Oracle® R Enterprise

User's Guide

Release 1.5.1

E83204-02

April 2017

Oracle R Enterprise User's Guide, Release 1.5.1

E83204-02

Primary Author: David McDermid

Contributing Authors: Mark Hornick

Contributors: Sherry Lamonica, Qin Wang, Lei Zhang

Contents

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 that have purchased support 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 Oracle R Enterprise documentation set includes the following:

- *Oracle R Enterprise Installation and Administration Guide*
- *Oracle R Enterprise Release Notes*

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/database-technologies/r/r-enterprise/overview/index.html`.
- The Oracle R Enterprise Discussion Forum at `https://community.oracle.com/community/developer/english/business_intelligence/data_warehousing/r` 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 1.5.1

Oracle R Enterprise 1.5.1 has some new features that are compatible with Oracle Database 12*c*, Release 12.1.0.2 and earlier, and other new features compatible with Oracle Database 12*c*, Release 12.2.

New Features for Oracle Database Release 12.1.0.2 and Earlier

Oracle R Enterprise 1.5.1 has the new `OREdplyr` package, improved performance of row ordering in `ore.frame` objects, and faster loading of the Oracle R Enterprise packages.

OREdplyr Package for Data Manipulation

The `dplyr` package provides a grammar of data manipulation functions for `data.frame` objects and `numeric` objects. The new `OREdplyr` package implements much of this functionality for `ore.frame` and `ore.numeric` objects. This enables in-database execution of `dplyr` functionality such as selecting, filtering, ordering, and grouping columns and rows, and joining, summarizing, sampling, and ranking rows.

Related Topics:

Data Manipulation Using OREdplyr (page 3-40)
`OREdplyr` package functions transparently implement `dplyr` functions for use with `ore.frame` and `ore.numeric` objects.

New Features for Oracle Database Release 12.2.0.1

Oracle R Enterprise 1.5.1 has the new graph analytics package `OAAgraph` and has new functions in the Oracle R Enterprise Data Mining package `OREdm`.

OAAgraph Package

The `OAAgraph` package provides an R interface to the powerful Oracle Spatial and Graph Property Graph In-Memory Analyst (PGX) for use in combination with Oracle R Enterprise and database tables.

The package provides a single, unified interface supporting the complementary use of machine learning and graph analytics technologies.

Graph analytics use graph representations of data, in which data entities are nodes and relationships are edges. Machine learning produces models that identify patterns in data for both descriptive and predictive analytics. Together, these technologies complement and augment one another.

Related Topics:

Graph Analysis Using OAAgraph (page 3-58)
Beginning with Oracle Database 12.2 , the `OAAgraph` package provides an R interface to the Oracle Spatial and Graph Property Graph In-Memory Analyst (PGX) for use with Oracle R Enterprise and database tables.

New Features of the OREdm Package

The `OREdm` package has some new functions that use in-database Oracle Data Mining algorithms to create models in the database and new arguments for some functions.

New Functions in the OREdm Package

New functions in the `OREdm` Oracle Data Mining package that use in-database algorithms are the following:

- `ore.odmEM`, Expectation Maximization Models
- `ore.odmESA`, Explicit Semantic Analysis Models
- `ore.odmRAlg`, Extensible R Algorithm Models
- `ore.odmSVD`, Singular Value Decomposition Models

The `ore.odmRAlg` enables users to use registered R scripts to create models that use the Oracle Data Mining in-database model framework.

Other new functions are the following:

- `partitions`, which returns partition names from a partitioned model
- `settings`, which returns the Oracle Data Mining parameter settings used to build the model.

New Arguments to Some Functions for Oracle Data Mining Model Build Configuration and Text Processing

The new arguments for some of the data mining model functions are:

- `odm.setting`

- `ctx.setting`

`odm.setting`

The `odm.setting` value is a list that specifies Oracle Data Mining parameter settings. Both Oracle Data Mining global and algorithm-specific parameters can be specified to configure the model build. Some new features are enabled through the parameter settings. For example, you can use this argument to specify the creation of a partitioned model, which is an ensemble model that consists of multiple sub-models. When you specify the parameter `ODMS_PARTITION_COLUMNS` and the names of the columns by which to partition the input data, the function returns a model with a sub-model for each partition. The partitions are based on the unique values found in the columns.

Partitioned models can automate scoring by allowing you to reference the top-level model only, which causes the proper sub-model to be chosen based on the values of the partitioned column or columns for each row of data to be scored.

`ctx.setting`

With this argument, you can specify Oracle Text attribute-specific settings. You specify the columns that should be treated as text and the type of text transformation to apply.

This argument applies to the following functions:

- `ore.odmESA`, Explicit Semantic Analysis
- `ore.odmGLM`, Generalized Linear Models
- `ore.odmKMeans`, *k*-Means
- `ore.odmNMF`, Non-Negative Matrix Factorization
- `ore.odmSVD`, Singular Value Decomposition
- `ore.odmSVM`, Support Vector Machines

> **Note:** To create an Oracle Text policy, the user must have the `CTXSYS.CTX_DDL` privilege.

Related Topics:

Building Oracle Data Mining Models (page 4-11)

Partitioning and Text Mining (page 4-13)
Beginning with Oracle Database 12*c*, Release 2 (12.2), functions in the Oracle Data Mining package have an argument that specifies settings for an Oracle Data Mining model and some have an argument for setting text mining parameters.

Changes in Oracle R Enterprise 1.5

Oracle R Enterprise 1.5 introduces functions for managing Oracle R Enterprise datastores and scripts in the Oracle Database script repository. It also contains a new function in the `OREmodel` package, new transparency layer methods for some functions in the `stats` package, and other enhancements.

New Features in Oracle R Enterprise 1.5

Oracle R Enterprise 1.5 has new R functions and SQL procedures, new transparency layer methods, and enhancements to some functions.

Oracle R Enterprise 1.5 has new R functions and SQL procedures for managing Oracle R Enterprise datastores and the Oracle R Enterprise R script repository. Users can now share with other users access to datastores and registered R scripts. This release also has a new modeling function, `ore.randomForest`, new `svd` and `prcomp` statistical function methods that take `ore.frame` objects and can use parallel processing, and other enhancements.

The following topics briefly describe the new features.

- R for Datastore and Script Repository Management (page xii)
- PL/SQL and Data Dictionary Views for Datastore and Script Repository Management (page xiii)
- ore.groupApply Function Changes (page xiv)
- ore.randomForest Modeling Function (page xv)
- ore.summary Function Changes (page xv)
- Statistical Function Method Changes (page xvi)
- Support for BLOB and CLOB Data Types (page xvi)

R for Datastore and Script Repository Management

Oracle R Enterprise 1.5 provides new R functions for managing Oracle R Enterprise datastores and the R script repository.

The owner of a datastore or registered R script can now share with other users read privilege access to the datastore or script. This release also has new arguments to functions that create datastores and scripts and that give information about them. This datastore and script management functionality has both R and SQL interfaces.

The R functions for managing datastores are the following:

- `ore.delete`, which deletes a datastore, is unchanged.
- `ore.grant` is a new function that grants read privilege access to a datastore or script.
- `ore.load`, which loads objects from a datastore into an R environment, has the new argument `owner` that specifies the datastore owner.
- `ore.revoke` is a new function that revokes read privilege access to a datastore or script.
- `ore.save`, which creates a datastore, has the new argument `grantable` that specifies whether read access can be granted to the datastore.
- `ore.datastore`, which lists information about a datastore, has the new argument `type` that specifies the type of datastore. The values of `type` are the character strings `user` (the default), `grant`, `granted`, and `all`.

- `ore.datastoreSummary`, which provides detailed information about datastores, has the new argument `owner` that specifies a datastore owner.

The R functions for managing scripts are the following:

- `ore.grant` is a new function that grants read privilege access to a datastore or script.
- `ore.revoke` is a new function that revokes read privilege access to a datastore or script.
- `ore.scriptCreate`, which adds an R function to the script repository, has the new arguments `global` and `overwrite`. The argument `global` specifies whether the script is public or private. A public script is available to all users. If `global = FALSE`, then access to the script must be granted by the owner to other users. The argument `overwrite` specifies whether the content of a script can be replaced.
- `ore.scriptDrop`, which deletes a script, has the new arguments `global`, which specifies whether the script to drop is public or not, and `silent`, which specifies whether to report an error if the script cannot be dropped.
- `ore.scriptList`, which lists information about scripts, has the new argument `type` that specifies the type of script. The values of `type` are the character strings `user` (the default), `global`, `grant`, `granted`, and `all`.
- `ore.scriptLoad` is a new function that loads a script into the R environment.

PL/SQL and Data Dictionary Views for Datastore and Script Repository Management

Oracle R Enterprise 1.5 provides new PL/SQL procedures for managing Oracle R Enterprise datastores and the R script repository.

The owner of a datastore or registered R script can now share with other users read privilege access to the datastore or script. This release also has new arguments to procedures that create datastores and scripts and that give information about them. This functionality has both R and SQL interfaces. Oracle Database data dictionary views provide information about datastores and scripts.

The SQL procedures for controlling access to Oracle R Enterprise datastores and registered R scripts are described in the following sections:

- PL/SQL Procedures for Managing Datastores (page xiii)
- PL/SQL Procedures for Managing Scripts (page xiv)
- Data Dictionary Views for Datastores (page xiv)
- Data Dictionary Views for Scripts (page xiv)

PL/SQL Procedures for Managing Datastores

The PL/SQL procedures for managing datastores are the following:

- `rqDropDataStore`, which deletes a datastore, is unchanged.
- `rqGrant` is a new procedure that grants read privilege access to a datastore or script.

- `rqRevoke` is a new procedure that revokes read privilege access to a datastore or script.

PL/SQL Procedures for Managing Scripts

The PL/SQL procedures for managing scripts are the following:

- `rqGrant` is a new procedure that grants read privilege access to a datastore or script.
- `rqRevoke` is a new procedure that revokes read privilege access to a datastore or script.
- `rqScriptCreate` has the new arguments `global` and `overwrite`. The argument `global` specifies whether the script is global or private. A global script is a public script that is available to all users. If `global = FALSE`, then access to the script must be granted by the owner to other users. The argument `overwrite` specifies whether the content of a script can be replaced.
- `rqScriptDrop` has the new arguments `global`, which specifies whether the script to drop is global or not, and `silent`, which specifies whether to report an error if a script cannot be dropped.

Data Dictionary Views for Datastores

The Oracle Database dictionary views related to datastores are the following:

- `ALL_RQ_DATASTORES`
- `RQUSER_DATASTORECONTENTS`
- `USER_RQ_DATASTORES`
- `USER_RQ_DATASTORE_PRIVS`

Data Dictionary Views for Scripts

The Oracle Database dictionary views related to scripts are the following:

- `ALL_RQ_SCRIPTS`
- `USER_RQ_SCRIPTS`
- `USER_RQ_SCRIPT_PRIVS`

ore.groupApply Function Changes

Function `ore.groupApply` now supports partitioning on multiple columns.

The `INDEX` argument can now take an `ore.vector` or `ore.frame` object that contains `ore.factor` objects or columns, each of which is the same length as argument `X`. Function `ore.groupApply` uses the `INDEX` object to partition the data in `X` before sending it to function `FUN`.

For an example of the use of the `INDEX` object to partition the data, see "Partitioning on Multiple Columns" (page 6-29).

ore.randomForest Modeling Function

The ore.randomforest function builds a random forest model on data in an ore.frame object.

It uses embedded R execution to grow random forest trees in parallel in R sessions on the database server. It returns an ore.randomforest object. In Oracle R Enterprise 1.5, function ore.randomForest supports classification but not regression.

The ore.randomforest function uses the same algorithm as that adopted by the CRAN R randomForest package but it has better runtime memory usage as well as ensemble tree size.

The scoring method predict on an ore.randomforest model also runs in parallel. Oracle recommends that you set the cache.model argument to TRUE when sufficient memory is available. Otherwise, you should set cache.model to FALSE to prevent memory overuse.

To use ore.randomforest, you must install either Oracle R Distribution (ORD) 3.2 or the CRAN R randomForest package. Oracle recommends that you use the function ore.randomforest in ORD 3.2, which offers better performance and scalability than the CRAN R randomForest. If you only install the R randomForest package, ore.randomForest issues a warning message at run time. The CRAN R randomForest package is one of the supporting packages in Oracle R Enterprise 1.5.

The global option ore.parallel determines the degree of parallelism to use in the Oracle R Enterprise server. The argument groups controls the granularity of the ore.randomForest model.

For an example of using ore.randomForest, see "Building a Random Forest Model (page 4-9)."

ore.summary Function Changes

Function ore.summary has improved performance. It also has a different signature and the data types for some arguments have changed.

The function’s syntax is now the following:

```
ore.summary(data, var, stats = c("n", "mean", "min", "max"),
            class = NULL, types = NULL, ways = NULL, weight = NULL,
            order = NULL, maxid = NULL, minid = NULL, mu = 0,
            no.type = FALSE, no.freq = FALSE)
```

The differences between the ore.summary in Oracle R Enterprise Release 1.5 and previous releases are the following:

- The performance of ore.summary has improved; it now returns over an order of magnitude faster.
- The arguments var, stats, and class now take a vector of character strings; previously, they took a concatenated string using a comma as the separator.
- Argument types is now a list of character string vectors that specifies combinations of columns in the class argument.
- Arguments maxid and minid are named vectors of character strings.
- Arguments group.by and no.level are not supported.
- Argument mu in previous releases was named mu0.

For examples of using `ore.summary`, see "Summarizing Data with ore.summary" (page 3-35).

Statistical Function Method Changes

Oracle R Enterprise 1.5 provides transparency layer methods for the `stats` package functions `prcomp` and `svd`.

The `prcomp` function performs principal components analysis and the `svd` function performs singular-value decomposition. Those functions now accept `ore.frame` objects and can use parallel execution in the database, which can improve scalability and performance.

Support for BLOB and CLOB Data Types

Some Oracle R Enterprise functions now support the Oracle Database data types BLOB and CLOB.

Functions `ore.push` and `ore.pull` now support the database data types BLOB and CLOB.

Embedded R execution R functions now support the database data types BLOB and CLOB for input and output objects.

For examples of using BLOB and CLOB data types, see "Example 6-11 (page 6-23)".

Changes in Oracle R Enterprise 1.4.1

The following topics describe the changes in Oracle R Enterprise 1.4.1:

- New Features in Oracle R Enterprise 1.4.1 (page xvi)
- Other Changes in Oracle R Enterprise 1.4.1 (page xvii)

New Features in Oracle R Enterprise 1.4.1

The following changes are in Oracle R Enterprise 1.4.1:

- The `ore.glm` function now accepts offset terms in the model formula and the function can now be used to fit negative binomial and tweedie families of generalized linear models.
- The `ore.sync` function has an additional optional argument, `query`, that creates an `ore.frame` object from an optimized SQL `SELECT` statement without creating a view in the database. You can use this argument to create a query even when you do not have the `CREATE VIEW` system privilege for the current schema.
- The new global option for serialization, `ore.envAsEmptyenv`, specifies whether referenced environments in an object should be replaced with an empty environment during serialization to an Oracle Database. This option is used by the following functions:
  - `ore.push`, which for a `list` object accepts `envAsEmptyenv` as an optional argument
  - `ore.save`, which has `envAsEmptyenv` as a named argument
  - `ore.doEval` and the other embedded R execution functions, which accept `ore.envAsEmptyenv` as a control argument.

The default values of the above arguments are regulated by the global option `ore.envAsEmptyenv`, but by using the argument you can override the global option value for a function.

See Also:

– "Oracle R Enterprise Global Options (page 1-12)"

– "Optional and Control Arguments (page 6-11)"

– The online help for the `ore.push` and `ore.save` functions

Other Changes in Oracle R Enterprise 1.4.1

Other changes in this release are the following:

- The `arules` and `statmod` packages from The Comprehensive R Archive Network (CRAN) are now included in the Oracle R Enterprise supporting packages.

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 (page xvii)

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.
- 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.

  - 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.
  - The new `ore.odmNMF` function, which builds an Oracle Data Mining model for feature extraction using the Non-Negative Matrix Factorization (NMF) algorithm
  - The new `ore.odmOC` function, which builds an Oracle Data Mining model for clustering using the Orthogonal Partitioning Cluster (O-Cluster) algorithm.
- An additional global option for Oracle R Enterprise, `ore.parallel`.

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 (page xviii)
- Other Changes in Oracle R Enterprise 1.3 (page xix)

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 operating 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:

1.1 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.

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 (The 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).

1.3 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, and for information on installing them, 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 Getting Help on Oracle R Enterprise Classes, Functions, and Methods

This example shows several ways 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.

```
# 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.frame class.
help("ore.frame")

# 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")

# 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 (page 1-4)

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

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.

# 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.
```

1.4 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 (page 1-6)
- Transparency Layer Support for R Data Types and Classes (page 1-7)

1.4.1 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`.

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 (page 1-7)" for information on Oracle R Enterprise data types and object mappings and on the correspondences between R, Oracle R Enterprise, and SQL data types and objects
- "Getting Started with Oracle R Enterprise (page 2-1)"

Example 1-2 Finding the Mean of the Petal Lengths by Species in R

This example 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.

```
ore.create(iris, table = 'IRIS_TABLE')
aggplen = aggregate(IRIS_TABLE$Petal.Length,
                    by = list(species = IRIS_TABLE$Species),
```

```
                    FUN = mean)
aggplen
```

Listing for This Example

```
R> ore.create(iris, table = 'IRIS_TABLE')
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 1-3 SQL Equivalent of the Previous Example

This example shows the SQL equivalent of the `aggregate` function in the previous example.

```
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
```

1.4.2 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 (page 1-7)
- About the ore.frame Class (page 1-8)
- Support for R Naming Conventions (page 1-10)
- About Coercing R and Oracle R Enterprise Class Types (page 1-10)

1.4.2.1 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.

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.

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`

None	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.
- The SQL VARCHAR2 data type is mapped to the R character data type through the embedded R input data argument. Users can convert the character variable to a factor in R if needed by using `as.factor()`.

Related Topics:

R Operators and Functions Supported by Oracle R Enterprise (page C-1)

1.4.2.2 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.

Oracle R Enterprise has both ordered and unordered `ore.frame` objects. 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`.

See Also:

"Moving Data to and from the Database (page 2-15)" for information on `ore.create`

"Creating Ordered and Unordered ore.frame Objects (page 2-9)".

Example 1-4 Classes of a data.frame and a Corresponding ore.frame

This example 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.

```
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 1-4 (page 1-9)

```
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"
```

1.4.2.3 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.

1.4.2.4 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 (page 2-15)".

Example 1-5 Coercing R and Oracle R Enterprise Class Types

This example 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 6-13 (page 6-27).

```
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 1-5 (page 1-11)

```
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
```

1.5 Typical Operations in Using Oracle R Enterprise

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

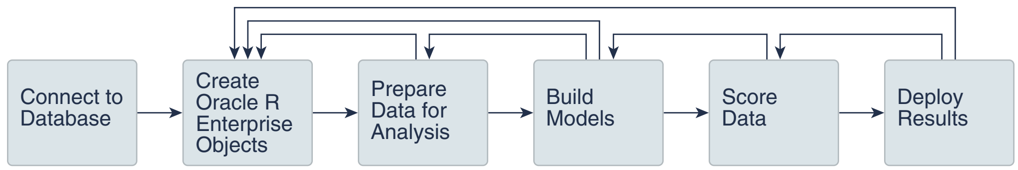
1. In an R session, connect to a schema in an Oracle Database instance.
2. Attach the schema and synchronize with the schema objects, which generates Oracle R Enterprise proxy objects for database tables.
3. Prepare the data for analysis and possibly perform exploratory data analysis and data visualization.
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 Typical Oracle R Enterprise Workflow

This figure illustrates these steps and typical reiterations of them.



1.6 Oracle R Enterprise Global Options

Oracle R Enterprise has global options that affect various functions. Table 1-2 (page 1-12) lists the Oracle R Enterprise global options and descriptions of them.

Table 1-2 Oracle R Enterprise Global Options

Global	Description
`ore.envAsEmptyenv`	A logical value that specifies whether an environment referenced in an object should be replaced with an empty environment during serialization to an Oracle Database. When `TRUE`, the referenced environment in the object is replaced with an empty environment whose parent is `.GlobalEnv`, and the objects in the original referenced environment are not serialized. In some cases, this can significantly reduce the size of serialized objects. When `FALSE`, all of the objects in the referenced environment are serialized, and can be unserialized and loaded into memory. The default value for this option is `FALSE`. The following Oracle R Enterprise functions use this global option: • `ore.push`, in saving a serialized `list` object to the database • `ore.save`, in saving objects to an Oracle R Enterprise datastore • `ore.doEval` and the other embedded R execution functions for serializing parameters of `list` type and for serializing some objects returned by an R function during embedded R execution
`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`.

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

Global	Description
`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 `|`.
`ore.trace`	A logical value that specifies 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 (page 2-10)" for information on using `ore.sep` and `ore.warn.order`
- "Support for Parallel Execution (page 6-4)"

1.7 Oracle R Enterprise Examples

Oracle R Enterprise includes several example scripts that demonstrate the use of Oracle R Enterprise functions. The following topics describe listing the example scripts and running a script:

- Listing the Oracle R Enterprise Examples (page 1-13)
- Running an Oracle R Enterprise Example Script (page 1-14)

1.7.1 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 the following example.

Example 1-6 Using demo to List Oracle R Enterprise Examples

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

Listing for Example 1-6 (page 1-13)

```
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     Hypothesis 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_em         Oracle Data Mining: expectation maximization (12.2)
odm_esa        Oracle Data Mining: explicit semantic analysis (12.2)
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_oc         Oracle Data Mining: o-cluster
odm_partition  Oracle Data Mining: partition model (12.2)
odm_ralg       Oracle Data Mining: extensible R algorithm (12.2)
odm_svd        Oracle Data Mining: singular value decomposition (12.2)
odm_svm        Oracle Data Mining: support vector machines
ore_dplyr      Data manipulation similar to dplyr
pca            Principal Component Analysis
push_pull      RDBMS <-> R data transfer
randomForest   Random forest model
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
script         Create, list, load, drop, grant, and revoke R scripts
sql_like       Mapping of R to SQL commands
stepwise       Stepwise OLS linear regression
summary        Summary functionality
table_apply    Embedded R processing of entire table
```

1.7.2 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 1-7 Running the basic.R Example Script

This example 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`, which is an `ore.frame` object. The script then demonstrates that the `iris` `data.frame` and the `IRIS` `ore.frame` have the same structure and contain the same data.

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

Listing for This Example

```
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 <- ore.push(iris)

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

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

R> nrow(IRIS)
[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)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

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

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:

2.1 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:

2.1.1 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 through the Oracle R Enterprise client interface.

This section has the following topics:

2.1.1.1 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.

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 `ore.is.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.

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. 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.

- 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.

  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 (page 2-3)" for examples of using the various connection identifiers
- "Creating R Objects for In-Database Data (page 2-4)"
- *Oracle Big Data Connectors User's Guide*
- *Oracle R Enterprise Installation and Administration Guide* for information on creating an Oracle wallet.

2.1.1.2 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.

2.1.2 Using the ore.connect and ore.disconnect Functions

The examples in this section demonstrate the various ways of specifying an Oracle R Enterprise connection to an Oracle Database instance.

The examples use sample values for the `ore.connect` argument values. Replace the sample values with the appropriate values for connecting to your database.

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

This example invokes the `ore.connect` function and specifies the `user`, `sid`, `host`, `password`, and `port` arguments.

```
ore.connect(user = "rquser", sid = "sales", host = "sales-server",
            password = "rquserStrongPassword", port = 1521 )
```

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

This example demonstrates using a service name rather than a SID. It also specifies connecting to the local host.

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

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

This example uses the `conn_string` argument to specify an easy connect string that identifies the connection.

```
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 Using ore.connect and Specifying a Full Connection String

This example uses the `conn_string` argument to specify a full connection string that identifies the connection.

```
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 Using the conn_string Argument to Specify an Oracle Wallet

This example uses the `conn_string` argument to specify an Oracle wallet. The `mydb_test` string is the connection identifier for the Oracle database. The Oracle wallet contains the information needed to create the connection. For information on creating an Oracle wallet for an Oracle R Enterprise connection, see *Oracle R Enterprise Installation and Administration Guide*.

```
ore.connect(conn_string = "mydb_test")
```

Example 2-6 Using the conn_string Argument and Specifying an Empty Connection String

This example uses an empty connection string to connect to the local host.

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

Example 2-7 Using the conn_string Argument in Connecting to a Pluggable Database

This example connects to a pluggable database using the `conn_string` argument to specify a service name.

```
ore.connect(conn_string = "pdb1.example.com")
```

Example 2-8 Using the service_name Argument in Connecting to a Pluggable Database

This example invokes `ore.connect` using a service name, host name, and port number to connect to a pluggable database.

```
ore.connect(service_name = "pdb1.example.com", host = "mypdb", port = 1521)
```

Example 2-9 Disconnecting an Oracle R Enterprise Session

This example explicitly disconnects an Oracle R Enterprise session from an Oracle database.

```
ore.disconnect()
```

2.2 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 (page 2-4)
- Moving Data to and from the Database (page 2-15)
- Creating and Deleting Database Tables (page 2-17)
- Saving and Managing R Objects in the Database (page 2-18)

2.2.1 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 (page 2-5)

- Using the ore.sync Function (page 2-6)
- Using the ore.get Function (page 2-7)
- Using the ore.attach Function (page 2-8)

2.2.1.1 About Creating R Objects for Database Objects

To gain access to the data in the database tables in the schema, you use the `ore.sync` function.

When you invoke `ore.connect` in an R session, Oracle R Enterprise creates a connection to a schema in an Oracle Database instance. The `ore.sync` 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 in the R search path.

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.

With the `query` argument, you can specify a SQL `SELECT` statement. This enables you to create an `ore.frame` for a query without creating a view in the database. This can be useful when you not have the `CREATE VIEW` system privilege for the current schema. You cannot use the `schema` argument and the `query` argument in the same `ore.sync` invocation.

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.

2.2.1.2 Using the ore.sync Function

The following example demonstrates the use of the `ore.sync` function.

The example first invokes the `ore.exec` function to create some tables to represent tables existing in the database schema for the rquser. The example then invokes `ore.sync` and specifies three tables of the schema. The `ore.sync` invocation 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 TABLE3 table exists in the schema but does not have an `ore.frame` proxy object because it was not included in the `ore.sync` invocation.

The example next invokes `ore.sync` with the `query` argument to create `ore.frame` objects for the specified SQL queries. The example lists the `ore.frame` objects again.

The example then 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 lists the R objects in the SH environment.

The example next removes the `ore.frame` objects `QUERY1`, `QUERY2`, and `TABLE4` from the rquser environment. Finally, the example lists the proxy objects in the environment again.

Note:

The `ore.rm` function invocation removes the `ore.frame` that is a proxy for the TABLE4 table from the environment. It does not delete the table from the schema.

Example 2-10 Using ore.sync to Add ore.frame Proxy Objects to an R Environment

```
# After connecting to a database as rquser, create some tables.
ore.exec("CREATE TABLE TABLE1 AS SELECT * FROM dual")
ore.exec("CREATE TABLE TABLE2 AS SELECT * FROM dual")
ore.exec("CREATE TABLE TABLE3 AS SELECT * FROM dual")
ore.exec("CREATE TABLE TABLE4 AS SELECT * FROM dual")
# 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()
# Create ore.frame objects for the specified queries.
ore.sync(query = c("QUERY1" = "SELECT 0 X, 1 Y FROM dual",
                   "QUERY2" = "SELECT 1 X, 0 Y FROM dual"))
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 objects for the specified objects.
ore.rm(c("QUERY1", "QUERY2", "TABLE4"))
# List the ore.frame proxy objects in the current environment again.
ore.ls()
```

Listing for This Example

```
R> # After connecting to a database as rquser, create some tables.
R> ore.exec("CREATE TABLE TABLE1 AS SELECT * FROM dual")
R> ore.exec("CREATE TABLE TABLE2 AS SELECT * FROM dual")
R> ore.exec("CREATE TABLE TABLE3 AS SELECT * FROM dual")
R> ore.exec("CREATE TABLE TABLE4 AS SELECT * FROM dual")
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> # Create ore.frame objects for the specified queries.
R> ore.sync(query = c("QUERY1" = "SELECT 0 X, 1 Y FROM dual",
+                     "QUERY2" = "SELECT 1 X, 0 Y FROM dual"))
R> ore.ls()
 [1] "QUERY1"     "QUERY2"     "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 objects for the specified objects.
R> ore.rm(c("QUERY1", "QUERY2", "TABLE4"))
R> # List the ore.frame proxy objects in the current environment again.
R> ore.ls()
 [1] "TABLE1"     TABLE3"
```

2.2.1.3 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. 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.

Example 2-11 Using ore.get to Get a Database Table

This 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.

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

Listing for Example 2-11 (page 2-7)

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

2.2.1.4 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 2-12 Using ore.attach to Add an Environment for a Database Schema

This example demonstrates the use of the `ore.attach` function. Comments in the example explain the function invocations.

```
# 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 2-12 (page 2-8)

```
R> # Connected as rquser.
R> # Add the environment for the rquser schema to the R search path.
R> 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
```

2.2.2 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 (page 2-9)
- Global Options Related to Ordering (page 2-10)
- Ordering Using Keys (page 2-10)
- Ordering Using Row Names (page 2-12)
- Using Ordered Frames (page 2-14)

2.2.2.1 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 2-13 (page 2-11). If the `ore.frame` object is unordered, `is.null` returns an error.

See Also:

"Indexing Data (page 3-5)"

2.2.2.2 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 2-13 (page 2-11) and Example 2-14 (page 2-12).

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" = "|")
```

2.2.2.3 Ordering Using Keys

You can use the primary key of a database table to order an `ore.frame` object.

The following example loads the spam data set from the `kernlab` package. It adds two columns to the data set.

The example invokes `ore.drop` to drop the named tables if they exist. It then invokes `ore.create` to create two tables from the data set. It invokes `ore.exec` to make the USERID and TS columns a composite primary key of the SPAM_PK table, and invokes `ore.sync` to synchronize the table with its `ore.frame` proxy.

Note:

The `ore.exec` function executes a SQL statement in the Oracle Database schema. The function is intended for database definition language (DDL) statements that have no return value.

The example then displays the first eight rows of each table. The proxy object for the SPAM_PK table is an ordered `ore.frame` object. It 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 2-13 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 This Example

```
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
```

2.2.2.4 Ordering Using Row Names

You can use row names to order an `ore.frame` object.

The following 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 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 2-14 Ordering Using Row Names

```
# 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")
# 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 This Example

```
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> # 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$b
 [1] a b c d e f g h i j j i h g f e d c b aLevels: a b c d e f g h i j
R> # Display the default row names for the first six rows of the a column.
R> row.names(head(a))
[1] 1 2 3 4 5 6
R> # SPAM_NOPK has no unique key, so row.names raises error messages.
R> row.names(head(SPAM_NOPK))
Error: ORE object has no unique key
In addition: Warning message:
ORE object has no unique key - using random order
R> # Row names consist of TS ‘|' USERID.
R> # For display on this page, only the first four row names are shown.
R> row.names(head(SPAM_PK))
        1001|351          1002|351          1003|352          1004|352
"1001|3.51E+002" "1002|3.51E+002" "1003|3.52E+002" "1004|3.52E+002"
R> # Reassign the row names to the TS column only
R> row.names(SPAM_PK) <- SPAM_PK$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
```

2.2.2.5 Using Ordered Frames

This example shows the result of merging two ordered `ore.frame` objects and two unordered `ore.frame` objects.

Example 2-15   Merging Ordered and Unordered ore.frame Objects

```
# 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")
# 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 This Example

```
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> # Uing 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> # 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
```

2.2.3 Moving Data to and from the Database

You can create a temporary database table, and its corresponding proxy `ore.frame` object, from a local R object with the `ore.push` function.

With the `ore.pull` function you can create a local R object that contains a copy of data represented by an Oracle R Enterprise proxy object

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.

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 (page 1-7)" for information on data type mappings
- "Saving and Managing R Objects in the Database (page 2-18)" for information on permanently saving the Oracle R Enterprise objects in the database
- The `push_pull.R` example script

Example 2-16 Using ore.push and ore.pull to Move Data

This example 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.

```
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 This Example

```
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"
```

2.2.4 Creating and Deleting Database Tables

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 and its associated `ore.frame` proxy object. Never use it to remove an `ore.frame` object that is not associated with a permanent database table. To remove an `ore.frame` object for a temporary database table, use the `ore.rm` function.

Example 2-17 Using ore.create and ore.drop to Create and Drop Tables

This example creates tables in the database and drops some of them.

```
# Create the AIRQUALITY table from the data.frame for the airquality data set.
ore.create(airquality, table = "AIRQUALITY")
# 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 and DF2 tables from the data.frame objects.
ore.create(df1, "DF1")
ore.create(df2, "DF2")
# Create the CARS93 table from the data.frame for the Cars93 data set.
ore.create(Cars93, table = "CARS93")
# List the Oracle R Enterprise proxy objects.
ore.ls()
# Drop the CARS93 object.
ore.drop(table = "CARS93")
# List the Oracle R Enterprise proxy objects again.
ore.ls()
```

Listing for This Example

```
R> # Create the AIRQUALITY table from the data.frame for the airquality data set.
R> ore.create(airquality, table = "AIRQUALITY")
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")
R> ore.create(df2, "DF2")
R> # Create the CARS93 table from the data.frame for the Cars93 data set.
R> ore.create(Cars93, table = "CARS93")
```

```
R> # List the Oracle R Enterprise proxy objects.
R> ore.ls()
[1] "AIRQUALITY"  "CARS93"  "DF1"  "DF2_"
R> # Drop the CARS93 object.
R> ore.drop(table = "CARS93")
R> # List the Oracle R Enterprise proxy objects again.
R> ore.ls()
[1] "AIRQUALITY"  "DF1_"  "DF2"
```

2.2.5 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 grant or revoke read privilege access to a datastore for one or more users. 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 (page 2-18)
- About Oracle R Enterprise Datastores (page 2-19)
- Saving Objects to a Datastore (page 2-19)
- Control Access to Datastores (page 2-21)
- Getting Information about Datastore Contents (page 2-22)
- Restoring Objects from a Datastore (page 2-24)
- Deleting a Datastore (page 2-26)
- About Using a datastore in Embedded R Execution (page 2-26)

2.2.5.1 About Persisting Oracle R Enterprise Objects

With Oracle R Enterprise datastores, you can save R objects in the database.

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 6-10 (page 6-22).

The following table lists the functions that manipulate datastores and provides brief descriptions of them.

Table 2-1 Functions that Manipulate Datastores

Function	Description
`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.
`ore.delete`	Deletes a datastore from the current Oracle database schema.
`ore.grant`	Grants read access to a datastore.
`ore.lazyLoad`	Lazily restores objects from a datastore into an R environment.
`ore.load`	Restores objects from a datastore into an R environment.
`ore.revoke`	Revokes read access to a datastore.
`ore.save`	Saves R objects in a new or existing datastore.

See Also:

"Using Oracle R Enterprise Embedded R Execution (page 6-1)" for information on using the R and the SQL interfaces to embedded R execution

2.2.5.2 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.

2.2.5.3 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 whether read privilege access to the datastore can be granted to other users. 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 2-18 Saving Objects and Creating a Datastore

This example demonstrates creating datastores using different combinations of arguments.

```
# 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.drop("AIRQUALITY")
ore.create(airquality, table = "AIRQUALITY")

# 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(AIRQUALITY, list = ls(), name = "ds1", description = "My private 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 This Example

```
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.drop("AIRQUALITY")
R> ore.create(airquality, table = "AIRQUALITY")
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] "AIRQUALITY"
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(AIRQUALITY, 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)
```

2.2.5.4 Control Access to Datastores

With the `ore.grant` and `ore.revoke` functions you can grant or revoke access to an Oracle R Enterprise datastore.

With the `ore.grant` and `ore.revoke` functions, you can control access to datastores. You can grant read access to a specified user to a datastore that you own or revoke the access privilege. The functions `ore.save`, `ore.load`, `ore.datastore`, and `ore.datastoreSummary` have arguments related to the accessibility of datastores.

Note:

If you use `ore.create` to create a persistent database table and its proxy `ore.frame` object, then save the proxy `ore.frame` object in a grantable datastore, and then use `ore.grant` to grant read privilege access to the datastore, the access applies only to the `ore.frame` object. The read access does not extend to the persistent database table. To grant read permission to the table itself, you must execute an appropriate SQL command.

Example 2-19 Granting and Revoking Access to a Datastore

This example pushes the airquality data set from the local R session to the Oracle database, where it exists as the `ore.frame` object `AIRQUALITY` and as a temporary database table with the same name. The example then saves the `AIRQUALITY` object to the datastore `ds3` and specifies that access to the datastore can be granted to other users. It invokes function `ore.datastore` with `type = grantable` to display all of the datastores to which read access has been granted. It grants the read privilege for the `ds3` datastore to SCOTT. It then invokes `ore.datastore` with `type = grant` to display the datastores to which read access has been granted. It revokes the read privilege for SCOTT, and again displays the datastores to which access has been granted.

```
AIRQUALITY <- ore.push(airquality)
ore.save(AIRQUALITY, name = "ds3",
```

```
         description = "My datastore 3", grantable  = TRUE)
ore.datastore(type = "grantable")
ore.datastore(type = "grant")
ore.grant("ds3", type = "datastore", user = "SCOTT")
ore.datastore(type = "grant")
ore.revoke("ds3", type = "datastore", user = "SCOTT")
ore.datastore(type = "grant")
```

Listing for This Example

```
R> AIRQUALITY <- ore.push(airquality)
R> ore.save(AIRQUALITY, name = "ds3",
+          description = "My datastore 3", grantable  = TRUE)
R> ore.datastore(type = "grantable")
  datastore.name object.count  size       creation.date    description
1            ds3            1  1451 2015-11-30 18:48:25 My datastore 3
R> ore.datastore(type = "grant")
[1] datastore.name grantee
<0 rows> (or 0-length row.names)
R> ore.grant("ds3", type = "datastore", user = "SCOTT")
R> ore.datastore(type = "grant")
  datastore.name grantee
1            ds3   SCOTT
R> ore.revoke("ds3", type = "datastore", user = "SCOTT")
R> ore.datastore(type = "grant")
[1] datastore.name grantee
<0 rows> (or 0-length row.names)
```

2.2.5.5 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. To get information about a specific type of datastore, you can use the optional character string `type` argument. The valid values for `type`are the following:

- `user`, which lists the datastores created by current session user. This is the default value.
- `private`, which lists the datastores for which read access cannot be granted by the current session user to other users.
- `all`, which lists all of the datastores to which the current session user has read access.
- `grantable`, which lists the datastores the read privilege for which can be granted by the current session user to other users.
- `grant`, which lists the datastores the read privilege for which has been granted by the current session user to other users.
- `granted`, which lists the datastores the read privilege for which has been granted by other users to the current session user.

