment and then in the SH environment. The example then removes a table from the rquser environment and lists the proxy objects in that environment.

Example 2–6 Using ore.sync to Add ore.frame Proxy Objects to an R Environment

```
# After connecting to a database as rquser. The tables TABLE1, TABLE2, TABLE3,
# and TABLE4 exist in the rquser schema.
# Create ore.frame objects for the specified tables.
ore.sync(table = c("TABLE1", "TABLE3", "TABLE4"))
# List the ore.frame proxy objects in the current environment.
ore.ls()
# The rquser user has been granted SELECT permission on the tables in the
# SH schema.
ore.sync("SH", table = c("CUSTOMERS", "SALES"))
# Find out if the CUSTOMERS ore.frame exists in the rquser environment.
ore.exists("CUSTOMERS")
# Find out if it exists in the SH environment.
ore.exists("CUSTOMERS", schema = "SH")
# List the ore.frame proxy objects in the SH environment.
ore.ls("SH")
# Remove the ore.frame proxy object for TABLE4 from the current environment.
ore.rm("TABLE4")
# List the ore.frame proxy objects in the current environment again.
ore.ls()
```

Listing for Example 2–6

```
R> # After connecting to a database as rquser. The tables TABLE1, TABLE2, TABLE3,
R> # and TABLE4 exist in the rquser schema.
R> # Create ore.frame objects for the specified tables.
R> ore.sync(table = c("TABLE1", "TABLE3", "TABLE4"))
```

```
R> # List the ore.frame proxy objects in the current environment.
R> ore.ls()
 [1] "TABLE1"     "TABLE3"     "TABLE4"
R> # The rquser user has been granted SELECT permission on the tables in the
R> # SH schema.
R> ore.sync("SH", table = c("CUSTOMERS", "SALES"))
R> # Find out if the CUSTOMERS ore.frame exists in the rquser environment.
R> ore.exists("CUSTOMERS")
[1] FALSE
R> # Find out if it exists in the SH environment.
R> ore.exists("CUSTOMERS", schema = "SH")
[1] TRUE
R> # List the ore.frame proxy objects in the SH environment.
R> ore.ls("SH")
[1] "CUSTOMERS" "SALES"
R> # Remove the ore.frame proxy object for TABLE4 from the current environment.
R> ore.rm("TABLE4")
R> # List the ore.frame proxy objects in the current environment again.
R> ore.ls()
 [1] "TABLE1"     TABLE3"
```

Using the ore.get Function

After you have created an R environment and `ore.frame` proxy objects with `ore.sync`, you can get a proxy object by name with the `ore.get` function, as shown in Example 2–7. The example invokes the `ore.sync` function to create an `ore.frame` object that is a proxy for the CUSTOMERS table in the SH schema. The example then gets the dimensions of the proxy object.

Example 2–7 Using ore.get to Get a Database Table

```
ore.sync(schema = "SH", table = "CUSTOMERS", use.keys = FALSE)
dim(ore.get(name = "CUSTOMERS", schema = "SH"))
```

Listing for Example 2–7

```
R> ore.sync(schema = "SH", table = "CUSTOMERS", use.keys = FALSE)
R> dim(ore.get(name = "CUSTOMERS", schema = "SH"))
[1] 630  15
```

You can use `ore.get` to get the proxy `ore.frame` for a table and assign it to a variable in R, as in `SH_CUST <- ore.get(name = "CUSTOMERS", schema = "SH")`. The `ore.frame` exists in the R global environment, which can be referred to using `.GlobalEnv`, and so it appears in the list returned by the `ls` function. Also, because this object exists in the R global environment, as opposed an R environment that represents a database schema, it is not listed by the `ore.ls` function.

Using the ore.attach Function

With `ore.attach`, you add an R environment for a database schema to the R search path. When you add the R environment, you have access to database tables by name through the proxy objects created by the `ore.sync` function without needing to specify the schema environment.

The default schema is the one specified in creating the connection and the default position in the search path is 2. You can specify the schema and the position in the ore.attach function invocation.. You can also specify whether you want the `ore.attach` function to indicate whether a naming conflict occurs when adding the environment. You can detach the environment for a schema from the R search path with the `ore.detach` function.

Example 2–8 demonstrates the use of the `ore.attach` function. Comments in the example explain the function invocations.

Example 2–8 Using ore.attach to Add an Environment for a Database Schema

```
# Connected as rquser.
# Add the environment for the rquser schema to the R search path.
ore.attach()
# Create an unordered ore.frame proxy object in the SH environment for the
# specifed table.
ore.sync(schema = "SH", table = "CUSTOMERS", use.keys = FALSE)
# Add the environment for the SH schema to the search path and warn if naming
# conflicts exist.
ore.attach("SH", 3, warn.conflicts = TRUE)
# Display the number of rows and columns in the proxy object for the table.
dim(CUSTOMERS)
# Remove the environment for the SH schema from the search path.
ore.detach("SH")
# Invoke the dim function again.
dim(CUSTOMERS)
```

Listing for Example 2–8

```
R> # Connected as rquser.
R> # Add the environment for the rquser schema to the R search path.
ore.attach()
R> # Create an unordered ore.frame proxy object in the SH environment for the
R> # specifed table.
R> ore.sync(schema = "SH", table = "CUSTOMERS", use.keys = FALSE)
R> # Add the environment for the SH schema to the search path and warn if naming
R> # conflicts exist.
R> ore.attach("SH", 3, warn.conflicts = TRUE)
R> # Display the number of rows and columns in the proxy object for the table.
R> dim(CUSTOMERS)
[1] 630  15
R> # Remove the environment for the SH schema from the search path.
R> ore.detach("SH")
R> # Invoke the dim function again.
R> dim(CUSTOMERS)
Error: object 'CUSTOMERS' not found
```

Creating Ordered and Unordered ore.frame Objects

Oracle R Enterprise provides the ability to create ordered or unordered `ore.frame` objects. The following topics describe this feature.

- About Ordering in ore.frame Objects
- Global Options Related to Ordering
- Ordering Using Keys
- Ordering Using Row Names
- Using Ordered Frames

About Ordering in ore.frame Objects

R objects such as `vector` and `data.frame` have an implicit ordering of their elements. The data in an Oracle Database table is not necessarily ordered. For some R operations, ordering is useful whereas for other operations it is unnecessary. By ordering an ore.frame, you are able to index the ore.frame object by using either integer or character indexes.

Using an ordered ore.frame object that is a proxy for a SQL query can be time-consuming for a large data set. Therefore, although Oracle R Enterprise attempts to create ordered ore.frame objects by default, it also provides the means of creating an unordered ore.frame object.

When you invoke the ore.sync function to create an Oracle R Enterprise ore.frame object as a proxy for a SQL query, you can use the use.keys argument to specify whether the ore.frame can be ordered or must be unordered.

An ore.frame object can be ordered if one or more of the following conditions are true:

- The value of the use.keys argument of the ore.sync function is TRUE and a primary key is defined on the underlying table
- The row names of the ore.frame constitute a unique tuple
- The ore.frame object is produced by certain functions such as aggregate and cbind
- All of the ore.frame objects that are input arguments to relevant Oracle R Enterprise functions are ordered

An ore.frame object is unordered if one or more of the following conditions are true:

- The value of the use.keys argument of the ore.sync function is FALSE
- No primary key is defined on the underlying table and either the row names of the ore.frame object are not specified or the row names of the ore.frame object are set to NULL
- One or more of the ore.frame objects that are input arguments to relevant Oracle R Enterprise functions are unordered

An unordered ore.frame object has null row names. You can determine whether an ore.frame object is ordered by invoking is.null on the row names of the objects, as shown in the last lines of Example 2–9 on page 2-9. If the ore.frame object is unordered, is.null returns an error.

See Also: "Indexing Data" on page 3-5

Global Options Related to Ordering

Oracle R Enterprise has options that relate to the ordering of an ore.frame object. The ore.warn.order global option specifies whether you want Oracle R Enterprise to display a warning message if you use an unordered ore.frame object in a function that requires ordering. If you know what to expect in an operation, then you might want to turn the warnings off so they do not appear in the output. For examples of the warning messages, see Example 2–9 and Example 2–10.

You can see what the current setting is, or turn the option on or off, as in the following example.

```
R> options("ore.warn.order")
$ore.warn.order
[1] TRUE
R> options("ore.warn.order" = FALSE)
R> options("ore.warn.order" = TRUE)
```

With the ore.sep option, you can specify the separator between the row name values that you use for multi-column keys, as in the following example.

```
R> options("ore.sep")
$ore.sep
[1] "|"

R> options("ore.sep" = "/")
R> options("ore.sep" = "|")
```

Ordering Using Keys

You can use the primary key of a database table to order an `ore.frame` object, as demonstrated in Example 2–9. The example loads the spam data set from the `kernlab` package. It creates two tables from the data set and then adds a primary key to the SPAM_PK table. It displays the first eight rows of each table. The proxy object for the SPAM_PK table is an ordered `ore.frame` object. It is has row names that are a combination of the TS and USERID column values separated by the "|" character. The proxy object for the SPAM_NOPK table is an unordered `ore.frame` object that has the symbol `SPAM_NOPK`. By default, `SPAM_NOPK` has row names that are sequential numbers.

Example 2–9 Ordering Using Keys

```
# Prepare the data.
library(kernlab)
data(spam)
s <- spam
# Create a column that has integer values.
s$TS <- 1001:(1000 + nrow(s))
# Create a column that has integer values with each number repeated twice.
s$USERID <- rep(351:400, each=2, len=nrow(s))
# Ensure that the database tables do not exist.
ore.drop(table='SPAM_PK')
ore.drop(table='SPAM_NOPK')
# Create database tables.
ore.create(s[,c(59:60,1:28)], table="SPAM_PK")
ore.create(s[,c(59:60,1:28)], table="SPAM_NOPK")
# Using a SQL statement, alter the SPAM_PK table to add a composite primary key.
ore.exec("alter table SPAM_PK add constraint SPAM_PK primary key
           (\"USERID\",\"TS\")")
# Synchronize the table to get the change to it.
ore.sync(table = "SPAM_PK")
# View the data in the tables.
# The row names of the ordered SPAM_PK are the primary key column values.
head(SPAM_PK[,1:8])
# The row names of the unordered SPAM_NOPK are sequential numbers.
# The first warning results from the inner accessing of SPAM_NOPK to subset
# the columns. The second warning is for the invocation of the head
# function on that subset.
head(SPAM_NOPK[,1:8])
# Verify that SPAM_NOPK is unordered.
is.null(row.names(SPAM_NOPK))
```

Listing for Example 2–9

```
R> # Prepare the data.
R> library(kernlab)
R> data(spam)
R> s <- spam
R> # Create a column that has integer values.
R> s$TS <- 1001:(1000 + nrow(s))
R> # Create a column that has integer values with each number repeated twice.
R> s$USERID <- rep(351:400, each=2, len=nrow(s))
```

```
R> # Ensure that the database tables do not exist.
R> ore.drop(table='SPAM_PK')
R> ore.drop(table='SPAM_NOPK')
R> # Create database tables.
R> ore.create(s[,c(59:60,1:28)], table="SPAM_PK")
R> ore.create(s[,c(59:60,1:28)], table="SPAM_NOPK")
R> # Using a SQL statement, alter the SPAM_PK table to add a composite primary
key.
R> ore.exec("alter table SPAM_PK add constraint SPAM_PK primary key
+  (\"USERID\",\"TS\")")
R> # Synchronize the table to get the change to it.
R> ore.sync(table = "SPAM_PK")
R> # View the data in the tables.
R> # The row names of the ordered SPAM_PK are the primary key column values.
R> head(SPAM_PK[,1:8])
           TS USERID make address  all num3d  our over
1001|351 1001    351 0.00    0.64 0.64     0 0.32 0.00
1002|351 1002    351 0.21    0.28 0.50     0 0.14 0.28
1003|352 1003    352 0.06    0.00 0.71     0 1.23 0.19
1004|352 1004    352 0.00    0.00 0.00     0 0.63 0.00
1005|353 1005    353 0.00    0.00 0.00     0 0.63 0.00
1006|353 1006    353 0.00    0.00 0.00     0 1.85 0.00
R> # The row names of the unordered SPAM_NOPK are sequential numbers.
R> # The first warning results from the inner accessing of SPAM_NOPK to subset
R> # the columns. The second warning is for the invocation of the head
R> # function on that subset.
R> head(SPAM_NOPK[,1:8])
    TS USERID make address  all num3d  our over
1 1001    351 0.00    0.64 0.64     0 0.32 0.00
2 1002    351 0.21    0.28 0.50     0 0.14 0.28
3 1003    352 0.06    0.00 0.71     0 1.23 0.19
4 1004    352 0.00    0.00 0.00     0 0.63 0.00
5 1005    353 0.00    0.00 0.00     0 0.63 0.00
6 1006    353 0.00    0.00 0.00     0 1.85 0.00
Warning messages:
1: ORE object has no unique key - using random order
2: ORE object has no unique key - using random order
R> # Verify that SPAM_NOPK is unordered.
R> is.null(row.names(SPAM_NOPK))
Error: ORE object has no unique key
```

Ordering Using Row Names

You can use row names to order an `ore.frame` object, as demonstrated in Example 2–10. The example creates a `data.frame` object in the local R session memory and pushes it to the `ore.frame` object with the symbol `a`, which exists in the memory of the Oracle database to which the R session is connected. The example shows that the `ore.frame` object has the default row names of the R `data.frame` object. Because the `ore.frame` object is ordered, invoking the `row.names` function on it does not produce a warning message.

The example uses the ordered `SPAM_PK` and unordered `SPAM_NOPK` `ore.frame` objects from Example 2–9 to show that invoking `row.names` on the unordered `SPAM_NOPK` produces a warning message but invoking it on the ordered `SPAM_PK` does not.

The `SPAM_PK` object is ordered by the row names, which are the combined values of the TS and USERID column values separated by the "`|`" character. The example shows that you can change the row names.

Example 2–10 Ordering Using Row Names

```
# Create an ordered ore.frame by default.
a <- ore.push(data.frame(a=c(1:10,10:1), b=letters[c(1:10,10:1)]))
# Display the values in the b column. Note that because the ore.frame is
# ordered, no warnings appear.
a$b
# Display the default row names for the first six rows of the a column.
row.names(head(a))
# SPAM_NOPK has no unique key, so row.names raises error messages.
row.names(head(SPAM_NOPK))
# Row names consist of TS '|' USERID.
# For display on this page, only the first four row names are shown.
row.names(head(SPAM_PK))
# Reassign the row names to the TS column only
row.names(SPAM_PK) <- SPAM_PK$TS
# The row names now correspond to the TS values only.
row.names(head(SPAM_PK[,1:4]))
head(SPAM_PK[,1:4])
```

Listing for Example 2–10

```
R> # Create an ordered ore.frame by default.
R> a <- ore.push(data.frame(a=c(1:10,10:1), b=letters[c(1:10,10:1)]))
R> # Display the values in the b column. Note that because the ore.frame is
R> # ordered, no warnings appear.
R> a$b
 [1] a b c d e f g h i j j i h g f e d c b a
Levels: a b c d e f g h i j
R> # Display the default row names for the first six rows of the a column.
R> row.names(head(a))
[1] 1 2 3 4 5 6
R> # SPAM_NOPK has no unique key, so row.names raises error messages.
R> row.names(head(SPAM_NOPK))
Error: ORE object has no unique key
In addition: Warning message:
ORE object has no unique key - using random order
R> # Row names consist of TS '|' USERID.
R> # For display on this page, only the first four row names are shown.
R> row.names(head(SPAM_PK))
        1001|351         1002|351         1003|352         1004|352
"1001|3.51E+002" "1002|3.51E+002" "1003|3.52E+002" "1004|3.52E+002"
R> # Reassign the row names to the TS column only
R> row.names(SPAM_PK) <- SPAM_PK$TS
R> # The row names now correspond to the TS values only.
R> row.names(head(SPAM_PK[,1:4]))
[1] 1001 1002 1003 1004 1005 1006
R> head(SPAM_PK[,1:4])
       TS USERID make address
1001 1001    351 0.00    0.64
1002 1002    351 0.21    0.28
1003 1003    352 0.06    0.00
1004 1004    352 0.00    0.00
1005 1005    353 0.00    0.00
1006 1006    353 0.00    0.00
```

Using Ordered Frames

Example 2–11 example uses the ordered `SPAM_PK` and unordered `SPAM_NOPK` `ore.frame` objects from Example 2–9 to show the result of merging two ordered `ore.frame` objects and two unordered `ore.frame` objects.

Example 2–11 Merging Ordered and Unordered ore.frame Objects

```
# Create objects for merging data from unordered ore.frame objects.
x <- SPAM_NOPK[,1:4]
y <- SPAM_NOPK[,c(1,2,4,5)]
m1 <- merge(x, y, by="USERID")
# The merged result m1 produces a warning because it is not an ordered frame.
head(m1,3)
# Create objects for merging data from ordered ore.frame objects.
x <- SPAM_PK[,1:4]
y <- SPAM_PK[,c(1,2,4,5)]
# The merged result m1 does not produce a warning now because it is an
# ordered frame.
m1 <- merge(x, y, by="USERID")
head(m1,3)
```

Listing for Example 2–11

```
R> # Create objects for merging data from unordered ore.frame objects.
R> x <- SPAM_NOPK[,1:4]
R> y <- SPAM_NOPK[,c(1,2,4,5)]
R> m1 <- merge(x, y, by="USERID")
R> # The merged result m1 produces a warning because it is not an ordered frame.
R> head(m1,3)
  USERID TS.x make address.x TS.y address.y  all
1    351 5601 0.00         0 1001      0.64 0.64
2    351 5502 0.00         0 1001      0.64 0.64
3    351 5501 0.78         0 1001      0.64 0.64
Warning messages:
1: ORE object has no unique key - using random order
2: ORE object has no unique key - using random order
R> # Create objects for merging data from ordered ore.frame objects.
R> x <- SPAM_PK[,1:4]
R> y <- SPAM_PK[,c(1,2,4,5)]
R> # The merged result m1 does not produce a warning now because it is an
R> # ordered frame.
R> m1 <- merge(x, y, by="USERID")
R> head(m1,3)
          USERID TS.x make address.x TS.y address.y  all
1001|1001    351 1001    0      0.64 1001      0.64 0.64
1001|1002    351 1001    0      0.64 1002      0.28 0.50
1001|1101    351 1001    0      0.64 1101      0.00 0.00
```

See Also: Example 3–4, "Indexing an ore.frame Object" on page 3-5

Moving Data to and from the Database

You can create a temporary database table, and corresponding proxy `ore.frame` object, from a local R object with the `ore.push` function. You can create a local R object that contains a copy of data represented by an Oracle R Enterprise proxy object with the `ore.pull` function.

The `ore.push` function translates an R object into an Oracle R Enterprise object of the appropriate data type. The `ore.pull` function takes an `ore` class object and returns an R object. If the input object is an `ore.list`, the `ore.pull` function creates a `data.frame` and translates each the data of each database column into the appropriate R representation.

Note: You can pull data to a local R `data.frame` only if the data can fit into the R session memory. Also, even if the data fits in memory but is still very large, you may not be able to perform many, or any, R functions in the client R session.

Example 2–12 demonstrates pushing an R `data.frame` object to the database as a temporary database table with an associated `ore.frame` object, `iris_of`, then creating another `ore.frame` object, `iris_of_setosa`, by selecting one column from `iris_of`, and then pulling the `iris_of_setosa` object into the local R session memory as a `data.frame` object. The example displays the class of some of the objects.

Example 2–12 Using ore.push and ore.pull to Move Data

```
class(iris)
# Push the iris data frame to the database.
iris_of <- ore.push(iris)
class(iris_of)
# Display the data type of the Sepal.Length column in the data.frame.
class(iris$Sepal.Length)
# Display the data type of the Sepal.Length column in the ore.frame.
class(iris_of$Sepal.Length)
# Filter one column of the data set.
iris_of_setosa <- iris_of[iris_of$Species == "setosa", ]
class(iris_of_setosa)
# Pull the selected column into the local R client memory.
local_setosa = ore.pull(iris_of_setosa)
class(local_setosa)
```

Listing for Example 2–12

```
R> class(iris)
[1] "data.frame"
R> # Push the iris data frame to the database.
R> iris_of <- ore.push(iris)
R> class(iris_of)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> # Display the data type of the Sepal.Length column in the data.frame.
R> class(iris$Sepal.Length)
[1] "numeric"
R> # Display the data type of the Sepal.Length column in the ore.frame.
R> class(iris_of$Sepal.Length)
[1] "ore.numeric"
attr(,"package")
[1] "OREbase"
R> # Filter one column of the data set.
R> iris_of_setosa <- iris_of[iris_of$Species == "setosa", ]
R> class(iris_of_setosa)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> # Pull the selected column into the local R client memory.
R> local_setosa = ore.pull(iris_of_setosa)
R> class(local_setosa)
[1] "data.frame"
```

Unless you explicitly save them, the temporary database tables and their corresponding Oracle R Enterprise proxy objects that you create with the `ore.push` function are discarded when you quit the R session.

See Also:

- "Transparency Layer Support for R Data Types and Classes" on page 1-7 for information on data type mappings
- "Saving and Managing R Objects in the Database" on page 2-15 for information on permanently saving the Oracle R Enterprise objects in the database
- The `push_pull.R` example script.

Creating and Deleting Database Tables

You can use the `ore.create` function to create a persistent table in an Oracle Database schema. Creating the table automatically creates an `ore.frame` proxy object for the table in the R environment that represents your database schema. The proxy `ore.frame` object has the same name as the table. You can delete the persistent table in an Oracle Database schema with the `ore.drop` function.

Caution: Only use the `ore.drop` function to delete a database table. Never use it to remove an `ore.frame` object. To remove an `ore.frame` object, use the `ore.rm` function.

Example 2–13 creates tables in the database and drops some of them.

Example 2–13 Using ore.create and ore.drop to Create and Drop Tables

```
# Create the ANIMALS_TABLE table from the Animals data.frame.
ore.create(Animals, table = "ANIMALS_TABLE")
# Create data.frame objects.
df1 <- data.frame(x1 = 1:5, y1 = letters[1:5])
df2 <- data.frame(x2 = 5:1, y2 = letters[11:15])
# Create the DF1_TABLE and DF2_TABLE tables from the data.frame objects.
ore.create(df1, "DF1_TABLE")
ore.create(df2, "DF2_TABLE")
# Create the CARS93_TABLE table from the Cars93 data.frame.
ore.create(Cars93, table = "CARS93_TABLE")
# List the Oracle R Enterprise proxy objects.
ore.ls()
# Drop the CARS93_TABLE object.
ore.drop(table = "CARS93_TABLE")
# List the Oracle R Enterprise proxy objects again.
ore.ls()
```

Listing for Example 2–13

```
R> # Create the ANIMALS_TABLE table from the Animals data.frame.
R> ore.create(Animals, table = "ANIMALS_TABLE")
R> # Create data.frame objects.
R> df1 <- data.frame(x1 = 1:5, y1 = letters[1:5])
R> df2 <- data.frame(x2 = 5:1, y2 = letters[11:15])
R> # Create the DF1_TABLE and DF2_TABLE tables from the data.frame objects.
R> ore.create(df1, "DF1_TABLE")
R> ore.create(df2, "DF2_TABLE")
R> # Create the CARS93_TABLE table from the Cars93 data.frame.
```

```
R> .create(Cars93, table = "CARS93_TABLE")
R> # List the Oracle R Enterprise proxy objects.
R> .ls()
[1] "ANIMALS_TABLE"  "CARS93_TABLE"  "DF1_TABLE"  "DF2_TABLE"
R> # Drop the CARS93_TABLE object.
R> ore.drop(table = "CARS93_TABLE")
R> # List the Oracle R Enterprise proxy objects again.
R> ore.ls()
[1] "ANIMALS_TABLE"  "DF1_TABLE"  "DF2_TABLE"
```

Saving and Managing R Objects in the Database

Oracle R Enterprise provides datastores that you can use to save Oracle R Enterprise proxy objects, as well as any R object, in an Oracle database. You can restore the saved objects in another R session. The objects in a datastore are also accessible to embedded R execution through both the R and the SQL interfaces.

This section describes the Oracle R Enterprise functions that you can use to create and manage datastores. The section contains the following topics:

- About Persisting Oracle R Enterprise Objects
- About Oracle R Enterprise Datastores
- Saving Objects to a Datastore
- Getting Information about Datastore Contents
- Restoring Objects from a Datastore
- Deleting a Datastore
- About Using a Datastore in Embedded R Execution

About Persisting Oracle R Enterprise Objects

R objects, including Oracle R Enterprise proxy objects, exist for the duration of the current R session unless you explicitly save them. The standard R functions for saving and restoring R objects, `save` and `load`, serialize objects in R memory to store them in a file and deserialize them to restore them in memory. However, for Oracle R Enterprise proxy objects, those functions do not save the database objects associated with the proxy objects in an Oracle database; therefore the saved proxy objects do not behave properly in a different R session.

You can save Oracle R Enterprise proxy objects, as well as any R object, with the `ore.save` function. The `ore.save` function specifies an Oracle R Enterprise datastore. A datastore persists in the database when you end the R session. The datastore maintains the referential integrity of the objects it contains. Using the `ore.load` function, you can restore in another R session the objects in the datastore.

Using a datastore, you can do the following:

- Save Oracle R Enterprise and other R objects that you create in one R session and restore them in another R session.
- Pass arguments to R functions for use in embedded R execution.
- Pass objects for use in embedded R execution. You could, for example, use a function in the `OREdm` package to build an Oracle Data Mining model and save it in a `datastore`. You could then use that model to score data in the database through embedded R execution. For an example of using a datastore in an embedded R execution function, see Example 6–7 on page 6-11.

Table 2–1 lists the functions that manipulate datastores and provides brief descriptions of them.

Table 2–1 Functions that Manipulate Datastores

Function	Description
`ore.save`	Saves R objects in a new or existing datastore.
`ore.load`	Restores objects from a datastore into an R environment.
`ore.lazyLoad`	Lazily restores objects from a datastore into an R environment.
`ore.delete`	Deletes a datastore from the current Oracle database schema.
`ore.datastore`	Lists information about a datastore in the current Oracle database schema.
`ore.datastoreSummary`	Provides detailed information about the specified datastore in the current Oracle database schema.

See Also: Chapter 6, "Using Oracle R Enterprise Embedded R Execution" for information on using the R and the SQL interfaces to embedded R execution

About Oracle R Enterprise Datastores

Each database schema has a table that stores named Oracle R Enterprise datastores. A datastore can contain Oracle R Enterprise objects and standard R objects.

You create a datastore with the `ore.save` function. When you create a datastore, you specify a name for it. You can save objects in one or more datastores.

As long as a datastore contains an Oracle R Enterprise proxy object for a database object, the database object persists between R sessions. For example, you could use the `ore.odmNB` function in the `OREdm` package to build an Oracle Data Mining Naive Bayes model. If you save the resulting `ore.odmNB` object in a datastore and end the R session, then Oracle Database does not delete the Oracle Data Mining model. If no datastore contains the `ore.odmNB` object and the R session ends, then the database automatically drops the model.

Saving Objects to a Datastore

The `ore.save` function saves one or more R objects in the specified datastore. By default, Oracle R Enterprise creates the datastore in the current user schema. With the arguments to `ore.save`, you can provide the names of specific objects, or provide a list of objects. You can specify a particular R environment to search for the objects you would like to save. The `overwrite` and `append` arguments are mutually exclusive. If you set the `overwrite` argument to `TRUE`, then you can replace an existing datastore with another datastore of the same name. If you set the `append` argument to `TRUE`, then you can add objects to an existing datastore. With the `description` argument, you can provide some descriptive text that appears when you get information about the datastore. The `description` argument has no effect when used with the `append` argument.

Example 2–14 demonstrates creating datastores using different combinations of arguments.

Example 2–14 Saving Objects and Creating a Datastore

```
# Create some R objects.
df1 <- data.frame(x1 = 1:5, y1 = letters[1:5])
```

```
df2 <- data.frame(x2 = 5:1, y2 = letters[11:15])
iris_of <- ore.push(iris)

# Create a database table and an Oracle R Enterprise proxy object for the table.
ore.create(longley, table = "LONGLEY")

# List the R objects.
ls()

# List the Oracle R Enterprise proxy objects.
ore.ls()

# Save the proxy object and all objects in the current workspace environment
# to the datastore named ds1 and supply a description.
ore.save(LONGLEY, list = ls(), name = "ds1", description = "My datastore")

# Create some more objects.
x <- stats::runif(20)  # x is an object of type numeric.
y <- list(a = 1, b = TRUE, c = "hoopsa")
z <- ore.push(x)  # z is an object of type ore.numeric.

# Create another datastore.
ore.save(x, y, name = "ds2", description = "x and y")

# Overwrite the contents of datastore ds2.
ore.save(x, name = "ds2", overwrite = TRUE, description = "only x")

# Append object z to datastore ds2.
ore.save(z, name = "ds2", append = TRUE)
```

Listing for Example 2–14

```
R> # Create some R objects.
R> df1 <- data.frame(x1 = 1:5, y1 = letters[1:5])
R> df2 <- data.frame(x2 = 5:1, y2 = letters[11:15])
R> iris_of <- ore.push(iris)
R>
R> # Create a database table and an Oracle R Enterprise proxy object for the
table.
R> ore.create(longley, table = "LONGLEY")
R>
R> # List the R objects.
R> ls()
[1] "df1"     "df2"     "iris_of"
R>
R> # List the Oracle R Enterprise proxy objects.
R> ore.ls()
[1] "LONGLEY"
R>
R> # Save the proxy object and all objects in the current workspace environment
R> # to the datastore named ds1 and supply a description.
R> ore.save(LONGLEY, list = ls(), name = "ds1", description = "My datastore")
R>
R> # Create some more objects.
R> x <- stats::runif(20)  # x is an object of type numeric.
R> y <- list(a = 1, b = TRUE, c = "hoopsa")
R> z <- ore.push(x)  # z is an object of type ore.numeric.
R>
R> # Create another datastore.
R> ore.save(x, y, name = "ds2", description = "x and y")
R>
```

```
R> # Overwrite the contents of datastore ds2.
R> ore.save(x, name = "ds2", overwrite = TRUE, description = "only x")
R>
R> # Append object z to datastore ds2.
R> ore.save(z, name = "ds2", append = TRUE)
```

Getting Information about Datastore Contents

You can get information about a datastore in the current user schema by using the `ore.datastore` and `ore.datastoreSummary` functions.

Using the `ore.datastore` function, you can list basic information about datastores. The function returns a `data.frame` object with columns that correspond to the datastore name, the number of objects in the datastore, the datastore size, the creation date, and a description. Rows are sorted by column `datastore.name` in alphabetical order. You can search for a datastore by name or by using a regular expression pattern.

Example 2–15 demonstrates using the `ore.datastore` function. The example uses some of the R objects created in Example 2–14 on page 2-16.

Example 2–15 Using the ore.datastore Function

```
# The datastore objects ds1 and ds2 and objects data.frame objects df1 and df2
# were created in Example 2-14.
ore.save(df1, df2, name = "dfobj", description = "df objects"
ore.save(x, y, z, name = "another_ds", description = "For pattern matching")

# List all of the datastore objects.
ore.datastore()

# List the specified datastore.
ore.datastore("ds1")

# List the datastore objects with names that include "ds".
ore.datastore(pattern = "ds")
```

Listing for Example 2–15

```
R> # The datastore objects ds1 and ds2 and objects data.frame objects df1 and df2
R> # were created in Example 2-14.
R> ore.save(df1, df2, name = "dfobj", description = "df objects"
R> ore.save(x, y, z, name = "another_ds", description = "For pattern matching")
R>
R> # List all of the datastore objects.
R> ore.datastore()
  datastore.name object.count size       creation.date  description
1     another_ds            3 1217 2013-11-08 13:36:19 For pattern mattching
2          dfobj            2  656 2013-11-08 13:27:26            df objects
3            ds1            4 2908 2013-11-08 10:34:38          My datastore
4            ds2            2 1085 2013-11-08 10:46:08                only x

R> # List the specified datastore.
R> ore.datastore("ds1")
  datastore.name object.count size       creation.date  description
1            ds1            4 2908 2013-11-08 10:41:09 My datastore
R>
R> # List the datastore objects with names that include "ds".
R> ore.datastore(pattern = "ds")
  datastore.name object.count size       creation.date            description
1     another_ds            3 1217 2013-11-08 13:36:19  For pattern mattching
2            ds1            4 2908 2013-11-08 10:34:38           My datastore
```

```
3            ds2            2 1085 2013-11-08 10:46:08                 only x
```

The `ore.datastoreSummary` function returns information about the R objects saved within a datastore in the user schema in the connected database. The function returns a `data.frame` with columns that correspond to object name, object class, object size, and either the length of the object, if it is a `vector`, or the number of rows and columns, if it is a `data.frame` object. It takes one argument, the name of a datastore.

Example 2–16 demonstrates using the `ore.datastoreSummary` function. The example uses the datastores created in Example 2–14 on page 2-16.