If you do not specify a type, then function `ore.datastore` 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. If you do specify a type, then the function returns a `data.frame`that has a column for the specified type.

You can search for a datastore by name or by using a regular expression pattern.

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 required argument, the name of a datastore, and has an optional argument, the owner of the datastore.

Example 2-20 Using the ore.datastore Function

This example demonstrates using the `ore.datastore` function.

```
# 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.drop("AIRQUALITY")
ore.create(airquality, table = "AIRQUALITY")

# Save the objects to a datastore named ds1 and supply a description.
ore.save(AIRQUALITY, list = ls(), name = "ds1", description = "My private 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 other datastores.
ore.save(x, y, name = "ds2", description = "x and y")
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 This Example

```
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.drop("AIRQUALITY")
R> ore.create(airquality, table = "AIRQUALITY")
R>
R> # Save the objects to a datastore named ds1 and supply a description.
R> ore.save(AIRQUALITY, list = ls(), name = "ds1", description = "My private
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 other datastores.
R> ore.save(x, y, name = "ds2", description = "x and y")
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 1284 2017-04-21 16:08:57 For pattern matching
2          dfobj            2  656 2017-04-21 16:08:38           df objects
3            ds1            4 3439 2017-04-21 16:03:55 My private datastore
4            ds2            2  314 2017-04-21 16:04:32              x and y

R> # List the specified datastore.
R> ore.datastore("ds1")
  datastore.name object.count size       creation.date          description
1            ds1            4 3439 2017-04-21 16:03:55 My private datastore

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 1284 2017-04-21 16:08:57 For pattern matching
2            ds1            4 3439 2017-04-21 16:03:55 My private datastore
3            ds2            2  314 2017-04-21 16:04:32              x and y
```

Example 2-21   Using the ore.datastoreSummary Function

This example demonstrates using the `ore.datastoreSummary` function. The example uses the datastores created in the previous example.

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

Listing for This Example

```
R> ore.datastoreSummary("ds1")
  object.name      class size length row.count col.count
1  AIRQUALITY  ore.frame 1213      6        NA         6
2         df1 data.frame  328      2         5         2
3         df2 data.frame  328      2         5         2
4     iris_of  ore.frame 1570      5        NA         5
R> ore.datastoreSummary("ds2")
  object.name   class size length row.count col.count
1           x numeric  182     20        NA        NA
2           y    list  132      3        NA        NA
```

2.2.5.6 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 2-22   Using the ore.load Function to Restore Objects from a Datastore

This example demonstrates using the `ore.load` function to restore objects from datastores that were created in Example 2-20 (page 2-23). The example runs in the same R session as that example.

```
# List the R objects.
ls()

# List the datastores.
ore.datastore()

# Delete the x and z objects.
rm(x, z)
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 2-22 (page 2-24)

```
R> # List the R objects.
R> ls()
[1] "df1"     "df2"     "iris_of" "x"       "y"       "z"
R>
R> # List the datastores.
R> ore.datastore()
  datastore.name object.count size       creation.date  description
1     another_ds            3 1243 2014-07-24 13:31:56 For pattern mattching
2          dfobj            2  656 2014-07-24 13:31:46           df objects
3            ds1            4 3162 2014-07-24 13:25:17         My datastore
4            ds2            2 1111 2014-07-24 13:27:26               only x
R>
R> # Delete the x and z objects.
R> rm(x, z)
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 1243 2014-07-24 13:31:56 For pattern mattching
2          dfobj            2  656 2014-07-24 13:31:46            df objects
3            ds1            4 3162 2014-07-24 13:25:17          My datastore
4            ds2            2 1111 2014-07-24 13:27:26                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"
R> ls()
[1] "df1"     "df2"     "iris_of"
```

2.2.5.7 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.

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.

Example 2-23 Using the ore.delete Function

This example 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 2-18 (page 2-20).

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

Listing for Example 2-23 (page 2-26)

```
R> # Delete 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"
```

2.2.5.8 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.

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:

3.1 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:

3.1.1 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.

3.1.2 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. The examples are in the following topics:

- Selecting Data by Column (page 3-2)
- Selecting Data by Row (page 3-3)
- Selecting Data by Value (page 3-4)

3.1.2.1 Selecting Data by Column

This example selects columns from an `ore.frame` object.

Example 3-1 Selecting Data by Column

This example first creates a temporary database table, with the corresponding proxy `ore.frame` object `iris_of`, from the `iris data.frame` object. It displays the first three rows of `iris_of`. The example selects two columns from `iris_of` and creates the `ore.frame` object `iris_projected` with them. It then displays the first three rows of `iris_projected`.

```
iris_of <- ore.push(iris)
head(iris_of, 3)

iris_projected = iris_of[, c("Petal.Length", "Species")]

head (iris_projected, 3)
```

Listing for This Example

```
iris_of <- ore.push(iris)
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

R> iris_projected = iris_of[, c("Petal.Length", "Species")]

R> head (iris_projected, 3)
```

```
  Petal.Length Species
1          1.4  setosa
2          1.4  setosa
3          1.3  setosa
```

3.1.2.2 Selecting Data by Row

This example selects rows from an ordered `ore.frame` object.

Example 3-2 Selecting Data by Row

This 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.

The example 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.

```
# 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 This Example

```
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
```

3.1.2.3 Selecting Data by Value

This example selects portions of a data set.

Example 3-3 Selecting Data by Value

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.

```
iris_of <- ore.push(iris)
# Select sepal length and species where petal length is less than 1.5.
iris_of_filtered <- iris_of[iris_of$Petal.Length < 1.5,
                            c("Sepal.Length", "Species")]
names(iris_of_filtered)
nrow(iris_of_filtered)
head(iris_of_filtered, 3)
# Alternate syntax filtering.
iris_of_filtered <- subset(iris_of, Petal.Length < 1.5)
nrow(iris_of_filtered)
head(iris_of_filtered, 3)
# Using the AND and OR conditions in filtering.
# Select all rows with in which the species is setosa or versicolor.
# and the petal width is less than 2.0.
iris_of_filtered <- iris_of[(iris_of$Species == "setosa" |
                             iris_of$Species == "versicolor") &
                             iris_of$Petal.Width < 2.0,]
nrow(iris_of_filtered)
head(iris_of, 3)
```

Listing for This Example

```
R> iris_of <- ore.push(iris)
R> # Select sepal length and species where petal length is less than 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 in which the species is setosa or versicolor.
R> # and the petal width is less than 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
```

3.1.3 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 (page 3-10)" and "Partitioning Data (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 Indexing an ore.frame Object

This example demonstrates character and integer indexing. The example uses the ordered `SPAM_PK` `ore.frame` object from Example 2-13 (page 2-11). 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.

```
# 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 This Example

```
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
```

3.1.4 Combining Data

You can join data from `ore.frame` objects that represent database tables by using the `merge` function.

Example 3-5 Joining Data from Two Tables

This 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 generates 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.

```
# 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 local 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 This Example

```
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 local 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
```

3.1.5 Summarizing Data

You can summarize data by using the `aggregate` function.

Example 3-6 Aggregating Data

This 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.

```
# 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 This Example

```
# 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` example script

3.1.6 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 Formatting Data

This example creates a function to format the data in a column.

```
# 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 This Example

```
R> # Create a function for formatting data.
R> petalCategory_fmt <- function(x) {
+    ifelse(x > 5, 'LONG',
+    ifelse(x > 2, 'MEDIUM', 'SMALL'))
+ }
R> # Create an ore.frame in database memory with the iris data set.
R> iris_of <- ore.push(iris)
R> # 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
R> # Format the data in Petal.Length column.
R> iris_of$Petal.Length <- petalCategory_fmt(iris_of$Petal.Length)
R> # 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 Using the transform Function

This example does the same thing as the previous example except that it uses the `transform` function to reformat the data in a column of the data set.

```
# 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 This Example

```
R> # Create an ore.frame in database memory with the iris data set.
R> iris_of2 <- ore.push(iris)
R> # 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 Adding Derived Columns

This example uses the `transform` function to add a derived column to the data set and then to add additional columns to it.

```
# 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 This Example

```
R> # Set the page width.
R> options(width = 80)
R> # 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` example script

3.1.7 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 (page 2-9).

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

Example 3-10   Simple Random Sampling

This example 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.

```
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 This Example

```
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 Split Data Sampling

This example 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 the previous example.

This example 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.

```
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 This Example

```
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 Systematic Sampling

This example 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 the first example.

```
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 This Example

```
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 Stratified Sampling

This example 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`.

```
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 This Example

```
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 Cluster Sampling

This example 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.

```
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 This Example

```
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 Quota Sampling

This example 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.

```
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 This Example

```
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
```

3.1.8 Partitioning Data

In analyzing large data sets, a typical operation is to randomly partition the data set into subsets.

You can analyze the partitions by using Oracle R Enterprise embedded R execution, as shown in the following example.

Example 3-16 Randomly Partitioning Data

This 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.

```
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 This Example

```
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
```

3.1.9 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, 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 Aggregating Date and Time Data

This example 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`.

```
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 This Example

```
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   Using Date and Time Arithmetic

This example creates a one day shift by taking the `datetime` column of the `MYDATA` `ore.frame` object created in the previous example 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.

This example 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.

```
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 This Example

```
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   Comparing Dates and Times

This example demonstrates date and time comparisons. The example uses the `datetime` column of the `MYDATA` `ore.frame` object created in the first example. This example 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`.

```
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 This Example

```
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
```

Example 3-20 Using Date and Time Accessors

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. This example demonstrates the use of these functions. The example uses the `datetime` column of the `MYDATA` `ore.frame` object created in the first example.

This example 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.

```
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 This Example

```
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 Coercing Date and Time Data Types

This example 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 the first example. That column contains `ore.datetime` values. This example 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.

```
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 This Example

```
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   Using a Window Function

This example 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 the first example. This 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 the third example 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.

```
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 This Example

```
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
```

3.2 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:

3.2.1 About the Exploratory Data Analysis Functions

The Oracle R Enterprise functions for exploratory data analysis are in the `OREeda` 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.

Table 3-1 (Cont.) Functions in the OREeda Package

Function	Description
`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.

3.2.2 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 the following example. Some of the columns are numeric, others are not.

Example 3-23 The NARROW Data Set

This example shows the class, dimensions, and names of the `NARROW` object.

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

Listing for This Example

```
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"
```

3.2.3 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.

Example 3-24 Performing Basic Correlation Calculations

This example demonstrates how to specify the different types of correlation statistics.

```
# 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 This Example

```
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 Creating Correlation Matrices

This example 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.

```
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 This Example

```
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` example script

3.2.4 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 (page 3-23)".

The most basic use case is to create a single-column frequency table, as shown in Example 3-26 (page 3-25).

Example 3-26 Creating a Single Column Frequency Table

This example filters the NARROW `ore.frame`, grouping by GENDER.

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

Listing for Example 3-26 (page 3-25)

```
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 Analyzing Two Columns

This example analyses AGE by GENDER and AGE by CLASS.

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

Listing for Example 3-27 (page 3-26)

```
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.
```

Example 3-28 Weighting Rows

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

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

Listing for Example 3-28 (page 3-26)

```
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 Ordering Cross-Tabulated Data

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

This example orders by frequency count and then by reverse order by frequency count.

```
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 (page 3-26)

```
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 Analyzing Three or More Columns

This example demonstrates analyzing three or more columns. The result is similar to what the SQL `GROUPING SETS` clause accomplishes.

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

Listing for Example 3-30 (page 3-27)

```
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.
```

Example 3-31 Specifying a Range of Columns

You can specify a range of columns instead of having to type all the column names, as demonstrated in this example.

```
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 (page 3-28)

```
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 Producing One Cross-Tabulation Table for Each Value of Another Column

This example produces one cross-tabulation table (AGE, GENDER) for *each* unique value of another column COUNTRY.

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

Listing for Example 3-32 (page 3-28)

```
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
```

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

You can extend the cross-tabulation to more than one column, as shown in this example, which produces one (AGE, EDUCATION) table for each unique combination of (COUNTRY, GENDER).

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

Listing for Example 3-33 (page 3-28)

```
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
```

Example 3-34 Augmenting Cross-Tabulation with Stratification

All of the cross-tabulation tables in the previous examples can be augmented with stratification, as shown in this example.

```
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 (page 3-29)

```
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 Binning Followed by Cross-Tabulation

This example does a custom binning by AGE and then calculates the cross-tabulation for GENDER and the bins.

```
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 (page 3-29)

```
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
```

3.2.5 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 (page 3-30).

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

Example 3-36 Using the ore.freq Function

This example pushes the `iris` data set to the database and gets the `ore.frame` object `iris_of`. The example gets a crosstab and invokes the `ore.freq` function on it.

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

Listing for Example 3-36 (page 3-30)

```
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
```

3.2.6 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.

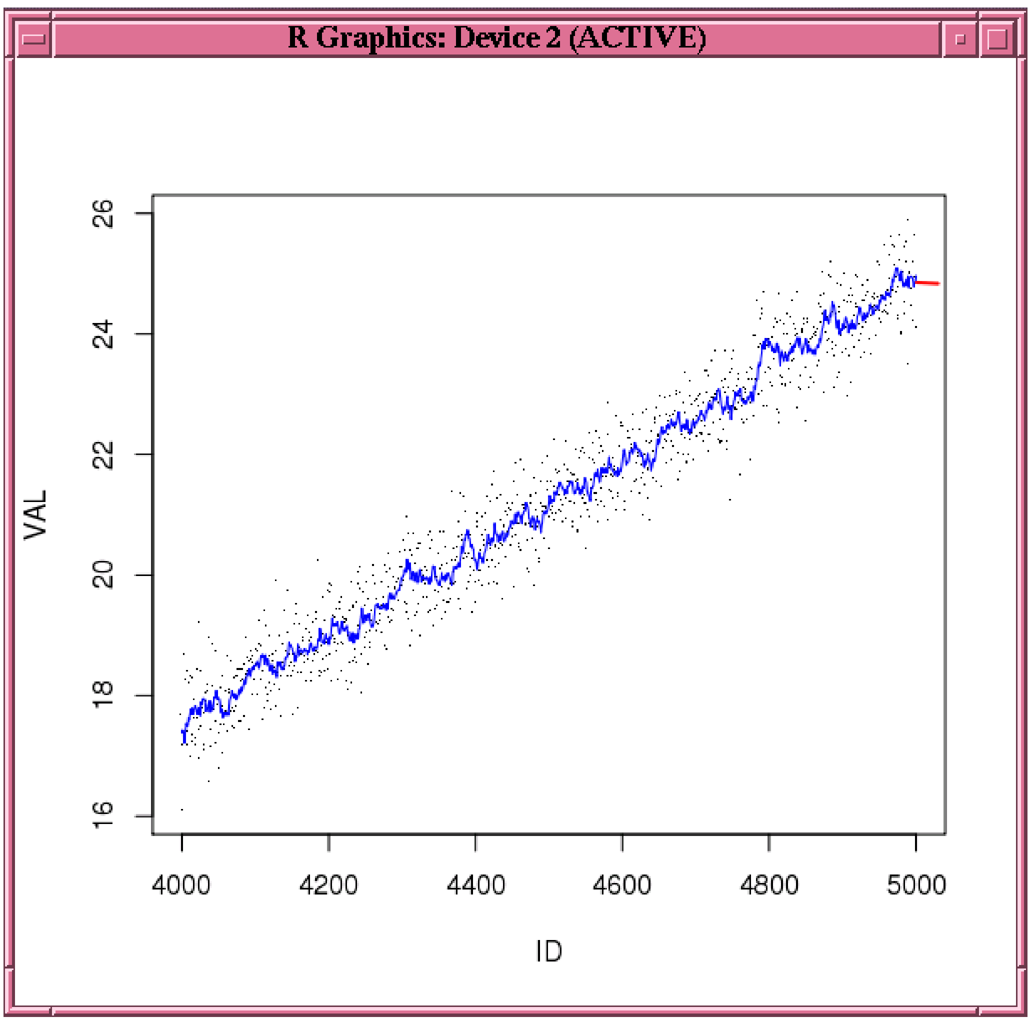
For information about the arguments of the `ore.esm` function, invoke `help(ore.esm)`.

Example 3-37 Building a Double Exponential Smoothing Model

This example 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 (page 3-31) shows the observations, fitted values, and the predictions.

```
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 Building a Time Series Model with Transactional Data

This example 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.

```
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 (page 3-31)

```
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
```

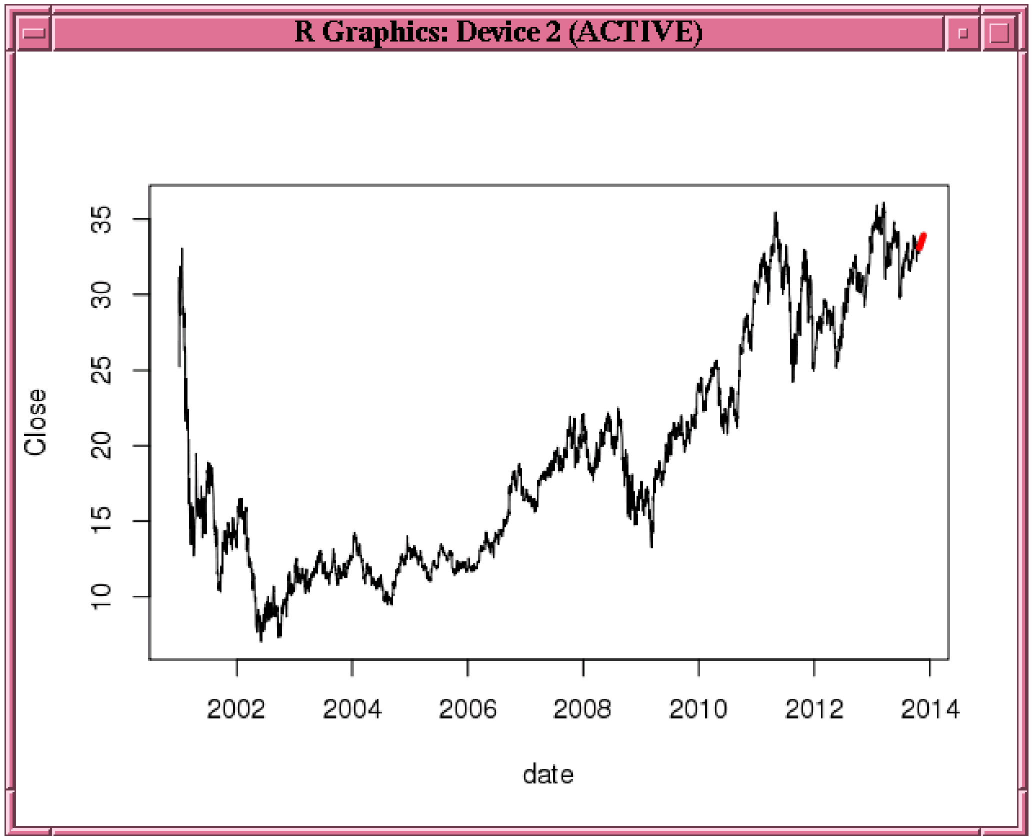
Example 3-39 Building a Double Exponential Smoothing Model Specifying an Interval

This example 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 (page 3-33).

```
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



3.2.7 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 Ranking Two Columns

This example ranks the two columns AGE and CLASS and reports the results as derived columns; values are ranked in the default order, which is ascending.

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

Example 3-41 Handling Ties in Ranking

This example ranks the two columns AGE and CLASS. If there is a tie, the smallest value is assigned to all tied values.

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

Example 3-42 Ranking by Groups

This example ranks the two columns AGE and CLASS and then ranks the resulting values according to COUNTRY.

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

Example 3-43 Partitioning into Deciles

To partition the columns into a different number of partitions, change the value of `groups`. For example, `groups=4` partitions into quartiles. This example ranks the two columns AGE and CLASS and partitions the columns into deciles (10 partitions).

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

Example 3-44 Estimating Cumulative Distribution Function

This example ranks the two columns AGE and CLASS and estimates the cumulative distribution function for both column.

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

Example 3-45 Scoring Ranks

This example 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.

```
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')
```

3.2.8 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 sorting done by the `ore.sort` function takes place 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. Some examples use the `ONTIME_S` data set.

Example 3-46 Sorting Columns in Descending Order

This example sorts the columns AGE and GENDER in descending order.

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

Example 3-47 Sorting Different Columns in Different Orders

This example sorts AGE in descending order and GENDER in ascending order.

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

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

This example sorts by AGE and keep one row per unique value of AGE:

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

Example 3-49 Removing Duplicate Columns

This example sorts by AGE and removes duplicate rows:

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

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

This example sorts by AGE, removes duplicate rows, and returns one row per unique value of AGE.

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

Example 3-51 Preserving Relative Order in the Output

This example maintains the relative order in the sorted output.

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

Example 3-52 Sorting Two Columns in Different Orders

This example sorts ONTIME_S by airline name in descending order and departure delay in ascending order.

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

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

This example sorts ONTIME_S by airline name and departure delay and selects one of each combination (that is, returns a unique key).

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

3.2.9 Summarizing Data with ore.summary

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 Calculating Default Statistics

This example calculates the mean, minimum, and maximum values for columns AGE and CLASS and rolls up (aggregates) the GENDER column.

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

Example 3-55 Calculating Skew and Probability for t Test

This example calculates the skew of AGE as column A and the probability of the Student's *t* distribution for CLASS as column B.

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

Example 3-56 Calculating the Weighted Sum

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

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

Example 3-57 Grouping by Two Columns

This example groups CLASS by GENDER and MARITAL_STATUS.

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

Example 3-58 Grouping by All Possible Ways

This example groups CLASS in all possible ways by GENDER and MARITAL_STATUS.

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

Example 3-59 Getting the Maximum Values of Columns Using ore.summary

This example lists the maximum value and corresponding species of the Sepal.Length and Sepal.Width columns in the IRIS `ore.frame`.

```
IRIS <- ore.push(iris)
ore.summary(IRIS, c("Sepal.Length", "Sepal.Width"),
                    "max",
                    maxid=c(Sepal.Length="Species", Sepal.Width="Species"))
```

Listing for Example 3-59 (page 3-37)

```
R> IRIS <- ore.push(iris)
R> ore.summary(IRIS, c("Sepal.Length", "Sepal.Width"),
+                       "max",
+                       maxid=c(Sepal.Length="Species", Sepal.Width="Species"))
  FREQ MAX(Sepal.Length) MAX(Sepal.Width) MAXID(Sepal.Length->Species)
MAXID(Sepal.Width->Species)
1  150               7.9             4.4
virginica                      setosa
Warning message:
ORE object has no unique key - using random order
```

3.2.10 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 3-60 Calculating the Default Univariate Statistics

This example calculates the default univariate statistics for AGE, YRS_RESIDENCE, and CLASS.

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

Example 3-61 Calculating the Default Univariate Statistics

This example calculates location statistics for YRS_RESIDENCE.

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

Example 3-62 Calculating the Complete Quantile Statistics

This example calculates complete quantile statistics for AGE and YRS_RESIDENCE.

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

3.2.11 Principal Component Analysis

The overloaded `prcomp` and `princomp` functions perform principal component analysis in parallel in the database.

The `prcomp` function uses a singular value decomposition of the covariance and correlations between variables. The `princomp` function uses eigen decomposition of the covariance and correlations between samples.

The transparency layer methods `ore.frame-prcomp` and `ore.frame-princomp` enable you to use the generic functions `prcomp` and `princomp` on data in an `ore.frame` object. This allows the functions to execute in parallel processes in the database.

For both functions, the methods support the function signature that accepts an `ore.frame` as the `x` argument and the signature that accepts a formula. The `ore.frame` must contain only numeric data. The formula must refer only to numeric variables and have no response variable.

Function `prcomp` returns a `prcomp` object and function `princomp` returns a `princomp` object.

For details about the function arguments, invoke `help('ore.frame-prcomp')` and `help('ore.frame-princomp')`.

Note:

The `biplot` function is not supported for the objects returned by these transparency layer methods.

Example 3-63 Using the prcomp and princomp Functions

```
USARRESTS <- ore.push(USArrests)

# Using prcomp

prcomp(USARRESTS)
prcomp(USARRESTS, scale. = TRUE)

# Formula interface
prcomp(~ Murder + Assault + UrbanPop, data = USARRESTS, scale. = TRUE)

# Using prcomp

princomp(USARRESTS)
princomp(USARRESTS, cor = TRUE)

# Formula interface
princomp(~ Murder + Assault + UrbanPop, data = USARRESTS, cor = TRUE)
```

Listing for Example 3-63 (page 3-38)

```
R> USARRESTS <- ore.push(USArrests)
R>
R> # Using prcomp
R>
R> prcomp(USARRESTS)
Standard deviations:
[1] 83.732400 14.212402  6.489426  2.482790

Rotation:
                PC1         PC2         PC3         PC4
Murder   0.04170432 -0.04482166  0.07989066 -0.99492173
Assault  0.99522128 -0.05876003 -0.06756974  0.03893830
UrbanPop 0.04633575  0.97685748 -0.20054629 -0.05816914
```

```
Rape     0.07515550  0.20071807  0.97408059  0.07232502

R> prcomp(USARRESTS, scale. = TRUE)
Standard deviations:
[1] 1.5748783 0.9948694 0.5971291 0.4164494

Rotation:
               PC1        PC2        PC3         PC4
Murder   0.5358995 -0.4181809  0.3412327  0.64922780
Assault  0.5831836 -0.1879856  0.2681484 -0.74340748
UrbanPop 0.2781909  0.8728062  0.3780158  0.13387773
Rape     0.5434321  0.1673186 -0.8177779  0.08902432
R>
R> # Formula interface
R> prcomp(~ Murder + Assault + UrbanPop, data = USARRESTS, scale. = TRUE)
Standard deviations:
[1] 1.3656547 0.9795415 0.4189100

Rotation:
               PC1         PC2        PC3
Murder   0.6672955 -0.30345520  0.6801703
Assault  0.6970818 -0.06713997 -0.7138411
UrbanPop 0.2622854  0.95047734  0.1667309
R>
R> # Using princomp
R>
R> princomp(USARRESTS)
Call:
princomp(USARRESTS)

Standard deviations:
   Comp.1    Comp.2    Comp.3    Comp.4
82.890847 14.069560  6.424204  2.457837

 4  variables and  50 observations.
R> princomp(USARRESTS, cor = TRUE)
Call:
princomp(USARRESTS, cor = TRUE)

Standard deviations:
   Comp.1    Comp.2    Comp.3    Comp.4
1.5748783 0.9948694 0.5971291 0.4164494

 4  variables and  50 observations.
R>
R> # Formula interface
R> princomp(~ Murder + Assault + UrbanPop, data = USARRESTS, cor = TRUE)
Call:
princomp(~Murder + Assault + UrbanPop, data = USARRESTS, cor = TRUE)

Standard deviations:
   Comp.1    Comp.2    Comp.3
1.3656547 0.9795415 0.4189100

 3  variables and  50 observations.
```

3.2.12 Singular Value Decomposition

The overloaded `svd` function performs singular value decomposition in parallel in the database.

The `svd` function accepts an `ore.frame` or an `ore.tblmatrix` object as the `x` argument. The `ore.frame-svd` method distributes block SVD computation to parallel processes executing in the database. The method uses the global option `ore.parallel` to determine the degree of parallelism to employ.

The function returns a `list` object that contains the `d` vector and `v` matrix components of a singular value decomposition of argument `x`. It does not return the left singular vector matrix `u`, therefore the argument `nu` is not used.

For details about the function arguments, invoke `help('ore.frame-svd')`.

Example 3-64 Using the svd Function

```
USARRESTS <- ore.push(USArrests)
svd(USARRESTS)
```

Listing for Example 3-64 (page 3-40)

```
R> USARRESTS <- ore.push(USArrests)
R> svd(USARRESTS)
$d
[1] 1419.06140  194.82585   45.66134   18.06956

$v
           [,1]        [,2]        [,3]        [,4]
[1,] 0.04239181 -0.01616262  0.06588426  0.99679535
[2,] 0.94395706 -0.32068580 -0.06655170 -0.04094568
[3,] 0.30842767  0.93845891 -0.15496743  0.01234261
[4,] 0.10963744  0.12725666  0.98347101 -0.06760284
```

3.3 Data Manipulation Using OREdplyr

`OREdplyr` package functions transparently implement `dplyr` functions for use with `ore.frame` and `ore.numeric` objects.

Many of these functions have non-standard evaluation (NSE) and standard evaluation (SE) interfaces. The SE functions have an underscore (_) appended to the function name. NSE functions are useful in interactive R sessions; SE functions are convenient for use in programs.

The functions in the `OREdplyr` package are described in the following topics.

Related Topics:

Select and Order Data (page 3-41)
: `OREdplyr` functions for selecting and ordering data in columns and rows of an `ore.frame` object.

Join Rows (page 3-47)
: `OREdplyr` functions for joining rows.

Group Columns and Rows (page 3-48)
: `OREdplyr` functions for grouping columns and rows.

Aggregate Columns and Rows (page 3-51)
: `OREdplyr` functions for aggregating columns and rows.

Sample Rows (page 3-53)
: `OREdplyr` functions for sampling rows.

Rank Rows (page 3-55)
: `OREdplyr` functions for ranking rows.

3.3.1 Select and Order Data

`OREdplyr` functions for selecting and ordering data in columns and rows of an `ore.frame` object.

Table 3-2 Selecting and Ordering Columns and Rows

Function	Description
`arrange` `arrange_`	Orders rows by the specified columns.
`desc`	Sorts an `ore.number`, `ore.factor`, or `ore.character` object in descending order
`distinct` `distinct_`	Selects unique rows from an input `ore.frame` object over the specified columns.
`filter` `filter_`	Filters rows by matching the specified condition.
`mutate` `mutate_`	Adds new columns.
`rename` `rename_`	Renames the specified columns and keeps all columns.
`select` `select_`	Selects only the specified columns.
`slice` `slice_`	Selects rows by position; ignores the grouping of the input ordered `ore.frame` object.
`tranmute` `tranmute_`	Adds new columns and drops the existing columns.

3.3.1.1 Examples of Selecting Columns

Examples of the `select` and `rename` functions of the `OREdplyr` package.

Example 3-65 Selecting Columns

The following examples select columns from the IRIS `ore.frame` object that is created by using the `ore.push` function on the iris `data.frame` objects.

```
IRIS <- ore.push(iris)
# Select the specified column
names(select(IRIS, Petal.Length))
names(select(IRIS, petal_length = Petal.Length))

# Drop the specified column
names(select(IRIS, -Petal.Length))

# rename() keeps all variables
names(rename(IRIS, petal_length = Petal.Length))
```

Listing for This Example

```
R> IRIS <- ore.push(iris)
R> # Select the specified column
R> names(select(IRIS, Petal.Length))
[1] "Petal.Length"
R> names(select(IRIS, petal_length = Petal.Length))
[1] "petal_length"
R>
R> # Drop the specified column
R> names(select(IRIS, -Petal.Length))
[1] "Sepal.Length" "Sepal.Width"  "Petal.Width"  "Species"
R>
R> # rename() keeps all variables
R> names(rename(IRIS, petal_length = Petal.Length))
[1] "Sepal.Length" "Sepal.Width"  "petal_length" "Petal.Width"  "Species"
```

3.3.1.2 Examples of Programming with select_

Examples of the `select_` function of the `OREdplyr` package.

Example 3-66 Programming with select

This example uses the `select_` function to select columns from the IRIS `ore.frame` object that is created by using the `ore.push` function on the iris `data.frame` object.

```
IRIS <- ore.push(iris)
# Use ~, double quote, or quote function to specify the column to select
head(select_(IRIS, ~Petal.Length))
head(select_(IRIS, "Petal.Length"))
head(select_(IRIS, quote(-Petal.Length), quote(-Petal.Width)))
head(select_(IRIS, .dots = list(quote(-Petal.Length), quote(-Petal.Width))))
```

Listing for This Example

```
R> IRIS <- ore.push(iris)
R> # Use ~, double quote, or quote function to specify the column to select
R> head(select_(IRIS, ~Petal.Length))
  Petal.Length
1          1.4
2          1.4
3          1.3
4          1.5
5          1.4
6          1.7
R> head(select_(IRIS, "Petal.Length"))
  Petal.Length
1          1.4
2          1.4
3          1.3
4          1.5
5          1.4
6          1.7
R> head(select_(IRIS, quote(-Petal.Length), quote(-Petal.Width)))
  Sepal.Length Sepal.Width Species
1          5.1         3.5  setosa
2          4.9         3.0  setosa
3          4.7         3.2  setosa
4          4.6         3.1  setosa
5          5.0         3.6  setosa
6          5.4         3.9  setosa
R> head(select_(IRIS, .dots = list(quote(-Petal.Length), quote(-Petal.Width))))
```

```
  Sepal.Length Sepal.Width Species
1          5.1         3.5  setosa
2          4.9         3.0  setosa
3          4.7         3.2  setosa
4          4.6         3.1  setosa
5          5.0         3.6  setosa
6          5.4         3.9  setosa
```

3.3.1.3 Examples of Selecting Distinct Columns

Examples of the `distinct` and `arrange` functions of the `OREdplyr` package.

Example 3-67 Selecting Distinct Columns

```
df <- data.frame(
  x = sample(10, 100, rep = TRUE),
  y = sample(10, 100, rep = TRUE)
)
DF <- ore.push(df)
nrow(DF)
nrow(distinct(DF))
arrange(distinct(DF, x), x)
arrange(distinct(DF, y), y)

# Use distinct on computed variables
arrange(distinct(DF, diff = abs(x - y)), diff)
```

Listing for This Example

```
R> df <- data.frame(
+   x = sample(10, 100, rep = TRUE),
+   y = sample(10, 100, rep = TRUE)
+ )
R> DF <- ore.push(df)
R> nrow(DF)
[1] 100
R> nrow(distinct(DF))
[1] 66
R> arrange(distinct(DF, x), x)
    x
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
9   9
10 10
R> arrange(distinct(DF, y), y)
    y
1   1
2   2
3   3
4   4
5   5
6   6
7   7
8   8
9   9
R>
```

```
R> # Use distinct on computed variables
R> arrange(distinct(DF, diff = abs(x - y)), diff)
   diff
1     0
2     1
3     2
4     3
5     4
6     5
7     6
8     7
9     8
10    9
```

3.3.1.4 Examples of Selecting Rows by Position

Examples of the `slice` and `filter` functions of the `OREdplyr` package.

Example 3-68 Selecting Rows by Position

```
MTCARS <- ore.push(mtcars)
# Display the names of the rows in MTCARS
rownames(MTCARS)
# Select the first row
slice(MTCARS, 1L)

# Arrange the rows by horsepower, then select the first row by position
MTCARS <- arrange(MTCARS, hp)
slice(MTCARS, 1L)

by_cyl <- group_by(MTCARS, cyl)
# Grouping is ignored by slice.
slice(by_cyl, 1:2)
# Use filter and row_number to obtain slices per group.
filter(by_cyl, row_number(hp) < 3L)
```

Listing for This Example

```
R> MTCARS <- ore.push(mtcars)
R> # Display the names of the rows in MTCARS
R> rownames(MTCARS)
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"          "Hornet 4 
Drive"      "Hornet Sportabout"
 [6] "Valiant"             "Duster 360"          "Merc 240D"           "Merc 
230"            "Merc 280"
[11] "Merc 280C"           "Merc 450SE"          "Merc 450SL"          "Merc 
450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"            "Honda 
Civic"         "Toyota Corolla"
[21] "Toyota Corona"       "Dodge Challenger"    "AMC Javelin"         "Camaro 
Z28"          "Pontiac Firebird"
[26] "Fiat X1-9"           "Porsche 914-2"       "Lotus Europa"        "Ford Pantera 
L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"
R> # Select the first row
R> slice(MTCARS, 1L)
          mpg cyl disp  hp drat   wt  qsec vs am gear carb
Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
R>
R> # Arrange the rows by horsepower, then select the first row by position
R> MTCARS <- arrange(MTCARS, hp)
R> slice(MTCARS, 1L)
```

```
   mpg cyl disp hp drat    wt  qsec vs am gear carb
1 30.4   4 75.7 52 4.93 1.615 18.52  1  1    4    2
R>
R> by_cyl <- group_by(MTCARS, cyl)
R> # Grouping is ignored by slice
R> slice(by_cyl, 1:2)
   mpg cyl  disp hp drat    wt  qsec vs am gear carb
1 30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2
2 24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2
Warning message:
In slice_.ore.frame(.data, .dots = .ore.dplyr.exprall(..., env = parent.frame())) :
  grouping is ignored
R> # Use filter and row_number to obtain slices per group
R> filter(by_cyl, row_number(hp) < 3L)
   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
2 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
3 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
5 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
6 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
```

3.3.1.5 Examples of Arranging Columns

Examples of the `arrange` and `desc` functions of the `OREdplyr` package.

Example 3-69 Arranging Columns

This example arranges columns from the `ore.frame` object MTCARS that is created by using the `ore.push` function on the mtcars `data.frame` object. The second `arrange()` invocation calls the `desc()` function to arrange the values in descending order.