Example 2–16 Using the ore.datastoreSummary Function

```
ore.datastoreSummary("ds1")
ore.datastoreSummary("ds2")
```

Listing for Example 2–16

```
R> ore.datastoreSummary("ds1")
    object.name      class size length row.count col.count
1 ANIMALS_TABLE  ore.frame  849      2        28         2
2           df1 data.frame  328      2         5         2
3           df2 data.frame  328      2         5         2
4       iris_of  ore.frame 1403      5       150         5
R> ore.datastoreSummary("ds2")
  object.name       class size length row.count col.count
1           x     numeric  182     20        NA        NA
2           z ore.numeric  903     20        NA        NA
```

Restoring Objects from a Datastore

The `ore.load` function restores R objects saved in a datastore to the R global environment, `.GlobalEnv`. The function returns a character vector that contains the names of the restored objects.

You can load all of the saved objects or you can use the `list` argument to specify the objects to load. With the `envir` argument, you can specify an environment in which to load objects.

Example 2–17 demonstrates using the `ore.load` function to restore objects from datastore objects created in Example 2–15 on page 2-18.

Example 2–17 Using the ore.load Function to Restore Objects from a Datastore

```
# We are in the same R session as Example 2-15. List the R objects.
ls()

# List the datastore objects.
ore.datastore()

# Delete the x and z objects.
ls()

# Restore all of the objects in datastore ds2.
ore.load("ds2")

ls()

# After ending the R session and starting another session.
ls()
# The datastore objects persist between sessions.
```

```
ore.datastore()

# Restore some of the objects from datastore ds1.
ore.load("ds1", list = c("df1", "df2", "iris_of"))
ls()
```

Listing for Example 2–17

```
R> # We are in the same R session as Example 2-15. List the R objects.
R> ls()
[1] "df1"     "df2"     "iris_of" "x"       "y"       "z"
R>
R> # List the datastore objects.
R> ore.datastore()
  datastore.name object.count size       creation.date  description
1     another_ds            3 1217 2013-11-08 13:36:19 For pattern mattching
2          dfobj            2  656 2013-11-08 13:27:26            df objects
3            ds1            4 2908 2013-11-08 10:34:38          My datastore
4            ds2            2 1085 2013-11-08 10:46:08                only x
R>
R> # Delete the x and z objects.
R> ls()
[1] "df1"     "df2"     "iris_of" "y"
R>
R> # Restore all of the objects in datastore ds2.
R> ore.load("ds2")
[1] "x" "z"
R>
R> ls()
[1] "df1"     "df2"     "iris_of" "x"       "y"       "z"
R>
R> # After ending the R session and starting another session.
R> ls()
character(0)
R> # The datastore objects persist between sessions.
R> ore.datastore()
  datastore.name object.count size       creation.date           description
1     another_ds            3 1217 2013-11-08 14:16:19 For pattern mattching
2          dfobj            2  656 2013-11-08 13:27:26            df objects
3            ds1            4 2908 2013-11-08 13:54:38          My datastore
4            ds2            2 1085 2013-11-08 10:46:08                only x

R> # Restore some of the objects from datastore ds1.
R> ore.load("ds1", list = c("df1", "df2", "iris_of"))
[1] "df1"     "df2"     "iris_of"
ls()
[1] "df1"     "df2"     "iris_of"
```

Deleting a Datastore

With the `ore.delete` function, you can delete objects from an Oracle R Enterprise datastore or you can delete the datastore itself. To delete a datastore, you specify the name of it. To delete one or more objects from the datastore, you specify the `list` argument. The `ore.delete` function returns the name of the deleted objects or datastore.

Example 2–18 demonstrates using `ore.delete` to delete an object from a datastore and then to delete the entire datastore. The example uses objects created in Example 2–14 on page 2-16.

Example 2–18 Using the ore.delete Function

```
# Delete the the df2 object from the ds1 datastore.
ore.delete("ds1", list = "df2")
# Delete the datastore named ds1.
ore.delete("ds1")
```

Listing for Example 2–18

```
R> # Delete the the df2 object from the ds1 datastore.
R> ore.delete("ds1", list = "df2")
[1] "df2"
R> # Delete the datastore named ds1.
R> ore.delete("ds1")
[1] "ds1"
```

When you delete a datastore, Oracle R Enterprise discards all temporary database objects that were referenced by R objects in the deleted datastore. If you have saved an R object in more than one datastore, then Oracle R Enterprise discards a temporary database object only when no object in a datastore references the temporary database object.

About Using a Datastore in Embedded R Execution

Saving objects in a datastore makes it very easy to pass arguments to, and reference R objects with, embedded R execution functions. You can save objects that you create in one R session in a single datastore in the database. You can pass the name of this datastore to an embedded R function as an argument for loading within that function. You can use a datastore to easily pass one object or multiple objects.

> **See Also:** Chapter 6, "Using Oracle R Enterprise Embedded R Execution" for information on using the R and the SQL interfaces to embedded R execution

3

Preparing and Exploring Data in the Database

This chapter describes how to use Oracle R Enterprise objects to prepare data for analysis and to perform exploratory analysis of the data. All of these functions make it easier for you to prepare very large enterprise database-resident data for modeling. The chapter contains the following topics:

- Preparing Data in the Database Using Oracle R Enterprise
- Exploring Data
- Using a Third-Party Package on the Client

Preparing Data in the Database Using Oracle R Enterprise

Using Oracle R Enterprise, you can prepare data for analysis in the database, as described in the following topics:

- About Preparing Data in the Database
- Selecting Data
- Indexing Data
- Combining Data
- Summarizing Data
- Transforming Data
- Sampling Data
- Partitioning Data
- Preparing Time Series Data

About Preparing Data in the Database

Oracle R Enterprise provides functions that enable you to use R to prepare database data for analysis. Using these functions, you can perform typical data preparation tasks on `ore.frame` and other Oracle R Enterprise objects. You can perform data preparation operations on large quantities of data in the database and then pull the results to your local R session for analysis using functions in packages available from the Comprehensive R Archive Network (CRAN).

You can do operations on data such as the following.

- Selecting
- Binning

- Sampling
- Sorting and Ordering
- Summarizing
- Transforming
- Performing data preparation operations on date and time data

Performing these operations is described in the other topics in this chapter.

Selecting Data

A typical step in preparing data for analysis is selecting or filtering values of interest from a larger data set. The examples in this section demonstrate selecting data from an `ore.frame` object by column, by row, and by value.

Example 3–1 selects columns from an `ore.frame` object. It creates a database table, with the corresponding proxy `ore.frame` object `IRIS_TABLE`, from the `iris` `data.frame` object. It displays the first six rows of `IRIS_TABLE`. The example selects two columns from `IRIS_TABLE` and creates the unordered `ore.frame` object `iris_projected` with them. It sets the `ore.warn.order` global option to `FALSE` to avoid displaying warning messages when using an unordered `ore.frame` object. It then displays the first six rows of `iris_projected`.

Example 3–1 Selecting Data by Column

```
# Create a database table from the iris data set.
ore.create(iris, table ="IRIS_TABLE")
head(IRIS_TABLE, 3)

# Select columns from the table.
iris_projected = IRIS_TABLE[, c("Petal.Length", "Species")]

# Turn off warnings about using an unordered ore.frame.
options("ore.warn.order" = FALSE)
head (iris_projected)
```

Listing for Example 3–1

```
# Create a database table from the iris data set.
R> ore.create(iris, table ="IRIS_TABLE")
R> head(IRIS_TABLE, 3)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa

# Select columns from the table.
R> iris_projected = IRIS_TABLE[, c("Petal.Length", "Species")]

# Turn off warnings about using an unordered ore.frame.
R> options("ore.warn.order" = FALSE)

R> head (iris_projected)
  Petal.Length Species
1          1.4  setosa
2          1.4  setosa
```

```
3          1.3  setosa
4          1.5  setosa
5          1.4  setosa
6          1.7  setosa
```

Example 3–2 selects rows from an ordered `ore.frame` object. The example first adds a column to the `iris` `data.frame` object for use in creating an ordered `ore.frame` object. It invokes the `ore.drop` function to delete the database table IRIS_TABLE, if it exists. It then creates a database table, with the corresponding proxy `ore.frame` object `IRIS_TABLE`, from the `iris` `data.frame`. The example invokes the `ore.exec` function to execute a SQL statement that makes the RID column the primary key of the database table. It then invokes the `ore.sync` function to synchronize the `IRIS_TABLE` `ore.frame` object with the table and displays the first three rows of the proxy `ore.frame` object.

Example 3–2 next selects 51 rows from `IRIS_TABLE` by row number and creates the ordered `ore.frame` object `iris_selrows` with them. It displays the first six rows of `iris_selrows`. It then selects 3 rows by row name and displays the result.

Example 3–2 Selecting Data by Row

```
# Add a column to the iris data set to use as row identifiers.
iris$RID <- as.integer(1:nrow(iris) + 100)
ore.drop(table = 'IRIS_TABLE')
ore.create(iris, table = 'IRIS_TABLE')
ore.exec("alter table IRIS_TABLE add constraint IRIS_TABLE
          primary key (\"RID\")")
ore.sync(table = "IRIS_TABLE")
head(IRIS_TABLE, 3)

# Select rows by row number.
iris_selrows <- IRIS_TABLE[50:100,]
head(iris_selrows)

# Select rows by row name.
IRIS_TABLE[c("101", "151", "201"),]
```

Listing for Example 3–2

```
R> # Add a column to the iris data set to use as row identifiers.
R> iris$RID <- as.integer(1:nrow(iris) + 100)
R> ore.drop(table = 'IRIS_TABLE')
R> ore.create(iris, table = 'IRIS_TABLE')
R> ore.exec("alter table IRIS_TABLE add constraint IRIS_TABLE
+           primary key (\"RID\")")
R> ore.sync(table = "IRIS_TABLE")
R> head(IRIS_TABLE, 3)
    Sepal.Length Sepal.Width Petal.Length Petal.Width Species RID
101          5.1         3.5          1.4         0.2  setosa 101
102          4.9         3.0          1.4         0.2  setosa 102
103          4.7         3.2          1.3         0.2  setosa 103
R> # Select rows by row number.
R> iris_selrows <- IRIS_TABLE[50:100,]
R> head(iris_selrows)
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species RID
150          5.0         3.3          1.4         0.2     setosa 150
151          7.0         3.2          4.7         1.4 versicolor 151
152          6.4         3.2          4.5         1.5 versicolor 152
153          6.9         3.1          4.9         1.5 versicolor 153
154          5.5         2.3          4.0         1.3 versicolor 154
155          6.5         2.8          4.6         1.5 versicolor 155
```

```
R> # Select rows by row name.
R> IRIS_TABLE[c("101", "151", "201"),]
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species RID
101          5.1         3.5          1.4         0.2     setosa 101
151          7.0         3.2          4.7         1.4 versicolor 151
201          6.3         3.3          6.0         2.5  virginica 201
```

You can select portions of a data set, as shown in Example 3–3. The example pushes the iris data set to the database and gets the ore.frame object iris_of. It filters the data to produce iris_of_filtered, which contains the values from the rows of iris_of that have a petal length of less than 1.5 and that are in the Sepal.Length and Species columns. The example also filters the data using conditions, so that iris_of_filtered contains the values from iris_of that are of the setosa or versicolor species and that have a petal width of less than 2.0.

Example 3–3   Selecting Data by Value

```
# Create a temporary database table from the iris data set and get an ore.frame.
iris_of <- ore.push(iris)
# Select Sepal.Length, Species from iris_of where Petal.Length < 1.5
iris_of_filtered <- iris_of[iris_of$Petal.Length < 1.5,
                            c("Sepal.Length", "Species")]
names(iris_of_filtered)
nrow(iris_of_filtered)
head(iris_of_filtered, 3)
# Alternate syntax filtering.
iris_of_filtered <- subset(iris_of, Petal.Length < 1.5)
nrow(iris_of_filtered)
head(iris_of_filtered, 3)
# Using the AND and OR conditions in filtering.
# Select all rows with either species = setosa OR species = versicolor
# and Petal.Width < 2.0
iris_of_filtered <- iris_of[(iris_of$Species == "setosa" |
                             iris_of$Species == "versicolor") &
                             iris_of$Petal.Width < 2.0,]
nrow(iris_of_filtered)
head(iris_of, 3)
```

Listing for Example 3–3

```
R> # Create a temporary database table from the iris data set and get an
ore.frame.
R> iris_of <- ore.push(iris)
R> # Select Sepal.Length, Species from iris_of where Petal.Length < 1.5
R> iris_of_filtered <- iris_of[iris_of$Petal.Length < 1.5,
+                              c("Sepal.Length", "Species")]
R> names(iris_of_filtered)
[1] "Sepal.Length" "Species"
R> nrow(iris_of_filtered)
[1] 24
R> head(iris_of_filtered, 3)
  Sepal.Length Species
1          5.1  setosa
2          4.9  setosa
3          4.7  setosa
R> # Alternate syntax filtering.
R> iris_of_filtered <- subset(iris_of, Petal.Length < 1.5)
R> nrow(iris_of_filtered)
[1] 24
R> head(iris_of_filtered, 3)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
R> # Using the AND and OR conditions in filtering.
R> # Select all rows with either species = setosa OR species = versicolor
R> # and Petal.Width < 2.0
R> iris_of_filtered <- iris_of[(iris_of$Species == "setosa" |
+                              iris_of$Species == "versicolor") &
+                              iris_of$Petal.Width < 2.0,]
R> nrow(iris_of_filtered)
[1] 100
R> head(iris_of, 3)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
```

See Also:

- "Indexing Data" on page 3-5
- The `sql_like.R` example script

Indexing Data

You can use integer or character vectors to index an ordered `ore.frame` object. You can use the indexing to perform sampling and partitioning, as described in "Sampling Data" on page 3-10 and "Partitioning Data" on page 3-15.

Oracle R Enterprise supports functionality similar to R indexing with these differences:

- Integer indexing is not supported for `ore.vector` objects.
- Negative integer indexes are not supported.
- Row order is not preserved.

Example 3–4 demonstrates character and integer indexing. The example uses the ordered `SPAM_PK` and unordered `SPAM_NOPK` `ore.frame` objects from Example 2–9 on page 2-9. The example shows that you can access rows by name and that you can also access a set of rows by supplying a vector of character row names. The example then shows that you can supply the actual integer value. In the example this results in a set of different rows because the USERID values start at 1001, as opposed to 1.

Example 3–4 Indexing an ore.frame Object

```
# Index to a specifically named row.
SPAM_PK["2060", 1:4]
# Index to a range of rows by row names.
SPAM_PK[as.character(2060:2064), 1:4]
# Index to a range of rows by integer index.
SPAM_PK[2060:2063, 1:4]
```

Listing for Example 3–4

```
R> # Index to a specifically named row.
R> SPAM_PK["2060", 1:4]
       TS USERID make address
2060 2060    380    0       0
R> # Index to a range of rows by row names.
R> SPAM_PK[as.character(2060:2064), 1:4]
       TS USERID make address
```

```
2060 2060    380    0       0
2061 2061    381    0       0
2062 2062    381    0       0
2063 2063    382    0       0
2064 2064    382    0       0
R> # Index to a range of rows by integer index.
R> SPAM_PK[2060:2063, 1:4]
       TS USERID make address
3060 3060    380 0.00    0.00
3061 3061    381 0.00    1.32
3062 3062    381 0.00    2.07
3063 3063    382 0.34    0.00
```

Combining Data

You can join data from the `ore.frame` objects that represent database tables by using the `merge` function, as shown in Example 3–5. The example creates two `data.frame` objects and merges them. It then invokes the `ore.create` function to create a database table for each `data.frame` object. The `ore.create` function automatically creates an `ore.frame` object as a proxy object for the table. The `ore.frame` object has the same name as the table. The example merges the `ore.frame` objects. Note that the order of the results of the two `merge` operations is not the same because the `ore.frame` objects are unordered.

Example 3–5   Joining Data from Two Tables

```
# Create data.frame objects.
df1 <- data.frame(x1=1:5, y1=letters[1:5])
df2 <- data.frame(x2=5:1, y2=letters[11:15])

# Combine the data.frame objects.
merge (df1, df2, by.x="x1", by.y="x2")

# Create database tables and ore.frame proxy objects to correspond to
# the in-memory R objects df1 and df2.
ore.create(df1, table="DF1_TABLE")
ore.create(df2, table="DF2_TABLE")

# Combine the ore.frame objects.
merge (DF1_TABLE, DF2_TABLE, by.x="x1", by.y="x2")
```

Listing for Example 3–5

```
R> # Create data.frame objects.
R> df1 <- data.frame(x1=1:5, y1=letters[1:5])
R> df2 <- data.frame(x2=5:1, y2=letters[11:15])

R> # Combine the data.frame objects.
R> merge (df1, df2, by.x="x1", by.y="x2")
  x1 y1 y2
1  1  a  o
2  2  b  n
3  3  c  m
4  4  d  l
5  5  e  k

R> # Create database tables and ore.frame proxy objects to correspond to
R> # the in-memory R objects df1 and df2.
R> ore.create(df1, table="DF1_TABLE")
R> ore.create(df2, table="DF2_TABLE")
```

```
R> # Combine the ore.frame objects.
R> merge (DF1_TABLE, DF2_TABLE, by.x="x1", by.y="x2")
  x1 y1 y2
1  5  e  k
2  4  d  l
3  3  c  m
4  2  b  n
5  1  a  o
Warning message:
ORE object has no unique key - using random order
```

Summarizing Data

You can summarize data by using the `aggregate` function, as shown in Example 3–6. The example pushes the `iris` data set to database memory as the `ore.frame` object `iris_of`. It aggregates the values of `iris_of` by the Species column using the `length` function. It then displays the first three rows of the result.

Example 3–6 Aggregating Data

```
# Create a temporary database table from the iris data set and get an ore.frame.
iris_of <- ore.push(iris)
aggdata <- aggregate(iris_of$Sepal.Length,
                     by = list(species = iris_of$Species),
                     FUN = length)
head(aggdata, 3)
```

Listing for Example 3–6

```
# Create a temporary database table from the iris data set and get an ore.frame.
R> iris_of <- ore.push(iris)
R> aggdata <- aggregate(iris_of$Sepal.Length,
+                       by = list(species = iris_of$Species),
+                       FUN = length)
R> head(aggdata, 3)
             species  x
setosa         setosa 50
versicolor versicolor 50
virginica   virginica 50
```

See Also: The `aggregate.R` example script

Transforming Data

In preparing data for analysis, a typical step is to transform data by reformatting it or deriving new columns and adding them to the data set. The examples in this topic demonstrate two ways of formatting data and deriving columns. Example 3–7 creates a function to format the data in a column and Example 3–8 does the same thing by using the `transform` function. Example 3–9 uses the `transform` function to add columns to the data set.

Example 3–7 Formatting Data

```
# Create a function for formatting data.
petalCategory_fmt <- function(x) {
    ifelse(x > 5, 'LONG',
    ifelse(x > 2, 'MEDIUM', 'SMALL'))
  }
```

```
# Create an ore.frame in database memory with the iris data set.
iris_of <- ore.push(iris)
# Select some rows from iris_of.
iris_of[c(10, 20, 60, 80, 110, 140),]
# Format the data in Petal.Length column.
iris_of$Petal.Length <- petalCategory_fmt(iris_of$Petal.Length)
# Select the same rows from iris_of.
```

Listing for Example 3–7

```
# Create a function for formatting data.
R> petalCategory_fmt <- function(x) {
+    ifelse(x > 5, 'LONG',
+    ifelse(x > 2, 'MEDIUM', 'SMALL'))
+ }
# Create an ore.frame in database memory with the iris data set.
R> iris_of <- ore.push(iris)
# Select some rows from iris_of.
R> iris_of[c(10, 20, 60, 80, 110, 140),]
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
10           4.9         3.1          1.5         0.1     setosa
20           5.1         3.8          1.5         0.3     setosa
60           5.2         2.7          3.9         1.4 versicolor
80           5.7         2.6          3.5         1.0 versicolor
110          7.2         3.6          6.1         2.5  virginica
140          6.9         3.1          5.4         2.1  virginica
# Format the data in Petal.Length column.
R> iris_of$Petal.Length <- petalCategory_fmt(iris_of$Petal.Length)
# Select the same rows from iris_of.
R> iris_of[c(10, 20, 60, 80, 110, 140),]
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
10           4.9         3.1        SMALL         0.1     setosa
20           5.1         3.8        SMALL         0.3     setosa
60           5.2         2.7       MEDIUM         1.4 versicolor
80           5.7         2.6       MEDIUM         1.0 versicolor
110          7.2         3.6         LONG         2.5  virginica
140          6.9         3.1         LONG         2.1  virginica
```

Example 3–8 does the same thing as Example 3–7 except that it uses the `transform` function to reformat the data in a column of the data set.

Example 3–8 Using the transform Function

```
# Create an ore.frame in database memory with the iris data set.
iris_of2 <- ore.push(iris)
# Select some rows from iris_of.
iris_of2[c(10, 20, 60, 80, 110, 140),]
iris_of2 <- transform(iris_of2,
                  Petal.Length = ifelse(Petal.Length > 5, 'LONG',
                                 ifelse(Petal.Length > 2, 'MEDIUM', 'SMALL')))
iris_of2[c(10, 20, 60, 80, 110, 140),]
```

Listing for Example 3–8

```
# Create an ore.frame in database memory with the iris data set.
R> iris_of2 <- ore.push(iris)
# Select some rows from iris_of.
R> iris_of2[c(10, 20, 60, 80, 110, 140),]
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
10           4.9         3.1          1.5         0.1     setosa
20           5.1         3.8          1.5         0.3     setosa
```

```
60           5.2         2.7          3.9         1.4 versicolor
80           5.7         2.6          3.5         1.0 versicolor
110          7.2         3.6          6.1         2.5  virginica
140          6.9         3.1          5.4         2.1  virginica
R> iris_of2 <- transform(iris_of2,
+                  Petal.Length = ifelse(Petal.Length > 5, 'LONG',
+                                 ifelse(Petal.Length > 2, 'MEDIUM', 'SMALL')))
R> iris_of2[c(10, 20, 60, 80, 110, 140),]
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
10           4.9         3.1        SMALL         0.1     setosa
20           5.1         3.8        SMALL         0.3     setosa
60           5.2         2.7       MEDIUM         1.4 versicolor
80           5.7         2.6       MEDIUM         1.0 versicolor
110          7.2         3.6         LONG         2.5  virginica
140          6.9         3.1         LONG         2.1  virginica
```

Example 3–9 uses the `transform` function to add a derived column to the data set and then to add additional columns to it.

Example 3–9 Adding Derived Columns

```
# Set the page width.
options(width = 80)
# Create an ore.frame in database memory with the iris data set.
iris_of <- ore.push(iris)
names(iris_of)
# Add one column derived from another
iris_of <- transform(iris_of, LOG_PL = log(Petal.Length))
names(iris_of)
head(iris_of, 3)
# Add more columns.
iris_of <- transform(iris_of,
                    SEPALBINS = ifelse(Sepal.Length < 6.0, "A", "B"),
                    PRODUCTCOLUMN = Petal.Length * Petal.Width,
                    CONSTANTCOLUMN = 10)
names(iris_of)
# Select some rows of iris_of.
iris_of[c(10, 20, 60, 80, 110, 140),]
```

Listing for Example 3–9

```
R> # Set the page width.
R> options(width = 80)
# Create an ore.frame in database memory with the iris data set.
R> iris_of <- ore.push(iris)
R> names(iris_of)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
R> # Add one column derived from another
R> iris_of <- transform(iris_of, LOG_PL = log(Petal.Length))
R> names(iris_of)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
[6] "LOG_PL"
R> head(iris_of, 3)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species    LOG_PL
1          5.1         3.5          1.4         0.2  setosa 0.3364722
2          4.9         3.0          1.4         0.2  setosa 0.3364722
3          4.7         3.2          1.3         0.2  setosa 0.2623643
R> # Add more columns.
R> iris_of <- transform(iris_of,
                    SEPALBINS = ifelse(Sepal.Length < 6.0, "A", "B"),
                    PRODUCTCOLUMN = Petal.Length * Petal.Width,
```

```
                    CONSTANTCOLUMN = 10)
R> names(iris_of)
[1] "Sepal.Length"   "Sepal.Width"    "Petal.Length"   "Petal.Width"
[5] "Species"        "LOG_PL"         "CONSTANTCOLUMN" "SEPALBINS"
[9] "PRODUCTCOLUMN"
R> # Select some rows of iris_of.
R> iris_of[c(10, 20, 60, 80, 110, 140),]
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species    LOG_PL
10           4.9         3.1          1.5         0.1     setosa 0.4054651
20           5.1         3.8          1.5         0.3     setosa 0.4054651
60           5.2         2.7          3.9         1.4 versicolor 1.3609766
80           5.7         2.6          3.5         1.0 versicolor 1.2527630
110          7.2         3.6          6.1         2.5  virginica 1.8082888
140          6.9         3.1          5.4         2.1  virginica 1.6863990
    CONSTANTCOLUMN SEPALBINS PRODUCTCOLUMN
10              10         A          0.15
20              10         A          0.45
60              10         A          5.46
80              10         A          3.50
110             10         B         15.25
140             10         B         11.34
```

See Also: The derived.R example script

Sampling Data

Sampling is an important capability for statistical analytics. Typically, you sample data to reduce its size and to perform meaningful work on it. In R you usually must load data into memory to sample it. However, if the data is too large, this isn't possible.

In Oracle R Enterprise, instead of pulling the data from the database and then sampling, you can sample directly in the database and then pull only those records that are part of the sample. By sampling in the database, you minimize data movement and you can work with larger data sets. Note that it is the ordering framework integer row indexing in the transparency layer that enables this capability.

Note: Sampling requires using ordered ore.frame objects as described in "Creating Ordered and Unordered ore.frame Objects" on page 2-7.

The examples in this section illustrate several sampling techniques. Similar examples are in the sampling.R example script.

Example 3–10 demonstrates a simple selection of rows at random. The example creates a small data.frame object and pushes it to the database to create an ore.frame object, MYDATA. Out of 20 rows, the example samples 5. It uses the R sample function to produce a random set of indices that it uses to get the sample from MYDATA. The sample, simpleRandomSample, is an ore.frame object.

Example 3–10 Simple Random Sampling

```
set.seed(1)
N <- 20
myData <- data.frame(a=1:N,b=letters[1:N])
MYDATA <- ore.push(myData)
head(MYDATA)
sampleSize <- 5
simpleRandomSample <- MYDATA[sample(nrow(MYDATA), sampleSize), , drop=FALSE]
```

```
class(simpleRandomSample)
simpleRandomSample
```

Listing for Example 3–10

```
R> set.seed(1)
R> N <- 20
R> myData <- data.frame(a=1:N,b=letters[1:N])
R> MYDATA <- ore.push(myData)
R> head(MYDATA)
  a b
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
6 6 f
R> sampleSize <- 5
R> simpleRandomSample <- MYDATA[sample(nrow(MYDATA), sampleSize), , drop=FALSE]
R> class(simpleRandomSample)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> simpleRandomSample
    a b
2   2 b
7   7 g
10 10 j
12 12 l
19 19 s
```

Example 3–11 demonstrates randomly partitioning data into training and testing sets. This splitting of the data is normally done in classification and regression to assess how well a model performs on new data. The example uses the `MYDATA` object created in Example 3–10.

Example 3–11 produces a sample set of indices to use as the test data set. It then creates the logical vector `group` that is `TRUE` if the index is in the sample and is `FALSE` otherwise. Next, it uses row indexing to produce the training set where the group is `FALSE` and the test set where the group is `TRUE`. Notice that the number of rows in the training set is 15 and the number of rows in the test set is 5, as specified in the invocation of the `sample` function.

Example 3–11 Split Data Sampling

```
set.seed(1)
sampleSize <- 5
ind <- sample(1:nrow(MYDATA), sampleSize)
group <- as.integer(1:nrow(MYDATA) %in% ind)
MYDATA.train <- MYDATA[group==FALSE,]
dim(MYDATA.train)
MYDATA.test <- MYDATA[group==TRUE,]
dim(MYDATA.test)
```

Listing for Example 3–11

```
R> set.seed(1)
R> sampleSize <- 5
R> ind <- sample(1:nrow(MYDATA), sampleSize)
R> group <- as.integer(1:nrow(MYDATA) %in% ind)
R> MYDATA.train <- MYDATA[group==FALSE,]
```

```
dim(MYDATA.train)
[1] 15  2
R> MYDATA.test <- MYDATA[group==TRUE,]
R> dim(MYDATA.test)
[1] 5 2
```

Example 3–12 demonstrates systematic sampling, in which rows are selected at regular intervals. The example uses the `seq` function to create a sequence of values that start at 2 and increase by increments of 3. The number of values in the sequence is equal to the number of rows in `MYDATA`. The `MYDATA` object is created in Example 3–10.

Example 3–12 Systematic Sampling

```
set.seed(1)
N <- 20
myData <- data.frame(a=1:20,b=letters[1:N])
MYDATA <- ore.push(myData)
head(MYDATA)
start <- 2
by <- 3
systematicSample <- MYDATA[seq(start, nrow(MYDATA), by = by), , drop = FALSE]
systematicSample
```

Listing for Example 3–12

```
R> set.seed(1)
R> N <- 20
R> myData <- data.frame(a=1:20,b=letters[1:N])
R> MYDATA <- ore.push(myData)
R> head(MYDATA)
  a b
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
6 6 f
R> start <- 2
R> by <- 3
R> systematicSample <- MYDATA[seq(start, nrow(MYDATA), by = by), , drop = FALSE]
systematicSample
    a b
2   2 b
5   5 e
8   8 h
11 11 k
14 14 n
17 17 q
20 20 t
```

Example 3–13 demonstrates stratified sampling, in which rows are selected within each group where the group is determined by the values of a particular column. The example creates a data set that has each row assigned to a group. The function `rnorm` produces random normal numbers. The argument 4 is the desired mean for the distribution. The example splits the data according to group and then samples proportionately from each partition. Finally, it row binds the list of subset `ore.frame` objects into a single `ore.frame` object and then displays the values of the result, `stratifiedSample`.

Example 3–13 Stratified Sampling

```
set.seed(1)
N <- 200
myData <- data.frame(a=1:N,b=round(rnorm(N),2),
                     group=round(rnorm(N,4),0))
MYDATA <- ore.push(myData)
head(MYDATA)
sampleSize <- 10
stratifiedSample <- do.call(rbind,
                            lapply(split(MYDATA, MYDATA$group),
                                   function(y) {
                                   ny <- nrow(y)
                                   y[sample(ny, sampleSize*ny/N), , drop = FALSE]
                              }))
stratifiedSample
```

Listing for Example 3–13

```
R> set.seed(1)
R> N <- 200
R> myData <- data.frame(a=1:N,b=round(rnorm(N),2),
+                       group=round(rnorm(N,4),0))
R> MYDATA <- ore.push(myData)
R> head(MYDATA)
  a     b group
1 1 -0.63     4
2 2  0.18     6
3 3 -0.84     6
4 4  1.60     4
5 5  0.33     2
6 6 -0.82     6
R> sampleSize <- 10
R> stratifiedSample <- do.call(rbind,
+                              lapply(split(MYDATA, MYDATA$group),
+                                function(y) {
+                                  ny <- nrow(y)
+                                  y[sample(ny, sampleSize*ny/N), , drop = FALSE]
+                             }))
R> stratifiedSample
          a     b group
173|173 173  0.46     3
9|9       9  0.58     4
53|53    53  0.34     4
139|139 139 -0.65     4
188|188 188 -0.77     4
78|78    78  0.00     5
137|137 137 -0.30     5
```

Example 3–14 demonstrates cluster sampling, in which entire groups are selected at random. The example splits the data according to group and then samples among the groups and row binds into a single `ore.frame` object. The resulting sample has data from two clusters, 6 and 7.

Example 3–14 Cluster Sampling

```
set.seed(1)
N <- 200
myData <- data.frame(a=1:N,b=round(runif(N),2),
                     group=round(rnorm(N,4),0))
MYDATA <- ore.push(myData)
```

```
head(MYDATA)
sampleSize <- 5
clusterSample <- do.call(rbind,
                         sample(split(MYDATA, MYDATA$group), 2))
unique(clusterSample$group)
```

Listing for Example 3–14

```
R> set.seed(1)
R> N <- 200
R> myData <- data.frame(a=1:N,b=round(runif(N),2),
+                       group=round(rnorm(N,4),0))
R> MYDATA <- ore.push(myData)
R> head(MYDATA)
  a    b group
1 1 0.27     3
2 2 0.37     4
3 3 0.57     3
4 4 0.91     4
5 5 0.20     3
6 6 0.90     6
R> sampleSize <- 5
R> clusterSample <- do.call(rbind,
+                          sample(split(MYDATA, MYDATA$group), 2))
R> unique(clusterSample$group)
[1] 6 7
```

Example 3–15 demonstrates quota sampling, in which a consecutive number of records are selected as the sample. The example uses the `head` function to select the sample. The `tail` function could also have been used.

Example 3–15 Quota Sampling

```
set.seed(1)
N <- 200
myData <- data.frame(a=1:N,b=round(runif(N),2))
MYDATA <- ore.push(myData)
sampleSize <- 10
quotaSample1 <- head(MYDATA, sampleSize)
quotaSample1
```

Listing for Example 3–15

```
R> set.seed(1)
R> N <- 200
R> myData <- data.frame(a=1:N,b=round(runif(N),2))
R> MYDATA <- ore.push(myData)
R> sampleSize <- 10
R> quotaSample1 <- head(MYDATA, sampleSize)
R> quotaSample1
    a    b
1   1 0.15
2   2 0.75
3   3 0.98
4   4 0.97
5   5 0.35
6   6 0.39
7   7 0.95
8   8 0.11
9   9 0.93
10 10 0.35
```

Partitioning Data

In analyzing large data sets, a typical operation is to randomly partitioning the data set into subsets. You can analyze the partitions by using Oracle R Enterprise embedded R execution, as shown in Example 3–16. The example creates a `data.frame` object with the symbol `myData` in the local R session and adds a column to it that contains a randomly generated set of values. It pushes the data set to database memory as the object `MYDATA`. The example invokes the embedded R execution function `ore.groupApply`, which partitions the data based on the partition column and then applies the `lm` function to each partition.

Example 3–16 Randomly Partitioning Data

```
N <- 200
k <- 5
myData <- data.frame(a=1:N,b=round(runif(N),2))
myData$partition <- sample(rep(1:k, each = N/k,
                               length.out = N), replace = TRUE)
MYDATA <- ore.push(myData)
head(MYDATA)
results <- ore.groupApply(MYDATA, MYDATA$partition,
                          function(y) {lm(b~a,y)}, parallel = TRUE)
length(results)
results[[1]]
```

Listing for Example 3–16

```
R> N <- 200
R> k <- 5
R> myData <- data.frame(a=1:N,b=round(runif(N),2))
R> myData$partition <- sample(rep(1:k, each = N/k,
+                               length.out = N), replace = TRUE)
R> MYDATA <- ore.push(myData)
R> head(MYDATA)
  a    b partition
1 1 0.89         2
2 2 0.31         4
3 3 0.39         5
4 4 0.66         3
5 5 0.01         1
6 6 0.12         4
R> results <- ore.groupApply(MYDATA, MYDATA$partition,
+                           function(y) {lm(b~a,y)}, parallel = TRUE)
R> length(results)
[1] 5
R> results[[1]]

Call:
lm(formula = b ~ a, data = y)

Coefficients:
(Intercept)            a
   0.388795     0.001015
```

See Also: Chapter 6, "Using Oracle R Enterprise Embedded R Execution"

Preparing Time Series Data

Oracle R Enterprise provides you with the ability to perform many data preparation operations on time series data, such as filtering, ordering, and transforming the data. Oracle R Enterprise maps R data types to SQL data types, as shown in Table 1–1 on page 1-8, which allows you to create Oracle R Enterprise objects and perform data preparation operations in database memory.

The following examples demonstrate some operations on time series data.

Example 3–17 illustrates some of the statistical aggregation functions. For a data set, the example first generates on the local client a sequence of five hundred dates spread evenly throughout 2001. It then introduces a random `difftime` and a vector of random normal values. The example then uses the `ore.push` function to create `MYDATA`, an in-database version of the data. The example invokes the `class` function to show that `MYDATA` is an `ore.frame` object and that the `datetime` column is of class `ore.datetime`. The example displays the first three rows of the generated data. It then uses the statistical aggregation operations of `min`, `max`, `range`, `median`, and `quantile` on the `datetime` column of `MYDATA`.

Example 3–17 Aggregating Date and Time Data

```
N <- 500
mydata <- data.frame(datetime =
                seq(as.POSIXct("2001/01/01"),
                    as.POSIXct("2001/12/31"),
                    length.out = N),
                difftime = as.difftime(runif(N),
                                 units = "mins"),
                x = rnorm(N))
MYDATA <- ore.push(mydata)
class(MYDATA)
class(MYDATA$datetime)
head(MYDATA,3)
# statistical aggregations
min(MYDATA$datetime)
max(MYDATA$datetime)
range(MYDATA$datetime)
quantile(MYDATA$datetime,
         probs = c(0, 0.05, 0.10))
```

Listing for Example 3–17

```
R> N <- 500
R> mydata <- data.frame(datetime =
+             seq(as.POSIXct("2001/01/01"),
+                 as.POSIXct("2001/12/31"),
+                 length.out = N),
+             difftime = as.difftime(runif(N),
+                              units = "mins"),
+             x = rnorm(N))
R> MYDATA <- ore.push(mydata)
R> class(MYDATA)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> class(MYDATA$datetime)
[1] "ore.datetime"
attr(,"package")
[1] "OREbase"
R> head(MYDATA,3)
```

```
             datetime       difftime           x
1 2001-01-01 00:00:00 16.436782 secs  0.68439244
2 2001-01-01 17:30:25  8.711562 secs  1.38481435
3 2001-01-02 11:00:50  1.366927 secs -0.00927078

R> # statistical aggregations
R> min(MYDATA$datetime)
[1] "2001-01-01 CST"
R> max(MYDATA$datetime)
[1] "2001-12-31 CST"
R> range(MYDATA$datetime)
[1] "2001-01-01 CST" "2001-12-31 CST"
R> quantile(MYDATA$datetime,
+           probs = c(0, 0.05, 0.10))
                       0%                        5%                       10%
"2001-01-01 00:00:00 CST" "2001-01-19 04:48:00 CST" "2001-02-06 09:36:00 CST"
```

Example 3–18 creates a one day shift by taking the `datetime` column of the `MYDATA` `ore.frame` object created in Example 3–17 and adding a `difftime` of one day. The result is `day1Shift`, which the example shows is of class `ore.datetime`. The example displays the first three elements of the `datetime` column of `MYDATA` and those of `day1Shift`. The first element of `day1Shift` is January 2, 2001.

Example 3–18 also computes lag differences using the overloaded `diff` function. The difference between the dates is all the same because the 500 dates in `MYDATA` are evenly distributed throughout 2001.

Example 3–18 Using Date and Time Arithmetic

```
day1Shift <- MYDATA$datetime + as.difftime(1, units = "days")
class(day1Shift)
head(MYDATA$datetime,3)
head(day1Shift,3)
lag1Diff <- diff(MYDATA$datetime)
class(lag1Diff)
head(lag1Diff,3)
```

Listing for Example 3–18

```
R> day1Shift <- MYDATA$datetime + as.difftime(1, units = "days")
R> class(day1Shift)
[1] "ore.datetime"
attr(,"package")
[1] "OREbase"
R> head(MYDATA$datetime,3)
[1] "2001-01-01 00:00:00 CST" "2001-01-01 17:30:25 CST" "2001-01-02 11:00:50 CST"
R> head(day1Shift,3)
[1] "2001-01-02 00:00:00 CST" "2001-01-02 17:30:25 CST" "2001-01-03 11:00:50 CST"
R> lag1Diff <- diff(MYDATA$datetime)
R> class(lag1Diff)
[1] "ore.difftime"
attr(,"package")
[1] "OREbase"
R> head(lag1Diff,3)
Time differences in secs
[1] 63025.25 63025.25 63025.25
```

Example 3–19 demonstrates date and time comparisons. The example uses the `datetime` column of the `MYDATA` `ore.frame` object created in Example 3–17. Example 3–19 selects the elements of `MYDATA` that have a date earlier than April 1, 2001.

The resulting `isQ1` is of class `ore.logical` and for the first three entries the result is `TRUE`. The example finds out how many dates matching `isQ1` are in March. It then sums the logical vector and displays the result, which is that 43 rows are in March. The example next filters rows based on dates that are the end of the year, after December 27. The result is `eoySubset`, which is an `ore.frame` object. The example displays the first three rows returned in `eoySubset`.

Example 3–19 Comparing Dates and Times

```
isQ1 <- MYDATA$datetime < as.Date("2001/04/01")
class(isQ1)
head(isQ1,3)
isMarch <- isQ1 & MYDATA$datetime > as.Date("2001/03/01")
class(isMarch)
head(isMarch,3)
sum(isMarch)
eoySubset <- MYDATA[MYDATA$datetime > as.Date("2001/12/27"), ]
class(eoySubset)
head(eoySubset,3)
```

Listing for Example 3–19

```
R> isQ1 <- MYDATA$datetime < as.Date("2001/04/01")
R> class(isQ1)
[1] "ore.logical"
attr(,"package")
[1] "OREbase"
R> head(isQ1,3)
[1] TRUE TRUE TRUE
R> isMarch <- isQ1 & MYDATA$datetime > as.Date("2001/03/01")
R> class(isMarch)
[1] "ore.logical"
attr(,"package")
[1] "OREbase"
R> head(isMarch,3)
[1] FALSE FALSE FALSE
R> sum(isMarch)
[1] 43
R> eoySubset <- MYDATA[MYDATA$datetime > as.Date("2001/12/27"), ]
R> class(eoySubset)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> head(eoySubset,3)
               datetime      difftime          x
495 2001-12-27 08:27:53 55.76474 secs -0.2740492
496 2001-12-28 01:58:18 15.42946 secs -1.4547270
497 2001-12-28 19:28:44 28.62195 secs  0.2929171
```

Oracle R Enterprise has accessor functions that you can use to extract various components from `datetime` objects, such as year, month, day of the month, hour, minute, and second. Example 3–20 demonstrates the use of these functions. The example uses the `datetime` column of the `MYDATA` `ore.frame` object created in Example 3–17.

Example 3–20 gets the year elements of the `datetime` column. The invocation of the `unique` function for `year` displays `2001` because it is the only year value in the column. However, for objects that have a range of values, as for example, `ore.mday`, the `range` function returns the day of the month. The result contains a vector with values that range from 1 through 31. Invoking the range function succinctly reports the range of values, as demonstrated for the other accessor functions.

Example 3–20 Using Date and Time Accessors

```
year <- ore.year(MYDATA$datetime)
unique(year)
month <- ore.month(MYDATA$datetime)
range(month)
dayOfMonth <- ore.mday(MYDATA$datetime)
range(dayOfMonth)
hour <- ore.hour(MYDATA$datetime)
range(hour)
minute <- ore.minute(MYDATA$datetime)
range(minute)
second <- ore.second(MYDATA$datetime)
range(second)
```

Listing for Example 3–20

```
R> year <- ore.year(MYDATA$datetime)
R> unique(year)
[1] 2001
R> month <- ore.month(MYDATA$datetime)
R> range(month)
[1]  1 12
R> dayOfMonth <- ore.mday(MYDATA$datetime)
R> range(dayOfMonth)
[1]  1 31
R> hour <- ore.hour(MYDATA$datetime)
R> range(hour)
[1]  0 23
R> minute <- ore.minute(MYDATA$datetime)
R> range(minute)
[1]  0 59
R> second <- ore.second(MYDATA$datetime)
R> range(second)
[1]  0.00000 59.87976
```

Example 3–21 uses the `as.ore` subclass objects to coerce an `ore.datetime` data type into other data types. The example uses the `datetime` column of the `MYDATA` `ore.frame` object created in Example 3–17. That column contains `ore.datetime` values. Example 3–21 first extracts the date from the `MYDATA$datetime` column. The resulting `dateOnly` object has `ore.date` values that contain only the year, month, and day, but not the time. The example then coerces the `ore.datetime` values into objects with `ore.character` and `ore.integer` values that represent the names of days, the number of the day of the year, and the quarter of the year.

Example 3–21 Coercing Date and Time Data Types

```
dateOnly <- as.ore.date(MYDATA$datetime)
class(dateOnly)
head(sort(unique(dateOnly)),3)
nameOfDay <- as.ore.character(MYDATA$datetime, format = "DAY")
class(nameOfDay)
sort(unique(nameOfDay))
dayOfYear <- as.integer(as.character(MYDATA$datetime, format = "DDD"))
class(dayOfYear)
range(dayOfYear)
quarter <- as.integer(as.character(MYDATA$datetime, format = "Q"))
```

```
class(quarter)
sort(unique(quarter))
```

Listing for Example 3–21

```
R> dateOnly <- as.ore.date(MYDATA$datetime)
R> class(dateOnly)
[1] "ore.date"
attr(,"package")
[1] "OREbase"
R> head(sort(unique(dateOnly)),3)
[1] "2001-01-01" "2001-01-02" "2001-01-03"
R> nameOfDay <- as.ore.character(MYDATA$datetime, format = "DAY")
R> class(nameOfDay)
[1] "ore.character"
attr(,"package")
[1] "OREbase"
R> sort(unique(nameOfDay))
[1] "FRIDAY " "MONDAY " "SATURDAY " "SUNDAY " "THURSDAY " "TUESDAY " "WEDNESDAY"
R> dayOfYear <- as.integer(as.character(MYDATA$datetime, format = "DDD"))
R> class(dayOfYear)
[1] "ore.integer"
attr(,"package")
[1] "OREbase"
R> range(dayOfYear)
[1]   1 365
R> quarter <- as.integer(as.character(MYDATA$datetime, format = "Q"))
R> class(quarter)
[1] "ore.integer"
attr(,"package")
[1] "OREbase"
R> sort(unique(quarter))
[1] 1 2 3 4
```

Example 3–22 uses the window functions `ore.rollmean` and `ore.rollsd` to compute the rolling mean and the rolling standard deviation. The example uses the `MYDATA` `ore.frame` object created in Example 3–17. The example ensures that `MYDATA` is an ordered `ore.frame` by assigning the values of the `datetime` column as the row names of `MYDATA`. The example computes the rolling mean and the rolling standard deviation over five periods. Next, to use the R time series functionality in the `stats` package, the example pulls data to the client. To limit the data pulled to the client, it uses the vector `is.March` from Example 3–19 to select only the data points in March. The example creates a time series object using the `ts` function, builds the Arima model, and predicts three points out.

Example 3–22 Using a Window Function

```
row.names(MYDATA) <- MYDATA$datetime
MYDATA$rollmean5 <- ore.rollmean(MYDATA$x, k = 5)
MYDATA$rollsd5 <- ore.rollsd (MYDATA$x, k = 5)
head(MYDATA)
marchData <- ore.pull(MYDATA[isMarch,])
tseries.x <- ts(marchData$x)
arima110.x <- arima(tseries.x, c(1,1,0))
predict(arima110.x, 3)
tseries.rm5 <- ts(marchData$rollmean5)
arima110.rm5 <- arima(tseries.rm5, c(1,1,0))
predict(arima110.rm5, 3)
```

Listing for Example 3–22

```
R> row.names(MYDATA) <- MYDATA$datetime
R> MYDATA$rollmean5 <- ore.rollmean(MYDATA$x, k = 5)
R> MYDATA$rollsd5 <- ore.rollsd (MYDATA$x, k = 5)
R> head(MYDATA)
                               datetime       difftime
2001-01-01 00:00:00 2001-01-01 00:00:00 39.998460 secs
                                                 x   rollmean5   rollsd5
                                        -0.3450421 -0.46650761 0.8057575
                               datetime       difftime
2001-01-01 17:30:25 2001-01-01 17:30:25  37.75568 secs
                                                 x   rollmean5   rollsd5
                                        -1.3261019  0.02877517 1.1891384
                               datetime       difftime
2001-01-02 11:00:50 2001-01-02 11:00:50  18.44243 secs
                                                 x   rollmean5   rollsd5
                                         0.2716211 -0.13224503 1.0909515
                               datetime       difftime
2001-01-03 04:31:15 2001-01-03 04:31:15 38.594384 secs
                                                 x   rollmean5   rollsd5
                                         1.5146235  0.36307913 1.4674456
                               datetime       difftime
2001-01-03 22:01:41 2001-01-03 22:01:41  2.520976 secs
                                                 x   rollmean5   rollsd5
                                        -0.7763258  0.80073340 1.1237925
                               datetime       difftime
2001-01-04 15:32:06 2001-01-04 15:32:06 56.333281 secs
                                                 x   rollmean5   rollsd5
                                         2.1315787  0.90287282 1.0862614
R> marchData <- ore.pull(MYDATA[isMarch,])
R> tseries.x <- ts(marchData$x)
R> arima110.x <- arima(tseries.x, c(1,1,0))
R> predict(arima110.x, 3)
$pred
Time Series:
Start = 44
End = 46
Frequency = 1
[1] 1.4556614 0.6156379 1.1387587

$se
Time Series:
Start = 44
End = 46
Frequency = 1
[1] 1.408117 1.504988 1.850830

R> tseries.rm5 <- ts(marchData$rollmean5)
R> arima110.rm5 <- arima(tseries.rm5, c(1,1,0))
R> predict(arima110.rm5, 3)
$pred
Time Series:
Start = 44
End = 46
Frequency = 1
[1] 0.3240135 0.3240966 0.3240922

$se
Time Series:
Start = 44
```

```
End = 46
Frequency = 1
[1] 0.3254551 0.4482886 0.5445763
```

Exploring Data

Oracle R Enterprise provides functions that enable you to perform exploratory data analysis. With these functions, you can perform common statistical operations.

The functions and their uses are described in the following topics:

- About the Exploratory Data Analysis Functions
- About the NARROW Data Set for Examples
- Correlating Data
- Cross-Tabulating Data
- Analyzing the Frequency of Cross-Tabulations
- Building Exponential Smoothing Models on Time Series Data
- Ranking Data
- Sorting Data
- Summarizing Data
- Analyzing Distribution of Numeric Variables

See Also: Chapter 4, "Building Models in Oracle R Enterprise"

About the Exploratory Data Analysis Functions

The Oracle R Enterprise functions for exploratory data analysis are in the `OREeda` package. Table 3–1 lists the functions in that package.

Table 3–1 Functions in the OREeda Package

Function	Description
`ore.corr`	Performs correlation analysis across numeric columns in an `ore.frame` object.
`ore.crosstab`	Expands on the `xtabs` function by supporting multiple columns with optional aggregations, weighting, and ordering options. Building a cross-tabulation is a pre-requisite to using the `ore.freq` function.
`ore.esm`	Builds exponential smoothing models on data in an ordered `ore.vector` object.
`ore.freq`	Operates on output from the `ore.crosstab` function and automatically determines techniques that are relevant for the table.
`ore.rank`	Enables the investigation of the distribution of values along numeric columns in an `ore.frame` object.
`ore.sort`	Provides flexible sorting for `ore.frame` objects.
`ore.summary`	Provides descriptive statistics for `ore.frame` objects within flexible row aggregations.
`ore.univariate`	Provides distribution analysis of numeric columns in an `ore.frame` object of. Reports all statistics from the `ore.summary` function plus signed-rank test and extreme values.

About the NARROW Data Set for Examples

Many of the examples of the exploratory data analysis functions use the `NARROW` data set. `NARROW` is an `ore.frame` that has 9 columns and 1500 rows, as shown in Example 3–23. Some of the columns are numeric, others are not.

Example 3–23 The NARROW Data Set

```
R> class(NARROW)
R> dim(NARROW)
R> names(NARROW)
```

Listing for Example 3–23

```
R> class(NARROW)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> dim(NARROW)
[1] 1500    9
R> names(NARROW)
[1] "ID"            "GENDER"        "AGE"           "MARITAL_STATUS"
[5] "COUNTRY"       "EDUCATION"     "OCCUPATION"    "YRS_RESIDENCE"
[9] "CLASS"
```

Correlating Data

You can use the `ore.corr` function to perform correlation analysis. With the `ore.corr` function, you can do the following:

- Perform Pearson, Spearman or Kendall correlation analysis across numeric columns in an `ore.frame` object.
- Perform partial correlations by specifying a control column.
- Aggregate some data prior to the correlations.
- Post-process results and integrate them into an R code flow.

  You can make the output of the `ore.corr` function conform to the output of the R `cor` function; doing so allows you to use any R function to post-process the output or to use the output as the input to a graphics function.

For details about the function arguments, invoke `help(ore.corr)`.

The following examples demonstrate these operations. Most of the examples use the `NARROW` data set; for more information, see "About the Exploratory Data Analysis Functions" on page 3-22.

Example 3–24 demonstrates how to specify the different types of correlation statistics.

Example 3–24 Performing Basic Correlation Calculations

```
# Before performing correlations, project out all non-numeric values
# by specifying only the columns that have numeric values.
names(NARROW)
NARROW_NUMS <- NARROW[,c(3,8,9)]
names(NARROW_NUMS)
# Calculate the correlation using the default correlation statistic, Pearson.
x <- ore.corr(NARROW_NUMS,var='AGE,YRS_RESIDENCE,CLASS')
head(x, 3)
# Calculate using Spearman.
x <- ore.corr(NARROW_NUMS,var='AGE,YRS_RESIDENCE,CLASS', stats='spearman')
```

```
head(x, 3)
# Calculate using Kendall
x <- ore.corr(NARROW_NUMS,var='AGE,YRS_RESIDENCE,CLASS', stats='kendall')
head(x, 3)
```

Listing for Example 3–24

```
R> # Before performing correlations, project out all non-numeric values
R> # by specifying only the columns that have numeric values.
R> names(NARROW)
 [1] "ID" "GENDER" "AGE" "MARITAL_STATUS" "COUNTRY" "EDUCATION" "OCCUPATION"
 [8] "YRS_RESIDENCE" "CLASS" "AGEBINS"
R> NARROW_NUMS <- NARROW[,c(3,8,9)]
R> names(NARROW_NUMS)
[1] "AGE" "YRS_RESIDENCE" "CLASS"

R> # Calculate the correlation using the default correlation statistic, Pearson.
R> x <- ore.corr(NARROW_NUMS,var='AGE,YRS_RESIDENCE,CLASS')
R> head(x, 3)
            ROW           COL PEARSON_T PEARSON_P PEARSON_DF
1           AGE         CLASS 0.2200960     1e-15       1298
2           AGE YRS_RESIDENCE 0.6568534     0e+00       1098
3 YRS_RESIDENCE         CLASS 0.3561869     0e+00       1298
R> # Calculate using Spearman.
R> x <- ore.corr(NARROW_NUMS,var='AGE,YRS_RESIDENCE,CLASS', stats='spearman')
R> head(x, 3)
            ROW           COL SPEARMAN_T SPEARMAN_P SPEARMAN_DF
1           AGE         CLASS  0.2601221      1e-15        1298
2           AGE YRS_RESIDENCE  0.7462684      0e+00        1098
3 YRS_RESIDENCE         CLASS  0.3835252      0e+00        1298
R> # Calculate using Kendall
R> x <- ore.corr(NARROW_NUMS,var='AGE,YRS_RESIDENCE,CLASS', stats='kendall')
R> head(x, 3)
            ROW           COL KENDALL_T    KENDALL_P KENDALL_DF
1           AGE         CLASS 0.2147107 4.285594e-31       <NA>
2           AGE YRS_RESIDENCE 0.6332196 0.000000e+00       <NA>
3 YRS_RESIDENCE         CLASS 0.3362078 1.094478e-73       <NA>
```

Example 3–25 pushes the `iris` data set to a temporary table in the database, which has the proxy `ore.frame` object `iris_of`. It creates correlation matrices grouped by species.

Example 3–25   Creating Correlation Matrices

```
iris_of <- ore.push(iris)
x <- ore.corr(iris_of, var = "Sepal.Length, Sepal.Width, Petal.Length",
              partial = "Petal.Width", group.by = "Species")
 class(x)
 head(x)
```

Listing for Example 3–25

```
R> iris_of <- ore.push(iris)
R> x <- ore.corr(iris_of, var = "Sepal.Length, Sepal.Width, Petal.Length",
+                 partial = "Petal.Width", group.by = "Species")
R> class(x)
[1] "list"
R> head(x)
$setosa
           ROW          COL PART_PEARSON_T PART_PEARSON_P PART_PEARSON_DF
1 Sepal.Length Petal.Length      0.1930601   9.191136e-02              47
2 Sepal.Length  Sepal.Width      0.7255823   1.840300e-09              47
```

```
3  Sepal.Width Petal.Length      0.1095503   2.268336e-01              47

$versicolor
           ROW          COL PART_PEARSON_T PART_PEARSON_P PART_PEARSON_DF
1 Sepal.Length Petal.Length     0.62696041   7.180100e-07              47
2 Sepal.Length  Sepal.Width     0.26039166   3.538109e-02              47
3  Sepal.Width Petal.Length     0.08269662   2.860704e-01              47

$virginica
           ROW          COL PART_PEARSON_T PART_PEARSON_P PART_PEARSON_DF
1 Sepal.Length Petal.Length      0.8515725   4.000000e-15              47
2 Sepal.Length  Sepal.Width      0.3782728   3.681795e-03              47
3  Sepal.Width Petal.Length      0.2854459   2.339940e-02              47
```

See Also: The cor.R example script

Cross-Tabulating Data

Cross-tabulation is a statistical technique that finds an interdependent relationship between two tables of values. The ore.crosstab function enables cross-column analysis of an ore.frame. This function is a sophisticated variant of the R table function.

You must use ore.crosstab function before performing frequency analysis using ore.freq.

If the result of the ore.crosstab function invocation is a single cross-tabulation, the function returns an ore.frame object. If the result is multiple cross-tabulations, then the function returns a list of ore.frame objects.

For details about function arguments, invoke help(ore.crosstab).

The examples of ore.corr use the NARROW data set; for more information, see "About the NARROW Data Set for Examples" on page 3-23.

The most basic use case is to create a single-column frequency table, as shown in Example 3–26. The example filters the NARROW ore.frame, grouping by GENDER.

Example 3–26 Creating a Single Column Frequency Table

```
ct <- ore.crosstab(~AGE, data=NARROW)
head(ct)
```

Listing for Example 3–26

```
R> ct <- ore.crosstab(~AGE, data=NARROW)
R> head(ct)
   AGE ORE$FREQ ORE$STRATA ORE$GROUP
17  17       14          1         1
18  18       16          1         1
19  19       30          1         1
20  20       23          1         1
21  21       22          1         1
22  22       39          1         1
```

Example 3–27 analyses AGE by GENDER and AGE by CLASS.

Example 3–27 Analyzing Two Columns

```
ct <- ore.crosstab(AGE~GENDER+CLASS, data=NARROW)
head(ct)
```

Listing for Example 3–27

```
R> ct <- ore.crosstab(AGE~GENDER+CLASS, data=NARROW)
R> head(ct)
$`AGE~GENDER`
     AGE GENDER ORE$FREQ ORE$STRATA ORE$GROUP
17|F  17      F        5          1         1
17|M  17      M        9          1         1
18|F  18      F        6          1         1
18|M  18      M        7          1         1
19|F  19      F       15          1         1
19|M  19      M       13          1         1
# The remaining output is not shown.
```

To weight rows, include a count based on another column as shown in Example 3–28. The example weights values in AGE and GENDER using values in YRS_RESIDENCE.

Example 3–28 Weighting Rows

```
ct <- ore.crosstab(AGE~GENDER*YRS_RESIDENCE, data=NARROW)
head(ct)
```

There are several possibilities for ordering rows in a cross-tabulated table, such as the following:

- Default or NAME orders by the columns being analyzed
- FREQ orders by frequency counts
- -NAME or -FREQ does reverse ordering
- INTERNAL bypasses ordering

Listing for Example 3–28

```
R> ct <- ore.crosstab(AGE~GENDER*YRS_RESIDENCE, data=NARROW)
R> head(ct)
     AGE GENDER ORE$FREQ ORE$STRATA ORE$GROUP
17|F  17      F        1          1         1
17|M  17      M        8          1         1
18|F  18      F        4          1         1
18|M  18      M       10          1         1
19|F  19      F       15          1         1
19|M  19      M       17          1         1
```

Example 3–29 orders by frequency count and then by reverse order by frequency count.

Example 3–29 Ordering Cross-Tabulated Data

```
ct <- ore.crosstab(AGE~GENDER|FREQ, data=NARROW)
head(ct)
ct <- ore.crosstab(AGE~GENDER|-FREQ, data=NARROW)
head(ct)
```

Listing for Example 3–29

```
R> ct <- ore.crosstab(AGE~GENDER|FREQ, data=NARROW)
R> head(ct)
     AGE GENDER ORE$FREQ ORE$STRATA ORE$GROUP
66|F  66      F        1          1         1
70|F  70      F        1          1         1
73|M  73      M        1          1         1
74|M  74      M        1          1         1
```

```
76|F  76      F        1           1          1
77|F  77      F        1           1          1

R> ct <- ore.crosstab(AGE~GENDER|-FREQ, data=NARROW)
R> head(ct)
     AGE GENDER ORE$FREQ ORE$STRATA ORE$GROUP
27|M  27      M       33           1          1
35|M  35      M       28           1          1
41|M  41      M       27           1          1
34|M  34      M       26           1          1
37|M  37      M       26           1          1
28|M  28      M       25           1          1
```

Example 3–30 demonstrates analyzing three or more columns. The result is similar to what the SQL `GROUPING SETS` clause accomplishes.

Example 3–30 Analyzing Three or More Columns

```
ct <- ore.crosstab(AGE+COUNTRY~GENDER, NARROW)
head(ct)
```

Listing for Example 3–30

```
R> ct <- ore.crosstab(AGE+COUNTRY~GENDER, NARROW)
R> head(ct)
$`AGE~GENDER`
     AGE GENDER ORE$FREQ ORE$STRATA ORE$GROUP
17|F  17      F        5           1          1
17|M  17      M        9           1          1
18|F  18      F        6           1          1
18|M  18      M        7           1          1
19|F  19      F       15           1          1
19|M  19      M       13           1          1
# The rest of the output is not shown.
$`COUNTRY~GENDER`
                        COUNTRY GENDER ORE$FREQ ORE$STRATA ORE$GROUP
Argentina|F            Argentina      F       14          1         1
Argentina|M            Argentina      M       28          1         1
Australia|M            Australia      M        1          1         1
# The rest of the output is not shown.
```

You can specify a range of columns instead of having to type all the column names, as illustrated Example 3–31.

Example 3–31 Specifying a Range of Columns

```
names(NARROW)
# Because AGE, MARITAL_STATUS and COUNTRY are successive columns,
# you can simply do the following:
ct <- ore.crosstab(AGE-COUNTRY~GENDER, NARROW)
# An equivalent invocation is the following:
ct <- ore.crosstab(AGE+MARITAL_STATUS+COUNTRY~GENDER, NARROW)
```

Listing for Example 3–31

```
R> names(NARROW)
R> names(NARROW)
[1] "ID"             "GENDER"         "AGE"            "MARITAL_STATUS"
[5] "COUNTRY"        "EDUCATION"      "OCCUPATION"     "YRS_RESIDENCE"
[9] "CLASS"
R> # Because AGE, MARITAL_STATUS and COUNTRY are successive columns,
```

```
R> # you can simply do the following:
R> ct <- ore.crosstab(AGE-COUNTRY~GENDER, NARROW)
R> # An equivalent invocation is the following:
R> ct <- ore.crosstab(AGE+MARITAL_STATUS+COUNTRY~GENDER, NARROW)
```

Example 3–32 produces one cross-tabulation table (AGE, GENDER) for *each* unique value of another column COUNTRY:

Example 3–32   Producing One Cross-Tabulation Table for Each Value of Another Column

```
ct <- ore.crosstab(~AGE/COUNTRY, data=NARROW)
head(ct)
```

Listing for Example 3–32

```
R> ct <- ore.crosstab(~AGE/COUNTRY, data=NARROW)
R> head(ct)
                             AGE ORE$FREQ ORE$STRATA ORE$GROUP
Argentina|17                  17        1          1         1
Brazil|17                     17        1          1         3
United States of America|17  17       12          1        19
United States of America|18  18       16          1        19
United States of America|19  19       30          1        19
United States of America|20  20       23          1        19
```

You can extend this to more than one column, as shown in Example 3–33. The example produces one (AGE, EDUCATION) table for each unique combination of (COUNTRY, GENDER).

Example 3–33   Producing One Cross-Tabulation Table for Each Set of Value of Two Columns

```
ct <- ore.crosstab(AGE~EDUCATION/COUNTRY+GENDER, data=NARROW)
head(ct)
```

Listing for Example 3–33

```
R> ct <- ore.crosstab(AGE~EDUCATION/COUNTRY+GENDER, data=NARROW)
R> head(ct)
                                    AGE EDUCATION ORE$FREQ ORE$STRATA ORE$GROUP
United States of America|F|17|10th  17      10th        3          1        33
United States of America|M|17|10th  17      10th        5          1        34
United States of America|M|17|11th  17      11th        1          1        34
Argentina|M|17|HS-grad              17   HS-grad        1          1         2
United States of America|M|18|10th  18      10th        1          1        34
United States of America|F|18|11th  18      11th        2          1        33
```

All of the above cross-tabulation tables can be augmented with stratification, as shown in Example 3–34.

Example 3–34   Augmenting Cross-Tabulation with Stratification

```
ct <- ore.crosstab(AGE~GENDER^CLASS, data=NARROW)
head(ct)
R> head(ct)
# The previous function invocation is the same as the following:
ct <- ore.crosstab(AGE~GENDER, NARROW, strata="CLASS")
```

Listing for Example 3–34

```
R> ct <- ore.crosstab(AGE~GENDER^CLASS, data=NARROW)
R> head(ct)
```

```
R> head(ct)
       AGE GENDER ORE$FREQ ORE$STRATA ORE$GROUP
0|17|F  17      F        5          1         1
0|17|M  17      M        9          1         1
0|18|F  18      F        6          1         1
0|18|M  18      M        7          1         1
0|19|F  19      F       15          1         1
0|19|M  19      M       13          1         1
# The previous function invocation is the same as the following:
ct <- ore.crosstab(AGE~GENDER, NARROW, strata="CLASS")
```

Example 3–35 does a custom binning by AGE and then calculates the cross-tabulation for GENDER and the bins.