```
MTCARS <- ore.push(mtcars)
head(arrange(mtcars, cyl, disp))
head(arrange(MTCARS, desc(disp)))
```

Listing for This Example

```
R> MTCARS <- ore.push(mtcars)
R> head(arrange(MTCARS, cyl, disp))
   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
2 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
3 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
4 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
5 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
6 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
R> head(arrange(MTCARS, desc(disp)))
   mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
2 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
3 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
4 19.2   8  400 175 3.08 3.845 17.05  0  0    3    2
5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
6 14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
```

3.3.1.6 Examples of Filtering Columns

Examples of the `filter` function of the `OREdplyr` package.

Example 3-70 Filtering Columns

This example filters columns from the MTCARS `ore.frame` object that is created by using the `ore.push` function on the mtcars `data.frame` object.

```
MTCARS <- ore.push(mtcars)
head(filter(MTCARS, cyl == 8))
# Using multiple criteria
head(filter(MTCARS, cyl < 6 & vs == 1))

# Using multiple arguments is the equivalent to using &
head(filter(MTCARS, cyl < 6, vs == 1))
```

Listing for This Example

```
R> MTCARS <- ore.push(mtcars)
R> head(filter(MTCARS, cyl == 8))
   mpg cyl  disp  hp drat   wt  qsec vs am gear carb
1 18.7   8 360.0 175 3.15 3.44 17.02  0  0    3    2
2 14.3   8 360.0 245 3.21 3.57 15.84  0  0    3    4
3 16.4   8 275.8 180 3.07 4.07 17.40  0  0    3    3
4 17.3   8 275.8 180 3.07 3.73 17.60  0  0    3    3
5 15.2   8 275.8 180 3.07 3.78 18.00  0  0    3    3
6 10.4   8 472.0 205 2.93 5.25 17.98  0  0    3    4
R> head(filter(MTCARS, cyl < 6 & vs == 1))
   mpg cyl  disp hp drat    wt  qsec vs am gear carb
1 22.8   4 108.0 93 3.85 2.320 18.61  1  1    4    1
2 24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2
3 22.8   4 140.8 95 3.92 3.150 22.90  1  0    4    2
4 32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1
5 30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2
6 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1
R>
R> # Using multiple arguments is the equivalent to using &
R> head(filter(MTCARS, cyl < 6, vs == 1))
   mpg cyl  disp hp drat    wt  qsec vs am gear carb
1 22.8   4 108.0 93 3.85 2.320 18.61  1  1    4    1
2 24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2
3 22.8   4 140.8 95 3.92 3.150 22.90  1  0    4    2
4 32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1
5 30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2
6 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1
```

3.3.1.7 Examples of Mutating Columns

Examples of the `mutate` and `transmute` functions of the `OREdplyr` package.

Example 3-71 Mutating Columns

This example uses the MTCARS `ore.frame` object that is created by using the `ore.push` function on the mtcars `data.frame` object.

The `mutate` function adds the extra column displ_1 with the value derived from that of column disp. Setting the column to NULL removes the column.

```
MTCARS <- ore.push(mtcars)
head(mutate(MTCARS, displ_l = disp / 61.0237))
head(transmute(MTCARS, displ_l = disp / 61.0237))
head(mutate(MTCARS, cyl = NULL))
head(mutate(MTCARS, cyl = NULL, hp = NULL, displ_l = disp / 61.0237))
```

Listing for This Example

```
R> MTCARS <- ore.push(mtcars)
R> head(mutate(MTCARS, displ_l = disp / 61.0237))
   mpg cyl disp  hp drat    wt  qsec vs am gear carb  displ_l
1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4 2.621932
2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4 2.621932
3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1 1.769804
4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1 4.227866
5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2 5.899347
6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1 3.687092
R> head(transmute(MTCARS, displ_l = disp / 61.0237))
   displ_l
1 2.621932
2 2.621932
3 1.769804
4 4.227866
5 5.899347
6 3.687092
R> head(mutate(mtcars, cyl = NULL))
   mpg disp  hp drat    wt  qsec vs am gear carb
1 21.0  160 110 3.90 2.620 16.46  0  1    4    4
2 21.0  160 110 3.90 2.875 17.02  0  1    4    4
3 22.8  108  93 3.85 2.320 18.61  1  1    4    1
4 21.4  258 110 3.08 3.215 19.44  1  0    3    1
5 18.7  360 175 3.15 3.440 17.02  0  0    3    2
6 18.1  225 105 2.76 3.460 20.22  1  0    3    1
R> head(mutate(mtcars, cyl = NULL, hp = NULL, displ_l = disp / 61.0237))
   mpg disp drat    wt  qsec vs am gear carb  displ_l
1 21.0  160 3.90 2.620 16.46  0  1    4    4 2.621932
2 21.0  160 3.90 2.875 17.02  0  1    4    4 2.621932
3 22.8  108 3.85 2.320 18.61  1  1    4    1 1.769804
4 21.4  258 3.08 3.215 19.44  1  0    3    1 4.227866
5 18.7  360 3.15 3.440 17.02  0  0    3    2 5.899347
6 18.1  225 2.76 3.460 20.22  1  0    3    1 3.687092
```

3.3.2 Join Rows

`OREdplyr` functions for joining rows.

Table 3-3 Joining Rows

Function	Description
`full_join`	Returns the union of rows from `left_join` and `right_join`.
`inner_join`	Returns all combination of rows from x and y over matched columns.
`left_join`	Returns rows from `inner_join` plus rows from y that do not match with x. For unmatched rows of y, NA is returned.
`right_join`	Returns rows from `inner_join` plus rows from x that do not match with y. For unmatched rows of x, NA is returned.

Example 3-72 Joining Rows

To join two tables, the `join` function selects the columns in each table that have the same name or uses the argument `by` to specify the columns.

```
MTCARS <- ore.push(mtcars)
M1 <- filter(select(MTCARS, mpg, cyl, carb), carb < 6L)
M2 <- filter(select(MTCARS, cyl, hp, carb), carb > 2L)

names(inner_join(M1, M2))
nrow(left_join(M1, M2))
nrow(right_join(M1, M2))
nrow(full_join(M1, M2))

names(M2) <- c("cyl", "hp", "carb2")
names(inner_join(M1, M2, by = c("cyl", carb="carb2")))
nrow(inner_join(M1, M2, by = c("cyl", carb="carb2")))
nrow(left_join(M1, M2, by = c("cyl", carb="carb2")))
nrow(right_join(M1, M2, by = c("cyl", carb="carb2")))
nrow(full_join(M1, M2, by = c("cyl", carb="carb2")))
```

Listing for This Example

```
R> MTCARS <- ore.push(mtcars)
R> M1 <- filter(select(MTCARS, mpg, cyl, carb), carb < 6L)
R> M2 <- filter(select(MTCARS, cyl, hp, carb), carb > 2L)
R>
R> names(inner_join(M1, M2))
[1] "cyl"  "carb" "mpg"  "hp"
R> nrow(left_join(M1, M2))
[1] 78
R> nrow(right_join(M1, M2))
[1] 63
R> nrow(full_join(M1, M2))
[1] 80
R>
R> names(M2) <- c("cyl", "hp", "carb2")
R> names(inner_join(M1, M2, by = c("cyl", carb="carb2")))
[1] "cyl"  "carb" "mpg"  "hp"
R> nrow(inner_join(M1, M2, by = c("cyl", carb="carb2")))
[1] 61
R> nrow(left_join(M1, M2, by = c("cyl", carb="carb2")))
[1] 78
R> nrow(right_join(M1, M2, by = c("cyl", carb="carb2")))
[1] 63
R> nrow(full_join(M1, M2, by = c("cyl", carb="carb2")))
[1] 80
```

3.3.3 Group Columns and Rows

`OREdplyr` functions for grouping columns and rows.

Table 3-4 Grouping Columns and Rows

Function	Description
`group_by` `group_by_`	Groups an `ore.frame` object over the specified columns.
`group_size`	Lists the number of rows in each group.
`groups`	Shows the names of the grouping columns.
`n_groups`	Returns the number of groups.

Table 3-4 (Cont.) Grouping Columns and Rows

Function	Description
`ungroup`	Drops the grouping from the input `ore.frame` object.

Example 3-73 Using Grouping Functions

The following examples use the `ore.frame` object MTCARS that is created by using the `ore.push` function on the mtcars `data.frame` object. They exemplify the use of the grouping functions `group_by`, `group_size`, `groups`, `n_group`, and `ungroup`. They also use the `OREdplyr` functions `arrange`, `rename`, and `summarize`.

```
MTCARS <- ore.push(mtcars)
by_cyl <- group_by(MTCARS, cyl)

# Apply the summarise function to each group
arrange(summarise(by_cyl, mean(disp), mean(hp)), cyl)

# Summarise drops one layer of grouping
by_vs_am <- group_by(MTCARS, vs, am)
by_vs <- summarise(by_vs_am, n = n())
arrange(by_vs, vs, am)
arrange(summarise(by_vs, n = sum(n)), vs)

# Remove grouping
summarise(ungroup(by_vs), n = sum(n))

# Group by expressions with mutate
arrange(group_size(group_by(mutate(MTCARS, vsam = vs + am), vsam)), vsam)

# Rename the grouping column
groups(rename(group_by(MTCARS, vs), vs2 = vs))

# Add more grouping columns
groups(group_by(by_cyl, vs, am))
groups(group_by(by_cyl, vs, am, add = TRUE))

# Drop duplicate groups
groups(group_by(by_cyl, cyl, cyl))

# Load the magrittr library to use the forward-pipe operator %>%
library(magrittr)
by_cyl_gear_carb <- MTCARS %>% group_by(cyl, gear, carb)
n_groups(by_cyl_gear_carb)
arrange(group_size(by_cyl_gear_carb), cyl, gear, carb)

by_cyl <- MTCARS %>% group_by(cyl)
# Number of groups
n_groups(by_cyl)

# Size of each group
arrange(group_size(by_cyl), cyl)
```

Listing for This Example

```
R> MTCARS <- ore.push(mtcars)
R> by_cyl <- group_by(MTCARS, cyl)
R>
R> # Apply the summarise function to each group
```

```
R> arrange(summarise(by_cyl, mean(disp), mean(hp)), cyl)
  cyl mean.disp.  mean.hp.
1   4   105.1364  82.63636
2   6   183.3143 122.28571
3   8   353.1000 209.21429
R>
R> # Summarise drops one layer of grouping
R> by_vs_am <- group_by(MTCARS, vs, am)
R> by_vs <- summarise(by_vs_am, n = n())
R> arrange(by_vs, vs, am)
  vs am  n
1  0  0 12
2  0  1  6
3  1  0  7
4  1  1  7
R> arrange(summarise(by_vs, n = sum(n)), vs)
  vs  n
1  0 18
2  1 14
R>
R> # Remove grouping
R> summarise(ungroup(by_vs), n = sum(n))
 n
32
R>
R> # Group by expressions with mutate
R> arrange(group_size(group_by(mutate(MTCARS, vsam = vs + am), vsam)), vsam)
  vsam  n
1    0 12
2    1 13
3    2  7
R>
R> # Rename the grouping column
R> groups(rename(group_by(MTCARS, vs), vs2 = vs))
[1] "vs2"
R>
R> # Add more grouping columns
R> groups(group_by(by_cyl, vs, am))
[[1]]
[1] "vs"

[[2]]
[1] "am"

R> groups(group_by(by_cyl, vs, am, add = TRUE))
[[1]]
[1] "cyl"

[[2]]
[1] "vs"

[[3]]
[1] "am"
R>
R> # Drop duplicate groups
R> groups(group_by(by_cyl, cyl, cyl))
[1] "cyl
R>
R> # Load the magrittr library to use the forward-pipe operator %>%
R> library(magrittr)
R> by_cyl_gear_carb <- MTCARS %>% group_by(cyl, gear, carb)
```

```
R> n_groups(by_cyl_gear_carb)
[1] 12
R> arrange(group_size(by_cyl_gear_carb), cyl, gear, carb)
   cyl gear carb n
1    4    3    1 1
2    4    4    1 4
3    4    4    2 4
4    4    5    2 2
5    6    3    1 2
6    6    4    4 4
7    6    5    6 1
8    8    3    2 4
9    8    3    3 3
10   8    3    4 5
11   8    5    4 1
12   8    5    8 1
R>
R> by_cyl <- MTCARS %>% group_by(cyl)
R> # Number of groups
R> n_groups(by_cyl)
[1] 3
R> # Number of groups
R> n_groups(by_cyl)
[1] 3
R>
R> # Size of each group
R> arrange(group_size(by_cyl), cyl)
  cyl  n
1   4 11
2   6  7
3   8 14
```

3.3.4 Aggregate Columns and Rows

`OREdplyr` functions for aggregating columns and rows.

Table 3-5 Aggregating Columns and Rows

Function	Description
`count` `count_`	Counts rows by group; similar to `tally`, but it does the `group_by` for you.
`summarise` `summarise_`	Summarizes columns by using aggregate functions. When an `ore.frame` object is grouped, the aggregate function is applied group-wise. The resulting `ore.frame` drops one grouping of the input `ore.frame`.
`tally`	Tallies rows by group; a convenient wrapper for `summarise` that either calls `n` or `sum(n)` depending on whether you're tallying for the first time or re-tallying.

Example 3-74 Aggregating Columns

The following examples use the `ore.frame` object MTCARS that is created by using the `ore.push` function on the mtcars `data.frame` object. They exemplify the use of the aggregation functions `count`, `summarize`, and `tally`. They also use the `OREdplyr` functions `arrange` and `group_by`.

```
MTCARS <- ore.push(mtcars)
arrange(tally(group_by(MTCARS, cyl)), cyl)
tally(group_by(MTCARS, cyl), sort = TRUE)

# Multiple tallys progressively roll up the groups
cyl_by_gear <- tally(group_by(MTCARS, cyl, gear), sort = TRUE)
tally(cyl_by_gear, sort = TRUE)
tally(tally(cyl_by_gear))

cyl_by_gear <- tally(group_by(MTCARS, cyl, gear), wt = hp, sort = TRUE)
tally(cyl_by_gear, sort = TRUE)
tally(tally(cyl_by_gear))

cyl_by_gear <- count(MTCARS, cyl, gear, wt = hp + mpg, sort = TRUE)
tally(cyl_by_gear, sort = TRUE)
tally(tally(cyl_by_gear))

# Load the magrittr library to use the forward-pipe operator %>%
library(magrittr)
MTCARS %>% group_by(cyl) %>% tally(sort = TRUE)

# count is more succinct and also does the grouping
MTCARS %>% count(cyl) %>% arrange(cyl)
MTCARS %>% count(cyl, wt = hp) %>% arrange(cyl)
MTCARS %>% count_("cyl", wt = hp, sort = TRUE)
```

Listing for This Example

```
R> MTCARS <- ore.push(mtcars)
R> arrange(tally(group_by(MTCARS, cyl)), cyl)
  cyl  n
1   4 11
2   6  7
3   8 14
R> tally(group_by(MTCARS, cyl), sort = TRUE)
  cyl  n
1   8 14
2   4 11
3   6  7
R>
R> # Multiple tallys progressively roll up the groups
R> cyl_by_gear <- tally(group_by(MTCARS, cyl, gear), sort = TRUE)
R> tally(cyl_by_gear, sort = TRUE)
Using n as weighting variable
  cyl  n
1   8 14
2   4 11
3   6  7
R> tally(tally(cyl_by_gear))
Using n as weighting variable
Using n as weighting variable
 n
32
R>
R> cyl_by_gear <- tally(group_by(MTCARS, cyl, gear), wt = hp, sort = TRUE)
R> tally(cyl_by_gear, sort = TRUE)
Using n as weighting variable
  cyl    n
1   8 2929
2   4  909
3   6  856
```

```
R> tally(tally(cyl_by_gear))
Using n as weighting variable
Using n as weighting variable
   n
4694
R>
R> cyl_by_gear <- count(MTCARS, cyl, gear, wt = hp + mpg, sort = TRUE)
R> tally(cyl_by_gear, sort = TRUE)
Using n as weighting variable
  cyl      n
1   8 3140.4
2   4 1202.3
3   6  994.2
R> tally(tally(cyl_by_gear))
Using n as weighting variable
Using n as weighting variable
     n
5336.9
R>
R> # Load the magrittr library to use the forward-pipe operator %>%
R> library(magrittr)
R> MTCARS %>% group_by(cyl) %>% tally(sort = TRUE)
  cyl  n
1   8 14
2   4 11
3   6  7
R>
R> # count is more succinct and also does the grouping
R> MTCARS %>% count(cyl) %>% arrange(cyl)
  cyl  n
1   4 11
2   6  7
3   8 14
R> MTCARS %>% count(cyl, wt = hp) %>% arrange(cyl)
  cyl    n
1   4  909
2   6  856
3   8 2929
R> MTCARS %>% count_("cyl", wt = hp, sort = TRUE)
  cyl    n
1   8 2929
2   4  909
3   6  856
```

3.3.5 Sample Rows

`OREdplyr` functions for sampling rows.

Table 3-6 Sampling Row Functions

Function	Description
`sample_frac`	Samples an `ore.frame` object by a fraction.
`sample_n`	Samples an `ore.frame` object by a fixed number of rows.

Example 3-75 Sampling Rows

These examples use the `ore.frame` object MTCARS that is created by using the `ore.push` function on the mtcars `data.frame` object. They exemplify the use of the sampling functions `sample_n` and `sample_frac`. They also use the `OREdplyr` functions `arrange` and `summarize`.

```
MTCARS <- ore.push(mtcars)
by_cyl <- group_by(MTCARS, cyl)

# Sample fixed number per group of rows from the entire dataset
sample_n(MTCARS, 10)
nrow(sample_n(MTCARS, 50, replace = TRUE))
sample_n(MTCARS, 10, weight = mpg)
sample_n(MTCARS, 10, weight = MTCARS[["mpg"]])

# Sample fixed number of rows per group with replacement and weight
arrange(sample_n(by_cyl, 3), cyl, mpg)
arrange(summarise(sample_n(by_cyl, 10, replace = TRUE), n = n()), cyl)
arrange(summarise(sample_n(by_cyl, 3, weight = mpg/mean(mpg)), n = n()), cyl)
arrange(summarise(sample_n(by_cyl, 3,
                           weight = by_cyl[["mpg"]]/mean(by_cyl[["mpg"]])), n = 
n()), cyl)

# Sample fixed fraction per group
nrow(sample_frac(MTCARS, 0.1))
nrow(sample_frac(MTCARS, 1.5, replace = TRUE))
nrow(sample_frac(MTCARS, 0.1, weight = 1/mpg))
```

Listing for This Example

```
R> MTCARS <- ore.push(mtcars)
R> by_cyl <- group_by(MTCARS, cyl)
R> 
R> # Sample fixed number per group of rows from the entire dataset
R> sample_n(MTCARS, 10)
                  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710|4     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Ford Pantera L|2 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Honda Civic|10   30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Lotus Europa|6   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Maserati Bora|3  15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Mazda RX4|5      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag|9  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Merc 280|8       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Toyota Corolla|7 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona|1  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
R> nrow(sample_n(MTCARS, 50, replace = TRUE))
[1] 50
R> 
R> # Sample fixed number of rows per group with replacement and weight
R> arrange(sample_n(by_cyl, 3), cyl, mpg)
  cyl  mpg  disp  hp drat    wt  qsec vs am gear carb
1   4 22.8 108.0  93 3.85 2.320 18.61  1  1    4    1
2   4 24.4 146.7  62 3.69 3.190 20.00  1  0    4    2
3   4 30.4  95.1 113 3.77 1.513 16.90  1  1    5    2
4   6 19.2 167.6 123 3.92 3.440 18.30  1  0    4    4
5   6 19.7 145.0 175 3.62 2.770 15.50  0  1    5    6
6   6 21.4 258.0 110 3.08 3.215 19.44  1  0    3    1
7   8 10.4 460.0 215 3.00 5.424 17.82  0  0    3    4
8   8 15.2 304.0 150 3.15 3.435 17.30  0  0    3    2
9   8 15.2 275.8 180 3.07 3.780 18.00  0  0    3    3
R> arrange(summarise(sample_n(by_cyl, 10, replace = TRUE), n = n()), cyl)
  cyl  n
1   4 10
```

```
2   6 10
3   8 10
R> arrange(summarise(sample_n(by_cyl, 3, weight = mpg/mean(mpg)), n = n()), cyl)
  cyl n
1   4 3
2   6 3
3   8 3
R> arrange(summarise(sample_n(by_cyl, 3, weight = by_cyl[["mpg"]]/
mean(by_cyl[["mpg"]])), n = n()), cyl)
  cyl n
1   4 3
2   6 3
3   8 3
R>
R> nrow(sample_frac(MTCARS, 0.1))
[1] 3
R> nrow(sample_frac(MTCARS, 1.5, replace = TRUE))
[1] 48
R> nrow(sample_frac(MTCARS, 0.1, weight = 1/mpg))
[1] 3
```

3.3.6 Rank Rows

`OREdplyr` functions for ranking rows.

The ranking functions rank the elements in an ordered `ore.vector` by its values. An `ore.character` is coerced to an `ore.factor`. The values of an `ore.factor` are based upon factor levels. To reverse the direction of the ranking, use the `desc` function.

Table 3-7 Ranking Rows

Function	Description
`cume_dist`	A cumulative distribution function: returns the proportion of all values that are less than or equal to the current rank.
`dense_rank`	Like `min_rank` but with no gaps between ranks.
`first`	Gets the first value from an ordered `ore.vector` object.
`last`	Gets the last value from an ordered `ore.vector` object.
`min_rank`	Equivalent to `rank(ties.method = "min")`.
`nth`	Obtains the value at the specified position in the order.
`ntile`	A rough ranking that breaks the input vector into *n* buckets.
`n_distinct`	Gets the *n*th value from an ordered `ore.vector` object.
`percent_rank`	Returns a number between 0 and 1 that is computed by rescaling `min_rank` to `[0, 1]`.
`row_number`	Equivalent to `rank(ties.method = "first")`.
`top_n`	Selects the top or bottom number of rows.

Example 3-76 Ranking Rows

These examples use the ranking functions `row_number`, `min_rank`, `dense_rank`, `percent_rank`, `cume_dist`, and `ntile`.

```
X <- ore.push(c(5, 1, 3, 2, 2, NA))

row_number(X)
row_number(desc(X))

min_rank(X)

dense_rank(X)

percent_rank(X)

cume_dist(X)

ntile(X, 2)
ntile(ore.push(runif(100)), 10)

MTCARS <- ore.push(mtcars)
by_cyl <- group_by(MTCARS, cyl)

# Using ranking functions with an ore.frame
head(mutate(MTCARS, rank = row_number(hp)))

head(mutate(MTCARS, rank = min_rank(hp)))

head(mutate(MTCARS, rank = dense_rank(hp)))

# Using ranking functions with a grouped ore.frame
head(mutate(by_cyl, rank = row_number(hp)))

head(mutate(by_cyl, rank = min_rank(hp)))

head(mutate(by_cyl, rank = dense_rank(hp)))
```

Listing for This Example

```
R> X <- ore.push(c(5, 1, 3, 2, 2, NA))
R>
R> row_number(X)
[1] 5 1 4 2 3 6
R> row_number(desc(X))
[1] 1 5 2 3 4 6
R>
R> min_rank(X)
[1] 5 1 4 2 2 6
R>
R> dense_rank(X)
[1] 4 1 3 2 2 6
R>
R> percent_rank(X)
[1] 0.8 0.0 0.6 0.2 0.2 1.0
R>
R> cume_dist(X)
[1] 0.8333333 0.1666667 0.6666667 0.5000000 0.5000000 1.0000000
R>
R> ntile(X, 2)
[1] 2 1 2 1 1 2
R> ntile(ore.push(runif(100)), 10)
```

```
  [1]  6 10  5  2  1  1  8  3  8  8  7  3 10  3  7  9  9  4  4 10 10  7  2  3  7  4
5  5  3  9  4  6  8  4 10  6  1  5  5  4  6  9
 [43]  5  8  2  7  7  1  2  9  1  2  8  5  6  5  3  4  7  1  3  1 10  1  5  5 10  9
2  3  9  6  6  8  8  6  3  7  2  2  8  4  1  9
 [85]  6 10  4 10  7  2  9 10  7  2  4  9  6  3  8  1
R>
R> MTCARS <- ore.push(mtcars)
R> by_cyl <- group_by(MTCARS, cyl)
R>
R> # Using ranking functions with an ore.frame
R> head(mutate(MTCARS, rank = row_number(hp)))
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb rank
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4   12
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4   13
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1    7
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1   14
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2   20
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1   10
R>
R> head(mutate(MTCARS, rank = min_rank(hp)))
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb rank
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4   12
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4   12
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1    7
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1   12
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2   20
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1   10
R>
R> head(mutate(MTCARS, rank = dense_rank(hp)))
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb rank
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4   11
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4   11
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1    6
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1   11
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2   15
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1    9
R>
R> # Using ranking functions with a grouped ore.frame
R> head(mutate(by_cyl, rank = row_number(hp)))
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb rank
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4    2
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4    3
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1    7
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1    4
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2    3
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1    1
R>
R> head(mutate(by_cyl, rank = min_rank(hp)))
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb rank
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4    2
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4    2
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1    7
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1    2
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2    3
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1    1
R>
R> head(mutate(by_cyl, rank = dense_rank(hp)))
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb rank
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4    2
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4    2
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1    6
```

```
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1    2
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1    1
```

3.4 Graph Analysis Using OAAgraph

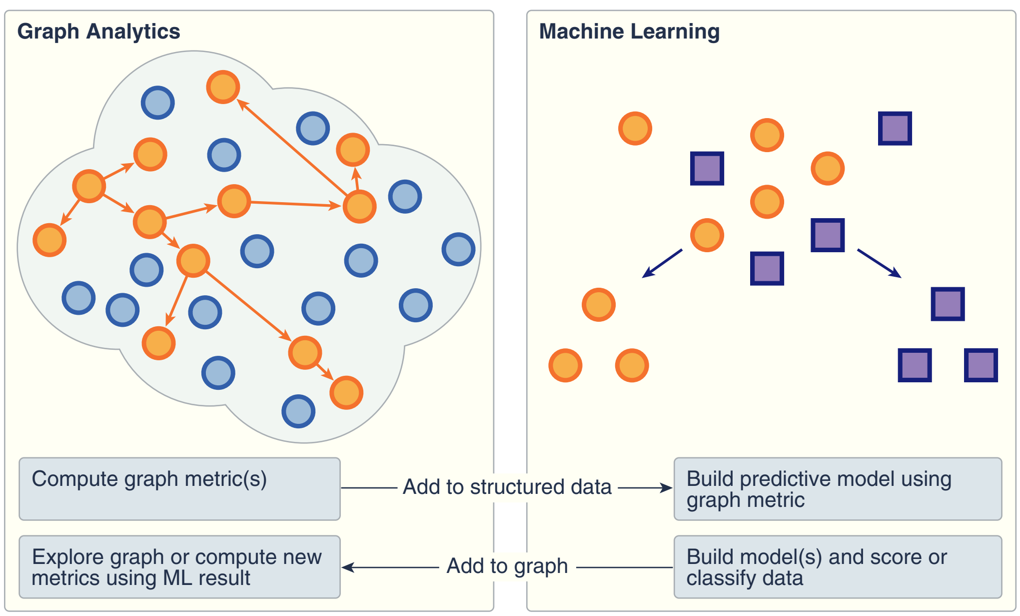
Beginning with Oracle Database 12.2 , the `OAAgraph` package provides an R interface to the Oracle Spatial and Graph Property Graph In-Memory Analyst (PGX) for use with Oracle R Enterprise and database tables.

About OAAgraph

PGX is an integrated set of Oracle Database functions, procedures, data types, and data models that support spatial and graph analytics. The `OAAgraph` package contains several graph algorithms, graph transformation operations, and graph querying capabilities.

With the `OAAgraph` functions in R, you can use the efficient PGX graph algorithms and representations to compute graph metrics and analysis in memory in the database. You can use the resulting `data.frame` objects to build models that include the graph metrics as predictors. You can use the models to score or classify data. You can then add the results to graph nodes where you can use the graph algorithms to further explore the graph or compute new metrics.

Figure 3-3 Graph Analytics and Machine Learning Interaction



Advantages of the `OAAgraph` package include:

- Access in R to the PGX in-memory graph analysis engine, which provides fast, parallel graph analysis
- Many graph algorithms
- Ability to query graphs and perform pattern matching
- Integration with Oracle Big Data Spatial and Graph and Oracle R Advanced Analytics for Hadoop

Graph analysis is a methodology in data analysis in which you represent your data as a graph. Data entities become nodes and relationships become edges. You can analyze fine-grained relationships through the graph and navigate multi-hop relationships quickly without needing to repeatedly compute joins.

Two major types of graph algorithms are:

- Computational graph analytics, which analyze an entire graph
- Graph pattern matching, which are queries that find sub-graphs that fit relationship patterns

OAAgraph Algorithms

The algorithms in the OAAgraph package are the following:

Ranking
Pagerank and variants

Vertex betweenness centrality, including approximations

Closeness centrality

Eigenvector centrality

Path Finding
Dijkstra and variants

Bellman Ford and variants

Hop distance and variants

Fattest path

Partitioning
Weakly and strongly connected components

Conductance and modularity

Community detection

Recommendation
Twitter's whom-to-follow

Matrix factorization

Other
Breadth first search with filter

Triangle counting

Degree distribution

K-core

Adamic Adar

See Also: For reference pages for the `OAAgraph` functions, see *Graph Analysis Function Reference* in the Oracle R Enterprise Documentation Media Library Release 1.5.1 .

Example 3-77 Using OAAgraph Functions

This example uses the graph capabilities of the OAAgraph package. The example does the following:

- Creates a graph from node and edge tables
- Creates a graph from a snaphot in-memory representation stored in the database
- Invokes the graph analytics algorithms `countTriangles`, `degree`, `pagerank`, and `adamicAdarCounting`
- Uses the `oaa.cursor` object
- Cleans up in-memory graphs and database objects

```
library(ORE)
library(OAAgraph)

#-- Replace the values in quotation marks with the values for your database
dbHost     <- "<DATABASE_HOST>"
dbUser     <- "<DATABASE_USERNAME>"
dbPassword <- "<DATABASE_PASSWORD>"
dbSid      <- "<DATABASE_SID>"
pgxBaseUrl <- "<PGX_BASE_URL>"

#-- Connect to the Oracle R Enterprise and PGX servers
ore.connect(host = dbHost, user = dbUser, password = dbPassword, sid = dbSid)
oaa.graphConnect(pgxBaseUrl = pgxBaseUrl, dbHost = dbHost,
                 dbSid = dbSid, dbUser = dbUser, dbPassword = dbPassword)

#-- Create the node table in Oracle Database

VID <- c(1, 2, 3, 4, 5)
NP1 <- c("node1", "node2", "node3", "node4", "node5")
NP2 <- c(111.11, 222.22, 333.33, 444.44, 555.55)
NP3 <- c(1, 2, 3, 4, 5)

nodes <- data.frame(VID, NP1, NP2, NP3)
ore.drop(table = "MY_NODES")
ore.create(nodes, table = "MY_NODES")

#-- Create the edge table in Oracle Database

EID <- c(1, 2, 3, 4, 5)
SVID <- c(1, 3, 3, 2, 4)
DVID <- c(2, 1, 4, 3, 2)
EP1 <- c("edge1", "edge2", "edge3", "edge4", "edge5")
EL <- c("label1", "label2", "label3", "label4", "label5")

edges <- data.frame(EID, SVID, DVID, EP1, EL)

ore.drop(table = "MY_EDGES")
ore.create(edges, table = "MY_EDGES")

#-- Verify that the tables exist as ore.frame objects

ore.ls()

#-- Create a graph in PGX from the node and edge tables in the database

graph <- oaa.graph(MY_EDGES, MY_NODES, "myPgxGraph")
names(graph, "nodes")
names(graph, "edges")

#-- See the result of the countTriangles function, which gives an
#-- overview of the number of connections between nodes in neighborhoods
```

```
countTriangles(graph, sortVerticesByDegree=FALSE)

#-- See the results from degree algorithm variants, note the graph nodes
#-- are augmented with new properties as indicated by the 'name' argument

degree(graph, name = "OutDegree")
degree(graph, name = "InDegree", variant = "in")
degree(graph, name = "InOutDegree", variant = "all")

#-- Create a cursor including the degree properties

cursor <- oaa.cursor(graph, c("OutDegree", "InOutDegree", "InDegree"), "nodes")
oaa.next(cursor, 5)

#-- Create a cursor over the degree properties using
#-- the PGX SQL-like query language PGQL

cursor <- oaa.cursor(graph,
                     query = "select n.OutDegree, n.InOutDegree, n.InDegree
                              where (n) order by n.OutDegree desc")
#-- View the first 5 entries from the cursor

oaa.next(cursor, 5)

#-- See results from the pagerank algorithm

pagerankCursor <- pagerank(graph, 0.085, 0.1, 100)
oaa.next(pagerankCursor, 5)

#-- Create a cursor over the pagerank property using PGQL

cursor <- oaa.cursor(graph,
                     query = "select n.pagerank where (n)
                              order by n.pagerank desc")

oaa.next(cursor, 5)

#-- You can create a cursor using the R interface as well

cursor <- oaa.cursor(graph, "pagerank", ordering = "desc")

oaa.next(cursor, 5)

#-- Compute the adamic adar index for edges

topEdges <- adamicAdarCounting(graph)
oaa.next(topEdges)

#-- List any graph snapshots available

oaa.graphSnapshotList()

#-- Export a binary snapshot of the whole graph into Oracle Database
#-- and view the listing again

oaa.graphSnapshotPersist(graph, nodeProperties = TRUE, edgeProperties = TRUE)
oaa.graphSnapshotList()

#-- Read the snapshot back into memory
```

```
graph2 <- oaa.graphSnapshot("myPgxGraph")

#-- Export the graph nodes and specific node properties from memory
#-- into a database table

oaa.create(graph2, nodeTableName = "RANKED_NODES", nodeProperties = TRUE)

#-- Export both nodes and edges as tables from memory into the database,
#-- but only export the pagerank node property

oaa.create(graph2, nodeTableName = "RANKED_GRAPH_N",
           nodeProperties = c("NP1", "pagerank"),
           edgeTableName = "RANKED_GRAPH_E")

#-- Export the graph edges and their properties from memory into a database table

oaa.create(graph2, edgeTableName = "RANKED_EDGES", edgeProperties = TRUE)

#-- Free the graphs at the PGX server

oaa.rm(graph)
oaa.rm(graph2)

#-- Clean up the tables created by this example

ore.drop("MY_NODES")
ore.drop("MY_EDGES")
ore.drop("RANKED_NODES")
ore.drop("RANKED_GRAPH_N")
ore.drop("RANKED_GRAPH_E")
ore.drop("RANKED_EDGES")

oaa.dropSnapshots("myPgxGraph")
```

Listing for This Example

```
R> library(ORE)
R> library(OAAgraph)
R>
R> #-- Replace the values in quotation marks with the values for your database
R> dbHost     <- "<DATABASE_HOST>"
R> dbUser     <- "<DATABASE_USERNAME>"
R> dbPassword <- "<DATABASE_PASSWORD>"
R> dbSid      <- "<DATABASE_SID>"
R> pgxBaseUrl <- "<PGX_BASE_URL>"
R>
R> #-- Connect to the Oracle R Enterprise and PGX servers
R> ore.connect(host = dbHost, user = dbUser, password = dbPassword, sid = dbSid)
R> oaa.graphConnect(pgxBaseUrl = pgxBaseUrl, dbHost = dbHost,
+                   dbSid = dbSid, dbUser = dbUser, dbPassword = dbPassword)
R>
R> #-- Create the node table in Oracle Database
R>
R> VID <- c(1, 2, 3, 4, 5)
R> NP1 <- c("node1", "node2", "node3", "node4", "node5")
R> NP2 <- c(111.11, 222.22, 333.33, 444.44, 555.55)
R> NP3 <- c(1, 2, 3, 4, 5)
R>
R> nodes <- data.frame(VID, NP1, NP2, NP3)
R> ore.drop(table = "MY_NODES")
R> ore.create(nodes, table = "MY_NODES")
```

```
R>
R> #-- Create the edge table in Oracle Database
R>
R> EID <- c(1, 2, 3, 4, 5)
R> SVID <- c(1, 3, 3, 2, 4)
R> DVID <- c(2, 1, 4, 3, 2)
R> EP1 <- c("edge1", "edge2", "edge3", "edge4", "edge5")
R> EL <- c("label1", "label2", "label3", "label4", "label5")
R>
R> edges <- data.frame(EID, SVID, DVID, EP1, EL)
R>
R> ore.drop(table = "MY_EDGES")
R> ore.create(edges, table = "MY_EDGES")
R>
R> #-- Verify that the tables exist as ore.frame objects
R>
R> ore.ls()
 [1] "ASSIGN_EDGES_SUBSET"   "ASSIGN_NODES_SUBSET"   "CALL_EDGES"
 [4] "DF_EDGES_140317215226" "DF_EDGES_150317002703" "DF_NODES_140317215226"
 [7] "DF_NODES_150317002703" "EDGES"                 "EDGES_KEY"
[10] "EDGES_T"               "MY_EDGES"              "MY_NODES"
[13] "MY_NODES1"             "N_H5855"               "NODES"
[16] "NODES2_T"              "NODES30174128"         "NODES30174506"
[19] "NODES30174740"         "NODES_KEY"             "NODES_T"
[22] "nyc20m"                "PERSON_NODES"          "PERSON_PAGERANK_NODES"
[25] "SCCE"                  "SCCN"                  "SUPERHERO_DATA"
[28] "SUPERHERO_DATA2"       "SUPERHERO_EDGES"       "SUPERHERO_IGNORE"
[31] "SUPERHERO_INFORMATION" "SUPERHERO_NODES"       "SUPERHERO_VALUES"
[34] "TABLE1"                "TABLE2"                "TEMP_EDGES"
[37] "TEMP_NODES"            "TMPN"