Example 3–35 Binning Followed by Cross-Tabulation

```
NARROW$AGEBINS <- ifelse(NARROW$AGE<20, 1, ifelse(NARROW$AGE<30,2,
                  ifelse(NARROW$AGE<40,3,4)))
ore.crosstab(GENDER~AGEBINS, NARROW)
```

Listing for Example 3–35

```
R> NARROW$AGEBINS <- ifelse(NARROW$AGE<20, 1, ifelse(NARROW$AGE<30,2,
+                     ifelse(NARROW$AGE<40,3,4)))
R> ore.crosstab(GENDER~AGEBINS, NARROW)
    GENDER AGEBINS ORE$FREQ ORE$STRATA ORE$GROUP
F|1      F       1       26          1         1
F|2      F       2      108          1         1
F|3      F       3       86          1         1
F|4      F       4      164          1         1
M|1      M       1       29          1         1
M|2      M       2      177          1         1
M|3      M       3      230          1         1
M|4      M       4      381          1         1
```

Analyzing the Frequency of Cross-Tabulations

The `ore.freq` function analyses the output of the `ore.crosstab` function and automatically determines the techniques that are relevant to an `ore.crosstab` result. The techniques depend on the kind of cross-tabulation tables, which are the following:

- 2-way cross-tabulation tables
  - Various statistics that describe relationships between columns in the cross-tabulation
  - Chi-square tests, Cochran-Mantel-Haenzsel statistics, measures of association, strength of association, risk differences, odds ratio and relative risk for 2x2 tables, tests for trend
- N-way cross-tabulation tables
  - N 2-way cross-tabulation tables
  - Statistics across and within strata

The `ore.freq` function uses Oracle Database SQL functions when available.

The `ore.freq` function returns an `ore.frame` in all cases.

Before you use `ore.freq`, you must calculate crosstabs, as shown in Example 3–36.

For details about the function arguments, invoke `ore.freq`.

Example 3–36 pushes the iris data set to the database and gets the ore.frame object iris_of. The example gets a crosstab and invoke the ore.freq function on it.

Example 3–36 Using the ore.freq Function

```
IRIS <- ore.push(iris)
ct <- ore.crosstab(Species ~ Petal.Length + Sepal.Length, data = IRIS)
ore.freq(ct)
```

Listing for Example 3–36

```
R> IRIS <- ore.push(iris)
R> ct <- ore.crosstab(Species ~ Petal.Length + Sepal.Length, data = IRIS)
R> ore.freq(ct)
$`Species~Petal.Length`
  METHOD     FREQ DF       PVALUE              DESCR GROUP
1 PCHISQ 181.4667 84 3.921603e-09 Pearson Chi-Square     1

$`Species~Sepal.Length`
  METHOD  FREQ DF      PVALUE              DESCR GROUP
1 PCHISQ 102.6 68 0.004270601 Pearson Chi-Square     1
```

Building Exponential Smoothing Models on Time Series Data

The ore.esm function builds a simple or a double exponential smoothing model for in-database time series observations in an ordered ore.vector object. The function operates on time series data, whose observations are evenly spaced by a fixed interval, or transactional data, whose observations are not equally spaced. The function can aggregate the transactional data by a specified time interval, as well as handle missing values using a specified method, before entering the modeling phase.

The ore.esm function processes the data in one or more R engines running on the database server. The function returns an object of class ore.esm.

You can use the predict method to predict the time series of the exponential smoothing model built by ore.esm. If you have loaded the forecast package, then you can use the forecast method on the ore.esm object. You can use the fitted method to generate the fitted values of the training time series data set.

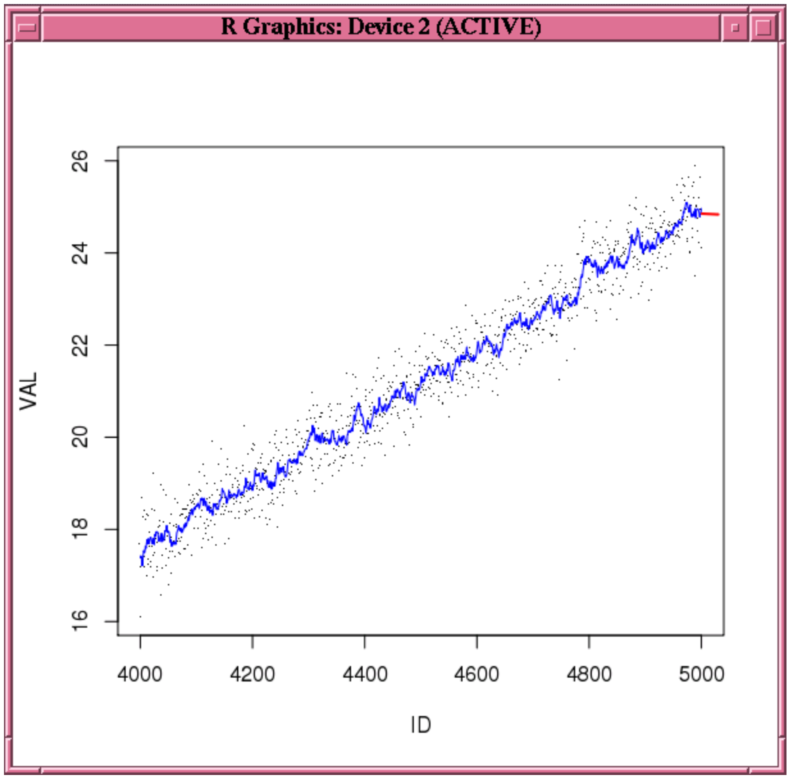
For information about the arguments of the ore.esm function, invoke help(ore.esm).

Example 3–37 builds a double exponential smoothing model on a synthetic time series data set. The predict and fitted functions are invoked to generate the predictions and the fitted values, respectively. Figure 3–1 shows the observations, fitted values, and the predictions.

Example 3–37 Building a Double Exponential Smoothing Model

```
N <- 5000
ts0 <- ore.push(data.frame(ID=1:N,
                           VAL=seq(1,5,length.out=N)^2+rnorm(N,sd=0.5)))
rownames(ts0) <- ts0$ID
x <- ts0$VAL
esm.mod <- ore.esm(x, model = "double")
esm.predict <- predict(esm.mod, 30)
esm.fitted <- fitted(esm.mod, start=4000, end=5000)
plot(ts0[4000:5000,], pch='.')
lines(ts0[4000:5000, 1], esm.fitted, col="blue")
lines(esm.predict, col="red", lwd=2)
```

Figure 3–1 Fitted and Predicted Values Based on the esm.mod Model



Example 3–38 builds a simple smoothing model based on a transactional data set. As preprocessing, it aggregates the values to the day level by taking averages, and fills missing values by setting them to the previous aggregated value. The model is then built on the aggregated daily time series. The function `predict` is invoked to generate predicted values on the daily basis.

Example 3–38 Building a Time Series Model with Transactional Data

```
ts01 <- data.frame(ID=seq(as.POSIXct("2008/6/13"), as.POSIXct("2011/6/16"),
                   length.out=4000), VAL=rnorm(4000, 10))
ts02 <- data.frame(ID=seq(as.POSIXct("2011/7/19"), as.POSIXct("2012/11/20"),
                   length.out=1500), VAL=rnorm(1500, 10))
ts03 <- data.frame(ID=seq(as.POSIXct("2012/12/09"), as.POSIXct("2013/9/25"),
                   length.out=1000), VAL=rnorm(1000, 10))
ts1 = ore.push(rbind(ts01, ts02, ts03))
rownames(ts1) <- ts1$ID
x <- ts1$VAL
esm.mod <- ore.esm(x, "DAY", accumulate = "AVG", model="simple",
                   setmissing="PREV")
esm.predict <- predict(esm.mod)
esm.predict
```

Listing for Example 3–38

```
R> ts01 <- data.frame(ID=seq(as.POSIXct("2008/6/13"), as.POSIXct("2011/6/16"),
+                       length.out=4000), VAL=rnorm(4000, 10))
R> ts02 <- data.frame(ID=seq(as.POSIXct("2011/7/19"), as.POSIXct("2012/11/20"),
+                      length.out=1500), VAL=rnorm(1500, 10))
R> ts03 <- data.frame(ID=seq(as.POSIXct("2012/12/09"), as.POSIXct("2013/9/25"),
+                      length.out=1000), VAL=rnorm(1000, 10))
R> ts1 = ore.push(rbind(ts01, ts02, ts03))
```

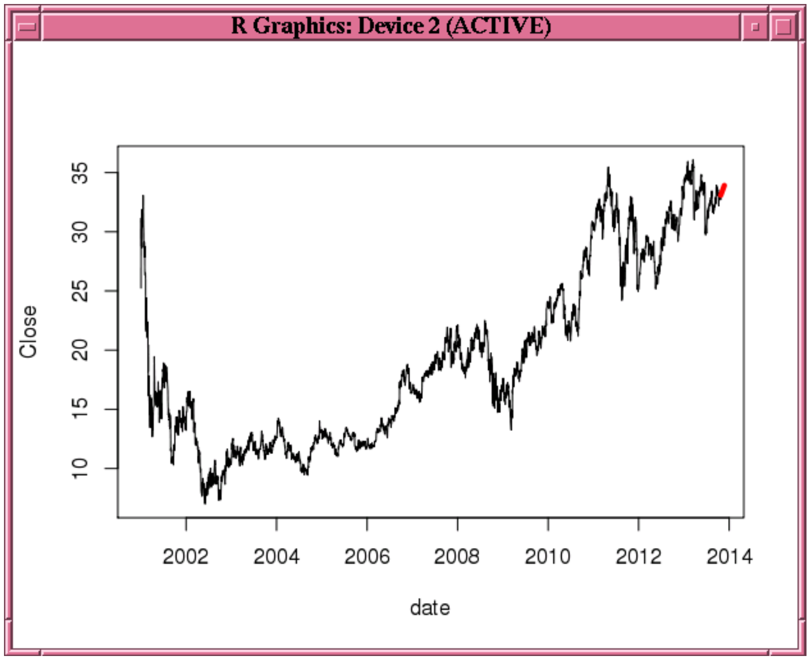
```
R> rownames(ts1) <- ts1$ID
R> x <- ts1$VAL
R> esm.mod <- ore.esm(x, "DAY", accumulate = "AVG", model="simple",
+                    setmissing="PREV")
R> esm.predict <- predict(esm.mod)
R> esm.predict
           ID      VAL
1  2013-09-26 9.962478
2  2013-09-27 9.962478
3  2013-09-28 9.962478
4  2013-09-29 9.962478
5  2013-09-30 9.962478
6  2013-10-01 9.962478
7  2013-10-02 9.962478
8  2013-10-03 9.962478
9  2013-10-04 9.962478
10 2013-10-05 9.962478
11 2013-10-06 9.962478
12 2013-10-07 9.962478
```

Figure 3–39 uses stock data from the `TTR` package. It builds a double exponential smoothing model based on the daily stock closing prices. The 30-day predicted stock prices, along with the original observations, are shown in Figure 3–2.

Example 3–39 Building a Double Exponential Smoothing Model Specifying an Interval

```
library(TTR)
stock <- "orcl"
xts.data <- getYahooData(stock, 20010101, 20131024)
df.data <- data.frame(xts.data)
df.data$date <- index(xts.data)
of.data <- ore.push(df.data[, c("date", "Close")])
rownames(of.data) <- of.data$date
esm.mod <- ore.esm(of.data$Close, "DAY", model = "double")
esm.predict <- predict(esm.mod, 30)
plot(of.data,type="l")
lines(esm.predict,col="red",lwd=4)
```

Figure 3–2 Stock Price Prediction



Ranking Data

The `ore.rank` function analyzes distribution of values in numeric columns of an `ore.frame`.

The `ore.rank` function supports useful functionality, including:

- Ranking within groups
- Partitioning rows into groups based on rank tiles
- Calculation of cumulative percentages and percentiles
- Treatment of ties
- Calculation of normal scores from ranks

The `ore.rank` function syntax is simpler than the corresponding SQL queries.

The `ore.rank` function returns an `ore.frame` in all instances.

You can use these R scoring methods with `ore.rank`:

- To compute exponential scores from ranks, use `savage`.
- To compute normal scores, use one of `blom`, `tukey`, or `vw` (van der Waerden).

For details about the function arguments, invoke `help(ore.rank)`.

The following examples illustrate using `ore.rank`. The examples use the `NARROW` data set.

Example 3–40 ranks the two columns AGE and CLASS and reports the results as derived columns; values are ranked in the default order, which is ascending.

Example 3–40 Ranking Two Columns

```
x <- ore.rank(data=NARROW, var='AGE=RankOfAge, CLASS=RankOfClass')
```

Example 3–41 ranks the two columns AGE and CLASS. If there is a tie, the smallest value is assigned to all tied values.

Example 3–41 Handling Ties in Ranking

```
x <- ore.rank(data=NARROW, var='AGE=RankOfAge, CLASS=RankOfClass', ties='low')
```

Example 3–42 ranks the two columns AGE and CLASS and then ranks the resulting values according to COUNTRY:

Example 3–42 Ranking by Groups

```
x <- ore.rank(data=NARROW, var='AGE=RankOfAge, CLASS=RankOfClass',
group.by='COUNTRY')
```

Example 3–43 ranks the two columns AGE and CLASS and partitions the columns into deciles (10 partitions):

Example 3–43 Partitioning into Deciles

```
x <- ore.rank(data=NARROW, var='AGE=RankOfAge, CLASS=RankOfClass',groups=10)
```

To partition the columns into a different number of partitions, change the value of `groups`. For example, `groups=4` partitions into quartiles.

Example 3–44 ranks the two columns AGE and CLASS and estimates the cumulative distribution function for both column.

Example 3–44 Estimating Cumulative Distribution Function

```
x <- ore.rank(data=NARROW, var='AGE=RankOfAge, CLASS=RankOfClass',nplus1=TRUE)
```

Example 3–45 ranks the two columns AGE and CLASS and scores the ranks in two different ways. The first command partitions the columns into percentiles (100 groups). The `savage` scoring method calculates exponential scores and `blom` scoring calculates normal scores:

Example 3–45 Scoring Ranks

```
x <- ore.rank(data=NARROW, var='AGE=RankOfAge,
          CLASS=RankOfClass', score='savage', groups=100, group.by='COUNTRY')
x <- ore.rank(data=NARROW, var='AGE=RankOfAge, CLASS=RankOfClass', score='blom')
```

Sorting Data

The `ore.sort` function enables flexible sorting of a data frame along one or more columns specified by the `by` argument.

The `ore.sort` function can be used with other data pre-processing functions. The results of sorting can provide input to R visualization.

The `ore.sort` function sorting takes places in the Oracle database. The `ore.sort` function supports the database `nls.sort` option.

The `ore.sort` function returns an `ore.frame`.

For details about the function arguments, invoke `help(ore.sort)`.

Most of the following examples use the `NARROW` data set. There are also examples that use the `ONTIME_S` data set.

Example 3–46 sorts the columns AGE and GENDER in descending order.

Example 3–46 Sorting Columns in Descending Order

```
x=ore.sort(data=NARROW, by='AGE,GENDER', reverse=TRUE)
```

Example 3–47 sorts AGE in descending order and GENDER in ascending order.

Example 3–47 Sorting Different Columns in Different Orders

```
x=ore.sort(data=NARROW, by='-AGE,GENDER')
```

Example 3–48 sorts by AGE and keep one row per unique value of AGE:

Example 3–48 Sorting and Returning One Row per Unique Value

```
x=ore.sort(data=NARROW, by='AGE', unique.key=TRUE)
```

Example 3–49 sorts by AGE and remove duplicate rows:

Example 3–49 Removing Duplicate Columns

```
x=ore.sort(data=NARROW, by='AGE', unique.data=TRUE)
```

Example 3–50 sorts by AGE, removes duplicate rows, and returns one row per unique value of AGE.

Example 3–50 Removing Duplicate Columns and Returning One Row per Unique Value

```
x=ore.sort(data=NARROW, by='AGE', unique.data=TRUE, unique.key = TRUE)
```

Example 3–51 maintains the relative order in the sorted output.

Example 3–51 Preserving Relative Order in the Output

```
x=ore.sort(data=NARROW, by='AGE', stable=TRUE)
```

The following examples use the `ONTIME_S` airline data set. Example 3–52 sorts ONTIME_S by airline name in descending order and departure delay in ascending order.

Example 3–52 Sorting Two Columns in Different Orders

```
sortedOnTime1 <- ore.sort(data=ONTIME_S, by='-UNIQUECARRIER,DEPDELAY')
```

Example 3–53 sorts ONTIME_S by airline name and departure delay and selects one of each combination (that is, returns a unique key).

Example 3–53 Sorting Two Columns in Different Orders and Producing Unique Combinations

```
sortedOnTime1 <- ore.sort(data=ONTIME_S, by='-UNIQUECARRIER,DEPDELAY',
                          unique.key=TRUE)
```

Summarizing Data

The `ore.summary` function calculates descriptive statistics and supports extensive analysis of columns in an `ore.frame`, along with flexible row aggregations.

The `ore.summary` function supports these statistics:

- Mean, minimum, maximum, mode, number of missing values, sum, weighted sum

- Corrected and uncorrected sum of squares, range of values, `stddev`, `stderr`, `variance`
- t-test for testing the hypothesis that the population mean is 0
- Kurtosis, skew, Coefficient of Variation
- Quantiles: p1, p5, p10, p25, p50, p75, p90, p95, p99, qrange
- 1-sided and 2-sided Confidence Limits for the mean: `clm`, `rclm`, `lclm`
- Extreme value tagging

The `ore.summary` function provides a relatively simple syntax compared with SQL queries that produce the same results.

The `ore.summary` function returns an `ore.frame` in all cases except when the `group.by` argument is used. If the `group.by` argument is used, then `ore.summary` returns a list of `ore.frame` objects, one `ore.frame` per stratum.

For details about the function arguments, invoke `help(ore.summary)`.

Example 3–54 calculates the mean, minimum, and maximum values for columns AGE and CLASS and rolls up (aggregates) the GENDER column.

Example 3–54 Calculating Default Statistics

```
ore.summary(NARROW, class='GENDER', var ='AGE,CLASS', order='freq')
```

Example 3–55 calculates the skew of AGE as column A and the probability of the Student's *t* distribution for CLASS as column B.

Example 3–55 Calculating Skew and Probability for t Test

```
ore.summary(NARROW, class='GENDER', var='AGE,CLASS', stats='skew(AGE)=A,
probt(CLASS)=B')
```

Example 3–56 calculates the weighted sum for AGE aggregated by GENDER with YRS_RESIDENCE as weights; in other words, it calculates `sum(var*weight)`.

Example 3–56 Calculating the Weighted Sum

```
ore.summary(NARROW, class='GENDER', var='AGE', stats='sum=X', weight='YRS_
RESIDENCE')
```

Example 3–57 groups CLASS by GENDER and MARITAL_STATUS.

Example 3–57 Grouping by Two Columns

```
ore.summary(NARROW, class='GENDER, MARITAL_STATUS', var='CLASS', ways=1)
```

Example 3–58 groups CLASS in all possible ways by GENDER and MARITAL_STATUS.

Example 3–58 Grouping by All Possible Ways

```
ore.summary(NARROW, class='GENDER, MARITAL_STATUS', var='CLASS', ways='nway')
```

Analyzing Distribution of Numeric Variables

The `ore.univariate` function provides distribution analysis of numeric variables in an `ore.frame`.

The `ore.univariate` function provides these statistics:

- All statistics reported by the `summary` function
- Signed rank test, Student's t-test
- Extreme values reporting

The `ore.univariate` function returns an ore.frame as output in all cases.

For details about the function arguments, invoke `help(ore.univariate)`.

Example 3–59 calculates the default univariate statistics for AGE, YRS_RESIDENCE, and CLASS.

Example 3–59 Calculating the Default Univariate Statistics

```
ore.univariate(NARROW, var="AGE,YRS_RESIDENCE,CLASS")
```

Example 3–60 calculates location statistics for YRS_RESIDENCE.

Example 3–60 Calculating the Default Univariate Statistics

```
ore.univariate(NARROW, var="YRS_RESIDENCE", stats="location")
```

Example 3–61 calculates complete quantile statistics for AGE and YRS_RESIDENCE.

Example 3–61 Calculating the Complete Quantile Statistics

```
ore.univariate(NARROW, var="AGE,YRS_RESIDENCE",stats="quantiles")
```

Using a Third-Party Package on the Client

In Oracle R Enterprise, if you want to use functions from an open source R package from the Comprehensive R Archive Network (CRAN) or other third-party R package, then you would generally do so in the context of embedded R execution. Using embedded R execution, you can take advantage of the likely greater amount of RAM on the database server.

However, if you want to use a third-party package function in your local R session on data from an Oracle database table, you must use the `ore.pull` function to get the data from an `ore.frame` object to your local session as a `data.frame` object. This is the same as using open source R except that you can extract the data from the database without needing the help of a DBA.

When pulling data from a database table to a local `data.frame`, you are limited to using the amount of data that can fit into the memory of your local machine. On your local machine, you do not have the benefits provided by embedded R execution.

To use a third-party package, you must install it on your system and load it in your R session. Example 3–62 demonstrates downloading and installing a CRAN package with the `install.packages` function and loading it in an R session.

Example 3–62 Loading a Third-Party Package on the Client

```
install.packages("arules")
library("arules")
```

For an example of using a CRAN package, see Example 4–7, "Using the ore.odmAssocRules Function" on page 4-11.

See Also:

- "Installing a Third-Party Package for Use in Embedded R Execution" on page 6-4
- `http://www.r-bloggers.com/installing-r-packages/`
- *Oracle R Enterprise Installation and Administration Guide*

4 Building Models in Oracle R Enterprise

Oracle R Enterprise provides functions for building regression models, neural network models, and models based on Oracle Data Mining algorithms.

This chapter has the following topics:

- Building Oracle R Enterprise Models
- Building Oracle Data Mining Models

Building Oracle R Enterprise Models

The Oracle R Enterprise package `OREmodels` contains functions with which you can create advanced analytical data models using `ore.frame` objects, as described in the following topics:

- About OREmodels Functions
- About the longley Data Set for Examples
- Building Linear Regression Models
- Building a Generalized Linear Model
- Building a Neural Network Model

About OREmodels Functions

The `OREmodels` package contains functions with which you can build advanced analytical data models using `ore.frame` objects. The `OREmodels` functions are the following:

Table 4–1 Functions in the OREmodels Package

Function	Description
`ore.glm`	Fits and uses a generalized linear model on data in an `ore.frame`.
`ore.lm`	Fits a linear regression model on data in an `ore.frame`.
`ore.neural`	Fits a neural network model on data in an `ore.frame`.
`ore.stepwise`	Fits a stepwise linear regression model on data in an `ore.frame`.

> **Note:** In R terminology, the phrase "fits a model" is often synonymous with "builds a model". In this document and in the online help for Oracle R Enterprise functions, the phrases are used interchangeably.

The `ore.glm`, `ore.lm` and `ore.stepwise` functions have the following advantages:

- The algorithms provide accurate solutions using out-of-core QR factorization. QR factorization decomposes a matrix into an orthogonal matrix and a triangular matrix.

  QR is an algorithm of choice for difficult rank-deficient models.

- You can process data that does not fit into memory, that is, out-of-core data. QR factors a matrix into two matrices, one of which fits into memory while the other is stored on disk.

  The `ore.glm`, `ore.lm` and `ore.stepwise` functions can solve data sets with more than one billion rows.

- The `ore.stepwise` function allows fast implementations of forward, backward, and stepwise model selection techniques.

The `ore.neural` function has the following advantages:

- It is a highly scalable implementation of neural networks, able to build a model on even billion row data sets in a matter of minutes. The `ore.neural` function can be run in two modes: in-memory for small to medium data sets and distributed (out-of-core) for large inputs.
- Users can specify the activation functions on neurons on a per-layer basis; `ore.neural` supports 15 different activation functions.
- Users can specify a neural network topology consisting of any number of hidden layers, including none.

About the longley Data Set for Examples

Most of the linear regression and `ore.neural` examples use the longley data set, which is provided by R. It is a small macroeconomic data set that provides a well-known example for collinear regression and consists of seven economic variables observed yearly over 16 years.

Example 4–1 pushes the longley data set to a temporary database table that has the proxy `ore.frame` object `longley_of` displays the first six rows of `longley_of`.

Example 4–1 Displaying Values from the longley Data Set

```
longley_of <- ore.push(longley)
head(longley_of)
```

Listing for Example 4–1

```
R> longley_of <- ore.push(longley)
R> dim(longley_of)
[1] 16  7
R> head(longley_of)
     GNP.deflator     GNP Unemployed Armed.Forces Population Year Employed
1947         83.0 234.289      235.6        159.0    107.608 1947   60.323
1948         88.5 259.426      232.5        145.6    108.632 1948   61.122
```

```
1949          88.2 258.054      368.2        161.6     109.773 1949     60.171
1950          89.5 284.599      335.1        165.0     110.929 1950     61.187
1951          96.2 328.975      209.9        309.9     112.075 1951     63.221
1952          98.1 346.999      193.2        359.4     113.270 1952     63.639
```

Building Linear Regression Models

The `ore.lm` and `ore.stepwise` functions perform least squares regression and stepwise least squares regression, respectively, on data represented in an `ore.frame` object. A model fit is generated using embedded R map/reduce operations where the map operation creates either QR decompositions or matrix cross-products depending on the number of coefficients being estimated. The underlying model matrices are created using either a `model.matrix` or `sparse.model.matrix` object depending on the sparsity of the model. Once the coefficients for the model have been estimated another pass of the data is made to estimate the model-level statistics.

When forward, backward, or stepwise selection is performed, the XtX and Xty matrices are subsetted to generate the F-test p-values based upon coefficient estimates that were generated using a Choleski decomposition of the XtX subset matrix.

If there are collinear terms in the model, functions `ore.lm` and `ore.stepwise` do not estimate the coefficient values for a collinear set of terms. For `ore.stepwise`, a collinear set of terms is excluded throughout the procedure.

For more information on `ore.lm` and `ore.stepwise`, invoke `help(ore.lm)`.

Example 4–2 pushes the `longley` data set to a temporary database table that has the proxy `ore.frame` object `longley_of`. The example builds a linear regression model using `ore.lm`.

Example 4–2 Using ore.lm

```
longley_of <- ore.push(longley)
# Fit full model
oreFit1 <- ore.lm(Employed ~ ., data = longley_of)
class(oreFit1)
summary(oreFit1)
```

Listing for Example 4–2

```
R> longley_of <- ore.push(longley)
R> # Fit full model
R>  oreFit1 <- ore.lm(Employed ~ ., data = longley_of)
R> class(oreFit1)
[1] "ore.lm"    "ore.model" "lm"
R> summary(oreFit1)

Call:
ore.lm(formula = Employed ~ ., data = longley_of)

Residuals:
     Min       1Q   Median       3Q      Max
-0.41011 -0.15767 -0.02816  0.10155  0.45539

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)  -3.482e+03  8.904e+02  -3.911 0.003560 **
GNP.deflator  1.506e-02  8.492e-02   0.177 0.863141
GNP          -3.582e-02  3.349e-02  -1.070 0.312681
Unemployed   -2.020e-02  4.884e-03  -4.136 0.002535 **
Armed.Forces -1.033e-02  2.143e-03  -4.822 0.000944 ***
```

```
Population   -5.110e-02  2.261e-01  -0.226 0.826212
Year          1.829e+00  4.555e-01   4.016 0.003037 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3049 on 9 degrees of freedom
Multiple R-squared:  0.9955,    Adjusted R-squared:  0.9925
F-statistic: 330.3 on 6 and 9 DF,  p-value: 4.984e-10
```

Example 4–3 pushes the `longley` data set to a temporary database table that has the proxy `ore.frame` object `longley_of`. The example builds linear regression models using the `ore.stepwise` function.

Example 4–3 Using the ore.stepwise Function

```
longley_of <- ore.push(longley)
# Two stepwise alternatives
oreStep1 <-
  ore.stepwise(Employed ~ .^2, data = longley_of, add.p = 0.1, drop.p = 0.1)
oreStep2 <-
  step(ore.lm(Employed ~ 1, data = longley_of),
             scope = terms(Employed ~ .^2, data = longley_of))
```

Listing for Example 4–3

```
R> longley_of <- ore.push(longley)
R> # Two stepwise alternatives
R> oreStep1 <-
+   ore.stepwise(Employed ~ .^2, data = longley_of, add.p = 0.1, drop.p = 0.1)
R> oreStep2 <-
+   step(ore.lm(Employed ~ 1, data = longley_of),
+              scope = terms(Employed ~ .^2, data = longley_of))
Start:  AIC=41.17
Employed ~ 1

               Df Sum of Sq     RSS     AIC
+ GNP           1   178.973   6.036 -11.597
+ Year          1   174.552  10.457  -2.806
+ GNP.deflator  1   174.397  10.611  -2.571
+ Population    1   170.643  14.366   2.276
+ Unemployed    1    46.716 138.293  38.509
+ Armed.Forces  1    38.691 146.318  39.411
<none>                      185.009  41.165

Step:  AIC=-11.6
Employed ~ GNP

               Df Sum of Sq     RSS     AIC
+ Unemployed    1     2.457   3.579 -17.960
+ Population    1     2.162   3.874 -16.691
+ Year          1     1.125   4.911 -12.898
<none>                        6.036 -11.597
+ GNP.deflator  1     0.212   5.824 -10.169
+ Armed.Forces  1     0.077   5.959  -9.802
- GNP           1   178.973 185.009  41.165
... The rest of the output is not shown.
```

Building a Generalized Linear Model

The `ore.glm` functions fits generalized linear models on data in an `ore.frame` object. The function uses a Fisher scoring iteratively re-weighted least squares (IRLS) algorithm.

Instead of the traditional step halving to prevent the selection of less optimal coefficient estimates, `ore.glm` uses a line search to select new coefficient estimates at each iteration, starting from the current coefficient estimates and moving through the Fisher scoring suggested estimates using the formula `(1 - alpha) * old + alpha * suggested` where `alpha in [0, 2]`. When the `interp` control argument is `TRUE`, the deviance is approximated by a cubic spline interpolation. When it is `FALSE`, the deviance is calculated using a follow-up data scan.

Each iteration consists of two or three embedded R execution map/reduce operations: an IRLS operation, an initial line search operation, and, if `interp = FALSE`, an optional follow-up line search operation. As with `ore.lm`, the IRLS map operation creates QR decompositions when `update = "qr"` or cross-products when `update = "crossprod"` of the `model.matrix`, or `sparse.model.matrix` if argument `sparse = TRUE`, and the IRLS reduce operation block updates those QR decompositions or cross-product matrices. After the algorithm has either converged or reached the maximum number of iterations, a final embedded R map/reduce operation is used to generate the complete set of model-level statistics.

The `ore.glm` function returns an `ore.glm` object.

For information on the `ore.glm` function arguments, invoke `help(ore.glm)`.

Example 4–4 loads the `rpart` package and then pushes the `kyphosis` data set to a temporary database table that has the proxy `ore.frame` object `KYPHOSIS`. The example builds a generalized linear model using the `ore.glm` function and one using the `glm` function and invokes the `summary` function on the models.