R>
R> #-- Create a graph in PGX from the node and edge tables in the database
R>
R> graph <- oaa.graph(MY_EDGES, MY_NODES, "myPgxGraph")
R> names(graph, "nodes")
[1] "NP1" "NP3" "NP2"
R> names(graph, "edges")
[1] "EP1"
R>
R> #-- See the result of the countTriangles function, which gives an
R> #-- overview of the number of connections between nodes in neighborhoods
R>
R> countTriangles(graph, sortVerticesByDegree=FALSE)
[1] 2
R>
R> #-- See the results from degree algorithm variants; note the graph nodes
R> #-- are augmented with new properties as indicated by the 'name' argument
R>
R> degree(graph, name = "OutDegree")
oaa.cursor over: ID, OutDegree
position: 0
size: 5
R> degree(graph, name = "InDegree", variant = "in")
oaa.cursor over: ID, InDegree
position: 0
size: 5

R> degree(graph, name = "InOutDegree", variant = "all")
oaa.cursor over: ID, InOutDegree
position: 0
```

```
size: 5
R>
R> #-- Create a cursor including the degree properties
R>
R> cursor <- oaa.cursor(graph, c("OutDegree", "InOutDegree", "InDegree"), "nodes")
R> oaa.next(cursor, 5)
  OutDegree InOutDegree InDegree
1         1           2        1
2         1           3        2
3         2           3        1
4         1           2        1
5         0           0        0
R>
R> #-- Create a cursor over the degree properties using
R> #-- the PGX SQL-like query language PGQL
R>
R> cursor <- oaa.cursor(graph,
+                        query = "select n.OutDegree, n.InOutDegree, n.InDegree
+                                 where (n) order by n.OutDegree desc")
R> #-- View the first 5 entries from the cursor
R>
R> oaa.next(cursor, 5)
  n.OutDegree n.InOutDegree n.InDegree
1           2             3          1
2           1             3          2
3           1             2          1
4           1             2          1
5           0             0          0
R>
R> #-- See the results from the pagerank algorithm
R>
R> pagerankCursor <- pagerank(graph, 0.085, 0.1, 100)
R> oaa.next(pagerankCursor, 5)
  pagerank
2     0.22
3     0.20
1     0.19
4     0.19
5     0.18
R>
R> #-- Create a cursor over the pagerank property using PGQL
R>
R> cursor <- oaa.cursor(graph,
+                         query = "select n.pagerank where (n)
+                                  order by n.pagerank descR> ")
R>
R> oaa.next(cursor, 5)
  n.pagerank
1       0.22
2       0.20
3       0.19
4       0.19
5       0.18
R>
R> #-- You can create a cursor using the R interface as well
R>
R> cursor <- oaa.cursor(graph, "pagerank", ordering = "desc")
R>
R> oaa.next(cursor, 5)
  pagerank
1     0.19
```

```
2     0.22
3     0.20
4     0.19
5     0.18

R>
R> #-- Compute the adamic adar index for edges
R>
R> topEdges <- adamicAdarCounting(graph)
R> oaa.next(topEdges)
  adamic_adar
0           0
1           0
2           0
3           0
4           0
R>
R> #-- List any graph snapshots available
R>
R> oaa.graphSnapshotList()
 [1] "ANONYMOUS_GRAPH_1"    "CONNECTIONS"          "EXAMPLE_GRAPH"
 [4] "GRAPH1"               "GRAPH_EXPORT_LABELED" "G_160317161147"
 [7] "G_160317201914"       "MYAWESOMEGRAPH"       "MYEXAMPLEGRAPH"
[10] "MY_GRAPH1"            "SAMPLE"               "SAMPLE_GRAPH"
[13] "SF"                   "SF_MUTATION"
R>
R> #-- Export a binary snapshot of the whole graph into Oracle Database
R> #-- and view the listing again
R>
R> oaa.graphSnapshotPersist(graph, nodeProperties = TRUE, edgeProperties = TRUE)
R> oaa.graphSnapshotList()
 [1] "ANONYMOUS_GRAPH_1"    "CONNECTIONS"          "EXAMPLE_GRAPH"
 [4] "GRAPH1"               "GRAPH_EXPORT_LABELED" "G_160317161147"
 [7] "G_160317201914"       "MYAWESOMEGRAPH"       "MYEXAMPLEGRAPH"
[10] "MYPGXGRAPH"           "MY_GRAPH1"            "SAMPLE"
[13] "SAMPLE_GRAPH"         "SF"                   "SF_MUTATION"
R>
R> #-- Read the snapshot back into memory
R>
R> graph2 <- oaa.graphSnapshot("myPgxGraph")
R>
R> #-- Export the graph nodes and specific node properties from memory
R> #-- into a database table
R>
R> oaa.create(graph2, nodeTableName = "RANKED_NODES", nodeProperties = TRUE)
R>
R> #-- Export both nodes and edges as tables from memory into the database,
R> #-- but only export the pagerank node property
R>
R> oaa.create(graph2, nodeTableName = "RANKED_GRAPH_N",
+            nodeProperties = c("NP1", "pagerank"),
+            edgeTableName = "RANKED_GRAPH_E")
R>
R> #-- Export the graph edges and their properties from memory into a database table
R>
R> oaa.create(graph2, edgeTableName = "RANKED_EDGES", edgeProperties = TRUE)
R>
R> #-- Free the graphs at the PGX server
R>
R> oaa.rm(graph)
R> oaa.rm(graph2)
```

```
R>
R> #-- Clean up the tables created by this example
R>
R> ore.drop("MY_NODES")
R> ore.drop("MY_EDGES")
R> ore.drop("RANKED_NODES")
R> ore.drop("RANKED_GRAPH_N")
R> ore.drop("RANKED_GRAPH_E")
R> ore.drop("RANKED_EDGES")
R>
R> oaa.dropSnapshots("myPgxGraph")
```

3.5 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.

For an example that uses the `kernlab` package, see Example 2-13 (page 2-11).

See Also:

- "Installing a Third-Party Package for Use in Embedded R Execution (page 6-5)"
- *R Administration and Installation Guide* at `http://cran.r-project.org/doc/manuals/r-release/R-admin.html`
- `http://www.r-bloggers.com/installing-r-packages/`

Example 3-78   Downloading, Installing, and Loading a Third-Party Package on the Client

This example demonstrates downloading, installing, and loading the CRAN package `kernlab`. The `kernlab` package contains kernel-based machine learning methods. The example invokes the `install.packages` function to download and install the package. It then invokes the `library` function to load the package.

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

Listing for Example 3-78 (page 3-66)

```
R> install.packages("kernlab")
trying URL 'http://cran.rstudio.com/bin/windows/contrib/3.0/kernlab_0.9-19.zip'
Content type 'application/zip' length 2029405 bytes (1.9 Mb)
opened URL
downloaded 1.9 Mb

package 'kernlab' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
    C:\Users\rquser\AppData\Local\Temp\RtmpSKVZql\downloaded_packages
R> library("kernlab")
```

Example 3-79   Using a kernlab Package Function

This example invokes the `demo` function to look for example programs in the `kernlab` package. Because the package does not have examples, this example then gets help for the `ksvm` function. The example invokes example code from the help.

```
demo(package = "kernlab")
help(package = "kernlab", ksvm)
data(spam)
index <- sample(1:dim(spam)[1])
spamtrain <- spam[index[1:floor(dim(spam)[1]/2)], ]
spamtest <- spam[index[((ceiling(dim(spam)[1]/2)) + 1):dim(spam)[1]], ]
filter <- ksvm(type~.,data=spamtrain,kernel="rbfdot",
+              kpar=list(sigma=0.05),C=5,cross=3)
filter
table(mailtype,spamtest[,58])
```

Listing for Example 3-79 (page 3-67)

```
> demo(package = "kernlab")
no demos found
> help(package = "kernlab", ksvm)    # Output not shown.
> data(spam)
> index <- sample(1:dim(spam)[1])
> spamtrain <- spam[index[1:floor(dim(spam)[1]/2)], ]
> spamtest <- spam[index[((ceiling(dim(spam)[1]/2)) + 1):dim(spam)[1]], ]
> filter <- ksvm(type~.,data=spamtrain,kernel="rbfdot",
+                kpar=list(sigma=0.05),C=5,cross=3)
> filter
Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 5

Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.05

Number of Support Vectors : 970

Objective Function Value : -1058.218
Training error : 0.018261
Cross validation error : 0.08696
> mailtype <- predict(filter,spamtest[,-58])
> table(mailtype,spamtest[,58])

mailtype  nonspam spam
  nonspam    1347  136
  spam         45  772
```

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:

4.1 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:

4.1.1 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.randomForest`	Creates a random forest classification model in parallel on data in an `ore.frame`.

Table 4-1 (Cont.) Functions in the OREmodels Package

Function	Description
`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 many different activation functions.
- Users can specify a neural network topology consisting of any number of hidden layers, including none.

4.1.2 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 Displaying Values from the longley Data Set

This example 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`.

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

Listing for Example 4-1 (page 4-2)

```
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
```

4.1.3 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 Using ore.lm

This example 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`.

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

Listing for Example 4-2 (page 4-3)

```
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 Using the ore.stepwise Function

This example 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.

```
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 (page 4-4)

```
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.
```

4.1.4 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 reweighted 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   Using the ore.glm Function

This example 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.

```
# 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 (page 4-5)

```
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
```

4.1.5 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`.

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

Example 4-5 Building a Neural Network Model

This example 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`.

```
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 (page 4-8)

```
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 Using ore.neural and Specifying Activations

This example 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.

```
IRIS <- ore.push(iris)
fit <- ore.neural(Petal.Length ~ Petal.Width + Sepal.Length,
                  data = IRIS,
                  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 (page 4-8)

```
R> IRIS <- ore.push(iris)
R> fit <- ore.neural(Petal.Length ~ Petal.Width + Sepal.Length,
+                    data = IRIS,
+                    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
```

4.1.6 Building a Random Forest Model

The `ore.randomForest` function provides an ensemble learning technique for classification of data in an `ore.frame` object.

Function `ore.randomForest` builds a random forest model by growing trees in parallel on the database server. It constructs many decision trees and outputs the class that is the mode of the classes of the individual trees. The function avoids overfitting, which is a common problem for decision trees.

The random forest algorithm, developed by Leo Breiman and Adele Cutler, combines the ideas of bagging and the random selection of variables, which results in a collection of decision trees with controlled variance. The random forest algorithm provides high accuracy, but performance and scalability can be issues for large data sets.

Function `ore.randomForest` executes in parallel for model building and scoring. Parallel execution can occur whether you are using the `randomForest` package in Oracle R Distribution (ORD) or the open source `randomForest` package 4.6-10. Using `ore.randomForest` and ORD can require less memory than using `ore.randomForest` with the open source alternative. If you use the open source `randomForest` package, Oracle R Enterprise issues a warning.

Function `ore.randomForest` uses the global option `ore.parallel` to determine the degree of parallelism to employ. The function returns an `ore.randomForest` object.

An invocation of the scoring method `predict` on an `ore.randomForest` object also runs in parallel on the database server. The `cache.model`argument specifies whether to cache the entire random forest model in memory during prediction. If sufficient memory is available, use the default `cache.model` value of `TRUE` for better performance.

The `grabTree` method returns an `ore.frame` object that contains information on the specified tree. Each row of the `ore.frame` represents one node of the tree.

Note:

Function `ore.randomForest` loads a copy of the training data for each embedded R session executing in parallel. For large datasets, this can exceed the amount of available memory. Oracle recommends that you adjust the number of parallel processes and the amount of available memory accordingly. The global option `ore.parallel` specifies the number of parallel processes. For information on controlling the amount of memory used by embedded R execution processes, see Controlling Memory Used by Embedded R in *Oracle R Enterprise Installation and Administration Guide*.

Example 4-7 Using ore.randomForest

```
# Using the iris dataset
IRIS <- ore.push(iris)
mod <- ore.randomForest(Species~., IRIS)
tree10 <- grabTree(mod, k = 10, labelVar = TRUE)
ans <- predict(mod, IRIS, type="all", supplemental.cols="Species")
table(ans$Species, ans$prediction)

# Using the infert dataset
INFERT <- ore.push(infert)
formula <- case ~ age + parity + education + spontaneous + induced

rfMod <- ore.randomForest(formula, INFERT, ntree=1000, nodesize = 2)
tree <- grabTree(rfMod, k = 500)

rfPred <- predict(rfMod, INFERT, supplemental.cols = "case")

confusion.matrix <- with(rfPred, table(case, prediction))

confusion.matrix
```

Listing for This Example

```
R> # Using the iris dataset
R> IRIS <- ore.push(iris)
R> mod <- ore.randomForest(Species~., IRIS)
R> tree10 <- grabTree(mod, k = 10, labelVar = TRUE)
R> ans <- predict(mod, IRIS, type="all", supplemental.cols="Species")
R> table(ans$Species, ans$prediction)

             setosa versicolor virginica
  setosa         50          0         0
  versicolor      0         50         0
  virginica       0          0        50

# Using the infert dataset
R> INFERT <- ore.push(infert)
R> formula <- case ~ age + parity + education + spontaneous + induced
R>
R> rfMod <- ore.randomForest(formula, INFERT, ntree=1000, nodesize = 2)
R> tree <- grabTree(rfMod, k = 500)
R>
R> rfPred <- predict(rfMod, INFERT, supplemental.cols = "case")
R>
R> confusion.matrix <- with(rfPred, table(case, prediction))

R> confusion.matrix
```

```
      prediction
case    0   1
   0 154  11
   1  27  56
```

4.2 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:

4.2.1 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:

4.2.1.1 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.

The following table 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.odmEM` (12.2 feature)	Expectation Maximization	Clustering
`ore.odmESA` (12.2 feature)	Explicit Semantic Analysis	Feature Extraction
`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.odmRAlg` (12.2 feature)	Extensible R Algorithm	Association Rules, Attribute Importance, Classification, Clustering, Feature Extraction, and Regression
`ore.odmSVD` (12.2 feature)	Singular Value Decomposition	Feature Extraction
`ore.odmSVM`	Support Vector Machines	Classification and Regression

4.2.1.2 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.

By default, 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. 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.

You can use the `MODEL_NAME` parameter in `odm.settings` to explicitly name an Oracle Data Mining object created in the database. The named Oracle Data Mining model object persists in the database just like those created using Oracle Data Miner or SQL.

Related Topics:

Saving and Managing R Objects in the Database (page 2-18)

4.2.1.3 Partitioning and Text Mining

Beginning with Oracle Database 12*c*, Release 2 (12.2), functions in the Oracle Data Mining package have an argument that specifies settings for an Oracle Data Mining model and some have an argument for setting text mining parameters.

With the `odm.setting` argument to an `OREdm` function, you can specify a list of Oracle Data Mining parameter settings. Each list element's name and value refer to the parameter setting name and value, respectively. The setting value must be numeric or string. Refer to *Oracle Data Mining User's Guide* for each algorithm's valid settings.

The `settings` function returns a `data.frame` that lists each Oracle Data Mining parameter setting name and value pair used to build the model.

Partitioned Oracle Data Mining Models

A partitioned model is an ensemble model that consists of multiple sub-models. To create a partitioned Oracle Data Mining model, use the `odm.setting` argument with `ODMS_PARTITION_COLUMNS` as the name and with the names of the columns by which to partition the input data as the value. The `OREdm` function returns a model with a sub-model for each partition. The partitions are based on the unique values found in the columns.

Partitioned models can automate scoring by allowing you to reference the top-level model only, which causes the proper sub-model to be chosen based on the values of the partitioned column or columns for each row of data to be scored.

The `partitions` function returns an `ore.frame` that lists each partition of the specified model object and the associated partition column values of the model.

Partition names are system-determined. The function returns `NULL` for a non-partitioned model.

Text Mining Attribute Settings

Some `OREdm` functions have a `ctx.settings` argument that specifies text mining attribute settings with which you can specify Oracle Text attribute-specific settings. With the `odm.settings` argument, you can specify the Oracle text policy, the minimal number of documents in which each token occurs, and the maximum number of distinct features for text mining. With the `ctx.settings` argument, you specify the columns that should be treated as text and the type of text transformation to apply.

The `ctx.settings` argument applies to the following functions:

- `ore.odmESA`, Explicit Semantic Analysis
- `ore.odmGLM`, Generalized Linear Models
- `ore.odmKMeans`, *k*-Means
- `ore.odmNMF`, Non-Negative Matrix Factorization
- `ore.odmSVD`, Singular Value Decomposition
- `ore.odmSVM`, Support Vector Machines

Note: To create an Oracle Text policy, the user must have the `CTXSYS.CTX_DDL` privilege.

See Also: Creating a Model that Includes Text Mining in *Oracle Data Mining User's Guide* for valid text attribute values.

Example 4-8 Example of Text Mining with ore.odmKMeans

This example uses the `odm.settings` and `ctx.settings` arguments. The figure following the example shows the output of the `histogram(km.mod1)` 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 <- ore.push (data.frame(x))
km.mod1 <- NULL
km.mod1 <- ore.odmKMeans(~., X, num.centers = 2)
km.mod1
summary(km.mod1)
rules(km.mod1)
clusterhists(km.mod1)
histogram(km.mod1)

km.res1 <- predict(km.mod1,X,type="class",supplemental.cols=c("x","y"))
head(km.res1,3)
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))
```

```
head(predict(km.mod1,X,type=c("class","raw"),supplemental.cols=c("x","y")),3)
head(predict(km.mod1,X,type="raw",supplemental.cols=c("x","y")),3)

# Text mining with ore.odmKMeans
title <- c('Aids in Africa: Planning for a long war',
             'Mars rover maneuvers for rim shot',
             'Mars express confirms presence of water at Mars south pole',
             'NASA announces major Mars rover finding',
             'Drug access, Asia threat in focus at AIDS summit',
             'NASA Mars Odyssey THEMIS image: typical crater',
             'Road blocks for Aids')
response <- c('Aids', 'Mars', 'Mars', 'Mars', 'Aids', 'Mars', 'Aids')

# TEXT contents in character column
KM_TEXT <- ore.push(data.frame(CUST_ID = seq(length(title)),
                                     RESPONSE = response, TITLE = title))

# Create text policy (CTXSYS.CTX_DDL privilege is required)
ore.exec("Begin ctx_ddl.create_policy('ESA_TXTPOL'); End;")

# Specify POLICY_NAME, MIN_DOCUMENTS, MAX_FEATURES and
# text column attribute specification
km.mod <- ore.odmKMeans( ~ TITLE, data = KM_TEXT, num.centers = 2L,
                odm.settings = list(ODMS_TEXT_POLICY_NAME = "ESA_TXTPOL",
                                       ODMS_TEXT_MIN_DOCUMENTS = 1,
                                       ODMS_TEXT_MAX_FEATURES = 3,
                                    kmns_distance = "dbms_data_mining.kmns_cosine",
                                    kmns_details = "kmns_details_all"),
                ctx.settings = list(TITLE = "TEXT(TOKEN_TYPE:STEM)"))
summary(km.mod)
settings(km.mod)
print(predict(km.mod, KM_TEXT, supplemental.cols = "RESPONSE"), digits = 3L)

ore.exec("Begin ctx_ddl.drop_policy('ESA_TXTPOL'); End;")
```

Listing for This Example

```
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>
R> X <- ore.push (data.frame(x))
R> km.mod1 <- NULL
R> km.mod1 <- ore.odmKMeans(~., X, num.centers = 2)
R> km.mod1

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

Settings:
                                               value
clus.num.clusters                                  2
block.growth                                       2
conv.tolerance                                  0.01
details                                  details.all
distance                                   euclidean
iterations                                         3
min.pct.attr.support                             0.1
num.bins                                          10
random.seed                                        0
split.criterion                             variance
```

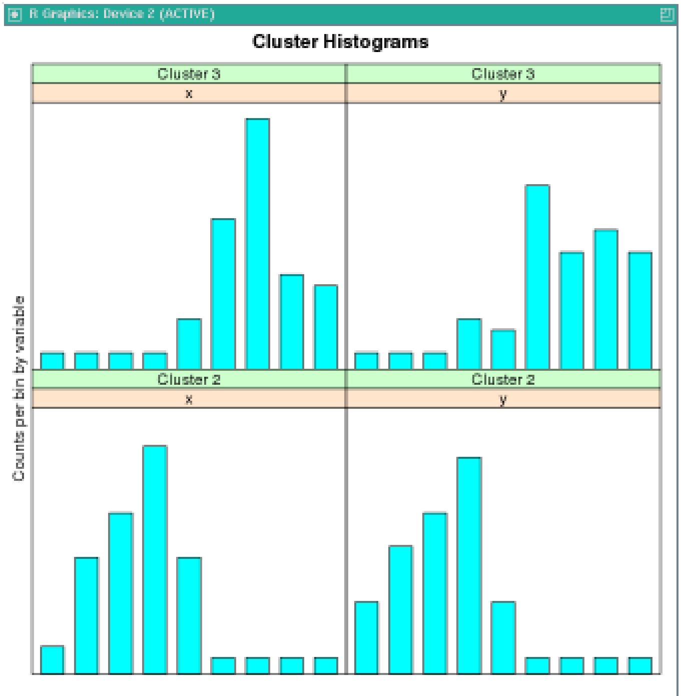
```
odms.missing.value.treatment odms.missing.value.auto
odms.sampling                 odms.sampling.disable
prep.auto                                        ON

R> summary(km.mod1)

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

Settings:
                                              value
clus.num.clusters                                 2
block.growth                                      2
conv.tolerance                                 0.01
details                                 details.all
distance                                  euclidean
iterations                                        3
min.pct.attr.support                            0.1
num.bins                                         10
random.seed                                       0
split.criterion                            variance
odms.missing.value.treatment odms.missing.value.auto
odms.sampling                 odms.sampling.disable
prep.auto                                        ON

Centers:
            x          y
2 -0.07638266 0.04449368
3  0.98493306 1.00864399

R> rules(km.mod1)
   cluster.id rhs.support rhs.conf lhr.support lhs.conf lhs.var lhs.var.support
lhs.var.conf   predicate
1           1         100      1.0          92     0.86       x              86
0.2222222 x <= 1.2209
2           1         100      1.0          92     0.86       x              86
0.2222222 x >= -.6188
3           1         100      1.0          86     0.86       y              86
0.4444444 y <= 1.1653
4           1         100      1.0          86     0.86       y              86
0.4444444  y > -.3053
5           2          50      0.5          48     0.96       x              48
0.0870793  x <= .4324
6           2          50      0.5          48     0.96       x              48
0.0870793 x >= -.6188
7           2          50      0.5          48     0.96       y              48
0.0893300  y <= .5771
8           2          50      0.5          48     0.96       y              48
0.0893300  y > -.5995
9           3          50      0.5          49     0.98       x              49
0.0852841 x <= 1.7465
10          3          50      0.5          49     0.98       x              49
0.0852841   x > .4324
11          3          50      0.5          50     0.98       y              49
0.0838225 y <= 1.7536
12          3          50      0.5          50     0.98       y              49
0.0838225   y > .2829

R> clusterhists(km.mod1)
   cluster.id variable bin.id lower.bound upper.bound               label count
1           1        x      1 -0.61884662 -0.35602715 -.6188466:-.3560272     6
```

```
2         1       x      2 -0.35602715 -0.09320769 -.3560272:-.0932077    17
3         1       x      3 -0.09320769  0.16961178  -.0932077:.1696118    15
4         1       x      4  0.16961178  0.43243125   .1696118:.4324312    11
5         1       x      5  0.43243125  0.69525071   .4324312:.6952507     8
6         1       x      6  0.69525071  0.95807018   .6952507:.9580702    17
7         1       x      7  0.95807018  1.22088965  .9580702:1.2208896    18
8         1       x      8  1.22088965  1.48370911 1.2208896:1.4837091     4
9         1       x      9  1.48370911  1.74652858 1.4837091:1.7465286     4
10        1       y      1 -0.89359597 -0.59946141  -.893596:-.5994614     2
11        1       y      2 -0.59946141 -0.30532685 -.5994614:-.3053269     4
12        1       y      3 -0.30532685 -0.01119230 -.3053269:-.0111923    11
13        1       y      4 -0.01119230  0.28294226  -.0111923:.2829423    24
14        1       y      5  0.28294226  0.57707682   .2829423:.5770768    13
15        1       y      6  0.57707682  0.87121138   .5770768:.8712114    12
16        1       y      7  0.87121138  1.16534593  .8712114:1.1653459    26
17        1       y      8  1.16534593  1.45948049 1.1653459:1.4594805     5
18        1       y      9  1.45948049  1.75361505  1.4594805:1.753615     3
19        2       x      1 -0.61884662 -0.35602715 -.6188466:-.3560272     6
20        2       x      2 -0.35602715 -0.09320769 -.3560272:-.0932077    17
21        2       x      3 -0.09320769  0.16961178  -.0932077:.1696118    15
22        2       x      4  0.16961178  0.43243125   .1696118:.4324312    10
23        2       x      5  0.43243125  0.69525071   .4324312:.6952507     2
24        2       x      6  0.69525071  0.95807018   .6952507:.9580702     0
25        2       x      7  0.95807018  1.22088965  .9580702:1.2208896     0
26        2       x      8  1.22088965  1.48370911 1.2208896:1.4837091     0
27        2       x      9  1.48370911  1.74652858 1.4837091:1.7465286     0
28        2       y      1 -0.89359597 -0.59946141  -.893596:-.5994614     2
29        2       y      2 -0.59946141 -0.30532685 -.5994614:-.3053269     4
30        2       y      3 -0.30532685 -0.01119230 -.3053269:-.0111923    11
31        2       y      4 -0.01119230  0.28294226  -.0111923:.2829423    24
32        2       y      5  0.28294226  0.57707682   .2829423:.5770768     9
33        2       y      6  0.57707682  0.87121138   .5770768:.8712114     0
34        2       y      7  0.87121138  1.16534593  .8712114:1.1653459     0
35        2       y      8  1.16534593  1.45948049 1.1653459:1.4594805     0
36        2       y      9  1.45948049  1.75361505  1.4594805:1.753615     0
37        3       x      1 -0.61884662 -0.35602715 -.6188466:-.3560272     0
38        3       x      2 -0.35602715 -0.09320769 -.3560272:-.0932077     0
39        3       x      3 -0.09320769  0.16961178  -.0932077:.1696118     0
40        3       x      4  0.16961178  0.43243125   .1696118:.4324312     1
41        3       x      5  0.43243125  0.69525071   .4324312:.6952507     6
42        3       x      6  0.69525071  0.95807018   .6952507:.9580702    17
43        3       x      7  0.95807018  1.22088965  .9580702:1.2208896    18
44        3       x      8  1.22088965  1.48370911 1.2208896:1.4837091     4
45        3       x      9  1.48370911  1.74652858 1.4837091:1.7465286     4
46        3       y      1 -0.89359597 -0.59946141  -.893596:-.5994614     0
47        3       y      2 -0.59946141 -0.30532685 -.5994614:-.3053269     0
48        3       y      3 -0.30532685 -0.01119230 -.3053269:-.0111923     0
49        3       y      4 -0.01119230  0.28294226  -.0111923:.2829423     0
50        3       y      5  0.28294226  0.57707682   .2829423:.5770768     4
51        3       y      6  0.57707682  0.87121138   .5770768:.8712114    12
52        3       y      7  0.87121138  1.16534593  .8712114:1.1653459    26
53        3       y      8  1.16534593  1.45948049 1.1653459:1.4594805     5
54        3       y      9  1.45948049  1.75361505  1.4594805:1.753615     3
R> histogram(km.mod1)
R>
R> km.res1 <- predict(km.mod1, X, type="class", supplemental.cols = c("x","y"))
R> head(km.res1, 3)
            x           y CLUSTER_ID
1 -0.43646407  0.26201831          2
2 -0.02797831  0.07319952          2
3  0.11998373 -0.08638716          2
```

```
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>
R>  head(predict(km.mod1, X))
        '2'          '3' CLUSTER_ID
1 0.9992236 0.0007763706          2
2 0.9971310 0.0028690375          2
3 0.9974216 0.0025783939          2
4 0.9997335 0.0002665114          2
5 0.9917773 0.0082226599          2
6 0.9771667 0.0228333398          2
R> head(predict(km.mod1,X,type=c("class","raw"),supplemental.cols=c("x","y")),3)
        '2'          '3'           x           y CLUSTER_ID
1 0.9992236 0.0007763706 -0.43646407  0.26201831          2
2 0.9971310 0.0028690375 -0.02797831  0.07319952          2
3 0.9974216 0.0025783939  0.11998373 -0.08638716          2
R> head(predict(km.mod1,X,type="raw",supplemental.cols=c("x","y")),3)
            x           y       '2'          '3'
1 -0.43646407  0.26201831 0.9992236 0.0007763706
2 -0.02797831  0.07319952 0.9971310 0.0028690375
3  0.11998373 -0.08638716 0.9974216 0.0025783939R>
R>
R> # Text mining with ore.odmKMeans
R> title <- c('Aids in Africa: Planning for a long war',
+             'Mars rover maneuvers for rim shot',
+             'Mars express confirms presence of water at Mars south pole',
+             'NASA announces major Mars rover finding',
+             'Drug access, Asia threat in focus at AIDS summit',
+             'NASA Mars Odyssey THEMIS image: typical crater',
+             'Road blocks for Aids')
R>  response <- c('Aids', 'Mars', 'Mars', 'Mars', 'Aids', 'Mars', 'Aids')
R>
R> KM_TEXT <- ore.push(data.frame(CUST_ID = seq(length(title)),
+                                RESPONSE = response, TITLE = title))
R>
R>  # Create text policy (CTXSYS.CTX_DDL privilege is required)
R> ore.exec("Begin ctx_ddl.create_policy('ESA_TXTPOL'); End;")
R>
R> # specify POLICY_NAME, MIN_DOCUMENTS, MAX_FEATURES and
R> # Text column attribute specification
R> km.mod <- ore.odmKMeans( ~ TITLE, data = KM_TEXT, num.centers = 2L,
+    odm.settings = list(ODMS_TEXT_POLICY_NAME = "ESA_TXTPOL",
+                        ODMS_TEXT_MIN_DOCUMENTS = 1,
+                        ODMS_TEXT_MAX_FEATURES = 3,
+                        kmns_distance = "dbms_data_mining.kmns_cosine",
+                        kmns_details = "kmns_details_all"),
+    ctx.settings = list(TITLE="TEXT(TOKEN_TYPE:STEM)"))
R> summary(km.mod)

Call:
ore.odmKMeans(formula = ~TITLE, data = KM_TEXT, num.centers = 2L,
    odm.settings = list(ODMS_TEXT_POLICY_NAME = "ESA_TXTPOL",
        ODMS_TEXT_MIN_DOCUMENTS = 1, ODMS_TEXT_MAX_FEATURES = 3,
        kmns_distance = "dbms_data_mining.kmns_cosine",
        kmns_details = "kmns_details_all"),
    ctx.settings = list(TITLE = "TEXT(TOKEN_TYPE:STEM)"))

Settings:
```

```
                                              value
clus.num.clusters                                 2
block.growth                                      2
conv.tolerance                                 0.01
details                                 details.all
distance                                     cosine
iterations                                        3
min.pct.attr.support                            0.1
num.bins                                         10
random.seed                                       0
split.criterion                            variance
odms.missing.value.treatment odms.missing.value.auto
odms.sampling                 odms.sampling.disable
odms.text.max.features                            3
odms.text.min.documents                           1
odms.text.policy.name                    ESA_TXTPOL
prep.auto                                        ON

Centers:
  TITLE.MARS TITLE.NASA TITLE.ROVER TITLE.AIDS
2  0.5292307  0.7936566   0.7936566         NA
3         NA         NA          NA          1
R> settings(km.mod)
                   SETTING_NAME           SETTING_VALUE SETTING_TYPE
1                     ALGO_NAME             ALGO_KMEANS        INPUT
2             CLUS_NUM_CLUSTERS                       2        INPUT
3             KMNS_BLOCK_GROWTH                       2        INPUT
4           KMNS_CONV_TOLERANCE                    0.01        INPUT
5                  KMNS_DETAILS       KMNS_DETAILS_ALL        INPUT
6                 KMNS_DISTANCE             KMNS_COSINE        INPUT
7               KMNS_ITERATIONS                       3        INPUT
8      KMNS_MIN_PCT_ATTR_SUPPORT                    0.1        INPUT
9                 KMNS_NUM_BINS                      10        INPUT
10             KMNS_RANDOM_SEED                       0      DEFAULT
11         KMNS_SPLIT_CRITERION           KMNS_VARIANCE        INPUT
12 ODMS_MISSING_VALUE_TREATMENT ODMS_MISSING_VALUE_AUTO      DEFAULT
13                ODMS_SAMPLING   ODMS_SAMPLING_DISABLE      DEFAULT
14       ODMS_TEXT_MAX_FEATURES                       3        INPUT
15      ODMS_TEXT_MIN_DOCUMENTS                       1        INPUT
16        ODMS_TEXT_POLICY_NAME              ESA_TXTPOL        INPUT
17                    PREP_AUTO                      ON        INPUT
R> print(predict(km.mod, KM_TEXT, supplemental.cols = "RESPONSE"), digits = 3L)
     '2'    '3' RESPONSE CLUSTER_ID
1 0.0213 0.9787     Aids          3
2 0.9463 0.0537     Mars          2
3 0.9325 0.0675     Mars          2
4 0.9691 0.0309     Mars          2
5 0.0213 0.9787     Aids          3
6 0.9463 0.0537     Mars          2
7 0.0213 0.9787     Aids          3
R>
R> ore.exec("Begin ctx_ddl.drop_policy('ESA_TXTPOL'); End;")
```

Figure 4-1 Cluster Histogram for km.mod1



4.2.2 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 4-9 Using the ore.odmAssocRules Function

This example 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 4-2 (page 4-23).

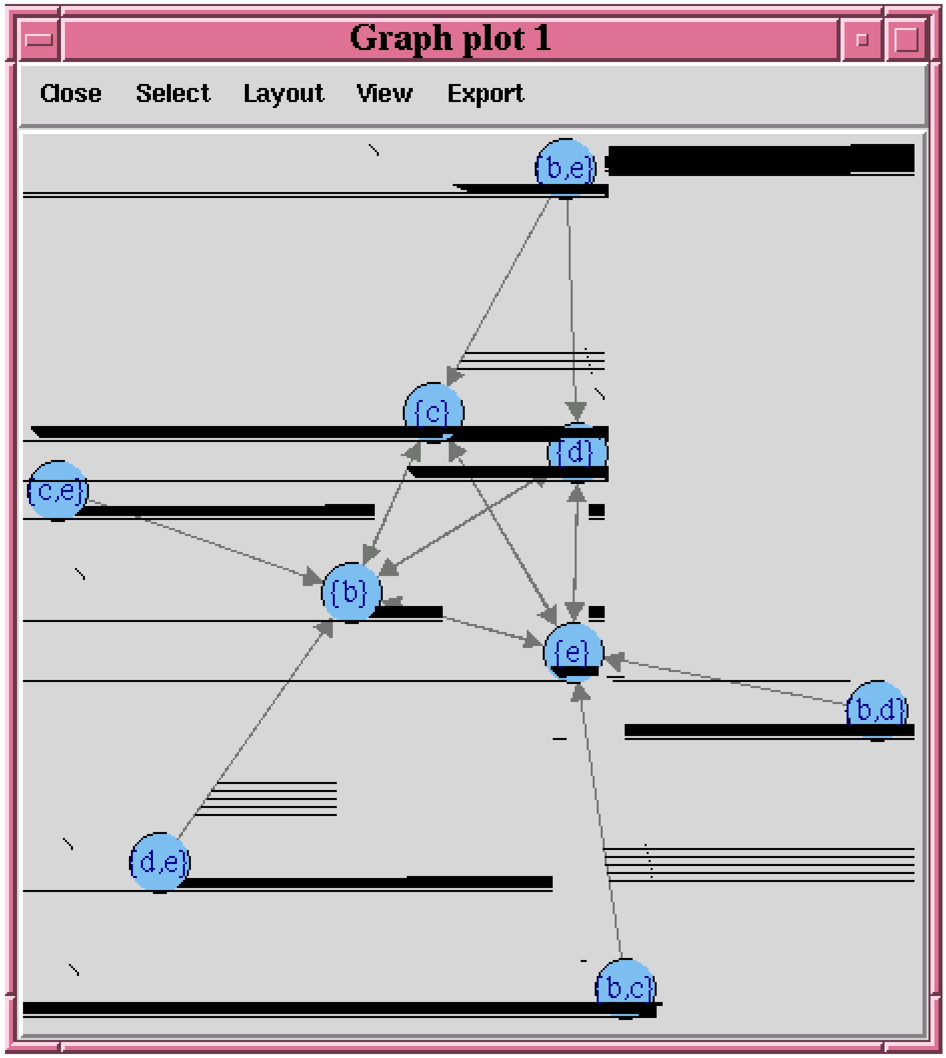
```
# Load the arules and arulesViz packages.
library(arules)
library(arulesViz)
# Create some transactional data.
id <- c(1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3)
item <- c("b", "d", "e", "a", "b", "c", "e", "b", "c", "d", "e")
# Push the data to the database as an ore.frame object.
transdata_of <- ore.push(data.frame(ID = 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 4-9 (page 4-21)

```
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 4-2 A Visual Demonstration of the Association Rules



4.2.3 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 4-10 Using the ore.odmAI Function

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

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

Listing for Example 4-10 (page 4-24)

```
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
```

4.2.4 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 4-11 Using the ore.odmDT Function

This example creates an input `ore.frame`, builds a model, makes predictions, and generates 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)
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 4-11 (page 4-24)

```
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
```

4.2.5 Building an Expectation Maximization Model

Beginning in Oracle Database 12*c*, Release 2 (12.2), the `ore.odmEM` function creates a model that uses the Oracle Data Mining Expectation Maximization algorithm.

Expectation Maximization (EM) is a density estimation algorithm that performs probabilistic clustering. In density estimation, the goal is to construct a density function that captures how a given population is distributed. The density estimate is based on observed data that represents a sample of the population.

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

Example 4-12 Using the ore.odmEM Function

```
## Synthetic 2-dimensional data set
set.seed(7654)

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 <- ore.push (data.frame(ID=1:100,x))
rownames(X) <- X$ID

em.mod <- NULL
em.mod <- ore.odmEM(~., X, num.centers = 2L)

summary(em.mod)
rules(em.mod)
clusterhists(em.mod)
histogram(em.mod)

em.res <- predict(em.mod, X, type="class", supplemental.cols=c("x", "y"))
head(em.res)
em.res.local <- ore.pull(em.res)
plot(data.frame(x=em.res.local$x, y=em.res.local$y), col=em.res.local$CLUSTER_ID)
points(em.mod$centers2, col = rownames(em.mod$centers2), pch=8, cex=2)

head(predict(em.mod,X))
head(predict(em.mod,X,type=c("class","raw")))
head(predict(em.mod,X,type=c("class","raw"),supplemental.cols=c("x","y")))
head(predict(em.mod,X,type="raw",supplemental.cols=c("x","y")))
```

Listing for This Example

```
R> ## Synthetic 2-dimensional data set
R>
R> set.seed(7654)
R>
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>
R> X <- ore.push (data.frame(ID=1:100,x))
R> rownames(X) <- X$ID
R>
R> em.mod <- NULL
R> em.mod <- ore.odmEM(~., X, num.centers = 2L)
R>
R> summary(em.mod)

Call:
ore.odmEM(formula = ~., data = X, num.centers = 2L)

Settings:
                                              value
clus.num.clusters                                 2
```

```
cluster.components              cluster.comp.enable
cluster.statistics                clus.stats.enable
cluster.thresh                                    2
linkage.function                     linkage.single
loglike.improvement                            .001
max.num.attr.2d                                  50
min.pct.attr.support                             .1
model.search                    model.search.disable
num.components                                   20
num.distribution                   num.distr.system
num.equiwidth.bins                               11
num.iterations                                  100
num.projections                                  50
random.seed                                       0
remove.components                remove.comps.enable
odms.missing.value.treatment odms.missing.value.auto
odms.sampling                  odms.sampling.disable
prep.auto                                        ON

Centers:
  MEAN.ID MEAN.x MEAN.y
2    25.5   4.03   3.96
3    75.5   1.93   1.99

R> rules(em.mod)
   cluster.id rhs.support rhs.conf lhr.support lhs.conf lhs.var lhs.var.support
lhs.var.conf   predicate
1          1         100      1.0         100     1.00      ID
100       0.0000   ID <= 100
2          1         100      1.0         100     1.00      ID
100       0.0000     ID >= 1
3          1         100      1.0         100     1.00       x
100       0.2500 x <= 4.6298
4          1         100      1.0         100     1.00       x
100       0.2500 x >= 1.3987
5          1         100      1.0         100     1.00       y
100       0.3000 y <= 4.5846
6          1         100      1.0         100     1.00       y
100       0.3000 y >= 1.3546
7          2          50      0.5          50     1.00      ID
50       0.0937  ID <= 50.5
8          2          50      0.5          50     1.00      ID
50       0.0937     ID >= 1
9          2          50      0.5          50     1.00       x
50       0.0937 x <= 4.6298
10         2          50      0.5          50     1.00       x
50       0.0937  x > 3.3374
11         2          50      0.5          50     1.00       y
50       0.0937 y <= 4.5846
12         2          50      0.5          50     1.00       y
50       0.0937  y > 2.9696
13         3          50      0.5          50     0.98      ID
49       0.0937   ID <= 100
14         3          50      0.5          50     0.98      ID
49       0.0937   ID > 50.5
15         3          50      0.5          49     0.98       x
49       0.0937  x <= 2.368
16         3          50      0.5          49     0.98       x
49       0.0937 x >= 1.3987
17         3          50      0.5          49     0.98       y
49       0.0937 y <= 2.6466
```

```
18          3          50      0.5          49     0.98       y
49      0.0937 y >= 1.3546
R> clusterhists(em.mod)
   cluster.id variable bin.id lower.bound upper.bound       label count
1           1       ID      1        1.00       10.90      1:10.9    10
2           1       ID      2       10.90       20.80   10.9:20.8    10
3           1       ID      3       20.80       30.70   20.8:30.7    10
4           1       ID      4       30.70       40.60   30.7:40.6    10
5           1       ID      5       40.60       50.50   40.6:50.5    10
6           1       ID      6       50.50       60.40   50.5:60.4    10
7           1       ID      7       60.40       70.30   60.4:70.3    10
8           1       ID      8       70.30       80.20   70.3:80.2    10
9           1       ID      9       80.20       90.10   80.2:90.1    10
10          1       ID     10       90.10      100.00    90.1:100    10
11          1       ID     11          NA          NA           :     0
12          1        x      1        1.40        1.72 1.399:1.722    11
13          1        x      2        1.72        2.04 1.722:2.045    22
14          1        x      3        2.04        2.37 2.045:2.368    16
15          1        x      4        2.37        2.69 2.368:2.691     1
16          1        x      5        2.69        3.01 2.691:3.014     0
17          1        x      6        3.01        3.34 3.014:3.337     0
18          1        x      7        3.34        3.66  3.337:3.66     4
19          1        x      8        3.66        3.98  3.66:3.984    18
20          1        x      9        3.98        4.31 3.984:4.307    22
21          1        x     10        4.31        4.63  4.307:4.63     6
22          1        x     11          NA          NA           :     0
23          1        y      1        1.35        1.68 1.355:1.678     7
24          1        y      2        1.68        2.00 1.678:2.001    18
25          1        y      3        2.00        2.32 2.001:2.324    18
26          1        y      4        2.32        2.65 2.324:2.647     6
27          1        y      5        2.65        2.97  2.647:2.97     1
28          1        y      6        2.97        3.29  2.97:3.293     4
29          1        y      7        3.29        3.62 3.293:3.616     3
30          1        y      8        3.62        3.94 3.616:3.939    16
31          1        y      9        3.94        4.26 3.939:4.262    16
32          1        y     10        4.26        4.58 4.262:4.585    11
33          1        y     11          NA          NA           :     0
34          2       ID      1        1.00       10.90      1:10.9    10
35          2       ID      2       10.90       20.80   10.9:20.8    10
36          2       ID      3       20.80       30.70   20.8:30.7    10
37          2       ID      4       30.70       40.60   30.7:40.6    10
38          2       ID      5       40.60       50.50   40.6:50.5    10
39          2       ID      6       50.50       60.40   50.5:60.4     0
40          2       ID      7       60.40       70.30   60.4:70.3     0
41          2       ID      8       70.30       80.20   70.3:80.2     0
42          2       ID      9       80.20       90.10   80.2:90.1     0
43          2       ID     10       90.10      100.00    90.1:100     0
44          2       ID     11          NA          NA           :     0
45          2        x      1        1.40        1.72 1.399:1.722     0
46          2        x      2        1.72        2.04 1.722:2.045     0
47          2        x      3        2.04        2.37 2.045:2.368     0
48          2        x      4        2.37        2.69 2.368:2.691     0
49          2        x      5        2.69        3.01 2.691:3.014     0
50          2        x      6        3.01        3.34 3.014:3.337     0
51          2        x      7        3.34        3.66  3.337:3.66     4
52          2        x      8        3.66        3.98  3.66:3.984    18
53          2        x      9        3.98        4.31 3.984:4.307    22
54          2        x     10        4.31        4.63  4.307:4.63     6
55          2        x     11          NA          NA           :     0
56          2        y      1        1.35        1.68 1.355:1.678     0
57          2        y      2        1.68        2.00 1.678:2.001     0
```

```
58          2        y      3        2.00        2.32 2.001:2.324     0
59          2        y      4        2.32        2.65 2.324:2.647     0
60          2        y      5        2.65        2.97  2.647:2.97     0
61          2        y      6        2.97        3.29  2.97:3.293     4
62          2        y      7        3.29        3.62 3.293:3.616     3
63          2        y      8        3.62        3.94 3.616:3.939    16
64          2        y      9        3.94        4.26 3.939:4.262    16
65          2        y     10        4.26        4.58 4.262:4.585    11
66          2        y     11          NA          NA           :     0
67          3       ID      1        1.00       10.90      1:10.9     0
68          3       ID      2       10.90       20.80   10.9:20.8     0
69          3       ID      3       20.80       30.70   20.8:30.7     0
70          3       ID      4       30.70       40.60   30.7:40.6     0
71          3       ID      5       40.60       50.50   40.6:50.5     0
72          3       ID      6       50.50       60.40   50.5:60.4    10
73          3       ID      7       60.40       70.30   60.4:70.3    10
74          3       ID      8       70.30       80.20   70.3:80.2    10
75          3       ID      9       80.20       90.10   80.2:90.1    10
76          3       ID     10       90.10      100.00    90.1:100    10
77          3       ID     11          NA          NA           :     0
78          3        x      1        1.40        1.72 1.399:1.722    11
79          3        x      2        1.72        2.04 1.722:2.045    22
80          3        x      3        2.04        2.37 2.045:2.368    16
81          3        x      4        2.37        2.69 2.368:2.691     1
82          3        x      5        2.69        3.01 2.691:3.014     0
83          3        x      6        3.01        3.34 3.014:3.337     0
84          3        x      7        3.34        3.66  3.337:3.66     0
85          3        x      8        3.66        3.98  3.66:3.984     0
86          3        x      9        3.98        4.31 3.984:4.307     0
87          3        x     10        4.31        4.63  4.307:4.63     0
88          3        x     11          NA          NA           :     0
89          3        y      1        1.35        1.68 1.355:1.678     7
90          3        y      2        1.68        2.00 1.678:2.001    18
91          3        y      3        2.00        2.32 2.001:2.324    18
92          3        y      4        2.32        2.65 2.324:2.647     6
93          3        y      5        2.65        2.97  2.647:2.97     1
94          3        y      6        2.97        3.29  2.97:3.293     0
95          3        y      7        3.29        3.62 3.293:3.616     0
96          3        y      8        3.62        3.94 3.616:3.939     0
97          3        y      9        3.94        4.26 3.939:4.262     0
98          3        y     10        4.26        4.58 4.262:4.585     0
99          3        y     11          NA          NA           :     0
R> histogram(em.mod)
R>
R> em.res <- predict(em.mod, X, type="class", supplemental.cols=c("x", "y"))
R> head(em.res)
     x    y CLUSTER_ID
1 4.15 3.63          2
2 3.88 4.13          2
3 3.72 4.10          2
4 3.78 4.14          2
5 4.22 4.35          2
6 4.07 3.62          2
R> em.res.local <- ore.pull(em.res)
R> plot(data.frame(x=em.res.local$x, y=em.res.local$y), col=em.res.local$CLUSTER_ID)
R> points(em.mod$centers2, col = rownames(em.mod$centers2), pch=8, cex=2)
R>
R> head(predict(em.mod,X))
  '2'      '3' CLUSTER_ID
1   1 1.14e-54          2
2   1 1.63e-55          2
```

```
3   1 1.10e-51         2
4   1 1.53e-52         2
5   1 9.02e-62         2
6   1 3.20e-49         2
R> head(predict(em.mod,X,type=c("class","raw")))
  '2'      '3' CLUSTER_ID
1   1 1.14e-54         2
2   1 1.63e-55         2
3   1 1.10e-51         2
4   1 1.53e-52         2
5   1 9.02e-62         2
6   1 3.20e-49         2
R> head(predict(em.mod,X,type=c("class","raw"),supplemental.cols=c("x","y")))
  '2'      '3'    x    y CLUSTER_ID
1   1 1.14e-54 4.15 3.63          2
2   1 1.63e-55 3.88 4.13          2
3   1 1.10e-51 3.72 4.10          2
4   1 1.53e-52 3.78 4.14          2
5   1 9.02e-62 4.22 4.35          2
6   1 3.20e-49 4.07 3.62          2
R> head(predict(em.mod,X,type="raw",supplemental.cols=c("x","y")))
     x    y '2'      '3'
1 4.15 3.63   1 1.14e-54
2 3.88 4.13   1 1.63e-55
3 3.72 4.10   1 1.10e-51
4 3.78 4.14   1 1.53e-52
5 4.22 4.35   1 9.02e-62
6 4.07 3.62   1 3.20e-49
```

4.2.6 Building an Explicit Semantic Analysis Model

Beginning in Oracle Database 12*c*, Release 2 (12.2), the `ore.odmESA` function creates a model that uses the Oracle Data Mining Explicit Semantic Analysis (ESA) algorithm.

ESA is an unsupervised algorithm used by Oracle Data Mining for feature extraction. ESA does not discover latent features but instead uses explicit features based on an existing knowledge base.

Explicit knowledge often exists in text form. Multiple knowledge bases are available as collections of text documents. These knowledge bases can be generic, for example, Wikipedia, or domain-specific. Data preparation transforms the text into vectors that capture attribute-concept associations.

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

Example 4-13   Using the ore.odmESA Function

```
title <- c('Aids in Africa: Planning for a long war',
               'Mars rover maneuvers for rim shot',
               'Mars express confirms presence of water at Mars south pole',
               'NASA announces major Mars rover finding',
               'Drug access, Asia threat in focus at AIDS summit',
               'NASA Mars Odyssey THEMIS image: typical crater',
               'Road blocks for Aids')

# TEXT contents in character column
df <- data.frame(CUST_ID = seq(length(title)), TITLE = title)
ESA_TEXT <- ore.push(df)

# TEXT contents in clob column
attr(df$TITLE, "ora.type") <- "clob"
```

```
ESA_TEXT_CLOB <- ore.push(df)

# Create text policy (CTXSYS.CTX_DDL privilege is required)
ore.exec("Begin ctx_ddl.create_policy('ESA_TXTPOL'); End;")

# Specify TEXT POLICY_NAME, MIN_DOCUMENTS, MAX_FEATURES and
# ESA algorithm settings in odm.settings
esa.mod <- ore.odmESA(~ TITLE, data = ESA_TEXT_CLOB,
 odm.settings = list(case_id_column_name = "CUST_ID",
                     ODMS_TEXT_POLICY_NAME = "ESA_TXTPOL",
                     ODMS_TEXT_MIN_DOCUMENTS = 1,
                     ODMS_TEXT_MAX_FEATURES = 3,
                     ESAS_MIN_ITEMS = 1,
                     ESAS_VALUE_THRESHOLD = 0.0001,
                     ESAS_TOPN_FEATURES = 3))
class(esa.mod)
summary(esa.mod)
settings(esa.mod)
features(esa.mod)
predict(esa.mod, ESA_TEXT, type = "class", supplemental.cols = "TITLE")

# Use ctx.settings to specify a character column as TEXT and
# the same settings as above as well as TOKEN_TYPE
esa.mod2 <- ore.odmESA(~ TITLE, data = ESA_TEXT,
  odm.settings = list(case_id_column_name = "CUST_ID", ESAS_MIN_ITEMS = 1),
  ctx.settings = list(TITLE =
    "TEXT(POLICY_NAME:ESA_TXTPOL)(TOKEN_TYPE:STEM)(MIN_DOCUMENTS:1)(MAX_FEATURES:
3)"))
summary(esa.mod2)
settings(esa.mod2)
features(esa.mod2)
predict(esa.mod2, ESA_TEXT_CLOB, type = "class", supplemental.cols = "TITLE")

ore.exec("Begin ctx_ddl.drop_policy('ESA_TXTPOL'); End;")
```

Listing for This Example

```
R> title <- c('Aids in Africa: Planning for a long war',
+             'Mars rover maneuvers for rim shot',
+             'Mars express confirms presence of water at Mars south pole',
+             'NASA announces major Mars rover finding',
+             'Drug access, Asia threat in focus at AIDS summit',
+             'NASA Mars Odyssey THEMIS image: typical crater',
+             'Road blocks for Aids')
R>
R> # TEXT contents in character column
R> df <- data.frame(CUST_ID = seq(length(title)), TITLE = title)
R> ESA_TEXT <- ore.push(df)
R>
R> # TEXT contents in clob column
R> attr(df$TITLE, "ora.type") <- "clob"
R> ESA_TEXT_CLOB <- ore.push(df)
R>
R> # Create a text policy (CTXSYS.CTX_DDL privilege is required)
R> ore.exec("Begin ctx_ddl.create_policy('ESA_TXTPOL'); End;")
R>
R> # Specify TEXT POLICY_NAME, MIN_DOCUMENTS, MAX_FEATURES and
R> # ESA algorithm settings in odm.settings
R> esa.mod <- ore.odmESA(~ TITLE, data = ESA_TEXT_CLOB,
+  odm.settings = list(case_id_column_name = "CUST_ID",
+                      ODMS_TEXT_POLICY_NAME = "ESA_TXTPOL",
```

```
+                       ODMS_TEXT_MIN_DOCUMENTS = 1,
+                       ODMS_TEXT_MAX_FEATURES = 3,
+                       ESAS_MIN_ITEMS = 1,
+                       ESAS_VALUE_THRESHOLD = 0.0001,
+                       ESAS_TOPN_FEATURES = 3))
R> class(esa.mod)
[1] "ore.odmESA" "ore.model"
R> summary(esa.mod)

Call:
ore.odmESA(formula = ~TITLE, data = ESA_TEXT_CLOB, odm.settings =
list(case_id_column_name = "CUST_ID",
    ODMS_TEXT_POLICY_NAME = "ESA_TXTPOL", ODMS_TEXT_MIN_DOCUMENTS = 1,
    ODMS_TEXT_MAX_FEATURES = 3, ESAS_MIN_ITEMS = 1, ESAS_VALUE_THRESHOLD = 1e-04,
    ESAS_TOPN_FEATURES = 3))

Settings:
                                               value
min.items                                          1
topn.features                                      3
value.threshold                                1e-04
odms.missing.value.treatment odms.missing.value.auto
odms.sampling                  odms.sampling.disable
odms.text.max.features                             3
odms.text.min.documents                            1
odms.text.policy.name                     ESA_TXTPOL
prep.auto                                         ON

Features:
   FEATURE_ID ATTRIBUTE_NAME ATTRIBUTE_VALUE COEFFICIENT
1           1     TITLE.AIDS            <NA>   1.0000000
2           2     TITLE.MARS            <NA>   0.4078615
3           2    TITLE.ROVER            <NA>   0.9130438
4           3     TITLE.MARS            <NA>   1.0000000
5           4     TITLE.NASA            <NA>   0.6742695
6           4    TITLE.ROVER            <NA>   0.6742695
7           5     TITLE.AIDS            <NA>   1.0000000
8           6     TITLE.MARS            <NA>   0.4078615
9           6     TITLE.NASA            <NA>   0.9130438
10          7     TITLE.AIDS            <NA>   1.0000000
R> settings(esa.mod)
                   SETTING_NAME                 SETTING_VALUE SETTING_TYPE
1                     ALGO_NAME ALGO_EXPLICIT_SEMANTIC_ANALYS        INPUT
2                ESAS_MIN_ITEMS                             1        INPUT
3            ESAS_TOPN_FEATURES                             3        INPUT
4          ESAS_VALUE_THRESHOLD                         1e-04        INPUT
5  ODMS_MISSING_VALUE_TREATMENT       ODMS_MISSING_VALUE_AUTO      DEFAULT
6                 ODMS_SAMPLING         ODMS_SAMPLING_DISABLE      DEFAULT
7        ODMS_TEXT_MAX_FEATURES                             3        INPUT
8       ODMS_TEXT_MIN_DOCUMENTS                             1        INPUT
9         ODMS_TEXT_POLICY_NAME                    ESA_TXTPOL        INPUT
10                    PREP_AUTO                            ON        INPUT
R> features(esa.mod)
   FEATURE_ID ATTRIBUTE_NAME ATTRIBUTE_VALUE COEFFICIENT
1           1     TITLE.AIDS            <NA>   1.0000000
2           2     TITLE.MARS            <NA>   0.4078615
3           2    TITLE.ROVER            <NA>   0.9130438
4           3     TITLE.MARS            <NA>   1.0000000
5           4     TITLE.NASA            <NA>   0.6742695
6           4    TITLE.ROVER            <NA>   0.6742695
7           5     TITLE.AIDS            <NA>   1.0000000
```

```
8           6     TITLE.MARS            <NA>   0.4078615
9           6     TITLE.NASA            <NA>   0.9130438
10          7     TITLE.AIDS            <NA>   1.0000000
R> predict(esa.mod, ESA_TEXT, type = "class", supplemental.cols = "TITLE")
                                                       TITLE FEATURE_ID
1                  Aids in Africa: Planning for a long war          1
2                        Mars rover maneuvers for rim shot          2
3 Mars express confirms presence of water at Mars south pole          3
4                  NASA announces major Mars rover finding          4
5           Drug access, Asia threat in focus at AIDS summit          1
6             NASA Mars Odyssey THEMIS image: typical crater          6
7                                    Road blocks for Aids          1
R>
R> # Use ctx.settings to specify a character column as TEXT and
R> # the same settings as above as well as TOKEN_TYPE
R> esa.mod2 <- ore.odmESA(~ TITLE, data = ESA_TEXT,
+    odm.settings = list(case_id_column_name = "CUST_ID", ESAS_MIN_ITEMS = 1),
+    ctx.settings = list(TITLE =
+      "TEXT(POLICY_NAME:ESA_TXTPOL)(TOKEN_TYPE:STEM)(MIN_DOCUMENTS:1)(MAX_FEATURES:
3)"))
R> summary(esa.mod2)

Call:
ore.odmESA(formula = ~TITLE, data = ESA_TEXT, odm.settings =
list(case_id_column_name = "CUST_ID",
    ESAS_MIN_ITEMS = 1), ctx.settings = list(TITLE = "TEXT(POLICY_NAME:ESA_TXTPOL)
(TOKEN_TYPE:STEM)(MIN_DOCUMENTS:1)(MAX_FEATURES:3)"))

Settings:
                                               value
min.items                                          1
topn.features                                   1000
value.threshold                            .00000001
odms.missing.value.treatment odms.missing.value.auto
odms.sampling                  odms.sampling.disable
odms.text.max.features                        300000
odms.text.min.documents                            3
prep.auto                                         ON

Features:
   FEATURE_ID ATTRIBUTE_NAME ATTRIBUTE_VALUE COEFFICIENT
1           1     TITLE.AIDS            <NA>   1.0000000
2           2     TITLE.MARS            <NA>   0.4078615
3           2    TITLE.ROVER            <NA>   0.9130438
4           3     TITLE.MARS            <NA>   1.0000000
5           4     TITLE.MARS            <NA>   0.3011997
6           4     TITLE.NASA            <NA>   0.6742695
7           4    TITLE.ROVER            <NA>   0.6742695
8           5     TITLE.AIDS            <NA>   1.0000000
9           6     TITLE.MARS            <NA>   0.4078615
10          6     TITLE.NASA            <NA>   0.9130438
11          7     TITLE.AIDS            <NA>   1.0000000
R> settings(esa.mod2)
                  SETTING_NAME                 SETTING_VALUE SETTING_TYPE
1                    ALGO_NAME ALGO_EXPLICIT_SEMANTIC_ANALYS        INPUT
2               ESAS_MIN_ITEMS                             1        INPUT
3           ESAS_TOPN_FEATURES                          1000      DEFAULT
4         ESAS_VALUE_THRESHOLD                     .00000001      DEFAULT
5 ODMS_MISSING_VALUE_TREATMENT       ODMS_MISSING_VALUE_AUTO      DEFAULT
6                ODMS_SAMPLING         ODMS_SAMPLING_DISABLE      DEFAULT
7       ODMS_TEXT_MAX_FEATURES                        300000      DEFAULT
```

```
8      ODMS_TEXT_MIN_DOCUMENTS                            3      DEFAULT
9                    PREP_AUTO                           ON        INPUT
R> features(esa.mod2)
   FEATURE_ID ATTRIBUTE_NAME ATTRIBUTE_VALUE COEFFICIENT
1           1     TITLE.AIDS           <NA>   1.0000000
2           2     TITLE.MARS           <NA>   0.4078615
3           2    TITLE.ROVER           <NA>   0.9130438
4           3     TITLE.MARS           <NA>   1.0000000
5           4     TITLE.MARS           <NA>   0.3011997
6           4     TITLE.NASA           <NA>   0.6742695
7           4    TITLE.ROVER           <NA>   0.6742695
8           5     TITLE.AIDS           <NA>   1.0000000
9           6     TITLE.MARS           <NA>   0.4078615
10          6     TITLE.NASA           <NA>   0.9130438
11          7     TITLE.AIDS           <NA>   1.0000000
R> predict(esa.mod2, ESA_TEXT_CLOB, type = "class", supplemental.cols = "TITLE")
                                                       TITLE FEATURE_ID
1                    Aids in Africa: Planning for a long war          1
2                          Mars rover maneuvers for rim shot          2
3 Mars express confirms presence of water at Mars south pole          3
4                    NASA announces major Mars rover finding          4
5           Drug access, Asia threat in focus at AIDS summit          1
6             NASA Mars Odyssey THEMIS image: typical crater          6
7                                        Road blocks for Aids          1
R>
R> ore.exec("Begin ctx_ddl.drop_policy('ESA_TXTPOL'); End;")
```

4.2.7 Building an Extensible R Algorithm Model

Beginning in Oracle Database 12*c*, Release 2 (12.2), the `ore.odmRAlg` function creates an Extensible R Algorithm model using Oracle Data Mining.

The Extensible R Algorithm builds, scores, and views an R model using registered R scripts. It supports classification, regression, clustering, feature extraction, attribute importance, and association mining functions.

For information on the `ore.odmRAlg` function arguments and for an example of using the function, invoke `help(ore.odmRAlg)`.

Example 4-14 Using the ore.odmRAlg Function

```
library(OREembed)

digits <- getOption("digits")
options(digits = 5L)

IRIS <- ore.push(iris)

# Regression with glm
ore.scriptCreate("glm_build",
                 function(data, form, family)
                 glm(formula = form, data = data, family = family))

ore.scriptCreate("glm_score",
                  function(mod, data)
                    { res <- predict(mod, newdata = data);
                      data.frame(res) })

ore.scriptCreate("glm_detail", function(mod)
                 data.frame(name=names(mod$coefficients),
                              coef=mod$coefficients))
```

```
ore.scriptList(name = "glm_build")
ore.scriptList(name = "glm_score")
ore.scriptList(name = "glm_detail")

ralg.glm <- ore.odmRAlg(IRIS, mining.function = "regression",
                        formula = c(form="Sepal.Length ~ ."),
                        build.function = "glm_build",
                        build.parameter = list(family="gaussian"),
                        score.function = "glm_score",
                        detail.function = "glm_detail",
                        detail.value = data.frame(name="a", coef=1))
summary(ralg.glm)
predict(ralg.glm, newdata = head(IRIS), supplemental.cols = "Sepal.Length")

ore.scriptDrop(name = "glm_build")
ore.scriptDrop(name = "glm_score")
ore.scriptDrop(name = "glm_detail")

# Classification with nnet
ore.scriptCreate("nnet_build",
                 function(dat, form, sz){
                   require(nnet);
                   set.seed(1234);
                   nnet(formula = formula(form), data = dat,
                           size = sz, linout = TRUE, trace = FALSE);
                  },
                  overwrite = TRUE)

ore.scriptCreate("nnet_detail", function(mod)
                 data.frame(conn = mod$conn, wts = mod$wts),
                 overwrite = TRUE)

ore.scriptCreate("nnet_score",
                 function(mod, data) {
                   require(nnet);
                   res <- data.frame(predict(mod, newdata = data));
                   names(res) <- sort(mod$lev); res
                 })

ralg.nnet <- ore.odmRAlg(IRIS, mining.function = "classification",
                         formula = c(form="Species ~ ."),
                         build.function = "nnet_build",
                         build.parameter = list(sz=2),
                         score.function = "nnet_score",
                         detail.function = "nnet_detail",
                         detail.value = data.frame(conn=1, wts =1))

summary(ralg.nnet)
predict(ralg.nnet, newdata = head(IRIS), supplemental.cols = "Species")

ore.scriptDrop(name = "nnet_build")
ore.scriptDrop(name = "nnet_score")
ore.scriptDrop(name = "nnet_detail")

# Feature extraction with pca
# Feature extraction with pca
ore.scriptCreate("pca_build",
                 function(dat){
                   mod <- prcomp(dat, retx = FALSE)
                   attr(mod, "dm$nfeat") <- ncol(mod$rotation)
                   mod},
```

```
                 overwrite = TRUE)

ore.scriptCreate("pca_score",
                 function(mod, data) {
                   res <- predict(mod, data)
                   as.data.frame(res)},
                 overwrite=TRUE)

ore.scriptCreate("pca_detail",
                 function(mod) {
                   rotation_t <- t(mod$rotation)
                   data.frame(id = seq_along(rownames(rotation_t)),
                                              rotation_t)},
                   overwrite = TRUE)

X <- IRIS[, -5L]
ralg.pca <- ore.odmRAlg(X,
                        mining.function = "feature_extraction",
                        formula = NULL,
                        build.function = "pca_build",
                        score.function = "pca_score",
                        detail.function = "pca_detail",
                        detail.value = data.frame(Feature.ID=1,
                                                  ore.pull(head(X,1L))))

summary(ralg.pca)
head(cbind(X, Pred = predict(ralg.pca, newdata = X)))

ore.scriptDrop(name = "pca_build")
ore.scriptDrop(name = "pca_score")
ore.scriptDrop(name = "pca_detail")

options(digits = digits)
```

Listing for This Example

```
R> library(OREembed)
R>
R> digits <- getOption("digits")
R> options(digits = 5L)
R>
R> IRIS <- ore.push(iris)
R>
R> # Regression with glm
R> ore.scriptCreate("glm_build",
+                  function(data, form, family)
+                  glm(formula = form, data = data, family = family))
R>
R> ore.scriptCreate("glm_score",
+                   function(mod, data)
+                     { res <- predict(mod, newdata = data);
+                       data.frame(res) })
R>
R> ore.scriptCreate("glm_detail", function(mod)
+                  data.frame(name=names(mod$coefficients),
+                                    coef=mod$coefficients))
R>
R> ore.scriptList(name = "glm_build")

NAME
SCRIPT
```

```
1 glm_build function (data, form, family) \nglm(formula = form, data = data, family
= family)

R> ore.scriptList(name = "glm_score")

NAME
   SCRIPT
1 glm_score function (mod, data) \n{\n    res <- predict(mod, newdata = data)\n
data.frame(res)\n}
R> ore.scriptList(name = "glm_detail")

NAME
SCRIPT
1 glm_detail function (mod) \ndata.frame(name = names(mod$coefficients), coef = mod
$coefficients)
R>
R> ralg.glm <- ore.odmRAlg(IRIS, mining.function = "regression",
+                          formula = c(form="Sepal.Length ~ ."),
+                          build.function = "glm_build",
+                          build.parameter = list(family="gaussian"),
+                          score.function = "glm_score",
+                          detail.function = "glm_detail",
+                          detail.value = data.frame(name="a", coef=1))
R>
R> summary(ralg.glm)

Call:
ore.odmRAlg(data = IRIS, mining.function = "regression", formula = c(form =
"Sepal.Length ~ ."),
    build.function = "glm_build", build.parameter = list(family = "gaussian"),
    score.function = "glm_score", detail.function = "glm_detail",
    detail.value = data.frame(name = "a", coef = 1))

Settings:

  value
odms.missing.value.treatment
odms.missing.value.auto
odms.sampling
odms.sampling.disable
prep.auto
    OFF
build.function
RQUSER.glm_build
build.parameter              select 'Sepal.Length ~ .' "form", 'gaussian' "family"
from dual
details.format                 select cast('a' as varchar2(4000)) "name", 1 "coef"
from dual
details.function
RQUSER.glm_detail
score.function
RQUSER.glm_score

               name     coef
1       (Intercept)  2.17127
2      Petal.Length  0.82924
3       Petal.Width -0.31516
4       Sepal.Width  0.49589
5 Speciesversicolor -0.72356
6  Speciesvirginica -1.02350
R> predict(ralg.glm, newdata = head(IRIS), supplemental.cols = "Sepal.Length")
```

```
  Sepal.Length PREDICTION
1          5.1     5.0048
2          4.9     4.7568
3          4.7     4.7731
4          4.6     4.8894
5          5.0     5.0544
6          5.4     5.3889
R>
R> ore.scriptDrop(name = "glm_build")
R> ore.scriptDrop(name = "glm_score")
R> ore.scriptDrop(name = "glm_detail")
R>
R> # Classification with nnet
R> ore.scriptCreate("nnet_build",
+                  function(dat, form, sz){
+                    require(nnet);
+                    set.seed(1234);
+                    nnet(formula = formula(form), data = dat,
+                           size = sz, linout = TRUE, trace = FALSE);
+                  },
+                  overwrite = TRUE)
R>
R> ore.scriptCreate("nnet_detail", function(mod)
+                  data.frame(conn = mod$conn, wts = mod$wts),
+                  overwrite = TRUE)
R>
R> ore.scriptCreate("nnet_score",
+                  function(mod, data) {
+                    require(nnet);
+                    res <- data.frame(predict(mod, newdata = data));
+                    names(res) <- sort(mod$lev); res
+                  })
R>
R> ralg.nnet <- ore.odmRAlg(IRIS, mining.function = "classification",
+                          formula = c(form="Species ~ ."),
+                          build.function = "nnet_build",
+                          build.parameter = list(sz=2),
+                          score.function = "nnet_score",
+                          detail.function = "nnet_detail",
+                          detail.value = data.frame(conn=1, wts =1))
R>
R> summary(ralg.nnet)

Call:
ore.odmRAlg(data = IRIS, mining.function = "classification",
    formula = c(form = "Species ~ ."), build.function = "nnet_build",
    build.parameter = list(sz = 2), score.function = "nnet_score",
    detail.function = "nnet_detail", detail.value = data.frame(conn = 1,
        wts = 1))

Settings:
                                                                    value
clas.weights.balanced                                                 OFF
odms.missing.value.treatment                      odms.missing.value.auto
odms.sampling                                       odms.sampling.disable
prep.auto                                                             OFF
build.function                                          RQUSER.nnet_build
build.parameter             select 'Species ~ .' "form", 2 "sz" from dual
details.format                         select 1 "conn", 1 "wts" from dual
details.function                                       RQUSER.nnet_detail
score.function                                          RQUSER.nnet_score
```

```
   conn       wts
1     0   1.46775
2     1 -12.88542
3     2  -4.38886
4     3   9.98648
5     4  16.57056
6     0   0.97809
7     1  -0.51626
8     2  -0.94815
9     3   0.13692
10    4   0.35104
11    0  37.22475
12    5 -66.49123
13    6  70.81160
14    0  -4.50893
15    5   7.01611
16    6  20.88774
17    0 -32.15127
18    5  58.92088
19    6 -91.96989
R> predict(ralg.nnet, newdata = head(IRIS), supplemental.cols = "Species")
  Species PREDICTION PROBABILITY
1  setosa     setosa     0.99999
2  setosa     setosa     0.99998
3  setosa     setosa     0.99999
4  setosa     setosa     0.99998
5  setosa     setosa     1.00000
6  setosa     setosa     0.99999
R>
R> ore.scriptDrop(name = "nnet_build")
R> ore.scriptDrop(name = "nnet_score")
R> ore.scriptDrop(name = "nnet_detail")
R>
R> ore.scriptCreate("pca_build",
+                  function(dat){
+                    mod <- prcomp(dat, retx = FALSE)
+                    attr(mod, "dm$nfeat") <- ncol(mod$rotation)
+                    mod},
+                  overwrite = TRUE)
R>
R> ore.scriptCreate("pca_score",
+                  function(mod, data) {
+                    res <- predict(mod, data)
+                    as.data.frame(res)},
+                  overwrite=TRUE)
R>
R> ore.scriptCreate("pca_detail",
+                  function(mod) {
+                    rotation_t <- t(mod$rotation)
+                    data.frame(id = seq_along(rownames(rotation_t)),
+                                                rotation_t)},
+                  overwrite = TRUE)
R>
R> X <- IRIS[, -5L]
R> ralg.pca <- ore.odmRAlg(X,
+                        mining.function = "feature_extraction",
+                        formula = NULL,
+                        build.function = "pca_build",
+                        score.function = "pca_score",
+                        detail.function = "pca_detail",
```

```
+                          detail.value = data.frame(Feature.ID=1,
+                                                    ore.pull(head(X,1L))))
R>
R> summary(ralg.pca)

Call:
ore.odmRAlg(data = X, mining.function = "feature_extraction",
    formula = NULL, build.function = "pca_build", score.function = "pca_score",
    detail.function = "pca_detail", detail.value = data.frame(Feature.ID = 1,
        ore.pull(head(X, 1L))))

Settings:
                                                                       value
odms.missing.value.treatment                        odms.missing.value.auto
odms.sampling                                         odms.sampling.disable
prep.auto                                                               OFF
build.function                                             RQUSER.pca_build
details.format    select 1 "Feature.ID", 5.1 "Sepal.Length", 3.5 "Sepal.Width", 1.4
"Petal.Length", 0.2 "Petal.Width" from dual
details.function                                          RQUSER.pca_detail
score.function                                             RQUSER.pca_score

  Feature.ID Sepal.Length Sepal.Width Petal.Length Petal.Width
1          1     0.856671    0.358289      0.36139   -0.084523
2          2    -0.173373   -0.075481      0.65659    0.730161
3          3     0.076236    0.545831     -0.58203    0.597911
4          4     0.479839   -0.753657     -0.31549    0.319723
R> head(cbind(X, Pred = predict(ralg.pca, newdata = X)))
  Sepal.Length Sepal.Width Petal.Length Petal.Width FEATURE_ID
1          5.1         3.5          1.4         0.2          2
2          4.9         3.0          1.4         0.2          4
3          4.7         3.2          1.3         0.2          3
4          4.6         3.1          1.5         0.2          4
5          5.0         3.6          1.4         0.2          2
6          5.4         3.9          1.7         0.4          2
R>
R> ore.scriptDrop(name = "pca_build")
R> ore.scriptDrop(name = "pca_score")
R> ore.scriptDrop(name = "pca_detail")
R>
R> options(digits = digits)
```

4.2.8 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 4-15 Building a Linear Regression Model

This example builds a linear regression model using the `longley` data set.