Example 4–4 Using the ore.glm Function

```
# Load the rpart library to get the kyphosis and solder data sets.
library(rpart)
# Logistic regression
KYPHOSIS <- ore.push(kyphosis)
kyphFit1 <- ore.glm(Kyphosis ~ ., data = KYPHOSIS, family = binomial())
kyphFit2 <- glm(Kyphosis ~ ., data = kyphosis, family = binomial())
summary(kyphFit1)
summary(kyphFit2)
```

Listing for Example 4–4

```
R> # Load the rpart library to get the kyphosis and solder data sets.
R> library(rpart)

R> # Logistic regression
R> KYPHOSIS <- ore.push(kyphosis)
R> kyphFit1 <- ore.glm(Kyphosis ~ ., data = KYPHOSIS, family = binomial())
R> kyphFit2 <- glm(Kyphosis ~ ., data = kyphosis, family = binomial())
R> summary(kyphFit1)

Call:
ore.glm(formula = Kyphosis ~ ., data = KYPHOSIS, family = binomial())

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.3124  -0.5484  -0.3632  -0.1659   2.1613
```

```
Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.036934   1.449622  -1.405  0.15998
Age          0.010930   0.006447   1.696  0.08997 .
Number       0.410601   0.224870   1.826  0.06786 .
Start       -0.206510   0.067700  -3.050  0.00229 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 83.234  on 80  degrees of freedom
Residual deviance: 61.380  on 77  degrees of freedom
AIC: 69.38

Number of Fisher Scoring iterations: 4

R> summary(kyphFit2)

Call:
glm(formula = Kyphosis ~ ., family = binomial(), data = kyphosis)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.3124  -0.5484  -0.3632  -0.1659   2.1613

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.036934   1.449575  -1.405  0.15996
Age          0.010930   0.006446   1.696  0.08996 .
Number       0.410601   0.224861   1.826  0.06785 .
Start       -0.206510   0.067699  -3.050  0.00229 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 83.234  on 80  degrees of freedom
Residual deviance: 61.380  on 77  degrees of freedom
AIC: 69.38

Number of Fisher Scoring iterations: 5

# Poisson regression
R> SOLDER <- ore.push(solder)
R> solFit1 <- ore.glm(skips ~ ., data = SOLDER, family = poisson())
R> solFit2 <- glm(skips ~ ., data = solder, family = poisson())
R> summary(solFit1)

Call:
ore.glm(formula = skips ~ ., data = SOLDER, family = poisson())

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.4105  -1.0897  -0.4408   0.6406   3.7927

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.25506    0.10069 -12.465  < 2e-16 ***
```

```
OpeningM     0.25851    0.06656   3.884 0.000103 ***
OpeningS     1.89349    0.05363  35.305  < 2e-16 ***
SolderThin   1.09973    0.03864  28.465  < 2e-16 ***
MaskA3       0.42819    0.07547   5.674 1.40e-08 ***
MaskB3       1.20225    0.06697  17.953  < 2e-16 ***
MaskB6       1.86648    0.06310  29.580  < 2e-16 ***
PadTypeD6   -0.36865    0.07138  -5.164 2.41e-07 ***
PadTypeD7   -0.09844    0.06620  -1.487 0.137001
PadTypeL4    0.26236    0.06071   4.321 1.55e-05 ***
PadTypeL6   -0.66845    0.07841  -8.525  < 2e-16 ***
PadTypeL7   -0.49021    0.07406  -6.619 3.61e-11 ***
PadTypeL8   -0.27115    0.06939  -3.907 9.33e-05 ***
PadTypeL9   -0.63645    0.07759  -8.203 2.35e-16 ***
PadTypeW4   -0.11000    0.06640  -1.657 0.097591 .
PadTypeW9   -1.43759    0.10419 -13.798  < 2e-16 ***
Panel        0.11818    0.02056   5.749 8.97e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 6855.7  on 719  degrees of freedom
Residual deviance: 1165.4  on 703  degrees of freedom
AIC: 2781.6

Number of Fisher Scoring iterations: 4
```

Building a Neural Network Model

Neural network models can be used to capture intricate nonlinear relationships between inputs and outputs or to find patterns in data. The `ore.neural` function builds a feed-forward neural network for regression on `ore.frame` data. It supports multiple hidden layers with a specifiable number of nodes. Each layer can have one of several activation functions.

The output layer is a single numeric or binary categorical target. The output layer can have any of the activation functions. It has the linear activation function by default

The output of `ore.neural` is an object of type `ore.neural`.

Modeling with the `ore.neural` function is well-suited for noisy and complex data such as sensor data. Problems that such data might have are the following:

- Potentially many (numeric) predictors, for example, pixel values
- The target may be discrete-valued, real-valued, or a vector of such values
- Training data may contain errors – robust to noise
- Fast scoring
- Model transparency is not required; models difficult to interpret

Typical steps in neural network modeling are the following:

1. Specifying the architecture
2. Preparing the data
3. Building the model
4. Specifying the stopping criteria: iterations, error on a validation set within tolerance

5. Viewing statistical results from model
6. Improving the model

For information about the arguments to the `ore.neural` function, invoke `help(ore.neural)`.

Example 4–5 builds a neural network with default values, including a hidden size of 1. The example pushes a subset of the `longley` data set to an `ore.frame` object in database memory as the object `trainData`. The example then pushes a different subset of `longley` to the database as the object `testData`. The example builds a neural network model with `trainData` and then predicts results using `testData`.

Example 4–5 Building a Neural Network Model

```
trainData <- ore.push(longley[1:11, ])
testData <- ore.push(longley[12:16, ])
fit <- ore.neural('Employed ~ GNP + Population + Year', data = trainData)
ans <- predict(fit, newdata = testData)
ans
```

Listing for Example 4–5

```
R> trainData <- ore.push(longley[1:11, ])
R> testData <- ore.push(longley[12:16, ])
R> fit <- ore.neural('Employed ~ GNP + Population + Year', data = trainData)
R> ans <- predict(fit, newdata = testData)
R> ans
  pred_Employed
1      67.97452
2      69.50893
3      70.28098
4      70.86127
5      72.31066
Warning message:
ORE object has no unique key - using random order
```

Example 4–6 pushes the `iris` data set to a temporary database table that has the proxy `ore.frame` object `IRIS`. The example builds a neural network model using the `ore.neural` function and specifies a different activation function for each layer.

Example 4–6 Using ore.neural and Specifying Activations

```
IRIS <- ore.push(iris)
fit <- ore.neural(Petal.Length ~ Petal.Width + Sepal.Length,
                  data = IRIS,
                  sparse = FALSE,
                  hiddenSizes = c(20, 5),
                  activations = c("bSigmoid", "tanh", "linear"))
ans <- predict(fit, newdata = IRIS,
               supplemental.cols = c("Petal.Length"))
options(ore.warn.order = FALSE)
head(ans, 3)
summary(ans)
```

Listing for Example 4–6

```
R> IRIS <- ore.push(iris)
R> fit <- ore.neural(Petal.Length ~ Petal.Width + Sepal.Length,
+                   data = IRIS,
+                   sparse = FALSE,
+                   hiddenSizes = c(20, 5),
```

```
+                   activations = c("bSigmoid", "tanh", "linear"))
R>
R> ans <- predict(fit, newdata = IRIS,
+                supplemental.cols = c("Petal.Length"))
R> options(ore.warn.order = FALSE)
R> head(ans, 3)
  Petal.Length pred_Petal.Length
1          1.4          1.416466
2          1.4          1.363385
3          1.3          1.310709
R> summary(ans)
  Petal.Length   pred_Petal.Length
 Min.   :1.000   Min.   :1.080
 1st Qu.:1.600   1st Qu.:1.568
 Median :4.350   Median :4.346
 Mean   :3.758   Mean   :3.742
 3rd Qu.:5.100   3rd Qu.:5.224
 Max.   :6.900   Max.   :6.300
```

Building Oracle Data Mining Models

This section describes using the functions in the OREdm package of Oracle R Enterprise to build Oracle Data Mining models in R. The section has the following topics:

- About Building Oracle Data Mining Models using Oracle R Enterprise
- Building an Association Rules Model
- Building an Attribute Importance Model
- Building a Decision Tree Model
- Building General Linearized Models
- Building a k-Means Model
- Building a Naive Bayes Model
- Building an Orthogonal Partitioning Cluster Model
- Building a Non-Negative Matrix Factorization Model
- Building a Support Vector Machine Model

See Also: *Oracle Data Mining Concepts*

About Building Oracle Data Mining Models using Oracle R Enterprise

Oracle Data Mining can mine tables, views, star schemas, transactional data, and unstructured data. The OREdm functions provide R interfaces that use arguments that conform to typical R usage for corresponding predictive analytics and data mining functions.

This section has the following topics:

- Oracle Data Mining Models Supported by Oracle R Enterprise
- About Oracle Data Mining Models Built by Oracle R Enterprise Functions

Oracle Data Mining Models Supported by Oracle R Enterprise

The functions in the OREdm package provide access to the in-database data mining functionality of Oracle Database. You use these functions to build data mining models in the database.

Table 4–2 lists the Oracle R Enterprise functions that build Oracle Data Mining models and the corresponding Oracle Data Mining algorithms and functions.

Table 4–2 Oracle R Enterprise Data Mining Model Functions

Oracle R Enterprise Function	Oracle Data Mining Algorithm	Oracle Data Mining Function
`ore.odmAI`	Minimum Description Length	Attribute Importance for Classification or Regression
`ore.odmAssocRules`	Apriori	Association Rules
`ore.odmDT`	Decision Tree	Classification
`ore.odmGLM`	Generalized Linear Models	Classification and Regression
`ore.odmKMeans`	*k*-Means	Clustering
`ore.odmNB`	Naive Bayes	Classification
`ore.odmNMF`	Non-Negative Matrix Factorization	Feature Extraction
`ore.odmOC`	Orthogonal Partitioning Cluster (O-Cluster)	Clustering
`ore.odmSVM`	Support Vector Machines	Classification and Regression

About Oracle Data Mining Models Built by Oracle R Enterprise Functions

In each `OREdm` R model object, the slot `name` (or `fit.name`) is the name of the underlying Oracle Data Mining model generated by the `OREdm` function. While the R model exists, the Oracle Data Mining model name can be used to access the Oracle Data Mining model through other interfaces, including:

- Oracle Data Miner
- Any SQL interface, such as SQL*Plus or SQL Developer

  In particular, the models can be used with the Oracle Data Mining SQL prediction functions.

With Oracle Data Miner you can do the following:

- Get a list of available models
- Use model viewers to inspect model details
- Score appropriately transformed data

> **Note:** Any transformations performed in the R space are not carried over into Oracle Data Miner or SQL scoring.

Users can also get a list of models using SQL for inspecting model details or for scoring appropriately transformed data.

Models built using `OREdm` functions are transient objects; they do not persist past the R session in which they were built unless they are explicitly saved in an Oracle R Enterprise datastore. Oracle Data Mining models built using Data Miner or SQL, on the other hand, exist until they are explicitly dropped.

Model objects can be saved or persisted, as described in "Saving and Managing R Objects in the Database" on page 2-15. Saving a model object generated by an `OREdm` function allows it to exist across R sessions and keeps the corresponding Oracle Data Mining object in place. While the `OREdm` model exists, you can export and import it; then you can use it apart from the Oracle R Enterprise R object existence.

Building an Association Rules Model

The ore.odmAssocRules function implements the apriori algorithm to find frequent itemsets and generate an association model. It finds the co-occurrence of items in large volumes of transactional data such as in the case of market basket analysis. An association rule identifies a pattern in the data in which the appearance of a set of items in a transactional record implies another set of items. The groups of items used to form rules must pass a minimum threshold according to how frequently they occur (the *support* of the rule) and how often the consequent follows the antecedent (the *confidence* of the rule). Association models generate all rules that have support and confidence greater than user-specified thresholds. The apriori algorithm is efficient, and scales well with respect to the number of transactions, number of items, and number of itemsets and rules produced.

The formula specification has the form ~ terms, where terms is a series of column names to include in the analysis. Multiple column names are specified using + between column names. Use ~ . if all columns in data should be used for model building. To exclude columns, use - before each column name to exclude. Functions can be applied to the items in terms to realize transformations.

The ore.odmAssocRules function accepts data in the following forms:

- Transactional data
- Multi-record case data using item id and item value
- Relational data

For examples of specifying the forms of data and for information on the arguments of the function, invoke help(ore.odmAssocRules).

The function rules returns an object of class ore.rules, which specifies a set of association rules. You can pull an ore.rules object into memory in a local R session by using ore.pull. The local in-memory object is of class rules defined in the arules package. See help(ore.rules).

The function itemsets returns an object of class ore.itemsets, which specifies a set of itemsets. You can pull an ore.itemsets object into memory in a local R session by using ore.pull. The local in-memory object is of class itemsets defined in the arules package. See help(ore.itemsets).

Example 4–7 builds an association model on a transactional data set. The packages arules and arulesViz are required to pull the resulting rules and itemsets into the client R session memory and be visualized. The graph of the rules appears in Figure 4–1.

Example 4–7 Using the ore.odmAssocRules Function

```
# Load the arules and arulesViz packages.
library(arules)
library(arulesViz)
# Create some transactional data.
id <- c(1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3)
item <- c("b", "d", "e", "a", "b", "c", "e", "b", "c", "d", "e")
# Push the data to the database as an ore.frame object.
transdata_of <- ore.push(data.frame(ID = id, ITEM = item))
# Build a model with specifications.
ar.mod1 <- ore.odmAssocRules(~., transdata_of, case.id.column = "ID",
             item.id.column = "ITEM", min.support = 0.6, min.confidence = 0.6,
             max.rule.length = 3)
# Generate itemsets and rules of the model.
itemsets <- itemsets(ar.mod1)
```

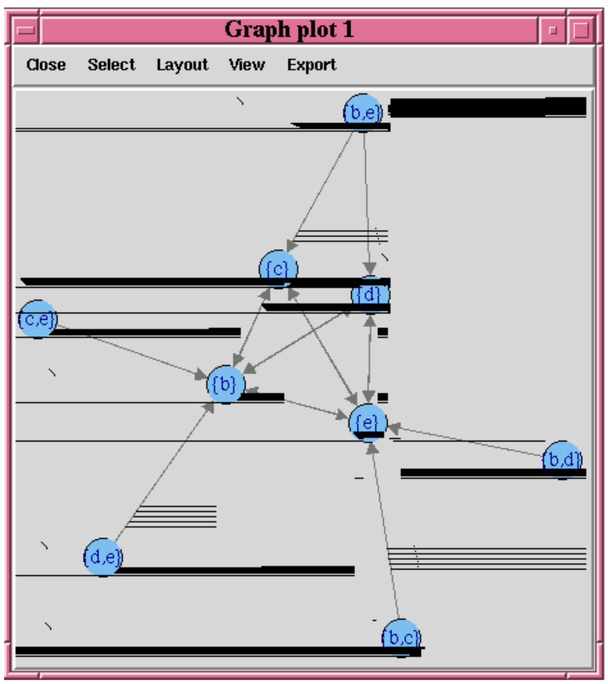
```
rules <- rules(ar.mod1)
# Convert the rules to the rules object in arules package.
rules.arules <- ore.pull(rules)
inspect(rules.arules)
# Convert itemsets to the itemsets object in arules package.
itemsets.arules <- ore.pull(itemsets)
inspect(itemsets.arules)
# Plot the rules graph.
plot(rules.arules, method = "graph", interactive = TRUE)
```

Listing for Example 4–7

```
R> # Load the arules and arulesViz packages.
R> library(arules)
R> library(arulesViz)
R> # Create some transactional data.
R> id <- c(1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3)
R> item <- c("b", "d", "e", "a", "b", "c", "e", "b", "c", "d", "e")
R> # Push the data to the database as an ore.frame object.
R> transdata_of <- ore.push(data.frame(ID = id, ITEM = item))
R> # Build a model with specifications.
R> ar.mod1 <- ore.odmAssocRules(~., transdata_of, case.id.column = "ID",
+              item.id.column = "ITEM", min.support = 0.6, min.confidence = 0.6,
+              max.rule.length = 3)
R> # Generate itemsets and rules of the model.
R> itemsets <- itemsets(ar.mod1)
R> rules <- rules(ar.mod1)
R> # Convert the rules to the rules object in arules package.
R> rules.arules <- ore.pull(rules)
R> inspect(rules.arules)
   lhs    rhs   support confidence lift
1  {b} => {e} 1.0000000  1.0000000    1
2  {e} => {b} 1.0000000  1.0000000    1
3  {c} => {e} 0.6666667  1.0000000    1
4  {d,
    e} => {b} 0.6666667  1.0000000    1
5  {c,
    e} => {b} 0.6666667  1.0000000    1
6  {b,
    d} => {e} 0.6666667  1.0000000    1
7  {b,
    c} => {e} 0.6666667  1.0000000    1
8  {d} => {b} 0.6666667  1.0000000    1
9  {d} => {e} 0.6666667  1.0000000    1
10 {c} => {b} 0.6666667  1.0000000    1
11 {b} => {d} 0.6666667  0.6666667    1
12 {b} => {c} 0.6666667  0.6666667    1
13 {e} => {d} 0.6666667  0.6666667    1
14 {e} => {c} 0.6666667  0.6666667    1
15 {b,
    e} => {d} 0.6666667  0.6666667    1
16 {b,
    e} => {c} 0.6666667  0.6666667    1
R> # Convert itemsets to the itemsets object in arules package.
R> itemsets.arules <- ore.pull(itemsets)
R> inspect(itemsets.arules)
   items   support
1  {b}   1.0000000
2  {e}   1.0000000
3  {b,
    e}   1.0000000
```

```
4  {c}    0.6666667
5  {d}    0.6666667
6  {b,
    c}    0.6666667
7  {b,
    d}    0.6666667
8  {c,
    e}    0.6666667
9  {d,
    e}    0.6666667
10 {b,
    c,
    e}    0.6666667
11 {b,
    d,
    e}    0.6666667

R> # Plot the rules graph.
R> plot(rules.arules, method = "graph", interactive = TRUE)
```

Figure 4–1   A Visual Demonstration of the Association Rules



Building an Attribute Importance Model

The `ore.odmAI` function uses the Oracle Data Mining Minimum Description Length algorithm to calculate attribute importance. Attribute importance ranks attributes according to their significance in predicting a target.

Minimum Description Length (MDL) is an information theoretic model selection principle. It is an important concept in information theory (the study of the quantification of information) and in learning theory (the study of the capacity for generalization based on empirical data).

MDL assumes that the simplest, most compact representation of the data is the best and most probable explanation of the data. The MDL principle is used to build Oracle Data Mining attribute importance models.

Attribute Importance models built using Oracle Data Mining cannot be applied to new data.

The `ore.odmAI` function produces a ranking of attributes and their importance values.

Note: `OREdm` AI models differ from Oracle Data Mining AI models in these ways: a model object is *not* retained, and an R model object is *not* returned. Only the importance ranking created by the model is returned.

For information on the `ore.odmAI` function arguments, invoke `help(ore.odmAI)`.

Example 4–8 pushes the `data.frame` `iris` to the database as the `ore.frame` `iris_of`. The example then builds an attribute importance model.

Example 4–8 Using the ore.odmAI Function

```
iris_of <- ore.push(iris)
ore.odmAI(Species ~ ., iris_of)
```

Listing for Example 4–8

```
R> iris_of <- ore.push(iris)
R> ore.odmAI(Species ~ ., iris_of)

Call:
ore.odmAI(formula = Species ~ ., data = iris_of)

Importance:
             importance rank
Petal.Width   1.1701851    1
Petal.Length  1.1494402    2
Sepal.Length  0.5248815    3
Sepal.Width   0.2504077    4
```

Building a Decision Tree Model

The `ore.odmDT` function uses the Oracle Data Mining Decision Tree algorithm, which is based on conditional probabilities. Decision trees generate rules. A rule is a conditional statement that can easily be understood by humans and be used within a database to identify a set of records.

Decision Tree models are classification models.

A decision tree predicts a target value by asking a sequence of questions. At a given stage in the sequence, the question that is asked depends upon the answers to the previous questions. The goal is to ask questions that, taken together, uniquely identify specific target values. Graphically, this process forms a tree structure.

During the training process, the Decision Tree algorithm must repeatedly find the most efficient way to split a set of cases (records) into two child nodes. The `ore.odmDT` function offers two homogeneity metrics, gini and entropy, for calculating the splits. The default metric is gini.

For information on the `ore.odmDT` function arguments, invoke `help(ore.odmDT)`.

Example 4–9 creates an input `ore.frame`, builds a model, makes predictions, and generates a confusion matrix.

Example 4–9 Using the ore.odmDT Function

```
m <- mtcars
m$gear <- as.factor(m$gear)
m$cyl  <- as.factor(m$cyl)
m$vs   <- as.factor(m$vs)
m$ID   <- 1:nrow(m)
mtcars_of <- ore.push(m)
row.names(mtcars_of) <- mtcars_of
# Build the model.
dt.mod  <- ore.odmDT(gear ~ ., mtcars_of)
summary(dt.mod)
# Make predictions and generate a confusion matrix.
dt.res  <- predict (dt.mod, mtcars_of, "gear")
with(dt.res, table(gear, PREDICTION))
```

Listing for Example 4–9

```
R> m <- mtcars
R> m$gear <- as.factor(m$gear)
R> m$cyl  <- as.factor(m$cyl)
R> m$vs   <- as.factor(m$vs)
R> m$ID   <- 1:nrow(m)
R> mtcars_of <- ore.push(m)
R> row.names(mtcars_of) <- mtcars_of
R> # Build the model.
R> dt.mod  <- ore.odmDT(gear ~ ., mtcars_of)
R> summary(dt.mod)

Call:
ore.odmDT(formula = gear ~ ., data = mtcars_of)

  n =  32

Nodes:
  parent node.id row.count prediction                         split
1     NA       0        32          3                          <NA>
2      0       1        16          4 (disp <= 196.299999999999995)
3      0       2        16          3  (disp > 196.299999999999995)
            surrogate                   full.splits
1                <NA>                          <NA>
2 (cyl in ("4" "6" )) (disp <= 196.299999999999995)
3     (cyl in ("8" ))  (disp > 196.299999999999995)

Settings:
                          value
prep.auto                    on
impurity.metric   impurity.gini
term.max.depth                7
term.minpct.node           0.05
term.minpct.split           0.1
term.minrec.node             10
term.minrec.split            20
R> # Make predictions and generate a confusion matrix.
R> dt.res  <- predict (dt.mod, mtcars_of, "gear")
R> with(dt.res, table(gear, PREDICTION))
    PREDICTION
gear  3  4
   3 14  1
   4  0 12
   5  2  3
```

Building General Linearized Models

The ore.odmGLM function builds Generalized Linear Models (GLM), which include and extend the class of linear models (linear regression). Generalized linear models relax the restrictions on linear models, which are often violated in practice. For example, binary (yes/no or 0/1) responses do not have same variance across classes.

The Oracle Data Mining GLM is a parametric modeling technique. Parametric models make assumptions about the distribution of the data. When the assumptions are met, parametric models can be more efficient than non-parametric models.

The challenge in developing models of this type involves assessing the extent to which the assumptions are met. For this reason, quality diagnostics are key to developing quality parametric models.

In addition to the classical weighted least squares estimation for linear regression and iteratively re-weighted least squares estimation for logistic regression, both solved through Cholesky decomposition and matrix inversion, Oracle Data Mining GLM provides a conjugate gradient-based optimization algorithm that does not require matrix inversion and is very well suited to high-dimensional data. The choice of algorithm is handled internally and is transparent to the user.

GLM can be used to build classification or regression models as follows:

- **Classification**: Binary logistic regression is the GLM classification algorithm. The algorithm uses the logit link function and the binomial variance function.
- **Regression:** Linear regression is the GLM regression algorithm. The algorithm assumes no target transformation and constant variance over the range of target values.

The ore.odmGLM function allows you to build two different types of models. Some arguments apply to classification models only and some to regression models only.

For information on the ore.odmGLM function arguments, invoke help(ore.odmGLM).

The following examples build several models using GLM. The input ore.frame objects are R data sets pushed to the database.

Example 4–10 builds a linear regression model using the longley data set.

Example 4–10 Building a Linear Regression Model

```
longley_of <- ore.push(longley)
longfit1 <- ore.odmGLM(Employed ~ ., data = longley_of)
summary(longfit1)
```

Listing for Example 4–10

```
R> longley_of <- ore.push(longley)
R> longfit1 <- ore.odmGLM(Employed ~ ., data = longley_of)
R> summary(longfit1)

Call:
ore.odmGLM(formula = Employed ~ ., data = longely_of)

Residuals:
     Min       1Q   Median       3Q      Max
-0.41011 -0.15767 -0.02816  0.10155  0.45539

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)  -3.482e+03  8.904e+02  -3.911 0.003560 **
```

```
GNP.deflator  1.506e-02  8.492e-02   0.177 0.863141
GNP          -3.582e-02  3.349e-02  -1.070 0.312681
Unemployed   -2.020e-02  4.884e-03  -4.136 0.002535 **
Armed.Forces -1.033e-02  2.143e-03  -4.822 0.000944 ***
Population   -5.110e-02  2.261e-01  -0.226 0.826212
Year          1.829e+00  4.555e-01   4.016 0.003037 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3049 on 9 degrees of freedom
Multiple R-squared:  0.9955,    Adjusted R-squared:  0.9925
F-statistic: 330.3 on 6 and 9 DF,  p-value: 4.984e-10
```

Example 4–11 uses the `longley_of` `ore.frame` from Example 4–10. Example 4–11 invokes the `ore.odmGLM` function and specifies using ridge estimation for the coefficients.

Example 4–11 Using Ridge Estimation for the Coefficients of the ore.odmGLM Model

```
longfit2 <- ore.odmGLM(Employed ~ ., data = longley_of, ridge = TRUE,
                       ridge.vif = TRUE)
summary(longfit2)
```

Listing for Example 4–11

```
R> longfit2 <- ore.odmGLM(Employed ~ ., data = longley_of, ridge = TRUE,
+                          ridge.vif = TRUE)
R> summary(longfit2)

Call:
ore.odmGLM(formula = Employed ~ ., data = longley_of, ridge = TRUE,
    ridge.vif = TRUE)

Residuals:
    Min      1Q  Median      3Q     Max
-0.4100 -0.1579 -0.0271  0.1017  0.4575

Coefficients:
               Estimate   VIF
(Intercept)  -3.466e+03 0.000
GNP.deflator  1.479e-02 0.077
GNP          -3.535e-02 0.012
Unemployed   -2.013e-02 0.000
Armed.Forces -1.031e-02 0.000
Population   -5.262e-02 0.548
Year          1.821e+00 2.212

Residual standard error: 0.3049 on 9 degrees of freedom
Multiple R-squared:  0.9955,    Adjusted R-squared:  0.9925
F-statistic: 330.2 on 6 and 9 DF,  p-value: 4.986e-10
```

Example 4–12 builds a logistic regression (classification) model. It uses the infert data set. The example invokes the `ore.odmGLM` function and specifies of `"logistic"` as the `type` argument, which builds a binomial GLM.

Example 4–12 Building a Logistic Regression GLM

```
infert_of <- ore.push(infert)
infit1 <- ore.odmGLM(case ~ age+parity+education+spontaneous+induced,
                     data = infert_of, type = "logistic")
infit1
```

Listing for Example 4–12

```
R> infert_of <- ore.push(infert)
R> infit1 <- ore.odmGLM(case ~ age+parity+education+spontaneous+induced,
+                        data = infert_of, type = "logistic")
R> infit1

Response:
case == "1"

Call:  ore.odmGLM(formula = case ~ age + parity + education + spontaneous +
    induced, data = infert_of, type = "logistic")

Coefficients:
     (Intercept)               age           parity   education0-5yrs
education12+ yrs       spontaneous          induced
        -2.19348           0.03958         -0.82828           1.04424
-0.35896           2.04590           1.28876

Degrees of Freedom: 247 Total (i.e. Null);  241 Residual
Null Deviance:      316.2
Residual Deviance: 257.8        AIC: 271.8
```

Example 4–13 builds a logistic regression (classification) model and specifies a reference value. The example uses the `infert_of` `ore.frame` from Example 4–12.

Example 4–13   Specifying a Reference Value in Building a Logistic Regression GLM

```
infit2 <- ore.odmGLM(case ~ age+parity+education+spontaneous+induced,
                     data = infert_of, type = "logistic", reference = 1)
infit2
```

Listing for Example 4–13

```
infit2 <- ore.odmGLM(case ~ age+parity+education+spontaneous+induced,
                     data = infert_of, type = "logistic", reference = 1)
infit2

Response:
case == "0"

Call:  ore.odmGLM(formula = case ~ age + parity + education + spontaneous +
    induced, data = infert_of, type = "logistic", reference = 1)

Coefficients:
     (Intercept)               age           parity   education0-5yrs
education12+ yrs       spontaneous          induced
         2.19348          -0.03958          0.82828          -1.04424
0.35896          -2.04590          -1.28876

Degrees of Freedom: 247 Total (i.e. Null);  241 Residual
Null Deviance:      316.2
Residual Deviance: 257.8        AIC: 271.8
```

Building a k-Means Model

The `ore.odmKM` function uses the Oracle Data Mining *k*-Means (KM) algorithm, a distance-based clustering algorithm that partitions data into a specified number of clusters. The algorithm has the following features:

- Several distance functions: Euclidean, Cosine, and Fast Cosine distance functions. The default is Euclidean.
- For each cluster, the algorithm returns the centroid, a histogram for each attribute, and a rule describing the hyperbox that encloses the majority of the data assigned to the cluster. The centroid reports the mode for categorical attributes and the mean and variance for numeric attributes.

For information on the `ore.odmKM` function arguments, invoke `help(ore.odmKM)`.

Example 4–14 demonstrates the use of the `ore.odmKMeans` function. The example creates two matrices that have 100 rows and two columns. The values in the rows are random variates. It binds the matrices into the `matrix` `x`. The example coerces `x` to a `data.frame` and pushes it to the database as `x_of`, an `ore.frame` object. The example invokes the `ore.odmKMeans` function to build the KM model, `km.mod1`. The example then invokes the `summary` and `histogram` functions on the model. Figure 4–2 on page 4-21 shows the graphic displayed by the `histogram` function.

The example then makes a prediction using the model and pulls the result to local memory. It plots the results. Figure 4–3 on page 4-21 shows the graphic displayed by the `points` function in the example.

Example 4–14 Using the ore.odmKM Function

```
x <- rbind(matrix(rnorm(100, sd = 0.3), ncol = 2),
           matrix(rnorm(100, mean = 1, sd = 0.3), ncol = 2))
colnames(x) <- c("x", "y")
x_of <- ore.push (data.frame(x))
km.mod1 <- NULL
km.mod1 <- ore.odmKMeans(~., x_of, num.centers=2)
summary(km.mod1)
histogram(km.mod1) # See Figure 4-2.
# Make a prediction.
km.res1 <- predict(km.mod1, x_of, type="class", supplemental.cols=c("x","y"))
head(km.res1, 3)
# Pull the results to the local memory and plot them.
km.res1.local <- ore.pull(km.res1)
plot(data.frame(x=km.res1.local$x, y=km.res1.local$y),
                col=km.res1.local$CLUSTER_ID)
points(km.mod1$centers2, col = rownames(km.mod1$centers2), pch = 8, cex=2)
head(predict(km.mod1, x_of, type=c("class","raw"),
             supplemental.cols=c("x","y")), 3)
```

Listing for Example 4–14

```
R> x <- rbind(matrix(rnorm(100, sd = 0.3), ncol = 2),
+             matrix(rnorm(100, mean = 1, sd = 0.3), ncol = 2))
R> colnames(x) <- c("x", "y")
R> x_of <- ore.push (data.frame(x))
R> km.mod1 <- NULL
R> km.mod1 <- ore.odmKMeans(~., x_of, num.centers=2)
R> summary(km.mod1)

Call:
ore.odmKMeans(formula = ~., data = x_of, num.centers = 2)

Settings:
                         value
clus.num.clusters            2
block.growth                 2
conv.tolerance            0.01
```

```
distance              euclidean
iterations                    3
min.pct.attr.support        0.1
num.bins                     10
split.criterion        variance
prep.auto                    on

Centers:
            x           y
2  0.99772307  0.93368684
3 -0.02721078 -0.05099784
R> histogram(km.mod1) # See Figure 4-2.
R> # Make a prediction.
R> km.res1 <- predict(km.mod1, x_of, type="class", supplemental.cols=c("x","y"))
R> head(km.res1, 3)
            x          y CLUSTER_ID
1 -0.03038444  0.4395409          3
2  0.17724606 -0.5342975          3
3 -0.17565761  0.2832132          3
# Pull the results to the local memory and plot them.
R> km.res1.local <- ore.pull(km.res1)
R> plot(data.frame(x=km.res1.local$x, y=km.res1.local$y),
+                  col=km.res1.local$CLUSTER_ID)
R> points(km.mod1$centers2, col = rownames(km.mod1$centers2), pch = 8, cex=2)
R> # See Figure 4-3.
R> head(predict(km.mod1, x_of, type=c("class","raw"),
                supplemental.cols=c("x","y")), 3)
           '2'       '3'           x          y CLUSTER_ID
1 8.610341e-03 0.9913897 -0.03038444  0.4395409          3
2 8.017890e-06 0.9999920  0.17724606 -0.5342975          3
3 5.494263e-04 0.9994506 -0.17565761  0.2832132          3
```

Figure 4–2 shows the graphic displayed by the invocation of the `histogram` function in Example 4–14.