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

Listing for Example 4-15 (page 4-41)

```
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 4-16 Using Ridge Estimation for the Coefficients of the ore.odmGLM Model

This example uses the `longley_of` `ore.frame` from Example 4-15 (page 4-41). Example 4-16 (page 4-42) invokes the `ore.odmGLM` function and specifies using ridge estimation for the coefficients.

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

Listing for Example 4-16 (page 4-42)

```
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 4-17 Building a Logistic Regression GLM

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

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

Listing for Example 4-17 (page 4-42)

```
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 4-18 Specifying a Reference Value in Building a Logistic Regression GLM

This example builds a logistic regression (classification) model and specifies a reference value. The example uses the `infert_of` `ore.frame` from Example 4-17 (page 4-42).

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

Listing for Example 4-18 (page 4-43)

```
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
```

4.2.9 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 4-19 Using the ore.odmKM Function

This example 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`, then coerces `x` to a `data.frame` and pushes it to the database as `x_of`, an `ore.frame` object. The example next invokes the `ore.odmKMeans` function to build the KM model, `km.mod1`. It then invokes the `summary` and `histogram` functions on the model. Figure 4-3 (page 4-45) shows the graphic displayed by the `histogram` function.

Finally, the example makes a prediction using the model, pulls the result to local memory, and plots the results.Figure 4-4 (page 4-46) shows the graphic displayed by the `points` 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)
# 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 4-19 (page 4-44)

```
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)
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> 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 4-3 (page 4-45) shows the graphic displayed by the invocation of the `histogram` function in Example 4-19 (page 4-44).

Figure 4-3   Cluster Histograms for the km.mod1 Model

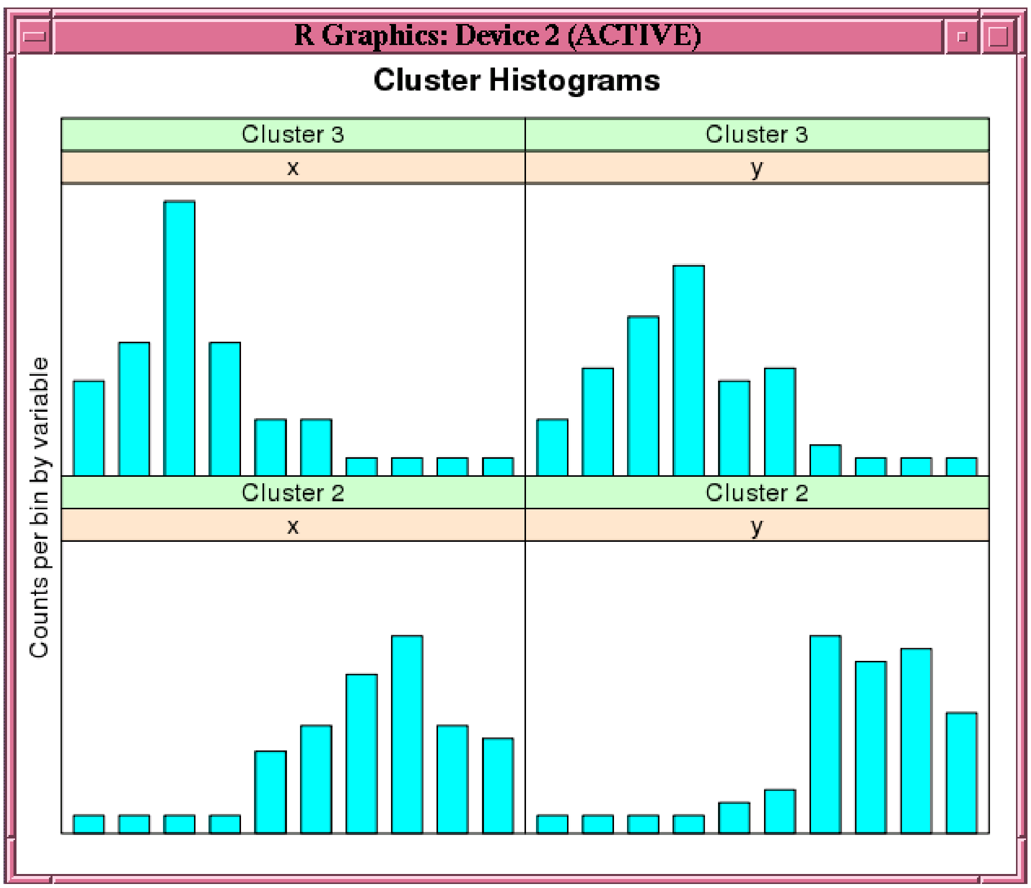
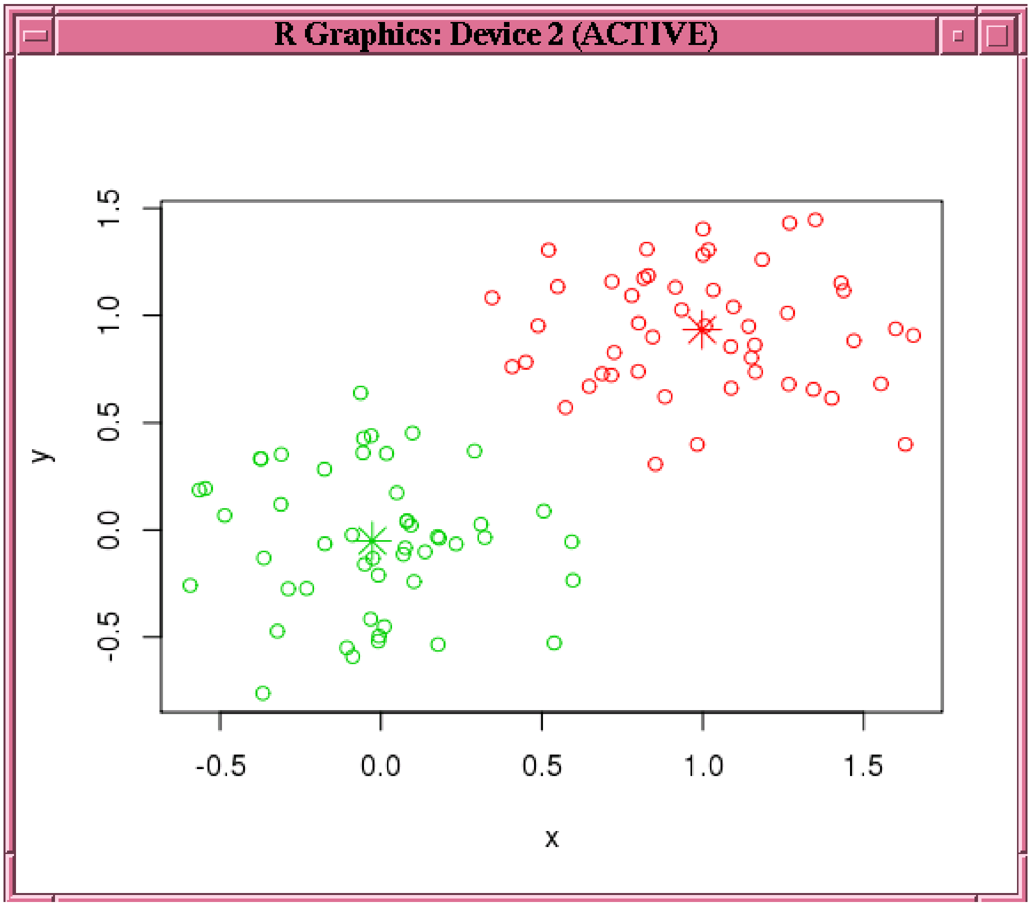


Figure 4-4 (page 4-46) shows the graphic displayed by the invocation of the `points` function in Example 4-19 (page 4-44).

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



4.2.10 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 4-20 Using the ore.odmNB Function

This example creates an input `ore.frame`, builds a Naive Bayes model, makes predictions, and generates 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)
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 4-11 (page 4-24)

```
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
```

4.2.11 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 4-21   Using the ore.odmNMF Function

This example creates an NMF model on a training data set and scores on a test data set.

```
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 4-21 (page 4-48)

```
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
```

4.2.12 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 4-22 Using the ore.odmOC Function

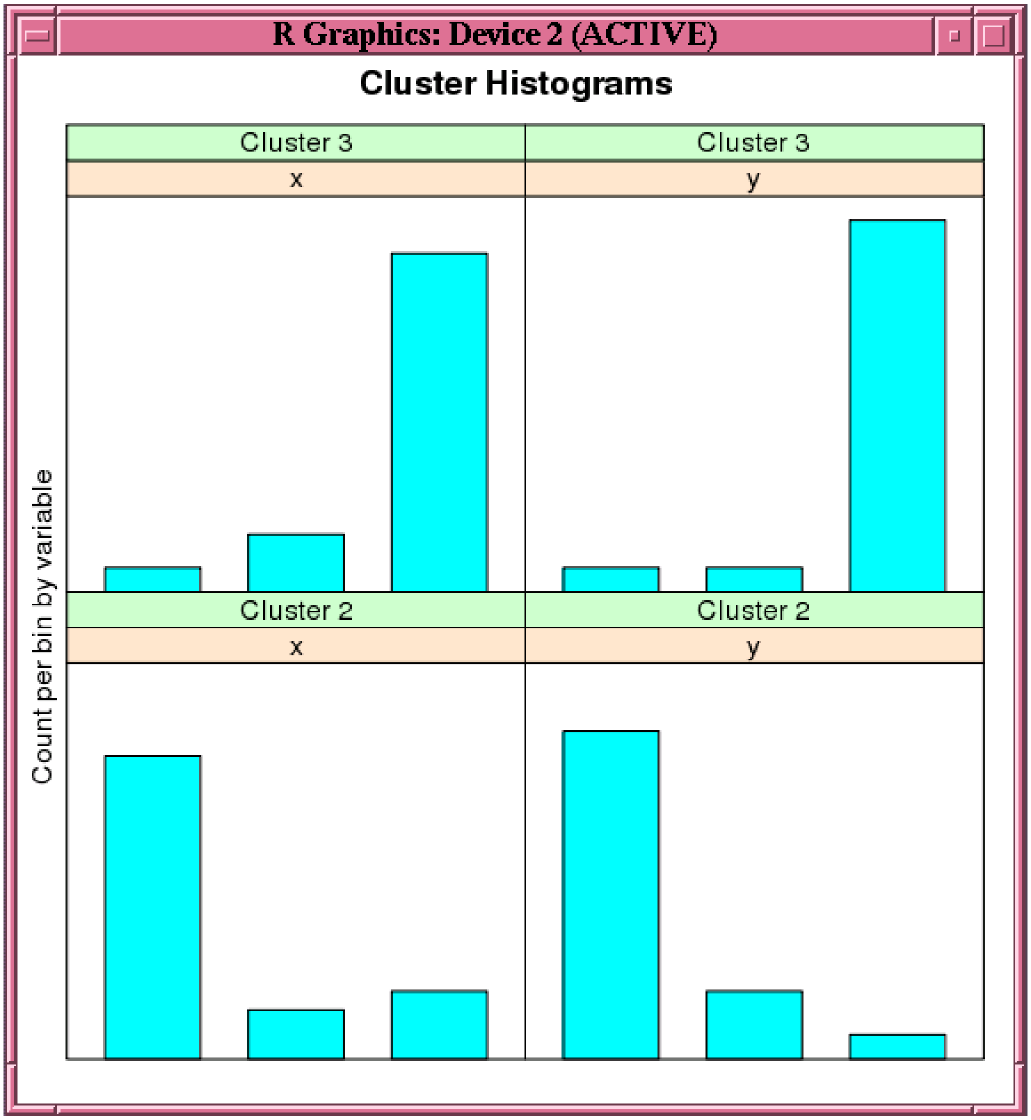
This example creates an OC model on a synthetic data set. Figure 4-5 (page 4-52) shows the histogram of the resulting clusters.

```
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 4-22 (page 4-51)

```
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 4-5 (page 4-52).
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 4-5   Output of the histogram Function for the ore.odmOC Model



4.2.13 Building a Singular Value Decomposition Model

Beginning in Oracle Database 12*c*, Release 2 (12.2), the `ore.odmSVD` function creates a model that uses the Oracle Data Mining Singular Value Decomposition (SVD) algorithm.

Singular Value Decomposition (SVD) is a feature extraction algorithm. SVD is orthogonal linear transformations that capture the underlying variance of the data by decomposing a rectangular matrix into three matrixes: 'U', 'D', and 'V'. Matrix 'D' is a diagonal matrix and its singular values reflect the amount of data variance captured by the bases.

Example 4-23   Using the ore.odmSVD Function

```
IRIS <- ore.push(cbind(Id = seq_along(iris[[1L]]), iris))

svd.mod <- ore.odmSVD(~. -Id, IRIS)
summary(svd.mod)
d(svd.mod)
v(svd.mod)
```

```
head(predict(svd.mod, IRIS, supplemental.cols = "Id"))

svd.pmod <- ore.odmSVD(~. -Id, IRIS,
                         odm.settings = list(odms_partition_columns = "Species"))
summary(svd.pmod)
d(svd.pmod)
v(svd.pmod)
head(predict(svd.pmod, IRIS, supplemental.cols = "Id"))
```

Listing for This Example

```
R> IRIS <- ore.push(cbind(Id = seq_along(iris[[1L]]), iris))
R> 
R> svd.mod <- ore.odmSVD(~. -Id, IRIS)
R> summary(svd.mod)
Call:
ore.odmSVD(formula = ~. - Id, data = IRIS)

Settings: 
                                                 value
odms.missing.value.treatment odms.missing.value.auto
odms.sampling                  odms.sampling.disable
prep.auto                                         ON
scoring.mode                             scoring.svd
u.matrix.output                     u.matrix.disable

d:
  FEATURE_ID      VALUE
1          1 96.2182677
2          2 19.0780817
3          3  7.2270380
4          4  3.1502152
5          5  1.8849634
6          6  1.1474731
7          7  0.5814097
v:
  ATTRIBUTE_NAME ATTRIBUTE_VALUE        '1'         '2'          '3'
'4'         '5'         '6'          '7'
1   Petal.Length            <NA> 0.51162932  0.65943465 -0.004420703  0.05479795
-0.51969015  0.17392232 -0.005674672
2    Petal.Width            <NA> 0.16745698  0.32071102  0.146484369  0.46553390
0.72685033  0.31962337 -0.021274748
3   Sepal.Length            <NA> 0.74909171 -0.26482593 -0.102057243 -0.49272847
0.31969417 -0.09379235 -0.067308615
4    Sepal.Width            <NA> 0.37906736 -0.50824062  0.142810811  0.69139828
-0.25849391 -0.17606099 -0.041908520
5        Species          setosa 0.03170407 -0.32247642  0.184499940 -0.12245506
-0.14348647  0.76017824  0.497502783
6        Species      versicolor 0.04288799  0.04054823 -0.780684855  0.19827972
0.07363250 -0.12354271  0.571881302
7        Species       virginica 0.05018593  0.16796988  0.551546107 -0.07177990
0.08109974 -0.48442099  0.647048040
Warning message:
In u.ore.odmSVD(object) : U matrix is not calculated.
R> d(svd.mod)
  FEATURE_ID      VALUE
1          1 96.2182677
2          2 19.0780817
3          3  7.2270380
4          4  3.1502152
5          5  1.8849634
```

```
6         6  1.1474731
7         7  0.5814097
Warning message:
ORE object has no unique key - using random order
R> v(svd.mod)
  ATTRIBUTE_NAME ATTRIBUTE_VALUE        '1'         '2'          '3'
'4'         '5'         '6'          '7'
1   Petal.Length           <NA> 0.51162932  0.65943465 -0.004420703  0.05479795
-0.51969015  0.17392232 -0.005674672
2    Petal.Width           <NA> 0.16745698  0.32071102  0.146484369  0.46553390
0.72685033  0.31962337 -0.021274748
3   Sepal.Length           <NA> 0.74909171 -0.26482593 -0.102057243 -0.49272847
0.31969417 -0.09379235 -0.067308615
4    Sepal.Width           <NA> 0.37906736 -0.50824062  0.142810811  0.69139828
-0.25849391 -0.17606099 -0.041908520
5        Species         setosa 0.03170407 -0.32247642  0.184499940 -0.12245506
-0.14348647  0.76017824  0.497502783
6        Species     versicolor 0.04288799  0.04054823 -0.780684855  0.19827972
0.07363250 -0.12354271  0.571881302
7        Species      virginica 0.05018593  0.16796988  0.551546107 -0.07177990
0.08109974 -0.48442099  0.647048040
Warning message:
ORE object has no unique key - using random order
R> head(predict(svd.mod, IRIS, supplemental.cols = "Id"))
  Id        '1'        '2'        '3'         '4'           '5'
'6'          '7' FEATURE_ID
1  1 0.06161595 -0.1291839 0.02586865 -0.01449182  1.536727e-05 -0.023495349
-0.007998605          2
2  2 0.05808905 -0.1130876 0.01881265 -0.09294788  3.466226e-02  0.069569113
0.051195429          2
3  3 0.05678818 -0.1190959 0.02565027 -0.01950986  8.851560e-04  0.040073030
0.060908867          2
4  4 0.05667915 -0.1081308 0.02496402 -0.02233741 -5.750222e-02  0.093904181
0.077741713          2
5  5 0.06123138 -0.1304597 0.02925687  0.02309694 -3.065834e-02 -0.030664898
-0.003629897          2
6  6 0.06747071 -0.1302726 0.03340671  0.06114966 -9.547838e-03 -0.008210224
-0.081807741          2
R>
R> svd.pmod <- ore.odmSVD(~. -Id, IRIS,
+                         odm.settings = list(odms_partition_columns = "Species"))
R> summary(svd.pmod)
$setosa

Call:
ore.odmSVD(formula = ~. - Id, data = IRIS, odm.settings =
list(odms_partition_columns = "Species"))

Settings:
                                               value
odms.max.partitions                             1000
odms.missing.value.treatment odms.missing.value.auto
odms.partition.columns                     "Species"
odms.sampling                  odms.sampling.disable
prep.auto                                         ON
scoring.mode                             scoring.svd
u.matrix.output                     u.matrix.disable

d:
  FEATURE_ID      VALUE
1          1 44.2872290
```

```
2          2  1.5719162
3          3  1.1458732
4          4  0.6836692
v:
  ATTRIBUTE_NAME ATTRIBUTE_VALUE       '1'         '2'        '3'         '4'
1   Petal.Length            <NA> 0.2334487  0.46456598  0.8317440 -0.19463332
2    Petal.Width            <NA> 0.0395488  0.04182015  0.1946750  0.97917752
3   Sepal.Length            <NA> 0.8010073  0.40303704 -0.4410167  0.03811461
4    Sepal.Width            <NA> 0.5498408 -0.78739486  0.2753323 -0.04331888

$versicolor

Call:
ore.odmSVD(formula = ~. - Id, data = IRIS, odm.settings =
list(odms_partition_columns = "Species"))

Settings:
                                                 value
odms.max.partitions                               1000
odms.missing.value.treatment odms.missing.value.auto
R> # xyz
R> d(svd.pmod)
   PARTITION_NAME FEATURE_ID      VALUE
1          setosa          1 44.2872290
2          setosa          2  1.5719162
3          setosa          3  1.1458732
4          setosa          4  0.6836692
5      versicolor          1 56.2523412
6      versicolor          2  1.9106625
7      versicolor          3  1.7015929
8      versicolor          4  0.6986103
9       virginica          1 66.2734064
10      virginica          2  2.4318639
11      virginica          3  1.6007740
12      virginica          4  1.2958261
Warning message:
ORE object has no unique key - using random order
R> v(svd.pmod)
   PARTITION_NAME ATTRIBUTE_NAME ATTRIBUTE_VALUE       '1'         '2'
'3'         '4'
1          setosa   Petal.Length            <NA> 0.2334487  0.46456598  0.83174398
-0.19463332
2          setosa    Petal.Width            <NA> 0.0395488  0.04182015  0.19467497
0.97917752
3          setosa   Sepal.Length            <NA> 0.8010073  0.40303704 -0.44101672
0.03811461
4          setosa    Sepal.Width            <NA> 0.5498408 -0.78739486  0.27533228
-0.04331888
5      versicolor   Petal.Length            <NA> 0.5380908  0.49576111 -0.60174021
-0.32029352
6      versicolor    Petal.Width            <NA> 0.1676394  0.36693207 -0.03448373
0.91436795
7      versicolor   Sepal.Length            <NA> 0.7486029 -0.64738491  0.06943054
0.12516311
8      versicolor    Sepal.Width            <NA> 0.3492119  0.44774385  0.79492074
-0.21372297
9       virginica   Petal.Length            <NA> 0.5948985 -0.26368708  0.65157671
-0.38988802
10      virginica    Petal.Width            <NA> 0.2164036  0.59106806  0.42921836
0.64774968
11      virginica   Sepal.Length            <NA> 0.7058813 -0.27846153 -0.53436210
```

```
0.37235450
12      virginica    Sepal.Width           <NA> 0.3177999  0.70962445 -0.32507927
-0.53829342
Warning message:
ORE object has no unique key - using random order
R> head(predict(svd.pmod, IRIS, supplemental.cols = "Id"))
  Id       '1'          '2'          '3'         '4' FEATURE_ID
1  1 0.1432539 -0.026487881 -0.071688339 -0.04956008          1
2  2 0.1334289  0.172689424 -0.114854368 -0.02902893          2
3  3 0.1317675 -0.008327214 -0.062409295 -0.02438248          1
4  4 0.1297716  0.075232572  0.097222019 -0.08055912          1
5  5 0.1426868 -0.102219140 -0.009172782 -0.06147133          1
6  6 0.1554060 -0.055950655  0.160698708  0.14286095          3
```

4.2.14 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 4-24 Using the ore.odmSVM Function and Generating a Confusion Matrix

This example 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.

```
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 4-24 (page 4-56)

```
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 4-25   Using the ore.odmSVM Function and Building a Regression Model

This example 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.

```
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 4-25 (page 4-57)

```
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
```

Example 4-26 Using the ore.odmSVM Function and Building an Anomaly Detection Model

This example demonstrates SVN anomaly detection. It uses `mtcars_of` created in the classification example and builds 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 4-26 (page 4-58)

```
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
```

4.3 Cross-Validating Models

Predictive models are usually built on given data and verified on held-aside or unseen data. Cross-validation is a model improvement technique that avoids the limitations of a single train-and-test experiment by building and testing multiple models through repeated sampling from the available data. It's purpose is to offer better insight into how well the model would generalize to new data and to avoid over-fitting and deriving wrong conclusions from misleading peculiarities of the seen data.

The `ore.CV` utility R function uses Oracle R Enterprise for performing cross-validation of regression and classification models. The function `ore.CV` is available for download from the following Oracle R Technologies blog post:

https://blogs.oracle.com/R/entry/model_cross_validation_with_ore

For a select set of algorithms and cases, the function `ore.CV` performs cross-validation for models that were generated by Oracle R Enterprise regression and classification functions using in-database data.

The `ore.CV` function works with models generated by the following Oracle R Enterprise functions:

- `ore.lm`
- `ore.stepwise`
- `ore.glm`
- `ore.neural`

- `ore.odmDT`
- `ore.odmGLM`
- `ore.odmNB`
- `ore.odmSVM`

You can also use `ore.CV` to cross-validate models generated with some R regression functions through Oracle R Enterprise embedded R execution. Those R functions are the following:

- `lm`
- `glm`
- `svm`

For more information on, and examples of, using `ore.CV`, and to download the function itself, see the blog post:

https://blogs.oracle.com/R/entry/model_cross_validation_with_ore

5

Predicting With R Models

This chapter describes the Oracle R Enterprise function `ore.predict` and provides some examples of its use. The chapter contains the following topics:

5.1 About the ore.predict Function

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.

The `ore.predict` function provides the fastest way to operationalize R-based models for scoring in Oracle Database. The function has no dependencies on PMML or any other plug-ins.

Some advantages of using the `ore.predict` function to score data in the database are the following:

- Uses R-generated models to score in-database data.

  The data to score is in an `ore.frame` object.

- Maximizes the use of Oracle Database as a compute engine.

  The database provides a commercial grade, high performance, scalable scoring engine.

- Simplifies application workflow.

  You can go from a model to SQL scoring in one step.

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 model objects listed in Table 5-1 (page 5-2). The value of the `newdata` argument is an `ore.frame` object that contains the data to score. The `ore.predict` function has methods for use with specific R model classes. The `...` argument represents the various additional arguments that are accepted by the different methods.

Function `ore.predict` has methods that support the model objects listed in Table 5-1 (page 5-2).

Table 5-1 Models Supported by the ore.predict Function

Class of Model	Description of Model
`glm`	Generalized linear model
`kmeans`	*k*-Means clustering model
`lm`	Linear regression model
`matrix`	A `matrix` with no more than 1000 rows, for use in an `hclust` hierarchical clustering model
`multinom`	Multinomial log-linear model
`nnet`	Neural network model
`ore.model`	An Oracle R Enterprise model from the `OREModels` package
`prcomp`	Principal components analysis on a matrix
`princomp`	Principal components analysis on a numeric matrix
`rpart`	Recursive partitioning and regression tree model

For the function signatures of the `ore.predict` methods, invoke the `help` function on the following, as in `help("ore.predict-kmeans")`:

- `ore.predict-glm`
- `ore.predict-kmeans`
- `ore.predict-lm`
- `ore.predict-matrix`
- `ore.predict-multinom`
- `ore.predict-nnet`
- `ore.predict-ore.model`
- `ore.predict-prcomp`
- `ore.predict-princomp`
- `ore.predict-rpart`

5.2 Using the ore.predict Function

These examples demonstrate the use of the `ore.predict` function.

Example 5-1 Using the ore.predict Function on a Linear Regression Model

This example builds a linear regression model, `irisModel`, using the `lm` function on the `iris` `data.frame`. It pushes the data set to the database as the temporary table IRIS and the corresponding `ore.frame` proxy, `IRIS`. The example scores the model by invoking `ore.predict` on it and then combines the prediction with `IRIS` `ore.frame` object. Finally, it displays the first six rows of the resulting object.

```
IRISModel <- lm(Sepal.Length ~ ., data = iris)
IRIS <- ore.push(iris)
IRIS_pred <- ore.predict(IRISModel, IRIS, se.fit = TRUE,
                            interval = "prediction")
IRIS <- cbind(IRIS, IRIS_pred)
head(IRIS)
```

Listing for This Example

```
R> IRISModel <- lm(Sepal.Length ~ ., data = iris)
R> IRIS <- ore.push(iris)
R> IRIS_pred <- ore.predict(IRISModel, IRIS, se.fit = TRUE,
+                              interval = "prediction")
R> IRIS <- cbind(IRIS, IRIS_pred)
R> head(IRIS)
  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)
  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 Using the ore.predict Function on a Generalized Linear Regression Model

This example builds a generalized linear model using the `infert` data set and then invokes the `ore.predict` function on the 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 This Example

```
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
```

Example 5-3   Using the ore.predict Function on an ore.model Model

This example pushes the `iris` data set to the database as the temporary table IRIS and the corresponding `ore.frame` proxy, `IRIS`. The example builds a linear regression model, `IRISModel2`, using the `ore.lm` function. It scores the model and adds a column to `IRIS`.

```
IRIS <- ore.push(iris)
IRISModel2 <- ore.lm(Sepal.Length ~ ., data = IRIS)
IRIS$PRED <- ore.predict(IRISModel2, IRIS)
head(IRIS, 3)
```

Listing for This Example

```
R> IRIS <- ore.push(iris)
R> IRISModel2 <- ore.lm(Sepal.Length ~ ., data = IRIS)
R> IRIS$PRED <- ore.predict(IRISModel, IRIS)
R> head(IRIS, 3)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species     PRED
1          5.1         3.5          1.4         0.2  setosa 5.004788
2          4.9         3.0          1.4         0.2  setosa 4.756844
3          4.7         3.2          1.3         0.2  setosa 4.773097
```

6

Using Oracle R Enterprise Embedded R Execution

Embedded R execution in Oracle R Enterprise enables you to invoke R scripts in R sessions that run on the Oracle Database server. This chapter discusses embedded R execution in the following topics:

6.1 About Oracle R Enterprise Embedded R Execution

In Oracle R Enterprise, embedded R execution is the ability to store R scripts in the Oracle R Enterprise R script repository and to invoke such scripts. When invoked, a script executes in one or more R engines that run on the database server 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:

6.1.1 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 R engines running on the server.

- 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 in R scripts running on the database server.
- 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.

6.1.2 APIs for Embedded R Execution

Oracle R Enterprise provides R and SQL application programming interfaces for embedded R execution. The following table lists the R functions and the equivalent SQL functions and procedures for embedded R execution and R script repository management. The function *f* refers to a named R function or an R function defined in a script in the Oracle R Enterprise 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 times to invoke the function. Supports parallel execution of each *f* invocation in the pool of R engines running on the database server.
`ore.grant`	`rqGrant`	Grants read privilege access to a datastore or script.

Table 6-1 (Cont.) R and SQL APIs for Embedded R Execution

R API	SQL API	Description
`ore.revoke`	`rqRevoke`	Revokes read privilege access to a datastore or script.
`ore.scriptCreate`	`sys.rqScriptCreate`	Adds the provided R function into the Oracle R Enterprise R script repository with the provided name.
`ore.scriptDrop`	`sys.rqScriptDrop`	Removes the named R function from the Oracle R Enterprise R script repository.
`ore.scriptList`	`ALL_RQ_SCRIPTS` `USER_RQ_SCRIPTS`	Lists information about scripts.
`ore.scriptLoad`	No equivalent.	Loads the R function of a script into the R environment.

See Also:

- "R Interface for Embedded R Execution (page 6-9)"
- "SQL Interface for Embedded R Execution (page 6-43)"

6.1.3 Security Considerations for Scripts

Because R 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 R Enterprise R script repository or drop scripts from the repository.

The installation of Oracle R Enterprise on the database server 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.

When creating a script, the owner can use the `global` argument to specify whether the script is public or private. If `global = TRUE`, then all users have read privilege access to the script. If `global = FALSE`, which is the default, then the owner can share the script by granting access to other users. The owner can revoke the access at any time.

See Also:

- "Manage Scripts in R (page 6-13)"
- "Manage Scripts in SQL (page 6-46)"

6.1.4 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`, `ore.rowApply`, *rq.groupEval*, and `rq.rowEval` 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 the `ore.groupApply`, `ore.rowApply`, and `ore.indexApply` 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.

For the *rq.groupEval*, and `rq.rowEval` functions, the degree of parallelism is specified by a `PARALLEL` hint in the input cursor argument.

In data-parallel execution for the `ore.groupApply` and *rq.groupEval* functions, 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` and `rq.rowEval` functions, one or more R engines perform 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 (page 1-12)"

6.1.5 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. To avoid incompatibilities, you must install the same version of the package on both the client and server machines.

An Oracle Database Administrator (DBA) can install a package on a database server so that it can be used by embedded R execution functions or by any R user. The DBA can install a package on a single database server or on multiple database servers.

A DBA would typically do the following:

1. Download and install the package from CRAN. Downloading a package from CRAN requires an Internet connection.

2. In an Oracle R Enterprise session running on the server, load the package. Verify that the package is installed correctly by using a function in the package.

To install a package on a single database server, do one of the following:

- In an Oracle R Enterprise session running on the server, invoke the `install.packages` function, as shown in Example 6-1 (page 6-5). The function downloads the package and installs dependencies automatically.

- 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 those packages, also.

  From the operating system command line, 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`. See Example 6-2 (page 6-6).

To install a package, and any dependent packages, 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, as shown in Example 6-3 (page 6-8). For detailed instructions on using DCLI to install packages, see *Oracle R Enterprise Installation and Administration Guide*.

To verify that the package is installed correctly, load the package and use a function in the package, as shown in Example 6-4 (page 6-8).

Example 6-1 Installing a Package for a Single Database in an Oracle R Enterprise Session

This example invokes the `install.packages` function to download the `C50` package from CRAN and to install it. The `C50` package contains functions for creating C5.0 decision trees and rule-based models for pattern recognition.

The output this example, which is not shown, is almost identical to the output of the `ORE CMD INSTALL` command in Example 6-2 (page 6-6).

```
install.packages("c50")
```

Example 6-2 Installing a Package for a Single Database from the Command Line

This example demonstrates downloading the `C50` package from CRAN and installing it with `ORE CMD INSTALL` from a Linux command line.

```
wget http://cran.r-project.org/src/contrib/C50_0.1.0-19.tar.gz
ORE CMD INSTALL C50_0.1.0-19.tar.gz
```

Lising for Example 6-2 (page 6-6)

```
$ wget http://cran.r-project.org/src/contrib/C50_0.1.0-19.tar.gz
# The output of wget is not shown.
$ ORE CMD INSTALL C50_0.1.0-19.tar.gz
* installing to library '/example/dbhome_1/R/library'
* installing *source* package 'C50' ...
** package 'C50' successfully unpacked and MD5 sums checked
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
configure: creating ./config.status
config.status: creating src/Makevars
** libs
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c attwinnow.c -o attwinnow.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c classify.c -o classify.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c confmat.c -o confmat.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c construct.c -o construct.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c contin.c -o contin.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c discr.c -o discr.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c formrules.c -o formrules.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c formtree.c -o formtree.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c getdata.c -o getdata.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c getnames.c -o getnames.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c global.c -o global.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c hash.c -o hash.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c hooks.c -o hooks.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c implicitatt.c -o implicitatt.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c info.c -o info.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c mcost.c -o mcost.o
```

```
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c modelfiles.c -o modelfiles.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c p-thresh.c -o p-thresh.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c prune.c -o prune.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c rc50.c -o rc50.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c redefine.c -o redefine.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c rsample.c -o rsample.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c rulebasedmodels.c -o rulebasedmodels.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c rules.c -o rules.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c ruletree.c -o ruletree.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c siftrules.c -o siftrules.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c sort.c -o sort.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c strbuf.c -o strbuf.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c subset.c -o subset.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c top.c -o top.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c trees.c -o trees.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c
update.c -o update.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c utility.c -o utility.o
gcc -m64 -std=gnu99 -I/usr/include/R -DNDEBUG -DNDEBUG -I/usr/local/include   -
ffloat-store -g -fpic  -g -O2  -c xval.c -o xval.o
gcc -m64 -std=gnu99 -shared -L/usr/local/lib64 -o C50.so attwinnow.o classify.o
confmat.o construct.o contin.o discr.o formrules.o formtree.o getdata.o getnames.o
global.o hash.o hooks.o implicitatt.o info.o mcost.o modelfiles.o p-thresh.o prune.o
rc50.o redefine.o rsample.o rulebasedmodels.o rules.o ruletree.o siftrules.o sort.o
strbuf.o subset.o top.o trees.o update.o utility.o xval.o -L/usr/lib64/R/lib -lR
installing to /example/dbhome_1/R/library/C50/libs
** R
** data
** preparing package for lazy loading
** help
*** installing help indices
  converting help for package 'C50'
    finding HTML links ... done
    C5.0                                    html
    C5.0Control                             html
    churn                                   html
    predict.C5.0                            html
    summary.C5.0                            html
    varImp.C5.0                             html
** building package indices
** testing if installed package can be loaded
* DONE (C50)
```

Example 6-3 Installing a Package Using DCLI

This example shows the DLCI command for installing the `C50` package. The `dcli -g` flag designates a file containing a list of nodes to install on, and the `-l` flag specifies the user ID to use when executing the commands. For more information on using DCLI, see *Oracle R Enterprise Installation and Administration Guide*.

```
dcli -g nodes -l oracle R CMD INSTALL C50_0.1.0-19.tar.gz
```

Example 6-4 Using a C50 Package Function

This example shows starting R, connecting to Oracle R Enterprise on the server, loading the `C50` package, and using a function in the package. The example starts R by executing the `ORE` command from the Linux command line. The example connects to Oracle R Enterprise and then loads the `C50` package. It invokes the `demo` function to look for example programs in the package. Because the package does not have examples, this example then gets help for the `C5.0` function. The example invokes example code from the help.

```
ORE

library(ORE)
ore.connect(user = "RQUSER", sid = "orcl", host = "myhost",
            password = "rquserStrongPassword", port = 1521, all=TRUE)

library(C50)
demo(package = "C50")
?C5.0
data(churn)
treeModel <- C5.0(x = churnTrain[, -20], y = churnTrain$churn)
treeModel
```

Listing for Example 6-4 (page 6-8)

```
$ ORE

R> library(ORE)
Loading required package: OREbase

Attaching package: 'OREbase'

The following objects are masked from 'package:base':

    cbind, data.frame, eval, interaction, order, paste, pmax, pmin,
    rbind, table

Loading required package: OREembed
Loading required package: OREstats
Loading required package: MASS
Loading required package: OREgraphics
Loading required package: OREeda
Loading required package: OREmodels
Loading required package: OREdm
Loading required package: lattice
Loading required package: OREpredict
Loading required package: ORExml

> ore.connect(user = "RQUSER", sid = "orcl", host = "myhost",
+             password = "rquserStrongPassword", port = 1521, all=TRUE)
Loading required package: ROracle
Loading required package: DBI
```

```
R> library(C50)
R> demo(package = "C50")
no demos found
R> ?C5.0     # Output not shown.
R> data(churn)
R> treeModel <- C5.0(x = churnTrain[, -20], y = churnTrain$churn)
R> treeModel
Call:
C5.0.default(x = churnTrain[, -20], y = churnTrain$churn)

Classification Tree
Number of samples: 3333
Number of predictors: 19

Tree size: 27

Non-standard options: attempt to group attributes
```

See Also:

- "Using a Third-Party Package on the Client (page 3-66)"
- *Oracle R Enterprise Installation and Administration Guide*
- *R Administration and Installation Guide* at `http://cran.r-project.org/doc/manuals/r-release/R-admin.html`
- `http://www.r-bloggers.com/installing-r-packages/`

6.2 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 and store an R function as a script in the Oracle R Enterprise R script repository, grant or revoke read access to a script, list the available scripts, load a script function into the R environment, or drop a script from the repository. This section describes these functions in the following topics:

6.2.1 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 (page 6-10)
- Optional and Control Arguments (page 6-11)
- Structure of Return Value (page 6-12)
- Input Data (page 6-12)
- Parallel Execution (page 6-13)
- Unique Arguments (page 6-13)

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 (page 6-19)" and the other topics on using the embedded R execution functions

6.2.1.1 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` (and optional `FUN.OWNER`)

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 Oracle R Enterprise 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 optional argument `FUN.OWNER` specifies the owner of a script in the R script repository. The owner is the user who created the script. Use this argument only with the `FUN.NAME` argument. When `FUN.NAME` is a private script to which you have been granted read privilege access, use `FUN.OWNNER` to specify the owner of the private script.

The RQSYS schema is the owner of public scripts and the predefined Oracle R Enterprise scripts. For a list of the predefined scripts, invoke `help(“ore.doEval”)` and see the description of the `FUN.NAME` argument. If `FUN.OWNNER` is not specified or is `NULL`, then Oracle R Enterprise looks for the owner in the following order: user of the current session, RQSYS. If the owner of the script is not current user or RQSYS, then an error occurs.

Note:

The Oracle R Enterprise advanced analytics functions in the `OREmodels` package, `ore.glm`, `ore.lm`, `ore.neural`, and `ore.randomForest`, use the embedded R execution framework internally and cannot be used in embedded R execution functions.

6.2.1.2 Optional and Control Arguments

All of the embedded R execution functions 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`.

  If an automatic connection is enabled, the following functionality occurs:

  – The embedded R script is connected to the database.

  – The connection has the same credentials as the session that invokes the embedded R SQL function.

  – The script runs in an autonomous transaction.

  – ROracle queries can work with the automatic connection.

  – Oracle R Enterprise transparency layer functionality is enabled in the embedded script.

- `ore.drop` controls the input data. If the option value is `TRUE`, a one column `data.frame` is converted to a `vector`. The default value is `TRUE`.

- `ore.envAsEmptyenv` controls whether an environment referenced in an object is replaced with an empty environment during serialization. Some types of input parameters and returned objects, such as `list` and `formula`, are serialized before being saved to the database. If the control argument value is `TRUE`, then the referenced environment in the object is replaced with an empty environment whose parent is `.GlobalEnv` and the objects in the original referenced environment are not serialized. In some cases, this can significantly reduce the size of serialized objects. If the control argument value is `FALSE`, then all of the objects in the referenced environment are serialized and can be unserialized and recovered later. The default value is regulated by the global option `ore.envAsEmptyenv`.

- `ore.na.omit` controls the handling of missing values in the input data. If you specify `ore.na.omit = TRUE`, then 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` is passed to the `png` function. If not set, the standard default values for the `png` function are used.

See Also:

For more details about control arguments, see the online help displayed by invoking `help(ore.doEval)`

6.2.1.3 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.

To specify that the corresponding output column of an `ore.frame` have a CLOB or BLOB database data type, you can apply the attribute `ora.type` to a column of a `FUN.VALUE` `data.frame`. For an example of using `ora.type`, see Example 6-11 (page 6-23).

6.2.1.4 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.

6.2.1.5 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 (page 6-4)".

6.2.1.6 Unique Arguments

The `ore.groupApply`, `ore.indexApply`, and `ore.rowApply` functions each take an 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.

6.2.2 Manage Scripts in R

Embedded R execution functions can invoke R functions that are stored as scripts in the Oracle R Enterprise R script repository. You can use the R functions described in this topic to create and manage scripts.

As mentioned in "Input Function to Execute (page 6-10)," the embedded R execution functions can take a `FUN.NAME` argument. That argument specifies the name of a script in the Oracle R Enterprise R script repository. Scripts in the R script repository are also available through the SQL API for embedded R execution.

The R functions for managing scripts are the following:

- `ore.grant`
- `ore.revoke`
- `ore.scriptCreate`
- `ore.scriptList`
- `ore.scriptLoad`
- `ore.scriptDrop`

These functions are described in the following sections:

- Adding a Script (page 6-14)
- Granting or Revoking Read Access to a Script (page 6-14)
- Listing the Available Scripts (page 6-15)
- Loading a Script into an R Environment (page 6-15)
- Dropping a Script (page 6-16)

For an example that uses these functions, see Example 6-5 (page 6-16).

Adding a Script

To add an R function as a script in the Oracle R Enterprise R script repository, use the `ore.createScript` function. To evoke this function, you must have the RQADMIN role. The `ore.createScript` function has the following syntax:

```
ore.scriptCreate(name, FUN, global, overwrite)
```

The arguments are the following:

Argument	Description
`name`	A name for the script in the Oracle R Enterprise R script repository.
`fun`	An R function.
`global`	A logical value that indicates whether the script is public (global) or private. `FALSE` (the default) specifies that the script is not public and is visible only to the owner or to users to whom the owner has granted read privilege access; `TRUE` specifies that the script is public and therefore visible to all users.
`overwrite`	A logical value that indicates whether to replace the R function of the script with the function specified in by the `fun` argument. `TRUE` specifies replacing the function, if it exists; `FALSE` (the default) specifies that the existing contents cannot be replaced.

If `overwrite = FALSE`, an error condition occurs if a script by the same name already exists in the R script repository; otherwise, `ore.scriptCreate` returns `NULL`.

Granting or Revoking Read Access to a Script

The creator of a script can use the `ore.grant` function to grant read access privilege to the script and the `ore.revoke` function to revoke that access. Those functions have the following syntax:

```
ore.grant(name, type = "rqscript", user)
ore.revoke(name, type = "rqscript", user)
```

The arguments are the following:

Argument	Description
`name`	The name of a script in the Oracle R Enterprise R script repository.
`type`	For a script, the type is rqscript.
`user`	The user to whom to grant or revoke read privilege access.

The `name` and `type` arguments are required. If argument `user` is not specified, then read privilege access is granted to or revoked from all users.

An error occurs when one of the following is true:

- The named script is not in the R script repository.
- The `type` argument is not specified.
- The user is not found.

- The read privilege has already been granted to or revoked from the user.
- The named script is public.

Listing the Available Scripts

To list the scripts available to you, use `ore.scriptList`. You can list scripts by name, by a pattern, or by type. If you have the RQADMIN role, you can list system scripts, as well. The function has the following syntax:

```
ore.scriptList(name, pattern, type)
```

The arguments are the following:

Argument	Description
`name`	The name of a script in the Oracle R Enterprise R script repository. Cannot be used when argument `pattern` is specified.
`pattern`	A regular expression pattern. Scripts that match the pattern are listed. Cannot be used when argument `name` is specified.
`type`	The type of the script, which can be one of the following: • `user`, which lists scripts owned by the current user • `global`, which lists public scripts, which are visible to all users • `grant`, which lists the scripts to which the current user has granted read access to others • `granted`, which lists the scripts to which the current user has been granted read access by another user • `all`, which lists all of the user, public, and granted scripts

The `ore.scriptList` function returns a `data.frame` that contains the names of the scripts in the R script repository and the function in the script.

Loading a Script into an R Environment

To load the R function of a script into an R environment, use `ore.scriptLoad`, which has the following syntax:

```
ore.scriptLoad(name, owner, newname, envir)
```

The arguments are the following:

Argument	Description
`name`	The name of a script in the Oracle R Enterprise R script repository.
`owner`	The owner of the script.
`newname`	A new function name in which to load the script.
`envir`	The R environment in which to load the script.

Specifying the owner of a script is useful when access to the script has been granted to the user who is invoking `ore.scriptLoad`.

Specifying a new function name is useful when the name of the script in the Oracle R Enterprise R script repository is not a valid R function name.

An error occurs when one of the following is true:

- The script is not in the R script repository.
- The current user does not have read access to the script.
- The function specified by the `name` argument is not a valid R function name.
- The `newname` argument is not a valid R function name.

Dropping a Script

To remove a script from the Oracle R Enterprise R script repository, use the `ore.scriptDrop` function. To invoke this function, you must have the RQADMIN role. The `ore.scriptDrop` function has the following syntax:

```
ore.scriptDrop(name, global, silent)
```

The arguments are the following:

Argument	Description
`name`	A name for the script in the Oracle R Enterprise R script repository.
`global`	A logical value that indicates whether the script is global (public) or private. `TRUE` specifies dropping a global script; `FALSE` (the default) specifies dropping a script owned by the current user.
`silent`	A logical value that indicates whether to display an error message if `ore.scriptDrop` encounters an error condition. `TRUE` specifies the display of error messages; `FALSE` (the default) specifies no display.

An error condition occurs when one of the following is true:

- The script is not in the R script repository.
- If `global = TRUE`, the script is a private script.
- If `global = FALSE`, the script is a public script.

If successful, `ore.scriptDrop` returns `NULL`.

Example 6-5 Using the R Script Management Functions

```
# Create an ore.frame object from the data.frame for the iris data set.
IRIS <- ore.push(iris)

# Create a private R script for the current user.
ore.scriptCreate("myRandomRedDots", function(divisor = 100){
                 id <- 1:10
                 plot(1:100, rnorm(100), pch = 21, bg = "red", cex = 2 )
                 data.frame(id = id, val = id / divisor)
                 })

# Create another private R script.
ore.scriptCreate("MYLM",
                 function(data, formula, ...) lm(formula, data, ...))

# Create a public script, available to any user.
ore.scriptCreate("GLBGLM",
                 function(data, formula, ...)
                 glm(formula = formula, data = data, ...),
                 global = TRUE)
```

```
# List only my private scripts.
ore.scriptList()

# List my private scripts and the public scripts.
ore.scriptList(type = "all")

# List my private scripts that have the specified pattern.
ore.scriptList(pattern = "MY")

# Grant read access to a private script to all users.
ore.grant("MYLM", type = "rqscript")

# Grant read access to a private script to a specific user.
ore.grant("myRandomRedDots", user = "SCOTT",  type = "rqscript")

# List the granted scripts.
ore.scriptList(type = "grant")

# Use the MYLM script in an embedded R execution function.
ore.tableApply(IRIS[1:4], FUN.NAME = "MYLM",
               formula = Sepal.Length ~ .)
# Use the GLBGLM script in an embedded R execution function.
ore.tableApply(IRIS[1:4], FUN.NAME = "GLBGLM",
               formula = Sepal.Length ~ .)

# Load an R script to an R function object
ore.scriptLoad(name = "MYLM")

# Invoke the function.
MYLM(iris, formula = Sepal.Length ~ .)

# Load another R script to an R function object
ore.scriptLoad(name = "GLBGLM", newname = "MYGLM")

# Invoke the function.
MYGLM(iris, formula = Sepal.Length ~ .)

# Drop some scripts.
ore.scriptDrop("MYLM")
ore.scriptDrop("GLBGLM", global = TRUE)

# List all scripts.
ore.scriptList(type = "all")
```

Listing for Example 6-5 (page 6-16)

```
R> # Create an ore.frame object from the data.frame for the iris data set.
R> IRIS <- ore.push(iris)
R>
R> # Create a private R script for the current user.
R> ore.scriptCreate("myRandomRedDots", 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>
R> # Create another private R script.
R> ore.scriptCreate("MYLM",
+                   function(data, formula, ...) lm(formula, data, ...))
R>
```

```
R> # Create a public script, available to any user.
R> ore.scriptCreate("GLBGLM",
+                  function(data, formula, ...)
+                  glm(formula = formula, data = data, ...),
+                  global = TRUE)
R>
R> # List only my private scripts.
R> ore.scriptList()
             NAME      SCRIPT
1            MYLM      function (data, formula, ...) \nlm(formula, data, ...)
2 myRandomRedDots      function (divisor = 100) \n{\n    id & lt\n    -1:10\n
                       plot(1:100, rnorm(100), pch = 21, bg = "red", cex = 2)\n
                       data.frame(id = id, val = id/divisor)\n}
R>
R> # List my private scripts and the public scripts.
R> ore.scriptList(type = "all")
   OWNER            NAME      SCRIPT
1  RQSYS          GLBGLM      function (data, formula, ...) \nglm(formula = formula,
data = data, ...)
2 RQUSER            MYLM      function (data, formula, ...) \nlm(formula, data, ...)
3 RQUSER myRandomRedDots      function (divisor = 100) \n{\n    id & lt\n    -1:10\n
                              plot(1:100, rnorm(100), pch = 21, bg = "red", cex =
2)\n
                              data.frame(id = id, val = id/divisor)\n}
R>
R> # List my private scripts that have the specified pattern.
R> ore.scriptList(pattern = "MY")
  NAME  SCRIPT
1 MYLM  function (data, formula, ...) \nlm(formula, data, ...)
R>
R> # Grant read access to a private script to all users.
R> ore.grant("MYLM", type = "rqscript")
R>
R> # Grant read access to a private script to a specific user.
R> ore.grant("myRandomRedDots", user = "SCOTT",  type = "rqscript")
R>
R> # List the granted scripts.
R> ore.scriptList(type = "grant")
             NAME GRANTEE
1            MYLM  PUBLIC
2 myRandomRedDots   SCOTT
R>
R> # Use the MYLM script in an embedded R execution function.
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>
R> # Use the GLBGLM script in an embedded R execution function.
R> ore.tableApply(IRIS[1:4], FUN.NAME = "GLBGLM",
+                formula = Sepal.Length ~ .)

Call:  glm(formula = formula, data = data)

Coefficients:
 (Intercept)   Sepal.Width  Petal.Length   Petal.Width
```

```
      1.8560        0.6508        0.7091       -0.5565

Degrees of Freedom: 149 Total (i.e. Null);  146 Residual
Null Deviance:      102.2
Residual Deviance: 14.45        AIC: 84.64
R>
R> # Load an R script to an R function object
R> ore.scriptLoad(name="MYLM")
R>
R> # Invoke the function.
R> MYLM(iris, formula = Sepal.Length ~ .)
R>
R> # Load another R script to an R function object
R> ore.scriptLoad(name = "GLBGLM", newname = "MYGLM")
R>
R> # Invoke the function.
R> MYGLM(iris, formula = Sepal.Length ~ .)
R>
R> # Drop some scripts.
R> ore.scriptDrop("MYLM")
R> ore.scriptDrop("GLBGLM", global = TRUE)
R>
R> # List all scripts.
R> ore.scriptList(type = "all")
   OWNER             NAME    SCRIPT
  RQUSER  myRandomRedDots    function (divisor = 100) \n{\n    id & lt\n    -1:10\n
                             plot(1:100, rnorm(100), pch = 21, bg = "red", cex =
                             2)\n data.frame(id = id, val = id/divisor)\n}
```

See Also:

- "Using the ore.doEval Function (page 6-19)" for examples that use the `myRandomRedDots` script
- Example 6-14 (page 6-30) for another example of using `ore.scriptCreate` and `ore.scriptDrop`
- "Manage Scripts in SQL (page 6-46)"

6.2.3 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.

The syntax of the `ore.doEval` function is the following:

```
ore.doEval(FUN, ..., FUN.VALUE = NULL, FUN.NAME = NULL, FUN.OWNER = NULL)
```

See Also:

"Arguments for Functions that Run Scripts (page 6-9)" for descriptions of the arguments to function `ore.doEval`

Example 6-6 Using the ore.doEval Function

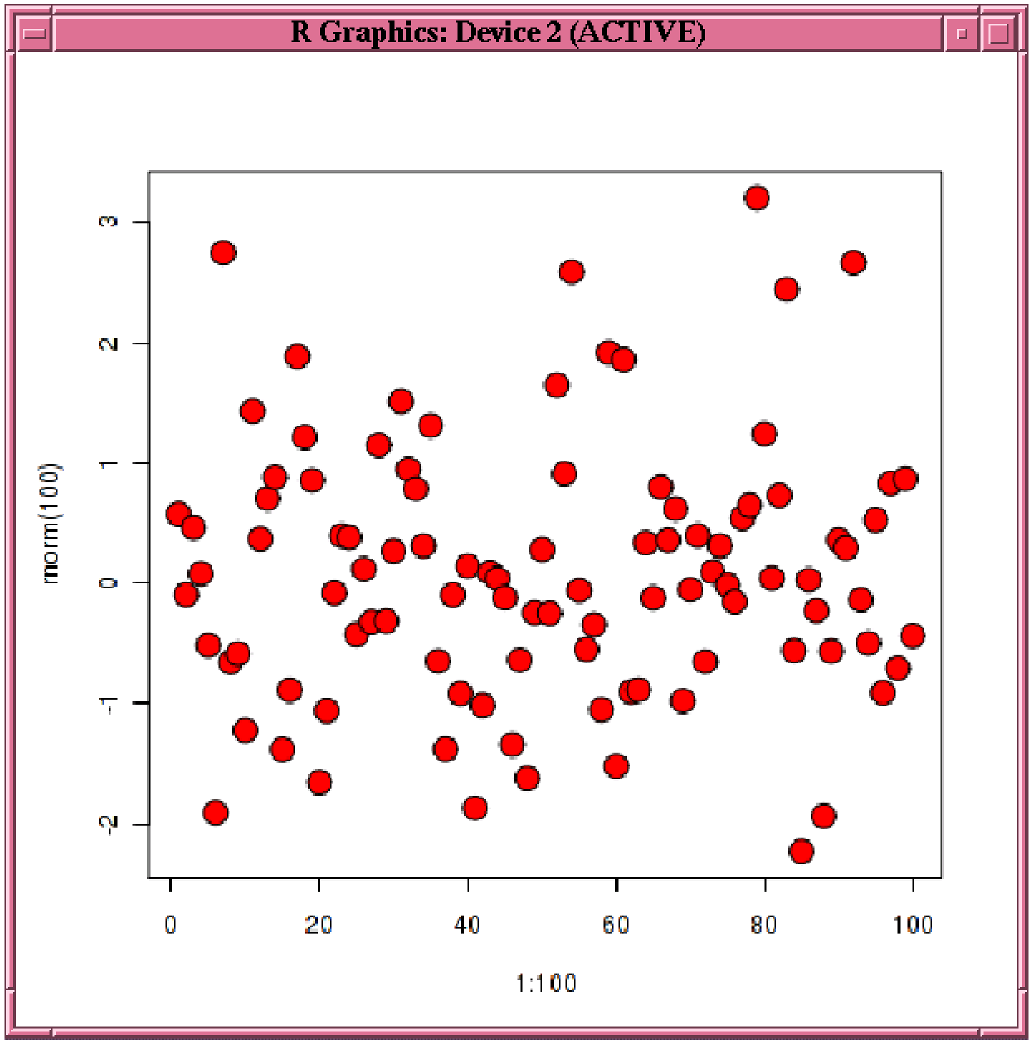
In this example, `RandomRedDots` gets a function that has an argument and that returns a `data.frame` object that has two columns and that plots 100 random normal values. The example then invokes `ore.doEval` function and passes it the `RandomRedDots function` object. The image is displayed at the client, but it is generated by the database server R engine that executed the `RandomRedDots` 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 6-6 (page 6-20)

```
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



Example 6-7 Using the ore.doEval Function with an Optional Argument

You can provide arguments to the input function as optional arguments to the `doEval` function. This example invokes the `doEval` function with an optional argument that overrides the `divisor` argument of the `RandomRedDots` function.

```
ore.doEval(RandomRedDots, divisor = 50)
```

Listing for Example 6-7 (page 6-21)

```
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.
```

Example 6-8 Using the ore.doEval Function with the FUN.NAME Argument

If the input function is stored in the Oracle R Enterprise R script repository, then you can invoke the `ore.doEval` function with the `FUN.NAME` argument. This example first invokes `ore.scriptDrop` to ensure that the R script repository does not contain a script with the name `myRandomRedDots`. The example adds the `RandomRedDots` function from Example 6-6 (page 6-20) to the repository under the name `myRandomRedDots`. This example invokes the `ore.doEval` function and specifies `myRandomRedDots`. The result is assigned to the variable `res`.

The return value of the `RandomRedDots` function is a `data.frame` but in this example 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 session.

```
ore.scriptDrop("myRandomRedDots")
ore.scriptCreate("myRandomRedDots", RandomRedDots)
res <- ore.doEval(FUN.NAME = "myRandomRedDots", divisor = 50)
class(res)
res.local <- ore.pull(res)
class(res.local)
```

Listing for Example 6-8 (page 6-22)

```
R> ore.scriptDrop("myRandomRedDots")
R> ore.scriptCreate("myRandomRedDots", RandomRedDots)
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"
```

Example 6-9 Using the ore.doEval Function with the FUN.VALUE Argument

To have the `doEval` function return an `ore.frame` object instead of an `ore.object`, use the argument `FUN.VALUE` to specify the structure of the result, as shown in this example.

```
res.of <- ore.doEval(FUN.NAME="myRandomRedDots", divisor = 50,
                     FUN.VALUE= data.frame(id = 1, val = 1))
class(res.of)
```

Listing for Example 6-9 (page 6-22)

```
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"
```

Example 6-10 Using the doEval Function with the ore.connect Argument

This example demonstrates using the special optional argument `ore.connect` to connect to the database in the embedded R function, which enables the use of objects stored in a datastore. The example creates the `RandomRedDots2` function object, which is the same as the `RandomRedDots` function from Example 6-6 (page 6-20) except the `RandomRedDots2` 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 passes the `ore.connect` control argument set to `TRUE`.

```
RandomRedDots2 <- 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 the numeric variable myVar.
  data.frame(id = id, val = id / divisor, num = myVar)
}
myVar <- 5
ore.save(myVar, name = "datastore_1")
ore.doEval(RandomRedDots2, datastore.name = "datastore_1", ore.connect = TRUE)
```

Listing for Example 6-10 (page 6-22)

```
R> RandomRedDots2 <- 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 the numeric variable myVar.
+   data.frame(id = id, val = id / divisor, num = myVar)
+ }
R> ore.doEval(RandomRedDots2, 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.
```

Example 6-11 Using the ora.type Attribute

This example demonstrates using the `ora.type` attribute to specify database data types of CLOB and BLOB for columns in the `data.frame` object specified by the `FUN.VALUE` argument.

```
eval1 <- ore.doEval(function() "Hello, world")
eval2 <-
  ore.doEval(function()
             data.frame(x = "Hello, world", stringsAsFactors = FALSE))
eval3 <-
  ore.doEval(function()
             data.frame(x = "Hello, world", stringsAsFactors = FALSE),
             FUN.VALUE =
             data.frame(x = character(), stringsAsFactors = FALSE))
out.df <- data.frame(x = character(), y = raw(), stringsAsFactors = FALSE)
attr(out.df$x, "ora.type") <- "clob"
attr(out.df$y, "ora.type") <- "blob"
eval4 <-
  ore.doEval(function() {
             res <- data.frame(x = "Hello, world",stringsAsFactors = FALSE)
             res$y[[1L]] <- charToRaw("Hello, world")
             res},
             FUN.VALUE = out.df)
eval1
class(eval1)  # ore.object
```

```
eval2
class(eval2)  # ore.object
eval3
class(eval3)  # ore.frame
eval4$x
rawToChar(ore.pull(eval4$y))
```

Listing for Example 6-11 (page 6-23)

```
R> eval1 <- ore.doEval(function() "Hello, world")
R> eval2 <-
+    ore.doEval(function()
+               data.frame(x = "Hello, world", stringsAsFactors = FALSE))
R> eval3 <-
+    ore.doEval(function()
+               data.frame(x = "Hello, world", stringsAsFactors = FALSE),
+               FUN.VALUE =
+               data.frame(x = character(), stringsAsFactors = FALSE))
R> out.df <- data.frame(x = character(), y = raw(), stringsAsFactors = FALSE)
R> attr(out.df$x, "ora.type") <- "clob"
R> attr(out.df$y, "ora.type") <- "blob"
R> eval4 <-
+    ore.doEval(function() {
+               res <- data.frame(x = "Hello, world",stringsAsFactors = FALSE)
+               res$y[[1L]] <- charToRaw("Hello, world")
+               res},
+               FUN.VALUE = out.df)
R> eval1
[1] "Hello, world"
R> class(eval1)
[1] "ore.object"
attr(,"package")
[1] "OREembed"
R> eval2
             x
1 Hello, world
R> class(eval2)
[1] "ore.object"
attr(,"package")
[1] "OREembed"
R> eval3
             x
1 Hello, world
Warning message:
ORE object has no unique key - using random order
R> class(eval3)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> eval4$x
[1] "Hello, world"
Warning message:
ORE object has no unique key - using random order
R> rawToChar(ore.pull(eval4$y))
[1] "Hello, world"
Warning message:
ORE object has no unique key - using random order
```

6.2.4 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.

The syntax of the `ore.tableApply` function is the following:

```
ore.tableApply(X, FUN, ..., FUN.VALUE = NULL, FUN.NAME = NULL, FUN.OWNER = NULL)
```

See Also:

- "Arguments for Functions that Run Scripts (page 6-9)" for descriptions of the arguments to function `ore.tableApply`
- "Installing a Third-Party Package for Use in Embedded R Execution (page 6-5)"

Example 6-12 Using the ore.tableApply Function

This example 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` table proxy 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.

```
library(e1071)
nbmod <- ore.tableApply(
  ore.push(iris),
  function(dat) {
    library(e1071)
    dat$Species <- as.factor(dat$Species)
    naiveBayes(Species ~ ., dat)
})
class(nbmod)
nbmod
```

Listing for Example 6-12 (page 6-25)

```
R> nbmod <- ore.tableApply(
+   ore.push(iris),
+   function(dat) {
+     library(e1071)
+     dat$Species <- as.factor(dat$Species)
+     naiveBayes(Species ~ ., dat)
+ })
R> class(nbmod)
[1] "ore.object"
attr(,"package")
[1] "OREembed"
R> nbmod

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
```

6.2.5 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 syntax of the `ore.groupApply` function is the following:

```
ore.groupApply(X, INDEX, FUN, ..., FUN.VALUE = NULL, FUN.NAME = NULL, FUN.OWNER =
NULL,
               parallel = getOption("ore.parallel", NULL))
```

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 (page 6-27)"
- "Partitioning on Multiple Columns (page 6-29)"

See Also:

- "Arguments for Functions that Run Scripts (page 6-9)" for descriptions of the arguments to function `ore.groupApply`
- "Installing a Third-Party Package for Use in Embedded R Execution (page 6-5)"

6.2.5.1 Partitioning on a Single Column

Example 6-13 (page 6-27) uses the `C50` package, which has functions that build decision tree and rule-based models. The package also provides training and testing data sets. Example 6-13 (page 6-27) builds C5.0 models on the `churnTrain` training data set from the `churn` data set of the `C50` package, 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 the `CHURN_TRAIN` database table and its proxy `ore.frame` object from `churnTrain`, a `data.frame` object.
- Specifies `CHURN_TRAIN`, the proxy `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.
- 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, with each model stored as an `ore.object` object.
- 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.

  – Converts 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.
- 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 6-13 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 6-13 (page 6-27)

```
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
```

6.2.5.2 Partitioning on Multiple Columns

The `ore.groupApply` function takes a single column or multiple columns as the `INDEX` argument.

Example 6-14 (page 6-30) 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.

The example next invokes the `ore.scriptDrop` function to ensure that no script by the specified name exists in the Oracle Database 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 by the values in the voice_mail_plan and international_plan columns. Because the source partition columns are constants, the function sets them to `NULL`. It converts the character vectors to factors, builds a model to predict churn, and saves it in 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 values from the voice_mail_plan and international_plan columns as the `INDEX` argument and `my_rpartFunction` as the value of the `FUN.NAME` argument 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 6-14 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)

options(width = 80)
head(CHURN_TRAIN, 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(CHURN_TRAIN,
      INDEX = CHURN_TRAIN[, c("voice_mail_plan", "international_plan")],
      FUN.NAME = "my_rpartFunction",
      datastorePrefix = datastorePrefix,
      ore.connect = TRUE)
res[[1]]
ore.load(name=paste(datastorePrefix, "yes", "yes", sep = "_"))
mod
```

Listing for Example 6-14 (page 6-30)

```
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>
R> options(width = 80)
R> head(CHURN_TRAIN, 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
1              2.70                             1    no
2              3.70                             1    no
3              3.29                             0    no
Warning messages:
1: ORE object has no unique key - using random order
2: ORE object has no unique key - using random order
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(CHURN_TRAIN,
+       INDEX = CHURN_TRAIN[, c("voice_mail_plan", "international_plan")],
+       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,WY
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,SC,S
D,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) *
```

6.2.6 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 syntax of the `ore.rowApply` function is the following:

```
ore.rowApply(X, FUN, ..., FUN.VALUE = NULL, FUN.NAME = NULL, rows = 1, FUN.OWNER =
NULL,
           parallel = getOption("ore.parallel", NULL))
```

The `ore.rowApply` function returns an `ore.list` object or an `ore.frame` object.

See Also:

- "Arguments for Functions that Run Scripts (page 6-9)" for descriptions of the arguments to function `ore.rowApply`

Example 6-15 (page 6-34) uses the `e1071` package, previously downloaded from CRAN. The example also uses the `nbmod` object, which is the Naive Bayes model created in Example 6-12 (page 6-25).

Example 6-15 (page 6-34) does the following:

- Loads the package `e1071`.
- Pushes the `iris` data set to the database as the `IRIS` temporary table and `ore.frame` object.
- 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 the user-defined R function itself.
- The user-defined function does the following:
  - Loads the package `e1071` so that it is available to the R engine or engines that run 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 6-15 Using the ore.rowApply Function

```
library(e1071)
IRIS <- ore.push(iris)
IRIS_PRED <- IRIS
IRIS_PRED$PRED <- "A"
res <- ore.rowApply(
  IRIS,
  function(dat, nbmod) {
    library(e1071)
    dat$Species <- as.factor(dat$Species)
    dat$PRED <- predict(nbmod, newdata = dat)
    dat
  },
  nbmod = ore.pull(nbmod),
  FUN.VALUE = IRIS_PRED,
  rows = 10)
class(res)
table(res$Species, res$PRED)
```

Listing for Example 6-15 (page 6-34)

```
R> library(e1071)
R> IRIS <- ore.push(iris)
R> IRIS_PRED <- IRIS
R> IRIS_PRED$PRED <- "A"
R> res <- ore.rowApply(
+   IRIS ,
+   function(dat, nbmod) {
+     library(e1071)
+     dat$Species <- as.factor(dat$Species)
+     dat$PRED <- predict(nbmod, newdata = dat)
+     dat
+   },
+   nbmod = ore.pull(nbmod),
+   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
```

As Example 6-13 (page 6-27) does, Example 6-16 (page 6-36) uses the `C50` package to score churn data (that is, to predict which customers are likely to churn) using C5.0 models. However, instead of partitioning the data by a column, Example 6-16 (page 6-36) partitions the data by a number of rows. The example scores the customers from the specified state in parallel. The example uses datastores and saves functions to the Oracle R Enterprise R script repository, which allows the functions to be used by the Oracle R Enterprise SQL API functions.

Example 6-16 (page 6-36) first loads `C50` package and the data sets. The example deletes the datastores with names containing `myC5.0modelFL`, if they exist. It invokes `ore.drop` to delete the CHURN_TEST table, if it exists, and then invokes `ore.create` to create the CHURN_TEST table from the `churnTest` data set.

The example next invokes `ore.getLevels`, which returns a `list` of the levels for each factor column. The invocation excludes the first column, which is state, because the levels for that column are not needed. Getting the levels first can ensure that all possible levels are provided during model building, even if some rows do not have values for some of the levels. The `ore.delete` invocation ensures that no datastore with the specified name exists and the `ore.save` invocation saves the `xlevels` object in the datastore named `myXLevels`.

Example 6-16 (page 6-36) creates a user-defined function, `myC5.0FunctionForLevels`, that generates a C5.0 model. The function uses the list of levels returned by function `ore.getXlevels` instead of computing the levels using the `as.factor` function as the user-defined function does in Example 6-13 (page 6-27). It uses the levels to convert the column type from character vector to factor. The function `myC5.0FunctionForLevels` returns the value `TRUE.` The example saves the function in the Oracle R Enterprise R script repository.

The example next gets a list of datastores that have names that include the specified string and deletes those datastores if they exist.

The example then invokes `ore.groupApply`, which invokes function `myC5.0FunctionForLevels` on each state in the `CHURN_TEST` data. To each `myC5.0FunctionForLevels` invocation, `ore.groupApply` passes the datastore that contains the `xlevels` object and a prefix to use in naming the datastore generated by `myC5.0FunctionForLevels`. It also passes the `ore.connect` control argument to connect to the database in the embedded R function, which enables the use of objects stored in a datastore. The `ore.groupApply` invocation returns a list that contains the results of all of the invocations of `myC5.0FunctionForLevels`.

The example pulls the result over to the local R session and verifies that `myC5.0FunctionForLevels` returned `TRUE` for each state in the data source.

Example 6-16 (page 6-36) next creates another user-defined another function, `myScoringFunction`, and stores it in the Oracle R Enterprise R script repository. The function scores a C5.0 model for the levels of a state and returns the results in a `data.frame`.

The example then invokes function `ore.rowApply`. It filters the input data to use only data for the state of Massachusetts. It specifies `myScoringFunction` as the function to invoke and passes that user-defined function the name of the datastore that contains the `xlevels` object and a prefix to use in loading the datastore that contains the C5.0 model for the state. The `ore.rowApply` invocation specifies invoking `myScoringFunction` on 200 rows of the data set in each parallel R engine. It uses the `FUN.VALUE` argument so that `ore.rowApply` returns an `ore.frame` that contains the results of all of the `myScoringFunction` invocations. The variable `scores` gets the results of the `ore.rowApply` invocation.

Finally, Example 6-16 (page 6-36) prints the `scores` object and then uses the table function to display the confusion matrix for the scoring.

See Also:

Example A-8 (page A-11) for an invocation of the `rqRowEval` function that produces the same result as the `ore.rowApply` function in Example 6-16 (page 6-36)

Example 6-16 Using the ore.rowApply Function with Datastores and Scripts

```
library(C50)
data(churn)

ore.drop("CHURN_TEST"
ore.create(churnTest, "CHURN_TEST")

xlevels <- ore.getXlevels(~ ., CHURN_TEST[,-1])
ore.delete("myXLevels")
ore.save(xlevels, name = "myXLevels")

ore.scriptDrop("myC5.0FunctionForLevels")
ore.scriptCreate("myC5.0FunctionForLevels",
  function(dat, xlevelsDatastore, datastorePrefix) {
    library(C50)
    state <- dat[1,"state"]
    datastoreName <- paste(datastorePrefix, dat[1, "state"], sep = "_")
    dat$state <- NULL
    ore.load(name = xlevelsDatastore)
    for (j in names(xlevels))
       dat[[j]] <- factor(dat[[j]], levels = xlevels[[j]])
    c5mod <- C5.0(churn ~ ., data = dat, rules = TRUE)
    ore.save(c5mod, name = datastoreName)
    TRUE
  })

ds.v <- ore.datastore(pattern= "myC5.0modelFL")$datastore.name
for (ds in ds.v) ore.delete(name = ds)

res <- ore.groupApply(CHURN_TEST,
          INDEX=CHURN_TEST$state,
          FUN.NAME = "myC5.0FunctionForLevels",
          xlevelsDatastore = "myXLevels",
          datastorePrefix = "myC5.0modelFL",
          ore.connect = TRUE)
res <- ore.pull(res)
all(as.logical(res) == TRUE)

ore.scriptDrop("myScoringFunction")
ore.scriptCreate("myScoringFunction",
                 function(dat, xlevelsDatastore, datastorePrefix) {
                    library(C50)
                    state <- dat[1,"state"]
                    datastoreName <- paste(datastorePrefix,state,sep="_")
                    dat$state <- NULL
                    ore.load(name = xlevelsDatastore)
                    for (j in names(xlevels))
                       dat[[j]] <- factor(dat[[j]], levels = xlevels[[j]])
                    ore.load(name = datastoreName)
                    res <- data.frame(pred = predict(c5mod, dat, type = "class"),
                                         actual = dat$churn,
                                         state = state)
                    res
```

```
                }
              )

scores <- ore.rowApply(
  CHURN_TEST[CHURN_TEST$state == "MA",],
  FUN.NAME = "myScoringFunction",
  xlevelsDatastore = "myXLevels",
  datastorePrefix = "myC5.0modelFL",
  ore.connect = TRUE, parallel = TRUE,
  FUN.VALUE = data.frame(pred = character(0),
                         actual = character(0),
                         state = character(0)),
  rows=200)
scores
table(scores$actual, scores$pred)
```

Listing for Example 6-16 (page 6-36)