Figure 4–2 Cluster Histograms for the km.mod1 Model

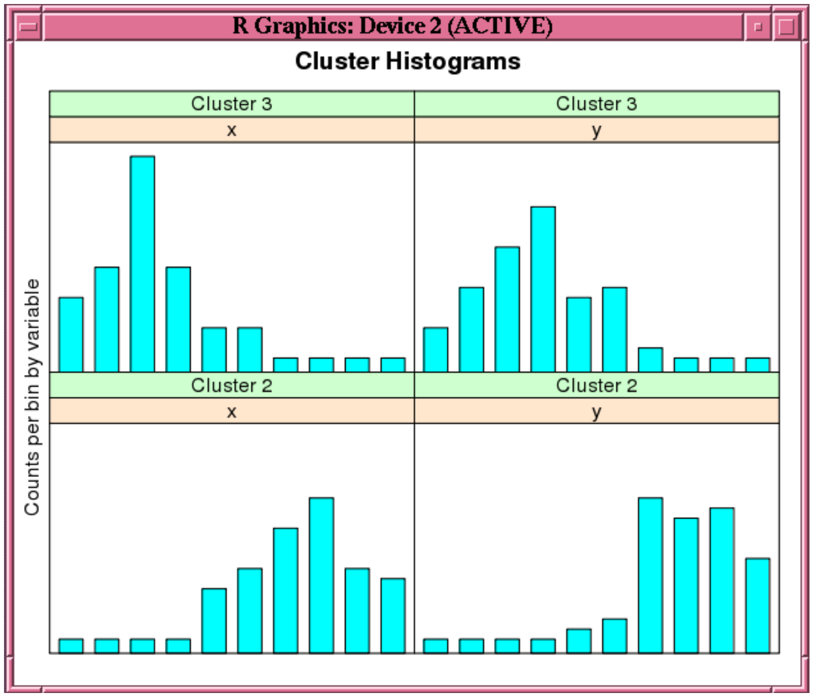
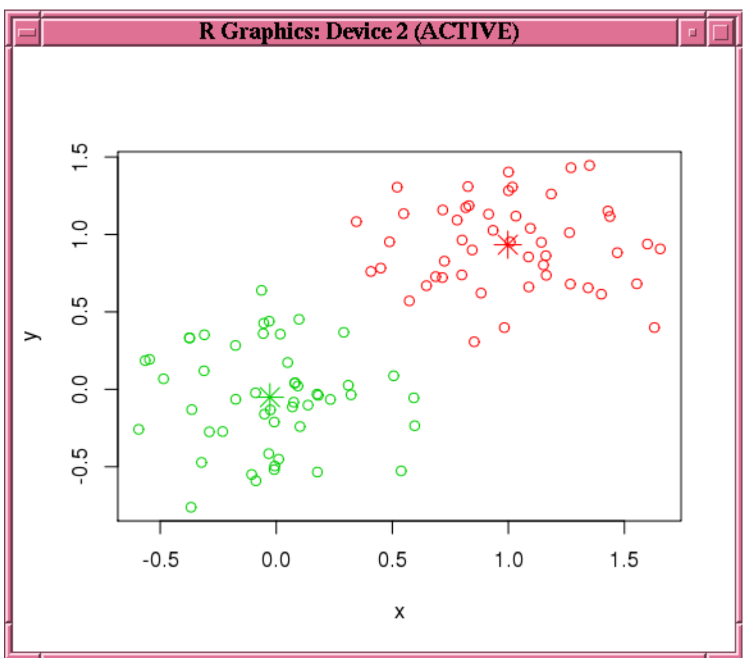


Figure 4–4 shows the graphic displayed by the invocation of the `points` function in Example 4–14.

Figure 4–3 Results of the points Function for the km.mod1 Model



Building a Naive Bayes Model

The `ore.odmNB` function builds an Oracle Data Mining Naive Bayes model. The Naive Bayes algorithm is based on conditional probabilities. Naive Bayes looks at the historical data and calculates conditional probabilities for the target values by observing the frequency of attribute values and of combinations of attribute values.

Naive Bayes assumes that each predictor is conditionally independent of the others. (Bayes' Theorem requires that the predictors be independent.)

For information on the `ore.odmNB` function arguments, invoke `help(ore.odmNB)`.

Example 4–9 creates an input `ore.frame`, builds a Naive Bayes model, makes predictions, and generates a confusion matrix.

Example 4–15 Using the ore.odmNB Function

```
m <- mtcars
m$gear <- as.factor(m$gear)
m$cyl  <- as.factor(m$cyl)
m$vs   <- as.factor(m$vs)
m$ID   <- 1:nrow(m)
mtcars_of <- ore.push(m)
row.names(mtcars_of) <- mtcars_of
# Build the model.
nb.mod  <- ore.odmNB(gear ~ ., mtcars_of)
summary(nb.mod)
# Make predictions and generate a confusion matrix.
nb.res  <- predict (nb.mod, mtcars_of, "gear")
with(nb.res, table(gear, PREDICTION))
```

Listing for Example 4–15

```
R> m <- mtcars
R> m$gear <- as.factor(m$gear)
R> m$cyl  <- as.factor(m$cyl)
R> m$vs   <- as.factor(m$vs)
R> m$ID   <- 1:nrow(m)
R> mtcars_of <- ore.push(m)
R> row.names(mtcars_of) <- mtcars_of
R> # Build the model.
R> nb.mod  <- ore.odmNB(gear ~ ., mtcars_of)
R> summary(nb.mod)

Call:
ore.odmNB(formula = gear ~ ., data = mtcars_of)

Settings:
          value
prep.auto    on

Apriori:
      3       4       5
0.46875 0.37500 0.15625
Tables:
$ID
  ( ; 26.5), [26.5; 26.5]  (26.5;  )
3              1.00000000
4              0.91666667 0.08333333
5                         1.00000000

$am
```

```
          0         1
3 1.0000000
4 0.3333333 0.6666667
5          1.0000000

$cyl
  '4', '6' '8'
3      0.2 0.8
4      1.0
5      0.6 0.4

$disp
  ( ; 196.299999999999995), [196.299999999999995; 196.299999999999995]
3                                                            0.06666667
4                                                            1.00000000
5                                                            0.60000000
  (196.299999999999995;  )
3              0.93333333
4
5              0.40000000

$drat
  ( ; 3.385), [3.385; 3.385] (3.385;  )
3                 0.8666667  0.1333333
4                            1.0000000
5                            1.0000000
$hp
  ( ; 136.5), [136.5; 136.5] (136.5;  )
3                       0.2        0.8
4                       1.0
5                       0.4        0.6

$vs
          0         1
3 0.8000000 0.2000000
4 0.1666667 0.8333333
5 0.8000000 0.2000000

$wt
  ( ; 3.2024999999999999), [3.2024999999999999; 3.2024999999999999]
3                                                         0.06666667
4                                                         0.83333333
5                                                         0.80000000
  (3.2024999999999999;  )
3             0.93333333
4             0.16666667
5             0.20000000


Levels:
[1] "3" "4" "5"

R> # Make predictions and generate a confusion matrix.
R> nb.res  <- predict (nb.mod, mtcars_of, "gear")
R> with(nb.res, table(gear, PREDICTION))
    PREDICTION
gear  3  4  5
   3 14  1  0
   4  0 12  0
   5  0  1  4
```

Building a Non-Negative Matrix Factorization Model

The `ore.odmNMF` function builds an Oracle Data Mining Non-Negative Matrix Factorization (NMF) model for feature extraction. Each feature extracted by NMF is a linear combination of the original attribution set. Each feature has a set of non-negative coefficients, which are a measure of the weight of each attribute on the feature. If the argument `allow.negative.scores` is `TRUE`, then negative coefficients are allowed.

For information on the `ore.odmNMF` function arguments, invoke `help(ore.odmNMF)`.

Example 4–16 creates an NMF model on a training data set and scores on a test data set.

Example 4–16 Using the ore.odmNMF Function

```
training.set <- ore.push(npk[1:18, c("N","P","K")])
scoring.set <- ore.push(npk[19:24, c("N","P","K")])
nmf.mod <- ore.odmNMF(~., training.set, num.features = 3)
features(nmf.mod)
summary(nmf.mod)
predict(nmf.mod, scoring.set)
```

Listing for Example 4–16

```
R> training.set <- ore.push(npk[1:18, c("N","P","K")])
R> scoring.set <- ore.push(npk[19:24, c("N","P","K")])
R> nmf.mod <- ore.odmNMF(~., training.set, num.features = 3)
R> features(nmf.mod)
   FEATURE_ID ATTRIBUTE_NAME ATTRIBUTE_VALUE  COEFFICIENT
1           1              K               0 3.723468e-01
2           1              K               1 1.761670e-01
3           1              N               0 7.469067e-01
4           1              N               1 1.085058e-02
5           1              P               0 5.730082e-01
6           1              P               1 2.797865e-02
7           2              K               0 4.107375e-01
8           2              K               1 2.193757e-01
9           2              N               0 8.065393e-03
10          2              N               1 8.569538e-01
11          2              P               0 4.005661e-01
12          2              P               1 4.124996e-02
13          3              K               0 1.918852e-01
14          3              K               1 3.311137e-01
15          3              N               0 1.547561e-01
16          3              N               1 1.283887e-01
17          3              P               0 9.791965e-06
18          3              P               1 9.113922e-01
R> summary(nmf.mod)

Call:
ore.odmNMF(formula = ~., data = training.set, num.features = 3)

Settings:
                                              value
feat.num.features                                 3
nmfs.conv.tolerance                             .05
nmfs.nonnegative.scoring nmfs.nonneg.scoring.enable
nmfs.num.iterations                              50
nmfs.random.seed                                 -1
prep.auto                                        on
```

```
Features:
   FEATURE_ID ATTRIBUTE_NAME ATTRIBUTE_VALUE  COEFFICIENT
1           1              K               0 3.723468e-01
2           1              K               1 1.761670e-01
3           1              N               0 7.469067e-01
4           1              N               1 1.085058e-02
5           1              P               0 5.730082e-01
6           1              P               1 2.797865e-02
7           2              K               0 4.107375e-01
8           2              K               1 2.193757e-01
9           2              N               0 8.065393e-03
10          2              N               1 8.569538e-01
11          2              P               0 4.005661e-01
12          2              P               1 4.124996e-02
13          3              K               0 1.918852e-01
14          3              K               1 3.311137e-01
15          3              N               0 1.547561e-01
16          3              N               1 1.283887e-01
17          3              P               0 9.791965e-06
18          3              P               1 9.113922e-01
R> predict(nmf.mod, scoring.set)
         '1'       '2'        '3' FEATURE_ID
19 0.1972489 1.2400782 0.03280919          2
20 0.7298919 0.0000000 1.29438165          3
21 0.1972489 1.2400782 0.03280919          2
22 0.0000000 1.0231268 0.98567623          2
23 0.7298919 0.0000000 1.29438165          3
24 1.5703239 0.1523159 0.00000000          1
```

Building an Orthogonal Partitioning Cluster Model

The `ore.odmOC` function builds an Oracle Data Mining model using the Orthogonal Partitioning Cluster (O-Cluster) algorithm. The O-Cluster algorithm builds a hierarchical grid-based clustering model, that is, it creates axis-parallel (orthogonal) partitions in the input attribute space. The algorithm operates recursively. The resulting hierarchical structure represents an irregular grid that tessellates the attribute space into clusters. The resulting clusters define dense areas in the attribute space.

The clusters are described by intervals along the attribute axes and the corresponding centroids and histograms. The `sensitivity` argument defines a baseline density level. Only areas that have a peak density above this baseline level can be identified as clusters.

The *k*-Means algorithm tessellates the space even when natural clusters may not exist. For example, if there is a region of uniform density, *k*-Means tessellates it into n clusters (where n is specified by the user). O-Cluster separates areas of high density by placing cutting planes through areas of low density. O-Cluster needs multi-modal histograms (peaks and valleys). If an area has projections with uniform or monotonically changing density, O-Cluster does not partition it.

The clusters discovered by O-Cluster are used to generate a Bayesian probability model that is then used during scoring by the `predict` function for assigning data points to clusters. The generated probability model is a mixture model where the mixture components are represented by a product of independent normal distributions for numeric attributes and multinomial distributions for categorical attributes.

If you choose to prepare the data for an O-Cluster model, keep the following points in mind:

- The O-Cluster algorithm does not necessarily use all the input data when it builds a model. It reads the data in batches (the default batch size is 50000). It only reads another batch if it believes, based on statistical tests, that there may still exist clusters that it has not yet uncovered.
- Because O-Cluster may stop the model build before it reads all of the data, it is highly recommended that the data be randomized.
- Binary attributes should be declared as categorical. O-Cluster maps categorical data to numeric values.
- The use of Oracle Data Mining equi-width binning transformation with automated estimation of the required number of bins is highly recommended.
- The presence of outliers can significantly impact clustering algorithms. Use a clipping transformation before binning or normalizing. Outliers with equi-width binning can prevent O-Cluster from detecting clusters. As a result, the whole population appears to fall within a single cluster.

The specification of the `formula` argument has the form `~ terms` where `terms` are the column names to include in the model. Multiple `terms` items are specified using `+` between column names. Use `~ .` if all columns in `data` should be used for model building. To exclude columns, use `-` before each column name to exclude.

For information on the `ore.odmOC` function arguments, invoke `help(ore.odmOC)`.

Example 4–17 creates an OC model on a synthetic data set. Figure 4–4 on page 4-27 shows the histogram of the resulting clusters.

Example 4–17 Using the ore.odmOC Function

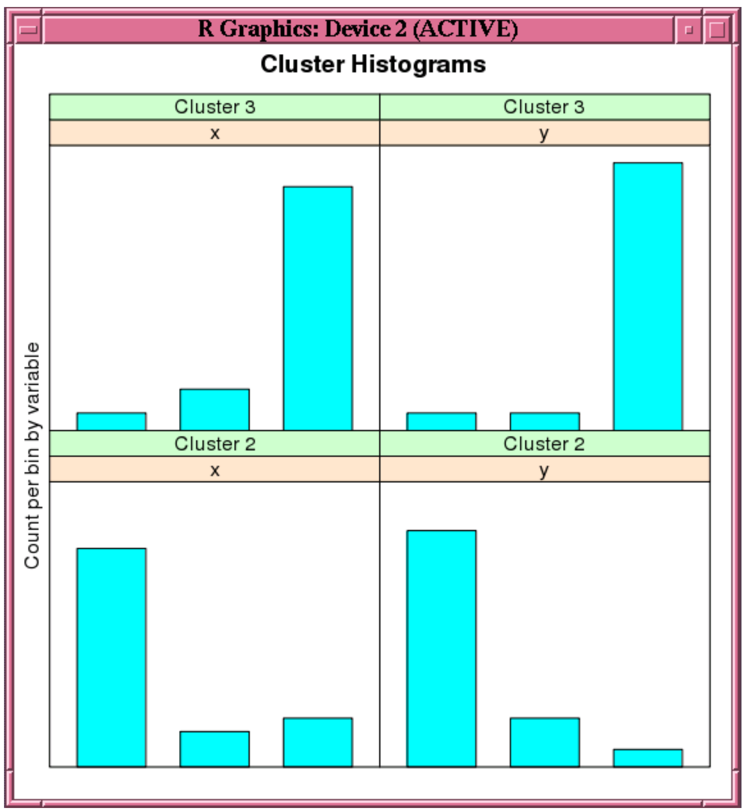
```
x <- rbind(matrix(rnorm(100, mean = 4, sd = 0.3), ncol = 2),
           matrix(rnorm(100, mean = 2, sd = 0.3), ncol = 2))
colnames(x) <- c("x", "y")
x_of <- ore.push (data.frame(ID=1:100,x))
rownames(x_of) <- x_of$ID
oc.mod <- ore.odmOC(~., x_of, num.centers=2)
summary(oc.mod)
```

Listing for Example 4–17

```
R> x <- rbind(matrix(rnorm(100, mean = 4, sd = 0.3), ncol = 2),
+              matrix(rnorm(100, mean = 2, sd = 0.3), ncol = 2))
R> colnames(x) <- c("x", "y")
R> x_of <- ore.push (data.frame(ID=1:100,x))
R> rownames(x_of) <- x_of$ID
R> oc.mod <- ore.odmOC(~., x_of, num.centers=2)
R> summary(oc.mod)

Call:
ore.odmOC(formula = ~., data = x_of, num.centers = 2)

Settings:
                  value
clus.num.clusters     2
max.buffer        50000
sensitivity         0.5
prep.auto            on

Clusters:
  CLUSTER_ID ROW_CNT PARENT_CLUSTER_ID TREE_LEVEL DISPERSION IS_LEAF
1          1     100                NA          1         NA   FALSE
```

```
2          2      56                 1         2         NA     TRUE
3          3      43                 1         2         NA     TRUE

Centers:
   MEAN.x   MEAN.y
2 1.85444 1.941195
3 4.04511 4.111740
R> histogram(oc.mod)     # See Figure 4-4.
R> predict(oc.mod, x_of, type=c("class","raw"), supplemental.cols=c("x","y"))
             '2'          '3'        x        y CLUSTER_ID
1   3.616386e-08 9.999999e-01 3.825303 3.935346          3
2   3.253662e-01 6.746338e-01 3.454143 4.193395          3
3   3.616386e-08 9.999999e-01 4.049120 4.172898          3
# ... Intervening rows not shown.
98  1.000000e+00 1.275712e-12 2.011463 1.991468          2
99  1.000000e+00 1.275712e-12 1.727580 1.898839          2
100 1.000000e+00 1.275712e-12 2.092737 2.212688          2
```

Figure 4–4 Output of the histogram Function for the ore.odmOC Model



Building a Support Vector Machine Model

The `ore.odmSVM` function builds an Oracle Data Mining Support Vector Machine (SVM) model. SVM is a powerful, state-of-the-art algorithm with strong theoretical foundations based on the Vapnik-Chervonenkis theory. SVM has strong regularization properties. Regularization refers to the generalization of the model to new data.

SVM models have similar functional form to neural networks and radial basis functions, both popular data mining techniques.

SVM can be used to solve the following problems:

- **Classification**: SVM classification is based on decision planes that define decision boundaries. A decision plane is one that separates between a set of objects having different class memberships. SVM finds the vectors ("support vectors") that define the separators that give the widest separation of classes.

  SVM classification supports both binary and multiclass targets.

- **Regression**: SVM uses an epsilon-insensitive loss function to solve regression problems.

  SVM regression tries to find a continuous function such that the maximum number of data points lie within the epsilon-wide insensitivity tube. Predictions falling within epsilon distance of the true target value are not interpreted as errors.

- **Anomaly Detection**: Anomaly detection identifies identify cases that are unusual within data that is seemingly homogeneous. Anomaly detection is an important tool for detecting fraud, network intrusion, and other rare events that may have great significance but are hard to find.

  Anomaly detection is implemented as one-class SVM classification. An anomaly detection model predicts whether a data point is typical for a given distribution or not.

The `ore.odmSVM` function builds each of these three different types of models. Some arguments apply to classification models only, some to regression models only, and some to anomaly detection models only.

For information on the `ore.odmSVM` function arguments, invoke `help(ore.odmSVM)`.

Example 4–18 demonstrates the use of SVM classification. The example creates `mtcars` in the database from the R `mtcars` data set, builds a classification model, makes predictions, and finally generates a confusion matrix.

Example 4–18 Using the ore.odmSVM Function and Generating a Confusion Matrix

```
m <- mtcars
m$gear <- as.factor(m$gear)
m$cyl  <- as.factor(m$cyl)
m$vs   <- as.factor(m$vs)
m$ID   <- 1:nrow(m)
mtcars_of <- ore.push(m)
svm.mod  <- ore.odmSVM(gear ~ .-ID, mtcars_of, "classification")
summary(svm.mod)
svm.res  <- predict (svm.mod, mtcars_of,"gear")
with(svm.res, table(gear, PREDICTION))  # generate confusion matrix
```

Listing for Example 4–18

```
R> m <- mtcars
R> m$gear <- as.factor(m$gear)
R> m$cyl  <- as.factor(m$cyl)
R> m$vs   <- as.factor(m$vs)
R> m$ID   <- 1:nrow(m)
R> mtcars_of <- ore.push(m)
R>
R> svm.mod  <- ore.odmSVM(gear ~ .-ID, mtcars_of, "classification")
R> summary(svm.mod)
Call:
ore.odmSVM(formula = gear ~ . - ID, data = mtcars_of, type = "classification")

Settings:
                      value
```

```
prep.auto                 on
active.learning   al.enable
complexity.factor  0.385498
conv.tolerance        1e-04
kernel.cache.size  50000000
kernel.function    gaussian
std.dev            1.072341

Coefficients:
[1] No coefficients with gaussian kernel
R> svm.res  <- predict (svm.mod, mtcars_of,"gear")
R> with(svm.res, table(gear, PREDICTION))  # generate confusion matrix
    PREDICTION
gear  3  4
   3 12  3
   4  0 12
   5  2  3
```

Example 4–18 demonstrates SVM regression. The example creates a data frame, pushes it to a table, and then builds a regression model; note that `ore.odmSVM` specifies a linear kernel.

Example 4–19   Using the ore.odmSVM Function and Building a Regression Model

```
x <- seq(0.1, 5, by = 0.02)
y <- log(x) + rnorm(x, sd = 0.2)
dat <-ore.push(data.frame(x=x, y=y))

# Build model with linear kernel
svm.mod <- ore.odmSVM(y~x,dat,"regression", kernel.function="linear")
summary(svm.mod)
coef(svm.mod)
svm.res <- predict(svm.mod,dat, supplemental.cols="x")
head(svm.res,6)
```

Listing for Example 4–18

```
R> x <- seq(0.1, 5, by = 0.02)
R> y <- log(x) + rnorm(x, sd = 0.2)
R> dat <-ore.push(data.frame(x=x, y=y))
R>
R> # Build model with linear kernel
R> svm.mod <- ore.odmSVM(y~x,dat,"regression", kernel.function="linear")
R> summary(svm.mod)

Call:
ore.odmSVM(formula = y ~ x, data = dat, type = "regression",
    kernel.function = "linear")

Settings:
                      value
prep.auto                on
active.learning   al.enable
complexity.factor  0.620553
conv.tolerance        1e-04
epsilon            0.098558
kernel.function      linear

Residuals:
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.79130 -0.28210 -0.05592 -0.01420  0.21460  1.58400
```

```
Coefficients:
     variable value  estimate
1           x       0.6637951
2 (Intercept)       0.3802170

R> coef(svm.mod)
     variable value  estimate
1           x       0.6637951
2 (Intercept)       0.3802170
R> svm.res <- predict(svm.mod,dat, supplemental.cols="x")
R> head(svm.res,6)
     x PREDICTION
1 0.10 -0.7384312
2 0.12 -0.7271410
3 0.14 -0.7158507
4 0.16 -0.7045604
5 0.18 -0.6932702
6 0.20 -0.6819799
```

This example of SVN anomaly detection uses `mtcars_of` created in the classification example and builds an anomaly detection model.

Example 4–20 Using the ore.odmSVM Function and Building an Anomaly Detection Model

```
svm.mod  <- ore.odmSVM(~ .-ID, mtcars_of, "anomaly.detection")
summary(svm.mod)
svm.res  <- predict (svm.mod, mtcars_of, "ID")
head(svm.res)
table(svm.res$PREDICTION)
```

Listing for Example 4–18

```
R> svm.mod  <- ore.odmSVM(~ .-ID, mtcars_of, "anomaly.detection")
R> summary(svm.mod)

Call:
ore.odmSVM(formula = ~. - ID, data = mtcars_of, type = "anomaly.detection")

Settings:
                      value
prep.auto                on
active.learning   al.enable
conv.tolerance        1e-04
kernel.cache.size  50000000
kernel.function    gaussian
outlier.rate             .1
std.dev            0.719126

Coefficients:
[1] No coefficients with gaussian kernel

R> svm.res  <- predict (svm.mod, mtcars_of, "ID")
R> head(svm.res)
                        '0'       '1' ID PREDICTION
Mazda RX4         0.4999405 0.5000595  1          1
Mazda RX4 Wag     0.4999794 0.5000206  2          1
Datsun 710        0.4999618 0.5000382  3          1
Hornet 4 Drive    0.4999819 0.5000181  4          1
Hornet Sportabout 0.4949872 0.5050128  5          1
```

```
Valiant           0.4999415 0.5000585  6          1
R> table(svm.res$PREDICTION)

 0  1
 5 27
```

5 Predicting With R Models

Predictive models allow you to predict future behavior based on past behavior. After you build a model, you use it to score new data, that is, to make predictions.

R allows you to build many kinds of models. When you score data to predict new results using an R model, the data to score must be in an R `data.frame`. With the `ore.predict` function, you can use an R model to score database-resident data in an `ore.frame` object.

With the `ore.predict` function, you can only make predictions using `ore.frame` objects; you cannot rebuild the model. For scalability and performance, build models in the database table using the algorithms and functions described in Chapter 4, "Building Models in Oracle R Enterprise." These include both algorithms that are native to Oracle R Enterprise and those from Oracle Data Mining that are exposed in R.

The `ore.predict` function is a generic function. It has the following usage:

```
ore.predict(object, newdata, ...)
```

The value of the `object` argument is one of the R models or objects listed in Table 5–1. The value of the `newdata` argument is an `ore.frame` object that contains the data to score. The `OREpredict` package has methods for use with specific R model classes. The `...` argument represents the various additional arguments that are accepted by the different methods.

Table 5–1 lists the methods employed by the generic `ore.predict` function, the class of the object the method accepts as the `object` argument, and a description of the type of model or object.

Table 5–1 Methods of the Generic ore.predict Function

OREpredict Method	Class of Object	Description of Object
`ore.predict-glm`	`glm`	Generalized linear model
`ore.predict-kmeans`	`kmeans`	*k*-Means clustering model
`ore.predict-lm`	`lm`	Linear regression model
`ore.predict-matrix`	`matrix`	A `matrix` with no more than 1000 rows
`ore.predict-multinom`	`multinom`	Multinomial log-linear model
`ore.predict-nnet`	`nnet`	Neural network models
`ore.predict-ore.model`	`ore.model`	An Oracle R Enterprise model
`ore.predict-prcomp`	`prcomp`	Principal components analysis on a matrix

Table 5–1 (Cont.) Methods of the Generic ore.predict Function

OREpredict Method	Class of Object	Description of Object
`ore.predict-princomp`	`princomp`	Principal components analysis on a numeric matrix
`ore.predict-rpart`	`rpart`	Recursive partitioning and regression tree model

For the arguments of the `ore.predict` methods, invoke the `help` function on the method, such as `help("ore.predict-glm")`.

Example 5–1 builds a linear regression model, `irisModel`, using the `lm` function on the `iris` `data.frame`. It pushes the data set to the database as `iris_of`, an `ore.frame` object. It then scores the model by invoking `ore.predict` on it.

Example 5–1 Using the ore.predict Function on an LM Model

```
irisModel <- lm(Sepal.Length ~ ., data = iris)
iris_of <- ore.push(iris)
iris_of_pred <- ore.predict(irisModel, iris_of, se.fit = TRUE,
                            interval = "prediction")
iris_of <- cbind(iris_of, iris_of_pred)
head(iris_of)
```

Listing for Example 5–1

```
R> irisModel <- lm(Sepal.Length ~ ., data = iris)
R> iris_of <- ore.push(iris)
R> iris_of_pred <- ore.predict(irisModel, iris_of, se.fit = TRUE,
+                              interval = "prediction")
R> iris_of <- cbind(iris_of, iris_of_pred)
R> head(iris_of)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species     PRED    SE.PRED
1          5.1         3.5          1.4         0.2  setosa 5.004788 0.04479188
2          4.9         3.0          1.4         0.2  setosa 4.756844 0.05514933
3          4.7         3.2          1.3         0.2  setosa 4.773097 0.04690495
4          4.6         3.1          1.5         0.2  setosa 4.889357 0.05135928
5          5.0         3.6          1.4         0.2  setosa 5.054377 0.04736842
6          5.4         3.9          1.7         0.4  setosa 5.388886 0.05592364
  LOWER.PRED UPPER.PRED
1   4.391895   5.617681
2   4.140660   5.373027
3   4.159587   5.386607
4   4.274454   5.504259
5   4.440727   5.668026
6   4.772430   6.005342

R> head(iris_of)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species     PRED    SE.PRED
LOWER.PRED UPPER.PRED
1          5.1         3.5          1.4         0.2  setosa 5.004788 0.04479188
4.391895   5.617681
2          4.9         3.0          1.4         0.2  setosa 4.756844 0.05514933
4.140660   5.373027
3          4.7         3.2          1.3         0.2  setosa 4.773097 0.04690495
4.159587   5.386607
4          4.6         3.1          1.5         0.2  setosa 4.889357 0.05135928
4.274454   5.504259
5          5.0         3.6          1.4         0.2  setosa 5.054377 0.04736842
4.440727   5.668026
```

```
6          5.4         3.9          1.7         0.4  setosa 5.388886 0.05592364
4.772430   6.005342
```

Example 5–2 builds a generalized linear model using the infert data set and then invokes the `ore.predict` function on the model.

Example 5–2 Using the ore.predict Function on a GLM Model

```
infertModel <-
  glm(case ~ age + parity + education + spontaneous + induced,
  data = infert, family = binomial())
INFERT <- ore.push(infert)
INFERTpred <- ore.predict(infertModel, INFERT, type = "response",
                          se.fit = TRUE)
INFERT <- cbind(INFERT, INFERTpred)
head(INFERT)
```

Listing for Example 5–2

```
R> infertModel <-
+   glm(case ~ age + parity + education + spontaneous + induced,
+   data = infert, family = binomial())
R> INFERT <- ore.push(infert)
R> INFERTpred <- ore.predict(infertModel, INFERT, type = "response",
+                           se.fit = TRUE)
R> INFERT <- cbind(INFERT, INFERTpred)
R> head(INFERT)
  education age parity induced case spontaneous stratum pooled.stratum
1    0-5yrs  26      6       1    1           2       1              3
2    0-5yrs  42      1       1    1           0       2              1
3    0-5yrs  39      6       2    1           0       3              4
4    0-5yrs  34      4       2    1           0       4              2
5   6-11yrs  35      3       1    1           1       5             32
6   6-11yrs  36      4       2    1           1       6             36
       PRED    SE.PRED
1 0.5721916 0.20630954
2 0.7258539 0.17196245
3 0.1194459 0.08617462
4 0.3684102 0.17295285
5 0.5104285 0.06944005
6 0.6322269 0.10117919
```

6 Using Oracle R Enterprise Embedded R Execution

Embedded R execution is a significant feature of Oracle R Enterprise. This chapter discusses embedded R execution in the following topics:

- About Oracle R Enterprise Embedded R Execution
- Security Considerations for Scripts
- Support for Parallel Execution
- Installing a Third-Party Package for Use in Embedded R Execution
- R Interface for Embedded R Execution
- SQL Interface for Embedded R Execution

About Oracle R Enterprise Embedded R Execution

In Oracle R Enterprise, embedded R execution is the ability to store R scripts in Oracle Database and to invoke such scripts, which then execute in one or more R engines that run in the database and that are dynamically started and managed by the database. Oracle R Enterprise provides both an R interface and a SQL interface for embedded R execution. From the same R script you can get structured data, an XML representation of R objects and images, and even PNG images through a BLOB column in a database table.

This section has the following topics:

- Benefits of Embedded R Execution
- APIs for Embedded R Execution

Benefits of Embedded R Execution

Embedded R execution has the following benefits:

- Eliminates moving data from the Oracle Database server to your local R session.

  As well as being more secure, the transfer of database data between Oracle Database and an internal R engine is much faster than to a separate client R engine.
- Uses the database server to start, manage, and control the execution of R scripts in database-side R engines.
- Leverages the memory and processing power of the database server machine for R engine execution, which provides better scalability and performance.

- Enables data-parallel and task-parallel execution of user-defined R functions that correspond to special cases of Hadoop Map-Reduce jobs.
- Provides parallel simulations capability.
- Allows the use of open source CRAN packages at the database server machine.
- Provides the ability to develop and operationalize comprehensive scripts for analytical applications in a single step, without leaving the R environment.

  You can directly integrate R scripts used in exploratory analysis into application tasks. You can also immediately invoke R scripts in production to drastically reduce time to market by eliminating porting and enabling instantaneous updates of changes to application code.
- Executing R scripts from SQL enables integration of R script results with Oracle Business Intelligence Enterprise Edition (OBIEE), Oracle BI Publisher, and other SQL-enabled tools for structured data, R objects, and images.

APIs for Embedded R Execution

Oracle R Enterprise provides R and SQL application programming interfaces for embedded R execution. Table 6–1 provides a summary of the embedded R execution functions and the R script repository functions available. The function *f* refers to the user-defined R code, or script, that is provided as either an R function object or a named R function in the database R script repository.

Table 6–1 R and SQL APIs for Embedded R Execution

R API	SQL API	Description
`ore.doEval`	`rqEval`	Executes *f* with no automatic transfer of data.
`ore.tableApply`	`rqTableEval`	Executes *f* by passing all rows of the provided input `ore.frame` as the first argument of *f*. Provides the first argument of *f* as a `data.frame`.
`ore.groupApply`	*rqGroupEval* This function must be explicitly defined by the user.	Executes *f* by partitioning data according to the values of a grouping column. Provides each data partition as a `data.frame` in the first argument of *f*. Supports parallel execution of each *f* invocation in the pool of database server-side R engines.
`ore.rowApply`	`rqRowEval`	Executes *f* by passing a specified number of rows (a *chunk*) of the provided input `ore.frame`. Provides each chunk as a `data.frame` in the first argument of *f*. Supports parallel execution of each *f* invocation in the pool of database server-side R engines.
`ore.indexApply`	No equivalent.	Executes *f* with no automatic transfer of data, but provides the index of the invocation, 1 through *n*, where *n* is the number of functions to invoke. Supports parallel execution of each *f* invocation in the pool of database server-side R engines.
`ore.scriptCreate`	`sys.rqScriptCreate`	Loads the provided R function into the R script repository with the provided name.
`ore.scriptDrop`	`sys.rqScriptDrop`	Removes the named R function from the R script repository.

Security Considerations for Scripts

Because both R scripts and SQL scripts allow access to the database server, the creation of scripts must be controlled. The RQADMIN role is a collection of Oracle Database privileges that a user must have to create scripts and store them in the Oracle Database R script repository or drop scripts from the repository.

The installation of Oracle R Enterprise creates the RQADMIN role. The role must be explicitly granted to a user. To grant RQADMIN to a user, start SQL*Plus as `sysdba` and enter a `GRANT` statement such as the following, which grants the role to the user RQUSER:

```
GRANT RQADMIN to RQUSER
```

Note: You should grant RQADMIN only to those users who need it.

See Also:

- "APIs for Embedded R Execution" on page 6-2
- "Using the ore.scriptCreate and ore.scriptDrop Functions" on page 6-26
- "Registering and Managing SQL Scripts" on page 6-27

Support for Parallel Execution

Some of the Oracle R Enterprise embedded R execution functions support the use of parallel execution in the database. The `ore.groupApply` and `ore.rowApply` functions support data-parallel execution and the `ore.indexApply` function supports task-parallel execution. This parallel execution capability enables a script to take advantage of high-performance computing hardware such as an Oracle Exadata Database Machine.

The `parallel` argument of these functions specifies the degree of parallelism to use in the embedded R execution. The value of the argument can be one of the following:

- A positive integer greater than or equal to `2` for a specific degree of parallelism
- `FALSE` or `1` for no parallelism
- `TRUE` for the default parallelism of the `data` argument
- `NULL` for the database default for the operation

The default value of the argument is the value of the global option `ore.parallel` or `FALSE` if `ore.parallel` is not set.

A user-defined R function invoked using `ore.doEval` or `ore.tableApply` is not executed in parallel. The function executes in a single R engine.

In data-parallel execution for the `ore.groupApply` function, one or more R engines perform the same R function, or task, on different partitions of data. This functionality enables the building of large numbers of models, for example building tens or hundreds of thousands of predictive models, one model per customer.

In data-parallel execution for the `ore.rowApply` function, one or more R engines performs the same R function on disjoint chunks of data. This functionality enables scalable model scoring and predictions on large data sets.

In task-parallel execution for the `ore.indexApply` function, one or more R engines perform the same or different calculations, or task. A number, associated with the index of the execution, is provided to the function. This functionality is valuable in a variety of operations, such as in performing simulations.

Oracle Database handles the management and control of potentially multiple R engines at the database server, automatically partitioning and passing data to R engines executing in parallel. It ensures that all of the R function executions for all of the partitions complete; if not, the Oracle R Enterprise function returns an error. The result from the execution of each user-defined embedded R function is gathered in an `ore.list`. This list remains in the database until the user requires the result.

Embedded R execution also allows for data-parallel execution of user-defined R functions that may use functions from an open source R package from the Comprehensive R Archive Network (CRAN) or other third-party R package. However, third-party packages do not leverage in-database parallelism and are subject to the parallelism constraints of R. Third-party packages can benefit from the data-parallel and task-parallel execution supported in embedded R execution.

See Also: "Oracle R Enterprise Global Options" on page 1-12

Installing a Third-Party Package for Use in Embedded R Execution

Embedded R execution allows the use of CRAN or other third-party packages in user-defined R functions executed on the Oracle Database server. To use a third-party package in embedded R execution, the package must be installed on the database server. If you are going to use the package from the R interface for embedded R execution, then the package must also be installed on the client, as well.

Embedded R execution enables user-defined R functions to use CRAN packages; however, open source R packages do not leverage in-database parallelism and are subject to the parallelism constraints of R. CRAN packages can benefit through the data-parallel and task-parallel execution supported by embedded R execution.

Embedded R execution leverages what is likely a larger amount of memory and number of processors on the database server, such as an Oracle Exadata Database Machine, than is available on a typical R client machine. Embedded R execution provides for a more efficient transfer of data between the database and the R engine because they are on the same machine.

You can install a third-party package in an R session or from the command line. For information on installing a package in an R session, see "Using a Third-Party Package on the Client" on page 3-37.

To install a package on the server so that it can be used by any R user and for use in embedded R execution, an Oracle Database Administrator (DBA) typically executes commands from the command line or in an administrative tool.

A DBA would typically do the following:

1. Download the package source from CRAN using `wget`. If the package depends on any packages that are not in the R distribution in use, then download the sources for those packages, also.
2. Do one of the following:
   - For a single Oracle Database instance, use the `ORE CMD INSTALL` command to install the package or packages in the same location as the Oracle R Enterprise packages, which is `$ORACLE_HOME/R/library`.
   - For installing a package on multiple database servers, such as those in an Oracle Real Application Clusters (Oracle RAC) or a multinode Oracle Exadata Database Machine environment, use the Exadata Distributed Command Line Interface (DCLI) utility. To install a package on a single node, use the `ORE CMD INSTALL` command.

Example 6–1 demonstrates getting the source for the `arules` package from CRAN and installing it with `ORE CMD INSTALL` from a Linux command line.

Example 6–1 Installing a Package for a Single Database

```
wget http://cran.r-project.org/src/contrib/arules_1.1-1.tar.gz
ORE CMD INSTALL arules_1.1-1.tar.gz
```

Example 6–2 shows the DLCI command for installing the `arules` package.

Example 6–2 Installing a Package Using DCLI

```
dcli -t -g nodes -l oracle R CMD INSTALL arules_1.1-1.tar.gz
```

See Also:

- *Oracle R Enterprise Installation and Administration Guide* for information about setting up DCLI and about installing packages
- "Using a Third-Party Package on the Client" on page 3-37
- http://www.r-bloggers.com/installing-r-packages/

R Interface for Embedded R Execution

Oracle R Enterprise provides functions that invoke R scripts that run in one or more R engines that are embedded in the Oracle database. Other functions create R scripts that are stored in a database script repository or drop a script from the repository. This section describes those functions. It contains the following topics:

- Arguments for Functions that Run Scripts
- Automatic Database Connection in Embedded R Scripts
- Using the ore.doEval Function
- Using the ore.tableApply Function
- Using the ore.groupApply Function
- Using the ore.rowApply Function
- Using the ore.indexApply Function
- Using the ore.scriptCreate and ore.scriptDrop Functions

Arguments for Functions that Run Scripts

The Oracle R Enterprise embedded R execution functions `ore.doEval`, `ore.tableApply`, `ore.groupApply`, `ore.rowApply`, and `ore.indexApply` have arguments that are common to some or all of the functions. Some of the functions also have an argument that is unique to the function.

This section describes the arguments in the following topics:

- Input Function to Execute
- Optional Arguments
- Structure of Return Value
- Input Data
- Parallel Execution
- Unique Arguments

See Also:

- For function signatures and more details about function arguments, see the online help displayed by invoking `help(ore.doEval)`
- For examples of the use of the arguments, see "Using the ore.doEval Function" on page 6-9 and the other topics on using the embedded R execution functions.

Input Function to Execute

The embedded R execution functions all require a function to apply during the execution of the script. You specify the input function with one of the following mutually exclusive arguments:

- `FUN`
- `FUN.NAME`

The `FUN` argument takes a function object as a directly specified function or as one assigned to an R variable. Only a user with the RQADMIN role can use the `FUN` argument.when invoking an embedded R function.

The `FUN.NAME` argument specifies a script that is stored in the R script repository. A stored script contains the function to apply when the script runs. Any Oracle R Enterprise user can use the `FUN.NAME` argument when invoking an embedded R function.

The advanced Oracle R Enterprise analytics functions in the `OREmodels` package, `ore.glm`, `ore.lm`, and `ore.neural`, use the embedded R execution framework internally and cannot be used in embedded R execution functions.

Optional Arguments

All of the embedded R execution functions also take optional arguments, which can be named or not. Oracle R Enterprise passes user-defined optional arguments to the input function. You can pass any number of optional arguments to the input function, including complex R objects such as models.

Arguments that start with `ore.` are special control arguments. Oracle R Enterprise does not pass them to the input function, but instead uses them to control what happens before or after the execution of that function. The following control arguments are supported:

- `ore.connect` controls whether to automatically connect to Oracle R Enterprise inside the embedded R execution function. This is equivalent to doing an `ore.connect` call with the same credentials as the client session. The default value is `FALSE`.

  See Also: "Automatic Database Connection in Embedded R Scripts" on page 6-8

- `ore.drop` controls the input data. If `TRUE`, a one column `data.frame` is converted to a vector. The default value is `TRUE`.
- `ore.na.omit` controls the handling of missing values in the input data. If `TRUE`, rows or vector elements, depending on the `ore.drop` setting, that contain missing values are removed from the input data. If all of the rows in a chunk contain missing values, then the input data for that chunk will be an empty `data.frame` or `vector`. The default value is `FALSE`.

- `ore.graphics` controls whether to start a graphical driver and look for images. The default value is `TRUE`.
- `ore.png.*` specifies additional arguments for the `png` graphics driver if `ore.graphics` is `TRUE`. The naming convention for these arguments is to add an `ore.png.` prefix to the arguments of the `png` function. For example, if `ore.png.height` is supplied, argument `height` will be passed to the `png` function. If not set, the standard default values for the `png` function are used.

Structure of Return Value

Another argument that applies to all of the embedded R execution functions is `FUN.VALUE`. If the `FUN.VALUE` argument is `NULL`, then the `ore.doEval` and `ore.tableApply` function can return a serialized R object as an `ore.object` class object, and the `ore.groupApply`, `ore.indexApply`, and `ore.rowApply` functions return an `ore.list` object. However, if you specify a `data.frame` or an `ore.frame` with the `FUN.VALUE` argument, then the function returns an `ore.frame` that has the structure of the specified `data.frame` or `ore.frame` object.

Input Data

The `ore.doEval`,and `ore.indexApply` functions do not automatically receive any data from the database. They simply execute the function specified by the `FUN` or `FUN.NAME` argument. Any data needed by the input function is either generated within that function or explicitly retrieved from a data source such as Oracle Database, other databases, or flat files. The input function can load data from a file or a table using the `ore.pull` function or other transparency layer function.

The `ore.tableApply`, `ore.groupApply`, and `ore.rowApply` functions require a database table as input data. The table is represented by an `ore.frame`. You supply that data with an `ore.frame` object that you specify with the `X` argument, which is the first argument to the embedded R execution function. The embedded R execution function passes the `ore.frame` object to the user-defined input function as the first argument to that function.

> **Note:** The data represented by the `ore.frame` object passed to the user-defined R function is copied from Oracle Database to the database server R engine. The R memory limitations apply. If your database server machine has 32 GB RAM and your data table is 64 GB, then Oracle R Enterprise cannot load the data into the R engine memory.

The embedded R execution function passes the `ore.frame` object input data provided as the first argument to a user-defined R function invoked using `ore.tableApply` is physically being moved from Oracle Database to the database server R engine. Note that the R memory limitations still apply. If your database server machine has 32 GB RAM and your data table is 64 GB, Oracle R Enterprise cannot load the data into the R engine memory.

Parallel Execution

The `ore.groupApply`, `ore.indexApply`, and `ore.rowApply` functions take the `parallel` argument. That argument specifies the degree of parallelism to use in the embedded R execution of the input function. See "Support for Parallel Execution" on page 6-3.

Unique Arguments

The `ore.groupApply`, `ore.indexApply`, and `ore.rowApply` functions each take a argument unique to the function.

The `ore.groupApply` function takes the `INDEX` argument, which specifies the name of a column by which the rows of the input data are partitioned for processing by the input function.

The `ore.indexApply` function takes the `times` argument, which specifies the number of times to execute the input function.

The `ore.rowApply` function tales the `rows` argument, which specifies the number of rows to pass to each invocation of the input function.

Automatic Database Connection in Embedded R Scripts

An embedded R script can automatically connect to an Oracle database.

If automatic connections are enabled, the following functionality occurs:

- Embedded R scripts are automatically connected to the database.
- The automatic connection has the same credentials as the session that invokes the embedded R SQL functions.
- The script runs in an autonomous transaction.
- ROracle queries work with the automatic connection.
- Oracle R Enterprise transparency is enabled in the embedded script.
- User and site-wide R profile loading is disabled in embedded R.

  Profile loading was supported in earlier Oracle R Enterprise releases. An automatic connection provides a more secure connection.

Automatic connections are disabled by default. You can specify whether automatic connections are enabled or disabled by using the `ore.connect` control argument. Control arguments are documented in R online Help for `ore.doEval`.

To enable automatic connections, ROracle was extended by adding a new driver `ExtDriver` with the constructor `Extproc` that is initialized by passing an external pointer wrapping the `extproc` context. Similarly to `OraDriver`, `ExtDriver` is a singleton. Both drivers can exist simultaneously in a session because they are represented by two distinct singletons. This setup allows working with `extproc` and explicit `OraDriver` connections in the same R script as shown by the following example.

```
ore.doEval(function() {
  ore.disconnect()
  con1 <- dbConnect(Extproc())
  res1 <- dbGetQuery(con1, "select * from grade order by name")
  con2 <- dbConnect(Oracle(), "scott", "tiger")
  res2 <- dbGetQuery(con2, "select * from emp order by empno")
  dbDisconnect(con1)
  dbDisconnect(con2)
  cbind(head(res1)[,1:3], head(res2)[,1:3])
} }, ore.connect = TRUE)
```

Using the ore.doEval Function

The `ore.doEval` function executes the specified input function using data that is generated by the input function. It returns an `ore.frame` object or a serialized R object as an `ore.object` object.

See Also: "Arguments for Functions that Run Scripts" on page 6-5

Example 6–3 creates the `function` object `RandomRedDots` that gets a function that has an argument and that returns a `data.frame` object that has two columns and that plots 100 random normal values. The example then invokes `ore.doEval` function and passes it the `RandomRedDots` object. The image is displayed at the client, but it is generated by the database server R engine that executed the `RandomRedDots` function.

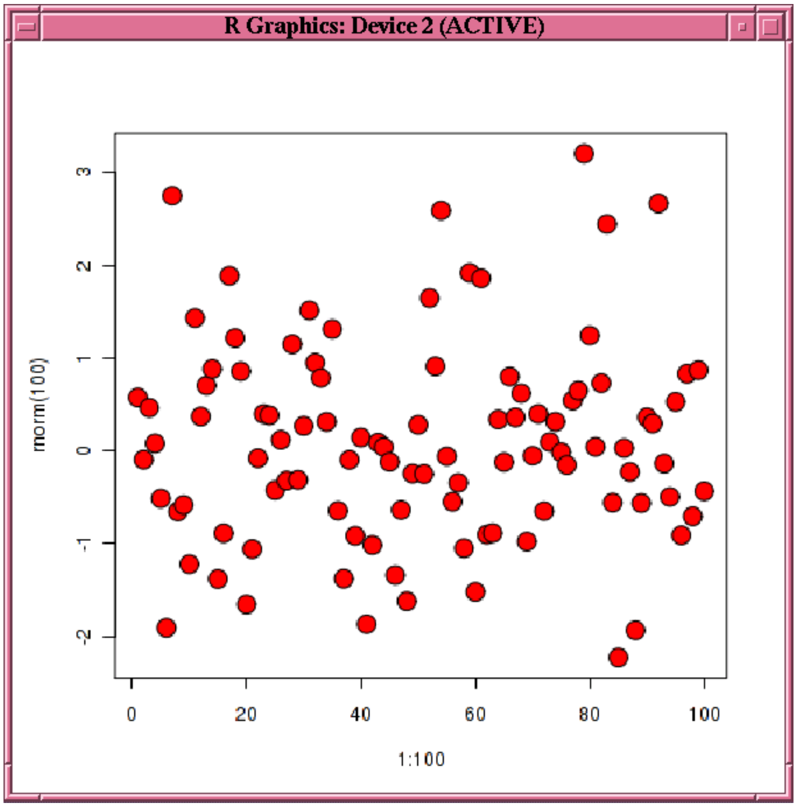
Example 6–3 Using the ore.doEval Function

```
RandomRedDots <- function(divisor=100){
  id<- 1:10
  plot(1:100, rnorm(100), pch = 21, bg = "red", cex = 2 )
  data.frame(id=id, val=id / divisor)
}
ore.doEval(RandomRedDots)
```

Listing for Example 6–3

```
R> RandomRedDots <- function(divisor=100){
+   id<- 1:10
+   plot(1:100, rnorm(100), pch = 21, bg = "red", cex = 2 )
+   data.frame(id=id, val=id / divisor)
+ }
R> ore.doEval(RandomRedDots)
   id  val
1   1 0.01
2   2 0.02
3   3 0.03
4   4 0.04
5   5 0.05
6   6 0.06
7   7 0.07
8   8 0.08
9   9 0.09
10 10 0.10
```

Figure 6–1 Display of Random Red Dots



You can provide arguments to the input function as optional arguments to the `doEval` function. Example 6–4 invokes the `doEval` function with an optional argument that overrides the divisor argument of the `RandomRedDots` function.

Example 6–4 Using the ore.doEval Function with an Optional Argument

```
ore.doEval(RandomRedDots, divisor=50)
```

Listing for Example 6–4

```
R> ore.doEval(RandomRedDots, divisor=50)
   id  val
1   1 0.02
2   2 0.04
3   3 0.06
4   4 0.08
5   5 0.10
6   6 0.12
7   7 0.14
8   8 0.16
9   9 0.18
10 10 0.20
# The graph displayed by the plot function is not shown.
```

If the input function is stored in the R script repository, then you can invoke the `ore.doEval` function with the `FUN.NAME` argument. Example 6–5 invokes the `ore.doEval` function and specifies `myRandomRedDots`, which is the `RandomRedDots` function that was added to the R script repository by that name. The result is assigned to the variable `res`.

The return value of the RandomRedDots function is a data.frame but in Example 6–5 the ore.doEval function returns an ore.object object. To get back the data.frame object, the example invokes ore.pull to pull the result to the client R engine.

Example 6–5 Using the ore.doEval Function with the FUN.NAME Argument

```
res <- ore.doEval(FUN.NAME="myRandomRedDots",divisor=50)
class(res)
res.local <- ore.pull(res)
class(res.local)
```

Listing for Example 6–5

```
R> res <- ore.doEval(FUN.NAME = "myRandomRedDots", divisor = 50)
R> class(res)
[1] "ore.object"
attr(,"package")
[1] "OREembed"
R> res.local <- ore.pull(res)
R> class(res.local)
[1] "data.frame"
```

To have the doEval function return an ore.frame object instead of an ore.object, specify the argument FUN.VALUE to describes the structure of the result, as shown in Example 6–6.

Example 6–6 Using the ore.doEval Function with the FUN.VALUE Argument

```
res.of <- ore.doEval(FUN.NAME="myRandomRedDots", divisor=50,
                     FUN.VALUE= data.frame(id=1, val=1))
class(res.of)
```

Listing for Example 6–6

```
R> res.of <- ore.doEval(FUN.NAME="myRandomRedDots", divisor=50,
+                         FUN.VALUE= data.frame(id=1, val=1))
R> class(res.of)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
```

To establish a connection to Oracle Database within the input function, set the special optional argument ore.connect TRUE. This uses the credentials of the user who invoked the function ore.doEval to establish a connection and also automatically load the Oracle R Enterprise package. This capability can be useful to explicitly use the Oracle R Enterprise transparency layer or to save and load objects from an Oracle R Enterprise datastore.

Example 6–7 creates the RandomRedDots function object as in Example 6–3 but this time the function has an argument that takes the name of a datastore. The example creates the myVar variable and saves it in the datastore named datastore_1. The example then invokes the doEval function and passes it the name of the datastore and specifies the ore.connect argument.

Example 6–7 Using the doEval Function with the ore.connect Argument

```
RandomRedDots <- function(divisor=100, datastore.name="myDatastore"){
  id <- 1:10
  plot(1:100, rnorm(100), pch = 21, bg = "red", cex = 2 )
  ore.load(datastore.name) # contains numeric variable myVar
  data.frame(id=id, val=id / divisor, num=myVar)
```

```
}
myVar <- 5
ore.save(myVar, name = "datastore_1")
ore.doEval(RandomRedDots, datastore.name="datastore_1", ore.connect=TRUE)
```

Listing for Example 6–7

```
R> RandomRedDots <- function(divisor=100, datastore.name="myDatastore"){
+   id <- 1:10
+   plot(1:100, rnorm(100), pch = 21, bg = "red", cex = 2 )
+   ore.load(datastore.name) # contains numeric variable myVar
+   data.frame(id=id, val=id / divisor, num=myVar)
+ }
R> ore.doEval(RandomRedDots, datastore.name="datastore_1", ore.connect=TRUE)
   id  val num
1   1 0.01   5
2   2 0.02   5
3   3 0.03   5
4   4 0.04   5
5   5 0.05   5
6   6 0.06   5
7   7 0.07   5
8   8 0.08   5
9   9 0.09   5
10 10 0.10   5
# The graph displayed by the plot function is not shown.
```

Using the ore.tableApply Function

The `ore.tableApply` function invokes an R script with an `ore.frame` as the input data. The `ore.tableApply` function passes the `ore.frame` to the user-defined input function as the first argument to that function. The `ore.tableApply` function returns an `ore.frame` object or a serialized R object as an `ore.object` object.

See Also:

- "Arguments for Functions that Run Scripts" on page 6-5
- "Installing a Third-Party Package for Use in Embedded R Execution" on page 6-4

Example 6–8 uses the `ore.tableApply` function to build a Naive Bayes model on the `iris` data set. The `naiveBayes` function is in the `e1071` package, which must be installed on both the client and database server machine R engines. As the first argument to the `ore.tableApply` function, the `ore.push(iris)` invocation creates a temporary database table and an `ore.frame` that is a proxy for the table. The second argument is the input function, which has as an argument `dat`. The `ore.tableApply` function passes the `ore.frame` to the input function as the `dat` argument. The input function creates a model, which the `ore.tableApply` function returns as an `ore.object` object.

Example 6–8 Using the ore.tableApply Function

```
library(e1071)
mod <- ore.tableApply(
  ore.push(iris),
  function(dat) {
    library(e1071)
    dat$Species <- as.factor(dat$Species)
    naiveBayes(Species ~ ., dat)
```

```
})
class(mod)
mod
```

Listing for Example 6–8

```
R> mod <- ore.tableApply(
+   ore.push(iris),
+   function(dat) {
+     library(e1071)
+     dat$Species <- as.factor(dat$Species)
+     naiveBayes(Species ~ ., dat)
+ })
R> class(mod)
[1] "ore.object"
attr(,"package")
[1] "OREembed"
R> mod

Naive Bayes Classifier for Discrete Predictors

Call:
naiveBayes.default(x = X, y = Y, laplace = laplace)

A-priori probabilities:
Y
    setosa versicolor  virginica
 0.3333333  0.3333333  0.3333333

Conditional probabilities:
            Sepal.Length
Y              [,1]      [,2]
  setosa     5.006 0.3524897
  versicolor 5.936 0.5161711
  virginica  6.588 0.6358796

            Sepal.Width
Y              [,1]      [,2]
  setosa     3.428 0.3790644
  versicolor 2.770 0.3137983
  virginica  2.974 0.3224966

            Petal.Length
Y              [,1]      [,2]
  setosa     1.462 0.1736640
  versicolor 4.260 0.4699110
  virginica  5.552 0.5518947

            Petal.Width
Y              [,1]      [,2]
  setosa     0.246 0.1053856
  versicolor 1.326 0.1977527
  virginica  2.026 0.2746501
```

Using the ore.groupApply Function

The `ore.groupApply` function invokes an R script with an `ore.frame` as the input data. The `ore.groupApply` function passes the `ore.frame` to the user-defined input function as the first argument to that function. The `INDEX` argument to the `ore.groupApply` function specifies the name of a column of the `ore.frame` by which Oracle Database partitions the rows for processing by the user-defined R function. The `ore.groupApply` function can use data-parallel execution, in which one or more R engines perform the same R function, or task, on different partitions of data.

The `ore.groupApply` function returns an `ore.list` object or an `ore.frame` object.

Examples of the use of the `ore.groupApply` function are in the following topics:

- Partitioning on a Single Column
- Partitioning on Multiple Columns

See Also:

- "Arguments for Functions that Run Scripts" on page 6-5
- "Installing a Third-Party Package for Use in Embedded R Execution" on page 6-4

Partitioning on a Single Column

Example 6–9 uses the `C50` package to build a C5.0 decision tree model on the `churn` data set from `C50`, with the goal of building one churn model on the data for each state. The example does the following:

- Loads the `C50` package and then the `churn` data set.
- Uses the `ore.create` function to create a database table and the proxy `ore.frame` object from `churnTrain`, a `data.frame` object.
- Specifies `CHURN_TRAIN`, an `ore.frame` object, as the first argument to the `ore.groupApply` function and specifies the `state` column as the `INDEX` argument. The `ore.groupApply` function partitions the data on the `state` column and invokes the user-defined function on each partition.
- Specifies the user-defined function. The first argument of the user-defined function receives one partition of the data, which in this case is all of the data associated with a single state.
- The user-defined function does the following:
  - Loads the C50 package so that it is available to the function when it executes in an R engine in the database.
  - Deletes the `state` column from the `data.frame` so that the column is not included in the model.
  - Convert the columns to factors because, although the `ore.frame` defined factors, when they are loaded to the user-defined function, factors appear as character vectors.
  - Builds a model for a state and returns it.
- The `ore.groupApply` function returns a list that contains the results from the execution of the user-defined function on each partition of the data. In this case, it is one C5.0 model per state.
- The example creates the variable `modList`, which gets the `ore.list` object returned by the `ore.groupApply` function. The `ore.list` object contains the results from the execution of the user-defined function on each partition of the data. In this case, it is one C5.0 model per state, each model stored as `ore.object` object.
- Uses the `ore.pull` function to retrieve the model from the database as the `mod.MA` variable and then invokes the `summary` function on it. The class of `mod.MA` is `C5.0`.

Example 6–9 Using the ore.groupApply Function

```
library(C50)
data("churn")

ore.create(churnTrain, "CHURN_TRAIN")

modList <- ore.groupApply(
  CHURN_TRAIN,
  INDEX=CHURN_TRAIN$state,
    function(dat) {
      library(C50)
      dat$state <- NULL
      dat$churn <- as.factor(dat$churn)
      dat$area_code <- as.factor(dat$area_code)
      dat$international_plan <- as.factor(dat$international_plan)
      dat$voice_mail_plan <- as.factor(dat$voice_mail_plan)
      C5.0(churn ~ ., data = dat, rules = TRUE)
    });
mod.MA <- ore.pull(modList$MA)
summary(mod.MA)
```

Listing for Example 6–9

```
R> library(C50)
R> data(churn)
R>
R> ore.create(churnTrain, "CHURN_TRAIN")
R>
R> modList <- ore.groupApply(
+   CHURN_TRAIN,
+   INDEX=CHURN_TRAIN$state,
+     function(dat) {
+       library(C50)
+       dat$state <- NULL
+       dat$churn <- as.factor(dat$churn)
+       dat$area_code <- as.factor(dat$area_code)
+       dat$international_plan <- as.factor(dat$international_plan)
+       dat$voice_mail_plan <- as.factor(dat$voice_mail_plan)
+       C5.0(churn ~ ., data = dat, rules = TRUE)
+     });
R> mod.MA <- ore.pull(modList$MA)
R> summary(mod.MA)

Call:
C5.0.formula(formula = churn ~ ., data = dat, rules = TRUE)


C5.0 [Release 2.07 GPL Edition]         Thu Feb 13 15:09:10 2014
-------------------------------

Class specified by attribute `outcome'

Read 65 cases (19 attributes) from undefined.data

Rules:

Rule 1: (52/1, lift 1.2)
        international_plan = no
        total_day_charge <= 43.04
        ->  class no  [0.963]
```

```
Rule 2: (5, lift 5.1)
        total_day_charge > 43.04
        ->  class yes  [0.857]

Rule 3: (6/1, lift 4.4)
        area_code in {area_code_408, area_code_415}
        international_plan = yes
        ->  class yes  [0.750]

Default class: no


Evaluation on training data (65 cases):

                Rules
          ----------------
            No      Errors

             3    2( 3.1%)   <<


           (a)   (b)    <-classified as
          ----  ----
            53     1    (a): class no
             1    10    (b): class yes


        Attribute usage:

         89.23% international_plan
         87.69% total_day_charge
          9.23% area_code


Time: 0.0 secs
```

Partitioning on Multiple Columns

The `ore.groupApply` function takes only a single column for the `INDEX` argument; however, you can create a new column that is the concatenation of the columns you want to use and provide this new column to the `INDEX` argument.

Example 6–10 uses data from the `CHURN_TRAIN` data set to build an `rpart` model that produces rules on the partitions of data specified, which are the `voice_mail_plan` and `international_plan` columns. The example uses the R `table` function to show the number of rows to expect in each partition. It then adds a new column that pastes together the two columns of interest to create a new column named `vmp_ip`.g1

The example next invokes the `ore.scriptDrop` to ensure that no script by the specified name exists in the R script repository. It then uses the `ore.scriptCreate` function to define a script named `my.rpartFunction` and to store it in the repository. The stored script defines a function that takes a data source and a prefix to use for naming Oracle R Enterprise datastore objects. Each invocation of the function `my.rpartFunction` receives data from one of the partitions identified in `vmp_ip`. Because the source partition columns are constants, the function sets them to `NULL`. It converts the character vectors to factors, builds a to predict churn, and saves in it an appropriately named datastore. The function creates a list to return the specific partition column values, the distribution of churn values, and the model itself.

The example then loads the `rpart` library, sets the datastore prefix, and invokes `ore.groupApply` using the derived column `vmp_ip` as the input to argument `INDEX` and `my.rpartFunction` as the input to argument `FUN.NAME` to invoke the user-defined function stored in the R script repository. The `ore.groupApply` function uses an optional argument to pass the `datastorePrefix` variable to the user-defined function. It uses the optional argument `ore.connect` to connect to the database when executing the user-defined function. The `ore.groupApply` function returns an `ore.list` object as the variable `res`.

The example displays the first entry in the list returned. It then invokes the `ore.load` function to load the model for the case where the customer has both the voice mail plan and the international plan.

Example 6–10 Using ore.groupApply for Partitioning Data on Multiple Columns

```
library(C50)
data(churn)
ore.drop("CHURN_TRAIN")
ore.create(churnTrain, "CHURN_TRAIN")

table(CHURN_TRAIN$international_plan, CHURN_TRAIN$voice_mail_plan)
CT <- CHURN_TRAIN
CT$vmp_ip <- paste(CT$voice_mail_plan,CT$international_plan,sep="-")
options(width = 80)
head(CT, 3)

ore.scriptDrop("my.rpartFunction")
ore.scriptCreate("my.rpartFunction",
  function(dat,datastorePrefix) {
    library(rpart)
    vmp <- dat[1,"voice_mail_plan"]
    ip <- dat[1,"international_plan"]
    datastoreName <- paste(datastorePrefix,vmp,ip,sep="_")
    dat$voice_mail_plan <- NULL
    dat$international_plan <- NULL
    dat$state <- as.factor(dat$state)
    dat$churn <- as.factor(dat$churn)
    dat$area_code <- as.factor(dat$area_code)
    mod <- rpart(churn ~ ., data = dat)
    ore.save(mod, name=datastoreName, overwrite=TRUE)
    list(voice_mail_plan=vmp,
        international_plan=ip,
        churn.table=table(dat$churn),
        rpart.model = mod)
  })

library(rpart)
datastorePrefix="my.rpartModel"

res <- ore.groupApply( CT, INDEX=CT$vmp_ip,
      FUN.NAME="my.rpartFunction",
      datastorePrefix=datastorePrefix,
      ore.connect=TRUE)
res[[1]]
ore.load(name=paste(datastorePrefix,"yes","yes",sep="_"))
mod
```

Listing for Example 6–10

```
R> library(C50)
```

```
R> data(churn)
R> ore.drop("CHURN_TRAIN")
R> ore.create(churnTrain, "CHURN_TRAIN")
R>
R> table(CHURN_TRAIN$international_plan, CHURN_TRAIN$voice_mail_plan)

        no  yes
  no  2180  830
  yes  231   92
R> CT <- CHURN_TRAIN
R> CT$vmp_ip <- paste(CT$voice_mail_plan,CT$international_plan,sep="-")
R> options(width = 80)
R> head(CT, 3)
  state account_length     area_code international_plan voice_mail_plan
1    KS            128 area_code_415                 no             yes
2    OH            107 area_code_415                 no             yes
3    NJ            137 area_code_415                 no              no
  number_vmail_messages total_day_minutes total_day_calls total_day_charge
1                    25             265.1             110            45.07
2                    26             161.6             123            27.47
3                     0             243.4             114            41.38
  total_eve_minutes total_eve_calls total_eve_charge total_night_minutes
1             197.4              99            16.78               244.7
2             195.5             103            16.62               254.4
3             121.2             110            10.30               162.6
  total_night_calls total_night_charge total_intl_minutes total_intl_calls
1                91              11.01               10.0                3
2               103              11.45               13.7                3
3               104               7.32               12.2                5
  total_intl_charge number_customer_service_calls churn vmp_ip
1              2.70                             1    no yes-no
2              3.70                             1    no yes-no
3              3.29                             0    no  no-no
R>
R> ore.scriptDrop("my.rpartFunction")
R> ore.scriptCreate("my.rpartFunction",
+   function(dat,datastorePrefix) {
+     library(rpart)
+     vmp <- dat[1,"voice_mail_plan"]
+     ip <- dat[1,"international_plan"]
+     datastoreName <- paste(datastorePrefix,vmp,ip,sep="_")
+     dat$voice_mail_plan <- NULL
+     dat$international_plan <- NULL
+     dat$state <- as.factor(dat$state)
+     dat$churn <- as.factor(dat$churn)
+     dat$area_code <- as.factor(dat$area_code)
+     mod <- rpart(churn ~ ., data = dat)
+     ore.save(mod, name=datastoreName, overwrite=TRUE)
+     list(voice_mail_plan=vmp,
+         international_plan=ip,
+         churn.table=table(dat$churn),
+         rpart.model = mod)
+   })
R>
R> library(rpart)
R> datastorePrefix="my.rpartModel"
R>
R> res <- ore.groupApply( CT, INDEX=CT$vmp_ip,
+       FUN.NAME="my.rpartFunction",
+       datastorePrefix=datastorePrefix,
```

```
+       ore.connect=TRUE)
R> res[[1]]
$voice_mail_plan
[1] "no"

$international_plan
[1] "no"

$churn.table

  no  yes
1878  302

$rpart.model
n= 2180

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 2180 302 no (0.86146789 0.13853211)
   2) total_day_minutes< 263.55 2040 192 no (0.90588235 0.09411765)
     4) number_customer_service_calls< 3.5 1876 108 no (0.94243070 0.05756930)
       8) total_day_minutes< 223.25 1599  44 no (0.97248280 0.02751720) *
       9) total_day_minutes>=223.25 277  64 no (0.76895307 0.23104693)
        18) total_eve_minutes< 242.35 210  18 no (0.91428571 0.08571429) *
        19) total_eve_minutes>=242.35 67  21 yes (0.31343284 0.68656716)
          38) total_night_minutes< 174.2 17   4 no (0.76470588 0.23529412) *
          39) total_night_minutes>=174.2 50   8 yes (0.16000000 0.84000000) *
     5) number_customer_service_calls>=3.5 164  80 yes (0.48780488 0.51219512)
      10) total_day_minutes>=160.2 95  22 no (0.76842105 0.23157895)
        20)
state=AL,AZ,CA,CO,DC,DE,FL,HI,KS,KY,MA,MD,ME,MI,NC,ND,NE,NH,NM,OK,OR,SC,TN,VA,VT,W
Y 56   2 no (0.96428571 0.03571429) *
        21) state=AK,AR,CT,GA,IA,ID,MN,MO,NJ,NV,NY,OH,RI,TX,UT,WA,WV 39  19 yes
(0.48717949 0.51282051)
          42) total_day_minutes>=182.3 21   5 no (0.76190476 0.23809524) *
          43) total_day_minutes< 182.3 18   3 yes (0.16666667 0.83333333) *
      11) total_day_minutes< 160.2 69   7 yes (0.10144928 0.89855072) *
   3) total_day_minutes>=263.55 140  30 yes (0.21428571 0.78571429)
     6) total_eve_minutes< 167.3 29   7 no (0.75862069 0.24137931)
      12) state=AK,AR,AZ,CO,CT,FL,HI,IN,KS,LA,MD,ND,NM,NY,OH,UT,WA,WV 21   0 no
(1.00000000 0.00000000) *
      13) state=IA,MA,MN,PA,SD,TX,WI 8   1 yes (0.12500000 0.87500000) *
     7) total_eve_minutes>=167.3 111   8 yes (0.07207207 0.92792793) *

R> ore.load(name=paste(datastorePrefix,"yes","yes",sep="_"))
[1] "mod"
R> mod
n= 92

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 92 36 no (0.60869565 0.39130435)
  2) total_intl_minutes< 13.1 71 15 no (0.78873239 0.21126761)
    4) total_intl_calls>=2.5 60  4 no (0.93333333 0.06666667)
      8)
state=AK,AR,AZ,CO,CT,DC,DE,FL,GA,HI,ID,IL,IN,KS,MD,MI,MO,MS,MT,NC,ND,NE,NH,NJ,OH,S
C,SD,UT,VA,WA,WV,WY 53  0 no (1.00000000 0.00000000) *
      9) state=ME,NM,VT,WI 7  3 yes (0.42857143 0.57142857) *
```

```
  5) total_intl_calls< 2.5 11  0 yes (0.00000000 1.00000000) *
3) total_intl_minutes>=13.1 21  0 yes (0.00000000 1.00000000) *
```

Using the ore.rowApply Function

The `ore.rowApply` function invokes an R script with an `ore.frame` as the input data. The `ore.rowApply` function passes the `ore.frame` to the user-defined input function as the first argument to that function. The `rows` argument to the `ore.rowApply` function specifies the number of rows to pass to each invocation of the user-defined R function. The last chunk or rows may have fewer rows than the number specified. The `ore.rowApply` function can use data-parallel execution, in which one or more R engines perform the same R function, or task, on different partitions of data.

The `ore.rowApply` function returns an `ore.list` object or an `ore.frame` object.

See Also:

- "Arguments for Functions that Run Scripts" on page 6-5
- "Installing a Third-Party Package for Use in Embedded R Execution" on page 6-4

Example 6–11 uses the `e1071` package, previously downloaded from CRAN. The example also uses the `mod` object, which is the a Naive Bayes model created in Example 6–8, "Using the ore.tableApply Function" on page 6-12.

Example 6–11 does the following:

- Loads the package `e1071`.
- Pushes the `iris` data set to the database as the `IRIS` temporary table and `ore.frame`.
- Creates a copy of `IRIS` as `IRIS_PRED` and adds the PRED column to `IRIS_PRED` to contain the predictions.
- Invokes the `ore.rowApply` function, passing the `IRIS` `ore.frame` as the data source for user-defined R function and defining the function.
- The user-defined function does the following:
  – Loads the package `e1071` so that it is available to the R engine or engines running in the database.
  – Converts the Species column to a factor because, although the `ore.frame` defined factors, when they are loaded to the user-defined function, factors appear as character vectors.
  – Invokes the `predict` method and returns the `res` object, which contains the predictions in the column added to the data set
- The example pulls the model to the client R session
- Passes `IRIS_PRED` as the argument `FUN.VALUE`, which specifies the structure of the object that the `ore.rowApply` function returns.
- Specifies the number of rows to pass to each invocation of the user-defined function.
- Displays the class of `res`, and invokes the `table` function to display the Species column and the PRED column of the `res` object.

Example 6–11 Using the ore.rowApply Function

```
library(e1071)
IRIS <- ore.push(iris)
IRIS_PRED <- IRIS
IRIS_PRED$PRED <- "A"
res <- ore.rowApply(
  IRIS,
  function(dat, mod) {
    library(e1071)
    dat$Species <- as.factor(dat$Species)
    dat$PRED <- predict(mod, newdata = dat)
    dat
  },
  mod = ore.pull(mod),
  FUN.VALUE = IRIS_PRED,
  rows=10)
class(res)
table(res$Species, res$PRED)
```

Listing for Example 6–11

```
R> library(e1071)
R> IRIS <- ore.push(iris)
R> IRIS_PRED <- IRIS
R> IRIS_PRED$PRED <- "A"
R> res <- ore.rowApply(
+   IRIS ,
+   function(dat, mod) {
+     library(e1071)
+     dat$Species <- as.factor(dat$Species)
+     dat$PRED <- predict(mod, newdata = dat)
+     dat
+   },
+   mod = ore.pull(mod),
+   FUN.VALUE = IRIS_PRED,
+   rows=10)
R> class(res)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> table(res$Species, res$PRED)

             setosa versicolor virginica
  setosa         50          0         0
  versicolor      0         47         3
  virginica       0          3        47
```

Using the ore.indexApply Function

The `ore.indexApply` function executes the specified user-defined input function using data that is generated by the input function. It supports task-parallel execution, in which one or more R engines perform the same or different calculations, or task. The `times` argument to the `ore.indexApply` function specifies the number of times that the input function executes in the database. Any required data must be explicitly generated or loaded within the input function.

The `ore.indexApply` function returns an `ore.list` object or an `ore.frame` object.

Examples of the use of the `ore.indexApply` function are in the following topics:

- Simple Example of Using the ore.indexApply Function

- Column-Parallel Use Case
- Simulations Use Case

See Also:

- "Arguments for Functions that Run Scripts" on page 6-5
- "Installing a Third-Party Package for Use in Embedded R Execution" on page 6-4

Simple Example of Using the ore.indexApply Function

Example 6–12 invokes `ore.indexApply` and specifies that it execute the input function five times in parallel. It displays the class of the result, which is `ore.list`, and then displays the result.

Example 6–12 Using the ore.indexApply Function

```
res <- ore.indexApply(5,
      function(index) {
        paste("IndexApply:",index)
      },
      parallel=TRUE)
class(res)
res

R> res <- ore.indexApply(5,
+       function(index) {
+         paste("IndexApply:",index)
+       },
+       parallel=TRUE)
R> class(res)
[1] "ore.list"
attr(,"package")
[1] "OREembed"
R> res
$`1`
[1] "IndexApply: 1"

$`2`
[1] "IndexApply: 2"

$`3`
[1] "IndexApply: 3"

$`4`
[1] "IndexApply: 4"

$`5`
[1] "IndexApply: 5"
```

Column-Parallel Use Case

Example 6–12 uses the R `summary` function to compute in parallel summary statistics on the first four numeric columns of the `iris` data set. The example combines the computations into a final result. The first argument to the `ore.indexApply` function is 4, which specifies the number of columns to summarize in parallel. The user-defined input function takes one argument, `index`, which will be a value between 1 and 4 and which specifies the column to summarize.

The example invokes the `summary` function on the specified column. The `summary` invocation returns a single row, which contains the summary statistics for the column. The example converts the result of the `summary` invocation into a `data.frame` and adds the column name to it.

The example next uses the `FUN.VALUE` argument to the `ore.indexApply` function to define the structure of the result of the function. The result is then returned as an `ore.frame` object with that structure.

Example 6–13 Using the ore.indexApply Function and Combining Results

```
res <- NULL
res <- ore.indexApply(4,
      function(index) {
        ss <- summary(iris[, index])
        attr.names <- attr(ss, "names")
        stats <- data.frame(matrix(ss, 1, length(ss)))
        names(stats) <- attr.names
        stats$col <- names(iris)[index]
        stats
      },
      FUN.VALUE=data.frame(Min.=numeric(0),
        "1st Qu."=numeric(0),
        Median=numeric(0),
        Mean=numeric(0),
        "3rd Qu."=numeric(0),
        Max.=numeric(0),
        col=character(0)),
      parallel=TRUE)
res
```

Listing for Example 6–13

```
R> res <- NULL
R> res <- ore.indexApply(4,
+       function(index) {
+         ss <- summary(iris[,index])
+         attr.names <- attr(ss,"names")
+         stats <- data.frame(matrix(ss,1,length(ss)))
+         names(stats) <- attr.names
+         stats$col <- names(iris)[index]
+         stats
+       },
+       FUN.VALUE=data.frame(Min.=numeric(0),
+         "1st Qu."=numeric(0),
+         Median=numeric(0),
+         Mean=numeric(0),
+         "3rd Qu."=numeric(0),
+         Max.=numeric(0),
+         col=character(0)),
+       parallel=TRUE)
R> res
  Min. X1st.Qu. Median  Mean X3rd.Qu. Max.          col
1  2.0      2.8   3.00 3.057      3.3  4.4  Sepal.Width
2  4.3      5.1   5.80 5.843      6.4  7.9 Sepal.Length
3  0.1      0.3   1.30 1.199      1.8  2.5  Petal.Width
4  1.0      1.6   4.35 3.758      5.1  6.9 Petal.Length
Warning message:
```

```
ORE object has no unique key - using random order
```

Simulations Use Case

You can use the `ore.indexApply` function in simulations, which can take advantage of high-performance computing hardware like an Oracle Exadata Database Machine. Example 6–14 takes multiple samples from a random normal distribution to compare the distribution of the summary statistics. Each simulation occurs in a separate R engine in the database, in parallel, up to the degree of parallelism allowed by the database.

Example 6–14 defines variables for the sample size, the mean and standard deviations of the random numbers, and the number of simulations to perform. The example specifies `num.simulations` as the first argument to the `ore.indexApply` function. The `ore.indexApply` function passes `num.simulations` to the user-defined function as the `index` argument. This input function then sets the random seed based on the index so that each invocation of the input functions generates a different set of random numbers.

The input function next uses the `rnorm` function to produce `sample.size` random normal values. It invokes the `summary` function on the vector of random numbers, and then prepares a `data.frame` as the result it returns. The `ore.indexApply` function specifies the `FUN.VALUE` argument so that it returns an `ore.frame` that structures the combined results of the simulations. The `res` variable gets the `ore.frame` returned by the `ore.indexApply` function.

To get the distribution of samples, the example invokes the `boxplot` function on the `data.frame` that is the result of using the `ore.pull` function to bring selected columns from `res` to the client.

Example 6–14 Using the ore.indexApply Function in a Simulation

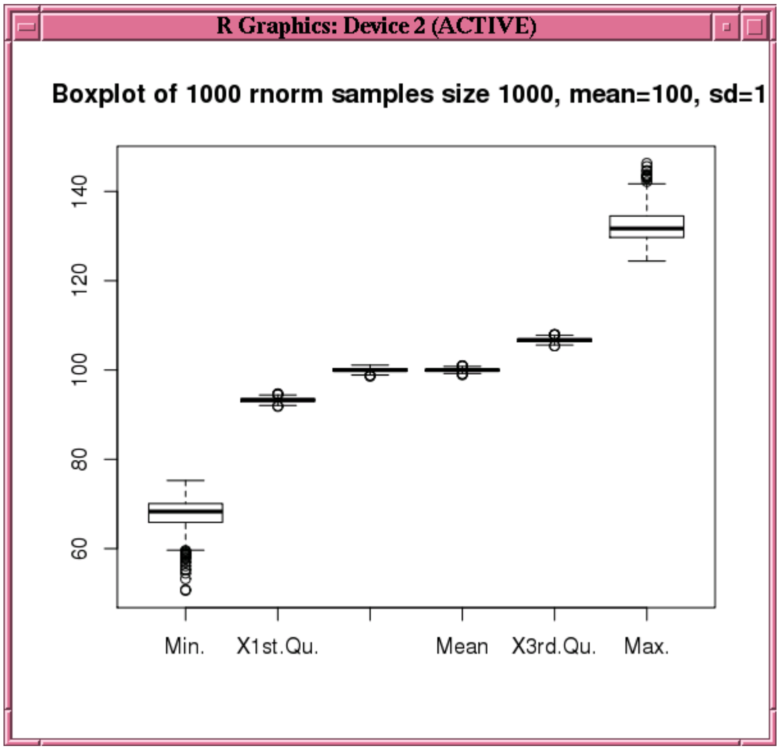
```
res <- NULL
sample.size = 1000
mean.val = 100
std.dev.val = 10
num.simulations = 1000

res <- ore.indexApply(num.simulations,
      function(index, sample.size=1000, mean=0, std.dev=1) {
        set.seed(index)
        x <- rnorm(sample.size, mean, std.dev)
        ss <- summary(x)
        attr.names <- attr(ss,"names")
        stats <- data.frame(matrix(ss,1,length(ss)))
        names(stats) <- attr.names
        stats$index <- index
        stats
      },
      FUN.VALUE=data.frame(Min.=numeric(0),
        "1st Qu."=numeric(0),
        Median=numeric(0),
        Mean=numeric(0),
        "3rd Qu."=numeric(0),
        Max.=numeric(0),
        index=numeric(0)),
      parallel=TRUE,
      sample.size=sample.size,
```

```
      mean=mean.val, std.dev=std.dev.val)
options("ore.warn.order" = FALSE)
head(res, 3)
tail(res, 3)
boxplot(ore.pull(res[,1:6]),
  main=sprintf("Boxplot of %d rnorm samples size %d, mean=%d, sd=%d",
               num.simulations, sample.size, mean.val, std.dev.val))
```

Listing for Example 6–14

```
R> res <- ore.indexApply(num.simulations,
+       function(index, sample.size=1000, mean=0, std.dev=1) {
+         set.seed(index)
+         x <- rnorm(sample.size, mean, std.dev)
+         ss <- summary(x)
+         attr.names <- attr(ss,"names")
+         stats <- data.frame(matrix(ss,1,length(ss)))
+         names(stats) <- attr.names
+         stats$index <- index
+         stats
+       },
+       FUN.VALUE=data.frame(Min.=numeric(0),
+         "1st Qu."=numeric(0),
+         Median=numeric(0),
+         Mean=numeric(0),
+         "3rd Qu."=numeric(0),
+         Max.=numeric(0),
+         index=numeric(0)),
+       parallel=TRUE,
+       sample.size=sample.size,
+       mean=mean.val, std.dev=std.dev.val)
R> options("ore.warn.order" = FALSE)
R> head(res, 3)
   Min. X1st.Qu. Median   Mean X3rd.Qu.  Max. index
1 67.56    93.11  99.42  99.30    105.8 128.0   847
2 67.73    94.19  99.86 100.10    106.3 130.7   258
3 65.58    93.15  99.78  99.82    106.2 134.3   264
R> tail(res, 3)
   Min. X1st.Qu. Median   Mean X3rd.Qu.  Max. index
1 65.02    93.44  100.2 100.20    106.9 134.0     5
2 71.60    93.34   99.6  99.66    106.4 131.7     4
3 69.44    93.15  100.3 100.10    106.8 135.2     3
R> boxplot(ore.pull(res[,1:6]),
+   main=sprintf("Boxplot of %d rnorm samples size %d, mean=%d, sd=%d",
+                num.simulations, sample.size, mean.val, std.dev.val))
```

Figure 6–2 Display of the boxplot Function in Example 6–14



Using the ore.scriptCreate and ore.scriptDrop Functions

The `ore.scriptCreate` function creates a script in R script repository of the Oracle database. Embedded R execution functions can use a script from the repository by specifying it with the `FUN.NAME` argument. Scripts in the R script repository are also available through the SQL interface for Oracle R Enterprise embedded R execution.

The `ore.scriptDrop` function removes the specified R script from the R script repository.

> **Note:** Invoking the `ore.scriptCreate` or `ore.scriptDrop` function requires the RQADMIN role.

Both the `ore.scriptCreate` and `ore.scriptDrop` functions return an invisible `NULL` value if it succeeds; if it does not succeed in creating or dropping the script, it returns an error.

Example 6–15 creates a script and stores it in the R script repository. The use-defined function in the script creates a linear model. The example pushes the `iris` data set to the database. It then invokes the `ore.tableApply` function and specifies the stored script with the `FUN.NAME` argument. The example then drops the script from the repository.

Example 6–15 Using the ore.scriptCreate and ore.scriptDrop Functions

```
ore.scriptCreate("MYLM",function(data, formula, ...) lm(formula, data, ...))
IRIS <- ore.push(iris)
ore.tableApply(IRIS[1:4], FUN.NAME = "MYLM", formula = Sepal.Length ~ .)
ore.scriptDrop("MYLM")
```

Listing for Example 6–15

```
R> ore.scriptCreate("MYLM",function(data, formula, ...) lm(formula, data, ...))
R> IRIS <- ore.push(iris)
R> ore.tableApply(IRIS[1:4], FUN.NAME = "MYLM", formula = Sepal.Length ~ .)

Call:
lm(formula = formula, data = data)

Coefficients:
 (Intercept)   Sepal.Width  Petal.Length   Petal.Width
      1.8560        0.6508        0.7091       -0.5565
R> ore.scriptDrop("MYLM")
```

See Also: Example 6–10 on page 6-17

SQL Interface for Embedded R Execution

The SQL interface for Oracle R Enterprise embedded R execution allows you to execute R scripts in production database applications.

The functions to use with the SQL interface must be stored in the database R repository, and referenced by name in SQL API functions. See Registering and Managing SQL Scripts for a description of how to add scripts to the repository, remove scripts from the repository, and list and use scripts in the repository.

For descriptions of the SQL functions, see About Oracle R Enterprise SQL Functions.

This section describes the SQL interface in the following topics:

- Registering and Managing SQL Scripts
- About Oracle R Enterprise SQL Functions
- Using the rqGroupEval Function
- Using SQL Functions and Objects in a Datastore
- Datastore Management in SQL

Registering and Managing SQL Scripts

The R functions to use with the SQL interface for embedded R execution must be stored in the database R script repository and be referenced by name in SQL API functions. For security purposes, you must first register the R script under some system unique name and then use new name instead of the actual script in call to SQL interface table functions.

The administrative functions `sys.rqScriptCreate` and `sys.rqScriptDrop` create and drop scripts. The `sys.rq_scripts` view allows you to list and use scripts that were created.

Script creation or deletion requires the RQADMIN role described in "Security Considerations for Scripts" on page 6-2.

When using the `sys.rqScriptCreate` function, you must specify a corresponding R closure of the function string.

Example 6–16 demonstrates registering an R script and using it.

Example 6–16   Registering and Using an R Script

```
begin
  sys.rqScriptCreate('tmrqfun2',
```

```
    'function() {
       ID <- 1:10
       res <- data.frame(ID = ID, RES = ID / 100)
       res
  }');
end;
/

select *
  from table(rqEval(
        NULL,
       'select 1 id, 1 res from dual',
       'tmrqfun2'));

begin
  sys.rqScriptDrop('tmrqfun2');
end;
/
```

About Oracle R Enterprise SQL Functions

Invoking an Oracle R Enterprise SQL function results in one or more R engines being started at the database depending on database parallelism settings. To enable execution of an R script in the database, Oracle R Enterprise provides SQL variants of the `ore.doEval`, `ore.tableApply`, `ore.groupApply`, and `ore.rowApply` functions. Those R functions are described in "R Interface for Embedded R Execution" on page 6-5.

The SQL functions for embedded R execution are:

- `rqTableEval`
- `rqEval`
- `rqRowEval`
- `rqGroupEval`

The `rqGroupEval` function requires additional SQL specification and is provided here as a virtual function that partitions the data according to the values of a specified column and invokes the R script on each partition. For more information, see "Using the rqGroupEval Function" on page 6-30.

You can also use these functions with objects in a datastore, as described in "Datastore Management in SQL" on page 6-31.

The `rqEval`, `rqTableEval`, `rqGroupEval`, and `rqRowEval`) functions have similar syntax:

```
rq*Eval(
     cursor(select * from table-1),
     cursor(select * from table-2),
     'select <column list> from table-3 t',
     <grouping col-name from table-1 or num_rows>,
     <R closure name of registered-R-code>
     )
```

The following are the components of the SQL function:

- The first cursor is the input cursor: Input is passed as a whole table, group, or N rows at a time to the R closure described in the fourth argument.

  The `rqEval` function does *not* have this cursor argument.

- The second cursor is the parameters cursor: One row of scalar values (string, numeric, or both) can be passed; for example, the name of the model and several numeric scalar values for model setting.
- The query specifies the output table definition; output can be 'SELECT statement', 'XML', or 'PNG'.
- `grouping col-name` applies to `rqGroupEval`; it provides the name of the grouping column.
- `num_rows` applies to `rqRowEval`; it provides the number of rows to provide to the functions at one time.
- <R closure name of registered-R-code> is a registered version of the R function to execute. See "Registering and Managing SQL Scripts" on page 6-27 for details.

The return values for all of the SQL functions specify one of these values:

- A table signature that is specified in a SELECT statement, which returns results as a table from the `rq` function.
- XML, returned as a CLOB that contains both structured and graph images in an XML string. The structured components are provided first, followed by the base 64 encoding of the `png` representation of the image.
- PNG, returned as a BLOB that contains graph images in `png` format.

The `rqEval`, `rqTableEval`, `rqGroupEval`, and `rqRowEval` functions must specify an R script by the name that is stored in the R script repository. The `sys.rq_scripts` view provides a list of registered scripts.

The following examples illustrate using these functions:

- This example uses all rows from the table `fish` as input to the R function that takes no other arguments and produces `output` that contains all input data plus the ROWSUM of values.

  Note that `param` argument to the R function is optional.

```
begin
sys.rqScriptCreate('tmrqfun2',
'function(x, param) {
dat <- data.frame(x, stringsAsFactors=F)
cbind(dat, ROWSUM = apply(dat,1,sum)+10)
}');
end;
/

select * from table(rqTableEval(
   cursor(select * from fish),
   NULL,
   'select t.*, 1 rowsum from fish t',
   'tmrqfun2' ));

begin
sys.rqScriptDrop('tmrqfun2');
end;
/
```

- This example illustrates passing n=1 (4th parameter) row at a time from the table `fish` to the R function. No parameters are required by the function. The function generates ROWSUM which is added as an extra column to `fish` in the output.

```
begin
```

```
sys.rqScriptCreate('tmrqfun2',
'function(x, param) {
dat <- data.frame(x, stringsAsFactors=F)
cbind(dat, ROWSUM = apply(dat,1,sum)+10)
}');
end;
/

select * from table(rqRowEval(
   cursor(select * from fish),
   NULL,
   'select t.*, 1 rowsum from fish t',
    1,
   'tmrqfun2' ));

begin
sys.rqScriptDrop('tmrqfun2');
end;
/
```

Using the rqGroupEval Function

The `rqGroupEval` function invokes an R script on data that is partitioned by a grouping column. The `rqGroupEval` function requires the creation of the following two PL/SQL objects:

- A PL/SQL package that specifies the types of the result to return.
- A function that takes the return value of the package and uses the return value with `PIPELINED_PARALLEL_ENABLE` set to indicate the column on which to partition data.

Suppose that ONTIME_S is a table that stores information about arrival of airplanes. The data cursor uses all data, but you could also define cursors that use some columns using PL/SQL records. Then you must define as many PL/SQL table functions as the number of grouping columns that you are interested in using for a particular data cursor.

```
CREATE PACKAGE ontimePkg AS
  TYPE cur IS REF CURSOR RETURN ontime_s%ROWTYPE;
END ontimePkg;
/

CREATE FUNCTION ontimeGroupEval(
  inp_cur  ontimePkg.cur,
  par_cur  SYS_REFCURSOR,
  out_qry  VARCHAR2,
  grp_col  VARCHAR2,
  exp_txt  CLOB)
RETURN SYS.AnyDataSet
PIPELINED PARALLEL_ENABLE (PARTITION inp_cur BY HASH (month))
CLUSTER inp_cur BY (month)
USING rqGroupEvalImpl;
/
```

At this time, only one grouping column is supported. If you have multiple columns, then combine the columns into one column and use the new column as a grouping column. The `PARALLEL_ENABLE` clause is optional but the `CLUSTER BY` clause is not.

Using SQL Functions and Objects in a Datastore

The SQL functions for embedded R execution allow you to use in a parameter cursor a serialized R object saved in a datastore. You can specify the association of object and datastore names of the serialized R objects with the R function parameter names in that parameter cursor.

Example 6–17 demonstrates using the `rqTableEval` function and specifying a datastore in a cursor. The example uses a datastore named `ontime_model` and gets the `lm.mod` model object from the datastore. The example uses the model in SQL for scoring using embedded R execution.

Example 6–17 Using rqTableEval and Specifying a Datastore

```
begin
  sys.rqScriptCreate('tmrqmodelscore',
    'function(dat, in.dsname, in.objname) {
       ore.load(name=in.dsname, list=in.objname)
       mod <- get(in.objname)
       prd <- predict(mod, newdata=dat)
       prd[as.integer(rownames(prd))] <- prd
       res <- cbind(dat, PRED = prd)
       res
    }');
end;
/ -- score model

select * from table(rqTableEval(
             cursor(select ARRDELAY, DISTANCE, DEPDELAY from ontime_s
                where year = 2003 and month = 5 and dayofmonth = 2),
             cursor(select 'ontime_model' as "in.dsname",
            'lm.mod' as "in.objname", 1 as "ore.connect" from dual),
             'select ARRDELAY, DISTANCE, DEPDELAY, 1 PRED from ontime_s',
            'tmrqmodelscore'))
order by 1, 2, 3;
```

Datastore Management in SQL

Oracle R Enterprise provides basic management for datastores in SQL. Basic datastore management includes show, search, and drop. The following functions and views are provided:

- `rqDropDataStore` deletes a datastore and all of the objects in the datastore.

  Syntax: `rqDropDataStore('<ds_name>')`, where `<ds_name>` is the name of the datastore to delete.

  The following example deletes the datastore ds_model from current user schema:

  ```
  rqDropDataStore('ds_model')
  ```

- `rquser_DataStoreList` is a view containing datastore-level information for all datastores in the current user schema. The information consists of datastore name, number of objects, size, creation date, and description.

  These examples illustrate using the view:

  ```
  select * from rquser_DataStoreList;
  select dsname, nobj, dssize from rquser_datastorelist where dsname = 'ds_1';
  ```

- `rquser_DataStoreContents` is a view containing object-level information about all datastores in the current user schema. The information consists of object name, size, class, length, number of rows and columns.

  This example lists the datastore contents for datastore `ds_1`:

  ```
  select * from rquser_DataStoreContents where dsname = 'ds_1';
  ```

A

R Operators and Functions Supported by Oracle R Enterprise

The Oracle R Enterprise packages support many R operators and functions that you can use with Oracle R Enterprise objects. This appendix lists the R operators and functions that Oracle R Enterprise supports.

The Oracle R Enterprise sample programs described in "Oracle R Enterprise Examples" on page 1-13 include several examples using each category of these functions with Oracle R Enterprise data types.

You are not restricted to using this list of functions. If a specific function that you need is not supported by Oracle R Enterprise, you can pull data from the database into the R engine memory using `ore.pull` to create an in-memory R object first, and use any R function.

The following operators and functions are supported. See R documentation for syntax and semantics of these operators and functions. Syntax and semantics for these items are unchanged when used on a corresponding database-mapped data type (also known as an Oracle R Enterprise data type). For a list of Oracle R Enterprise data types, see "Transparency Layer Support for R Data Types and Classes" on page 1-7.

- **Mathematical transformations**: abs, sign, sqrt, ceiling, floor, trunc, cummax, cummin, cumprod, cumsum, log, loglo, log10, log2, log1p, acos, acosh, asin, asinh, atan, atanh, exp, expm1, cos, cosh, sin, sinh, tan, atan2, tanh, gamma, lgamma, digamma, trigamma, factorial, lfactorial, round, signif, pmin, pmax, zapsmall, rank, diff, besselI, besselJ, besselK, besselY
- **Basic statistics**: mean, summary, min, max, sum, any, all, median, range, IQR, fivenum, mad, quantile, sd, var, table, tabulate, rowSums, colSums, rowMeans, colMeans, cor, cov
- **Arithmetic operators**: +, -, *, /, ^, %%, %/%
- **Comparison operators**: ==, >, <, !=, <=, >=
- **Logical operators**: &, |, xor
- **Set operations**: unique, %in%, subset
- **String operations**: tolower, toupper, casefold, toString, chartr, sub, gsub, substr, substring, paste, nchar, grepl
- **Combine Data Frame**: cbind, rbind, merge
- **Combine vectors**: append
- **Vector creation**: ifelse

- **Subset selection**: [, [[, $, head, tail, window, subset, Filter, na.omit, na.exclude, complete.cases
- **Subset replacement**: [<-, [[<-, $<-
- **Data reshaping**: split, unlist
- **Data processing**: eval, with, within, transform
- **Apply variants**: tapply, aggregate, by
- **Special value checks**: is.na, is.finite, is.infinite, is.nan
- **Metadata functions**: nrow, NROW, ncol, NCOL, nlevels, names, names<-, row, col, dimnames, dimnames<-, dim, length, row.names, row.names<-, rownames, rownames<-, colnames, levels, reorder
- **Graphics**: arrows, boxplot, cdplot, co.intervals, coplot, hist, identify, lines, matlines, matplot, matpoints, pairs, plot, points, polygon, polypath, rug, segments, smoothScatter, sunflowerplot, symbols, text, xspline, xy.coords
- **Conversion functions**: as.logical, as.integer, as.numeric, as.character, as.vector, as.factor, as.data.frame
- **Type check functions**: is.logical, is.integer, is.numeric, is.character, is.vector, is.factor, is.data.frame
- **Character manipulation**: nchar, tolower, toupper, casefold, chartr, sub, gsub, substr
- **Other ore.frame functions**: data.frame, max.col, scale
- **Hypothesis testing**: binom.test, chisq.test, ks.test, prop.test, t.test, var.test, wilcox.test
- **Various Distributions**: Density, cumulative distribution, and quantile functions for standard distributions
- **ore.matrix function**: show, is.matrix, as.matrix, %*% (matrix multiplication), t, crossprod (matrix cross-product), tcrossprod (matrix cross-product A times transpose of B), solve (invert), backsolve, forwardsolve, all appropriate mathematical functions (abs, sign, and so on), summary (max, min, all, and so on), mean

Index

T

U

V

W