```
R> library(C50)
R> data(churn)
R>
R> ore.drop("CHURN_TEST"
R> ore.create(churnTest, "CHURN_TEST")
R>
R> xlevels <- ore.getXlevels(~ ., CHURN_TEST[,-1])
R> ore.delete("myXLevels")
[1] "myXLevels
R> ore.save(xlevels, name = "myXLevels")
R>
R> ore.scriptDrop("myC5.0FunctionForLevels")
R> ore.scriptCreate("myC5.0FunctionForLevels",
+    function(dat, xlevelsDatastore, datastorePrefix) {
+      library(C50)
+      state <- dat[1,"state"]
+      datastoreName <- paste(datastorePrefix, dat[1, "state"], sep = "_")
+      dat$state <- NULL
+      ore.load(name = xlevelsDatastore)
+      for (j in names(xlevels))
+        dat[[j]] <- factor(dat[[j]], levels = xlevels[[j]])
+      c5mod <- C5.0(churn ~ ., data = dat, rules = TRUE)
+      ore.save(c5mod, name = datastoreName)
+      TRUE
+    })
R>
R> ds.v <- ore.datastore(pattern="myC5.0modelFL")$datastore.name
R> for (ds in ds.v) ore.delete(name=ds)
R>
R> res <- ore.groupApply(CHURN_TEST,
+                      INDEX=CHURN_TEST$state,
+                      FUN.NAME="myC5.0FunctionForLevels",
+                      xlevelsDatastore = "myXLevels",
+                      datastorePrefix = "myC5.0modelFL",
+                      ore.connect = TRUE)
R> res <- ore.pull(res)
R> all(as.logical(res) == TRUE)
[1] TRUE
R>
R> ore.scriptDrop("myScoringFunction")
R> ore.scriptCreate("myScoringFunction",
+                   function(dat, xlevelsDatastore, datastorePrefix) {
+                     library(C50)
```

```
+                       state <- dat[1,"state"]
+                       datastoreName <- paste(datastorePrefix,state,sep="_")
+                       dat$state <- NULL
+                       ore.load(name = xlevelsDatastore)
+                       for (j in names(xlevels))
+                           dat[[j]] <- factor(dat[[j]], levels = xlevels[[j]])
+                       ore.load(name = datastoreName)
+                       res <- data.frame(pred = predict(c5mod, dat, type="class"),
+                                         actual = dat$churn,
+                                         state = state)
+                       res
+                   }
+ )
R>
R> scores <- ore.rowApply(
+     CHURN_TEST[CHURN_TEST$state =="MA",],
+     FUN.NAME = "myScoringFunction",
+     xlevelsDatastore = "myXLevels",
+     datastorePrefix = "myC5.0modelFL",
+     ore.connect = TRUE, parallel = TRUE,
+     FUN.VALUE = data.frame(pred=character(0),
+                            actual=character(0),
+                            state=character(0)),
+     rows=200
R>
R> scores
   pred actual state
1    no     no    MA
2    no     no    MA
3    no     no    MA
4    no     no    MA
5    no     no    MA
6    no    yes    MA
7   yes    yes    MA
8   yes    yes    MA
9    no     no    MA
10   no     no    MA
11   no     no    MA
12   no     no    MA
13   no     no    MA
14   no     no    MA
15  yes    yes    MA
16   no     no    MA
17   no     no    MA
18   no     no    MA
19   no     no    MA
20   no     no    MA
21   no     no    MA
22   no     no    MA
23   no     no    MA
24   no     no    MA
25   no     no    MA
26   no     no    MA
27   no     no    MA
28   no     no    MA
29   no    yes    MA
30   no     no    MA
31   no     no    MA
32   no     no    MA
33  yes    yes    MA
34   no     no    MA
```

```
35   no      no     MA
36   no      no     MA
37   no      no     MA
38   no      no     MA
Warning message:
ORE object has no unique key - using random order
R> table(scores$actual, scores$pred)

      no yes
  no  32   0
  yes  2   4
```

6.2.7 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 syntax of the `ore.indexApply` function is the following:

```
ore.indexApply(times, FUN, ..., FUN.VALUE = NULL, FUN.NAME = NULL, FUN.OWNER = NULL,
               parallel = getOption("ore.parallel", NULL))
```

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 (page 6-39)"
- "Column-Parallel Use Case (page 6-40)"
- "Simulations Use Case (page 6-41)"

See Also:

- "Arguments for Functions that Run Scripts (page 6-9)" for descriptions of the arguments to function `ore.indexApply`

6.2.7.1 Simple Example of Using the ore.indexApply Function

Example 6-17 (page 6-39) 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 6-17 Using the ore.indexApply Function

```
res <- ore.indexApply(5,
      function(index) {
        paste("IndexApply:", index)
      },
      parallel = TRUE)
class(res)
res
```

Listing for Example 6-17 (page 6-39)

```
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"
```

6.2.7.2 Column-Parallel Use Case

Example 6-18 (page 6-40) 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 6-18 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 6-18 (page 6-40)

```
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
```

6.2.7.3 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 6-19 (page 6-42) 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 6-19 (page 6-42) 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 function 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 6-19 Using the ore.indexApply Function in a Simulation

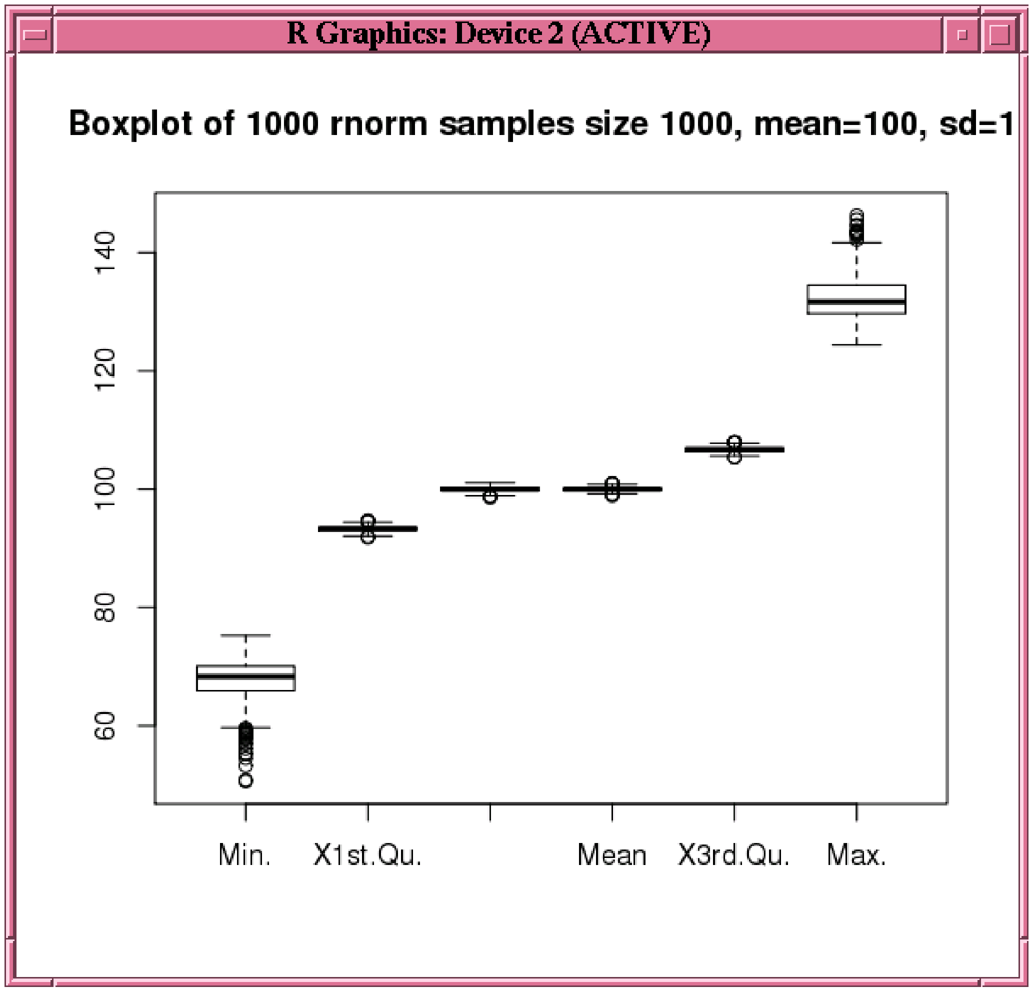
```
res <- NULL
sample.size = 1000
mean.val = 100
std.dev.val = 10
num.simulations = 1000

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 6-19 (page 6-42)

```
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 6-19 *(page 6-42)



6.3 SQL Interface for Embedded R Execution

The SQL interface for Oracle R Enterprise embedded R execution allows you to execute R functions in production database applications. The SQL interface has procedures for the following actions:

- Adding and removing a script from the Oracle R Enterprise R script repository
- Granting or revoking read privilege access to a script by the owner to other users
- Executing an R script in an embedded R session

- Deleting an Oracle R Enterprise datastore

Data dictionary views provide information about scripts and datastores.

This SQL interface is described in the following topics:

- About Oracle R Enterprise SQL Table Functions (page 6-44)
- Manage Scripts in SQL (page 6-46)
- Manage Datastores in SQL (page 6-48)
- rqEval Function (page A-2)
- rqGroupEval Function (page A-5)
- rqRowEval Function (page A-9)
- rqTableEval Function (page A-13)

6.3.1 About Oracle R Enterprise SQL Table Functions

Oracle R Enterprise provides SQL table functions that are equivalents of most of the R interface functions for embedded R execution. Executing a `SELECT FROM TABLE` statement and specifying one of the table functions results in the invocation of the specified R script. The script runs in one or more R engines on the Oracle Database server.

The SQL table functions for embedded R execution are:

- `rqEval`
- *`rqGroupEval`*
- `rqRowEval`
- `rqTableEval`

The R interface functions and the SQL equivalents are listed in Table 6-1 (page 6-2).

For the *`rqGroupEval`* function, Oracle R Enterprise provides a generic implementation of the group apply functionality in SQL. You must write a table function that captures the structure of the input cursor.

Some general aspects of the SQL table functions are described in the following topics:

- "Parameters of the SQL Table Functions (page 6-45)"
- "Return Value of SQL Table Functions (page 6-45)"
- "Connecting to Oracle R Enterprise in Embedded R Execution (page 6-46)"

See the reference pages for the functions for more information about them, including examples of their use.

Related Topics:

rqEval Function (page A-2)

rqGroupEval Function (page A-5)

rqRowEval Function (page A-9)

rqTableEval Function (page A-13)

6.3.1.1 Parameters of the SQL Table Functions

The SQL table functions have some parameters in common and some functions have parameters that are unique to that function. The parameters of the SQL table functions are the following.

Table 6-2 SQL Table Function Parameters

Parameter	Description
`INP_CUR`	A cursor that specifies the data that is input to the R function specified by `EXP_NAM`. For all of the SQL table functions except `rqEval`, the first argument is a cursor that specifies input data for the R function.
`PAR_CUR`	A cursor that specifies arguments to pass to the R function. The parameters cursor consists of a single row of scalar values. An argument can be a string or a numeric value. You can specify multiple arguments in the cursor. Arguments to an R function are case sensitive, so you should put names, such as a column name, in double quotes. In the cursor, you can also specify as scalar values an Oracle R Enterprise control argument or the names of serialized R objects, such as predictive models, that are in an Oracle R Enterprise datastore. The value of this parameters cursor can be `NULL` if you are not passing any arguments to the R function or any control arguments.
`OUT_QRY`	An output table definition. The value of this argument can be `NULL` or a string that defines the structure of the R `data.frame` returned by the R function specified by `EXP_NAM`. The string can be a `SELECT` statement, 'XML', or 'PNG'.
`GRP_COL`	For the *`rqGroupEval`* function, the name of the grouping column.
`ROW_NUM`	For the `rqRowEval` function, the number of rows to pass to each invocation of the R function.
`EXP_NAM`	The name of a script in the Oracle R Enterprise R script repository.

Related Topics:

Manage Scripts in SQL (page 6-46)

Manage Datastores in SQL (page 6-48)

6.3.1.2 Return Value of SQL Table Functions

The Oracle R Enterprise SQL table functions return a table. The structure and contents of the table are determined by the results of the R function passed to the SQL table function and by the `OUT_QRY` parameter. The R function can return a `data.frame` object, other R objects, and graphics. The structure of the table that represents the results of the R function is specified by one of the following `OUT_QRY` values:

- `NULL`, which results in a table that has a serialized object that can contain both data and image objects.
- A table signature specified in a `SELECT` statement, which results in a table that has the defined structure. The result of the R function must be a `data.frame`. No images are returned.
- The string `'XML'`, which results in a table that has a CLOB that can contain both structured data and graph images in an XML string. The non-image R objects, such as `data.frame` or `model` objects, are provided first, followed by the base 64 encoding of a PNG representation of the image.
- The string `'PNG'`, which results in a table that has a BLOB that contains graph images in PNG format. The table has the column names `name`, `id`, and `image`.

6.3.1.3 Connecting to Oracle R Enterprise in Embedded R Execution

To establish a connection to Oracle R Enterprise on the Oracle Database server during the embedded R execution, you can specify the control argument `ore.connect` in the parameters cursor. Doing so establishes a connection using the credentials of the user who invoked the embedded R function. It also automatically loads the `ORE` package. Establishing an Oracle R Enterprise connection is required to save objects in an Oracle R Enterprise R object datastore or to load objects from a datastore. It also allows you to explicitly use the Oracle R Enterprise transparency layer.

See Also:

"Optional and Control Arguments (page 6-11)" for information on other control arguments

6.3.2 Manage Scripts in SQL

This topic lists the PL/SQL procedures and Oracle Database data dictionary views for creating and managing R scripts.

The functions in the SQL API for embedded R execution require as an argument a named script that is stored in the Oracle R Enterprise R script repository. The PL/SQL procedures `sys.rqScriptCreate` and `sys.rqScriptDrop` create and drop scripts. To create a script or drop one from the Oracle R Enterprise R script repository requires the RQADMIN role.

When using the `sys.rqScriptCreate` function, you must specify a name for the script and an R function script that contains a single R function definition. Calls to the functions `sys.rqScriptCreate` and `sys.rqScriptDrop` must be wrapped in a `BEGIN-END` PL/SQL block. The R script repository stores the R function as a character large object (a `CLOB`), so you must enclose the function definition in single quotes to specify it as a string.

The owner of a script can use the `rqGrant` procedure to grant to another user read privilege access to a script or use the `rqRevoke` procedure to revoke the privilege. To use a script granted to you by another user, you must specify the owner by prepending the owner's name and a period to the name of the script, as in the following:

```
select * from table(rqEval(NULL, 'select 1 x from dual',
'owner_name.script_name'));
```

The owner prefix is not required for a public script or for a script owned by the user.

The following tables list the PL/SQL procedures for managing Oracle R Enterprise R script repository scripts and the data dictionary views that contain information about scripts.

Table 6-3 PL/SQL Procedures for Managing Scripts

PL/SQL Procedure	Description
`rqGrant`	Grants read privilege access to a datastore or script.
`rqRevoke`	Revokes read privilege access to a datastore or script.
`sys.rqScriptCreate`	Adds the provided R function into the Oracle R Enterprise R script repository with the provided name.
`sys.rqScriptDrop`	Removes the named R function from the Oracle R Enterprise R script repository.

Table 6-4 Data Dictionary Views for Scripts

Data Dictionary View	Description
`ALL_RQ_SCRIPTS`	Describes the scripts in the Oracle R Enterprise R script repository that are available to the current user
`USER_RQ_SCRIPTS`	Describes the scripts in the Oracle R Enterprise R script repository that are owned by the current user.
`USER_RQ_SCRIPT_PRIVS`	Describes the scripts in the Oracle R Enterprise R script repository to which the current user has granted read access and the users to whom access has been granted.
`SYS.RQ_SCRIPTS`	Describes the system scripts in the Oracle R Enterprise R script repository.

Related Topics:

Creating a Script with the SQL APIs (page 6-47)

SQL APIs for Oracle R Enterprise (page A-1)

Oracle Database Views for Oracle R Enterprise (page B-1)

Manage Scripts in R (page 6-13)

6.3.2.1 Creating a Script with the SQL APIs

This example uses the `sys.rqScriptCreate` procedure to create a script in the Oracle R Enterprise R script repository.

This example creates the user-defined function named `myRandomRedDots2`. The user-defined function accepts two arguments, and it returns a `data.frame` object that has two columns and that plots the specified number of random normal values. The `sys.rqScriptCreate` function stores the user-defined function in the Oracle R Enterprise script repository.

```
-- Create a script named myRandomRedDots2 and add it to the R script repository.
-- Specify that the script is private and to overwrite a script with the same name.
```

```
BEGIN
  sys.rqScriptCreate('myRandomRedDots2',
    'function(divisor = 100, numDots = 100) {
       id <- 1:10
       plot(1:numDots, rnorm(numDots), pch = 21, bg = "red", cex = 2 )
       data.frame(id = id, val = id / divisor)}',
       v_global => FALSE,
       v_overwrite => TRUE);
END;
/

-- Grant read privilege access to Scott.
BEGIN
  rqGrant('myRandomRedDots2', 'rqscript', 'SCOTT');
END;
 /

-- View the users granted read access to myRandomRedDots2.
select * from USER_RQ_SCRIPT_PRIVS;

NAME                GRANTEE
----------------    -------
myRandomRedDots       SCOTT

-- Revoke the read privilege access from Scott.
BEGIN
  rqRevoke('myRandomRedDots2', 'rqscript', 'SCOTT');
END;
 /

-- Remove the script from the R script repository.
BEGIN
  sys.rqScriptDrop('myRandomRedDots2');
 END;
 /
```

6.3.3 Manage Datastores in SQL

Oracle R Enterprise provides PL/SQL procedures and Oracle Database data dictionary views for the basic management of datastores in SQL. The following tables list the procedures and views.

Table 6-5 PL/SQL Procedures for Managing Datastores

PL/SQL Procedures	Description
`rqGrant`	Grants read privilege access to a datastore or script.
`rqRevoke`	Revokes read privilege access to a datastore or script.
`rqDropDataStore`	Deletes a datastore.

Table 6-6 Data Dictionary Views for Datastores

Views	Description
`ALL_RQ_DATASTORES`	Describes the datastores available to the current user, including whether the datastore is grantable.
`RQUSER_DATASTORELIST`	Describes the datastores in the Oracle Database schema..

Table 6-6 (Cont.) Data Dictionary Views for Datastores

Views	Description
RQUSER_DATASTORECONTENTS	Describes the objects in the datastores in the Oracle Database schema.
USER_RQ_DATASTORE_PRIVS	Describes the datastores and the users to whom the current user has granted read privilege access.
USER_RQ_DATASTORES	Describes the datastores owned by the current user, including whether the datastore is grantable.

Related Topics:

SQL APIs for Oracle R Enterprise (page A-1)

Oracle Database Views for Oracle R Enterprise (page B-1)

A

SQL APIs for Oracle R Enterprise

The Oracle R Enterprise SQL APIs comprise SQL table functions for executing R functions in one or more embedded R sessions on the Oracle R Enterprise Server database, and PL/SQL procedures for managing Oracle R Enterprise datastores and for managing scripts in the Oracle R Enterprise R script repository.

The SQL APIs for Oracle R Enterprise are described in the following topics:

SQL Table Functions for Embedded R Execution

PL/SQL Procedures for Managing Datastores and Scripts

A.1 rqDropDataStore Procedure

The `rqDropDataStore` procedure deletes a datastore from an Oracle Database schema.

Syntax

```
rqDropDataStore (
    DS_NAME      VARCHAR2      IN)
```

Parameters

Parameter	Description
DS_NAME	The name of the datastore to drop.

Example A-1 Dropping a Datastore

This example deletes the datastore `datastore_1` from the current user schema.

```
rqDropDataStore('datastore_1')
```

Related Topics:

Manage Datastores in SQL (page 6-48)

Oracle Database Views for Oracle R Enterprise (page B-1)

USER_RQ_DATASTORES (page B-4)

USER_RQ_DATASTORE_PRIVS (page B-3)

A.2 rqEval Function

The `rqEval` function executes the R function in the script specified by the `EXP_NAM` parameter. You can pass arguments to the R function with the `PAR_CUR` parameter.

The `rqEval` function does not automatically receive any data from the database. The R function generates the data that it uses or it explicitly retrieves it from a data source such as Oracle Database, other databases, or flat files.

The R function returns an R `data.frame` object, which appears as a SQL table in the database. You define the form of the returned value with the `OUT_QRY` parameter.

Syntax

```
rqEval (
     PAR_CUR     REF CURSOR     IN
     OUT_QRY     VARCHAR2       IN)
     EXP_NAM     VARCHAR2       IN)
```

Parameters

Parameter	Description
PAR_CUR	A cursor that contains argument values to pass to the R function specified by the EXP_NAME parameter.

Parameter	Description
`OUT_QRY`	One of the following: • NULL, which returns a serialized object that can contain both data and image objects. • A SQL `SELECT` statement that specifies the column names and data types of the table returned by `rqEval`. Any image data is discarded. You can provide a prototype row using the dual dummy table or you can base the `SELECT` statement on an existing table or view. The R function must return a `data.frame`. • The string `'XML'`, which specifies that the table returned contains a CLOB that is an XML string. The XML can contain both structured data and images, with structured or semi-structured R objects first, followed by the image or images generated by the R function. Images are returned as a base 64 encoding of the PNG representation. • The string `'PNG'`, which specifies that the table returned contains a BLOB that has the image or images generated by the R function in PNG format.
`EXP_NAM`	The name of a script in the Oracle R Enterprise R script repository.

Return Value

Function `rqEval` returns a table that has the structure specified by the `OUT_QRY` parameter value.

Examples

Example A-2 Using rqEval

This example creates the script `myRandomRedDots2`. The value of the first parameter to `rqEval` is `NULL`, which specifies that no arguments are supplied to the function `myRandomRedDots2`. The value of second parameter is a string that specifies a SQL statement that describes the column names and data types of the `data.frame` returned by `rqEval`. The value of third parameter is the name of the script in the Oracle R Enterprise R script repository.

```
-- Create a script named myRandomRedDots2 and add it to the R script repository.
-- Specify that the script is private and to overwrite a script with the same name.
BEGIN
  sys.rqScriptCreate('myRandomRedDots2',
    'function(divisor = 100, numDots = 100) {
       id <- 1:10
       plot(1:numDots, rnorm(numDots), pch = 21, bg = "red", cex = 2 )
       data.frame(id = id, val = id / divisor)}',
       v_global => FALSE,
       v_overwrite => TRUE);
END;
/

SELECT *
  FROM table(rqEval(NULL, 'SELECT 1 id, 1 val FROM dual', 'myRandomRedDots2'));
```

In Oracle SQL Developer, the results of the `SELECT` statement are:

```
        ID        VAL
---------- ----------
         1        .01
         2        .02
         3        .03
         4        .04
         5        .05
         6        .06
         7        .07
         8        .08
         9        .09
        10         .1

 10 rows selected
```

Example A-3 Passing Arguments to the R Function invoked by rqEval

This example provides arguments to the R function by specifying a cursor as the first parameter to `rqEval`. The cursor specifies multiple arguments in a single row of scalar values.

```
SELECT *
  FROM table(rqEval(cursor(SELECT 50 "divisor", 500 "numDots" FROM dual),
    'SELECT 1 id, 1 val FROM dual',
    'myRandomRedDots2'));
```

In Oracle SQL Developer, the results of the `SELECT` statement are:

```
        ID        VAL
---------- ----------
         1        .02
         2        .04
         3        .06
         4        .08
         5         .1
         6        .12
         7        .14
         8        .16
         9        .18
        10         .2

 10 rows selected
```

Example A-4 Specifying PNG as the Output Table Definition

This example creates a script named `PNG_Example` and stores it in the Oracle R Enterprise R script repository. The invocation of `rqEval` specifies an `OUT_QRY` value of `'PNG'`.

```
BEGIN
  sys.rqScriptDrop('PNG_Example');
  sys.rqScriptCreate('PNG_Example',
    'function(){
      dat <- data.frame(y = log(1:100), x = 1:100)
      plot(lm(y ~ x, dat))
      }');
END;
/
SELECT *
  FROM table(rqEval(NULL,'PNG','PNG_Example'));
```

In Oracle SQL Developer, the results of the `SELECT` statement are:

```
NAME     ID  IMAGE
------ ----  ------
          1  (BLOB)
          2  (BLOB)
          3  (BLOB)
          4  (BLOB)
```

A.3 rqGrant Procedure

The `rqGrant` procedure grants read privilege access to an Oracle R Enterprise datastore or to a script in the Oracle R Enterprise R script repository.

Syntax

```
rqGrant (
    V_NAME     VARCHAR2     IN
    V_TYPE     VARCHAR2     IN
    V_USER     VARCHAR2     IN     DEFAULT)
```

Parameters

Parameter	Description
V_NAME	The name of an Oracle R Enterprise datastore or a script in the Oracle R Enterprise R script repository.
V_TYPE	For a datastore, the type is `datastore`; for a script, the type is `rqscript`.
V_USER	The name of the user to whom to grant access.

Example A-5 Granting Read Access to a Script

```
-- Grant read privilege access to Scott.
BEGIN
  rqGrant('myRandomRedDots2', 'rqscript', 'SCOTT');
END;
/
```

Related Topics:

rqRevoke Procedure (page A-8)

A.4 rqGroupEval Function

The *`rqGroupEval`* function is a user-defined function that identifies a grouping column. The user defines an *`rqGroupEval`* function in PL/SQL using the SQL object `rqGroupEvalImpl`, which is a generic implementation of the group apply functionality in SQL. The implementation supports data-parallel execution, in which one or more R engines perform the same R function, or task, on different partitions of data. The data is partitioned according to the values of the grouping column.

Only one grouping column is supported. If you have multiple columns, then combine the columns into one column and use the new column as the grouping column.

The *rqGroupEval* function executes the R function in the script specified by the EXP_NAM parameter. You pass data to the R function with the INP_CUR parameter. You can pass arguments to the R function with the PAR_CUR parameter.

The R function returns an R data.frame object, which appears as a SQL table in the database. You define the form of the returned value with the OUT_QRY parameter.

To create an *rqGroupEval* function, you create 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.

Syntax

```
rqGroupEval (
     INP_CUR     REF CURSOR     IN
     PAR_CUR     REF CURSOR     IN
     OUT_QRY     VARCHAR2       IN
     GRP_COL     VARCHAR2       IN
     EXP_NAM     VARCHAR2       IN)
```

Parameters

Parameter	Description
INP_CUR	A cursor that specifies the data to pass to the R function specified by the EXP_NAME parameter.
PAR_CUR	A cursor that contains argument values to pass to the R function.
OUT_QRY	One of the following: • NULL, which returns a serialized object that can contain both data and image objects. • A SQL SELECT statement that specifies the column names and data types of the table returned by rqEval. Any image data is discarded. You can provide a prototype row using the dual dummy table or you can base the SELECT statement on an existing table or view. The R function must return a data.frame. • The string 'XML', which specifies that the table returned contains a CLOB that is an XML string. The XML can contain both structured data and images, with structured or semi-structured R objects first, followed by the image or images generated by the R function. Images are returned as a base 64 encoding of the PNG representation. • The string 'PNG', which specifies that the table returned contains a BLOB that has the image or images generated by the R function in PNG format.
GRP_COL	The name of the grouping column by which to partition the data.

Parameter	Description
EXP_NAM	The name of a script in the Oracle R Enterprise R script repository.

Return Value

The user-defined *rqGroupEval* function returns a table that has the structure specified by the OUT_QRY parameter value.

Examples

Example A-6 (page A-8) has a PL/SQL block that drops the script myC5.0Function to ensure that the script does not exist in the Oracle R Enterprise R script repository. It then creates a function and stores it as the script myC5.0Function in the R script repository.

The R function accepts two arguments: the data on which to operate and a prefix to use in creating datastores. The function uses the C50 package to build C5.0 models on the churn data set from C50. The function builds one churn model on the data for each state.

The myC5.0Function function loads the C50 package so that the function body has access to it when the function executes in an R engine on the database server. The function then creates a datastore name using the datastore prefix and the name of a state. To exclude the state name from the model, the function deletes the column from the data.frame. Because factors in the data.frame are converted to character vectors when they are loaded in the user-defined embedded R function, the myC5.0Function function explicitly converts the character vectors back to R factors.

The myC5.0Function function gets the data for the state from the specified columns and then creates a model for the state and saves the model in a datastore. The R function returns TRUE to have a simple value that can appear as the result of the function execution.

Example A-6 (page A-8) next creates a PL/SQL package, churnPkg, and a user-defined function, churnGroupEval. In defining an *rqGroupEval* function implementation, the PARALLEL_ENABLE clause is optional but the CLUSTER BY clause is required.

Finally, the example executes a SELECT statement that invokes the churnGroupEval function. In the INP_CUR argument of the churnGroupEval function, the SELECT statement specifies the PARALLEL hint to use parallel execution of the R function and the data set to pass to the R function. The INP_CUR argument of the churnGroupEval function specifies connecting to Oracle R Enterprise and the datastore prefix to pass to the R function. The OUT_QRY argument specifies returning the value in XML format, the GRP_NAM argument specifies using the state column of the data set as the grouping column, and the EXP_NAM argument specifies the myC5.0Function script in the R script repository as the R function to invoke.

For each of 50 states plus Washington, D.C., the SELECT statement returns from the churnGroupEval table function the name of the state and an XML string that contains the value TRUE.

See Also:

Example 6-13 (page 6-27)

Example A-6 Using an rqGroupEval Function

```
BEGIN
  sys.rqScriptDrop('myC5.0Function');
  sys.rqScriptCreate('myC5.0Function',
    'function(dat, datastorePrefix) {
       library(C50)
       datastoreName <- paste(datastorePrefix, dat[1, "state"], sep = "_")
       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)
       mod <- C5.0(churn ~ ., data = dat, rules = TRUE)
       ore.save(mod, name = datastoreName)
       TRUE
      }');
END;
/

CREATE OR REPLACE PACKAGE churnPkg AS
  TYPE cur IS REF CURSOR RETURN CHURN_TRAIN%ROWTYPE;
END churnPkg;
/
CREATE OR REPLACE FUNCTION churnGroupEval(
  inp_cur churnPkg.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 ("state"))
CLUSTER inp_cur BY ("state")
USING rqGroupEvalImpl;
/

SELECT *
  FROM table(churnGroupEval(
    cursor(SELECT * /*+ parallel(t,4) */ FROM CHURN_TRAIN t),
    cursor(SELECT 1 AS "ore.connect",
           'myC5.0model' AS "datastorePrefix" FROM dual),
    'XML', 'state', 'myC5.0Function'));
```

A.5 rqRevoke Procedure

The `rqRevoke` procedure revokes read privilege access to an Oracle R Enterprise datastore or to a script in the Oracle Database R script repository.

Syntax

```
rqGrant (
    V_NAME     VARCHAR2     IN
    V_TYPE     VARCHAR2     IN
    V_USER     VARCHAR2     IN     DEFAULT)
```

Parameters

Parameter	Description
V_NAME	The name of an Oracle R Enterprise datastore or a script in the Oracle Database R script repository.
V_TYPE	For a datastore, the type is datastore; for a script, the type is rqscript.
V_USER	The name of the user from whom to revoke access.

Example A-7 Revoking Read Access to a Script

```
-- Revoke read privilege access to Scott.
BEGIN
  rqRevoke('myRandomRedDots2', 'rqscript', 'SCOTT');
END;
/
```

Related Topics:

rqGrant Procedure (page A-5)

A.6 rqRowEval Function

The `rqRowEval` function executes the R function in the script specified by the `EXP_NAM` parameter. You pass data to the R function with the `INP_CUR` parameter. You can pass arguments to the R function with the `PAR_CUR` parameter. The `ROW_NUM` parameter specifies the number of rows that should be passed to each invocation of the R function. The last chunk may have fewer rows than the number specified.

The `rqRowEval` function supports data-parallel execution, in which one or more R engines perform the same R function, or task, on disjoint chunks of data. Oracle Database handles the management and control of the potentially multiple R engines that run on the database server machine, automatically chunking and passing data to the R engines executing in parallel. Oracle Database ensures that R function executions for all chunks of rows complete, or the `rqRowEval` function returns an error.

The R function returns an R `data.frame` object, which appears as a SQL table in the database. You define the form of the returned value with the `OUT_QRY` parameter.

Syntax

```
rqRowEval (
     INP_CUR     REF CURSOR     IN
     PAR_CUR     REF CURSOR     IN
     OUT_QRY     VARCHAR2       IN
     ROW_NUM     NUMBER         IN
     EXP_NAM     VARCHAR2       IN)
```

Parameters

Table A-1 Parameters of the rqRowEval Function

Parameter	Description
INP_CUR	A cursor that specifies the data to pass to the R function specified by the `EXP_NAME` parameter.

Table A-1 (Cont.) Parameters of the rqRowEval Function

Parameter	Description
`PAR_CUR`	A cursor that contains argument values to pass to the R function.
`OUT_QRY`	One of the following: • NULL, which returns a serialized object that can contain both data and image objects. • A SQL `SELECT` statement that specifies the column names and data types of the table returned by `rqEval`. Any image data is discarded. You can provide a prototype row using the dual dummy table or you can base the SELECT statement on an existing table or view. The R function must return a `data.frame`. • The string `'XML'`, which specifies that the table returned contains a CLOB that is an XML string. The XML can contain both structured data and images, with structured or semi-structured R objects first, followed by the image or images generated by the R function. Images are returned as a base 64 encoding of the PNG representation. • The string `'PNG'`, which specifies that the table returned contains a BLOB that has the image or images generated by the R function in PNG format.
`ROW_NUM`	The number of rows to include in each invocation of the R function.
`EXP_NAM`	The name of a script in the Oracle Database R script repository.

Return Value

Function `rqRowEval` returns a table that has the structure specified by the `OUT_QRY` parameter value.

Examples

Example A-8 (page A-11) uses the C50 package to score churn data (that is, to predict which customers are likely to churn) using C5.0 decision tree models. The example scores the customers from the specified state in parallel. This example produces the same result as the invocation of function `ore.rowApply` in Example 6-16 (page 6-36).

> **Tip:**
>
> Example A-8 (page A-11) uses the CHURN_TEST table and the `myXLevels` datastore created by Example 6-16 (page 6-36) so in R you should invoke the functions that create the table and that get the `xlevels` object and save it in the `myXLevels` datastore in Example 6-16 (page 6-36) before running Example A-8 (page A-11).

As Example A-6 (page A-8) does, Example A-8 (page A-11) creates a user-defined function and saves the function in the Oracle Database R script repository. The user-defined function creates a C5.0 model for a state and saves the model in a datastore. In Example A-8 (page A-11), however, the user-defined function myC5.0FunctionForLevels uses the list of levels created in Example 6-16 (page 6-36) instead of computing the levels using the as.factor function as function myC5.0Function does in Example A-6 (page A-8). The function myC5.0FunctionForLevels returns the value TRUE.

As Example A-6 (page A-8) does, Example A-8 (page A-11) creates the PL/SQL package churnPkg and the function churnGroupEval. Example A-6 (page A-8) declares a cursor to get the names of the datastores that include the string myC5.0modelFL and then executes a PL/SQL block that deletes those datastores. The example next executes a SELECT statement that invokes the churnGroupEval function. The churnGroupEval function invokes the myC5.0FunctionForLevels function to generate the C5.0 models and save them in datastores.

Example A-8 (page A-11) then creates the myScoringFunction function and stores it in the R script repository. The function scores a C5.0 model for the levels of a state and returns the results in a data.frame.

Finally, Example A-8 (page A-11) executes a SELECT statement that invokes the rqRowEval function. The input cursor to the rqRowEval function uses the PARALLEL hint to specify the degree of parallelism to use. The cursor specifies the CHURN_TEST table as the data source and filters the rows to include only those for Massachusetts. All rows processed use the same predictive model.

The parameters cursor specifies the ore.connect control argument to connect to Oracle R Enterprise on the database server and specifies values for the datastorePrefix and xlevelsDatastore arguments to the myScoringFunction function.

The SELECT statement for the OUT_QRY parameter specifies the format of the output. The ROW_NUM parameter specifies 200 as the number of rows to process at a time in each parallel R engine. The EXP_NAME parameter specifies myScoringFunction in the R script repository as the R function to invoke.

Example A-8 Using an rqRowEval Function

```
BEGIN
  sys.rqScriptDrop('myC5.0FunctionForLevels');
  sys.rqScriptCreate('myC5.0FunctionForLevels',
    'function(dat, xlevelsDatastore, datastorePrefix) {
       library(C50)
       state <- dat[1,"state"]
       datastoreName <- paste(datastorePrefix, dat[1, "state"], sep = "_")
       dat$state <- NULL
       ore.load(name = xlevelsDatastore) # To get the xlevels object.
       for (j in names(xlevels))
         dat[[j]] <- factor(dat[[j]], levels = xlevels[[j]])
       c5mod <- C5.0(churn ~ ., data = dat, rules = TRUE)
       ore.save(c5mod, name = datastoreName)
       TRUE
      }');
END;
/

CREATE OR REPLACE PACKAGE churnPkg AS
  TYPE cur IS REF CURSOR RETURN CHURN_TEST%ROWTYPE;
END churnPkg;
/
CREATE OR REPLACE FUNCTION churnGroupEval(
  inp_cur churnPkg.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 ("state"))
CLUSTER inp_cur BY ("state")
USING rqGroupEvalImpl;
/

DECLARE
 CURSOR c1
  IS
    SELECT dsname FROM rquser_DataStoreList WHERE dsname like 'myC5.0modelFL%';

BEGIN
  FOR dsname_st IN c1
  LOOP
    rqDropDataStore(dsname_st.dsname);
  END LOOP;
END;

SELECT *
  FROM table(churnGroupEval(
    cursor(SELECT * /*+ parallel(t,4) */ FROM CHURN_TEST t),
    cursor(SELECT 1 AS "ore.connect",
           'myXLevels' as "xlevelsDatastore",
           'myC5.0modelFL' AS "datastorePrefix" FROM dual),
    'XML', 'state', 'myC5.0FunctionForLevels'));

BEGIN
  sys.rqScriptDrop('myScoringFunction');
  sys.rqScriptCreate('myScoringFunction',
    'function(dat, xlevelsDatastore, datastorePrefix) {
       library(C50)
       state <- dat[1, "state"]
       datastoreName <- paste(datastorePrefix, state, sep = "_")
       dat$state <- NULL
       ore.load(name = xlevelsDatastore) # To get the xlevels object.
       for (j in names(xlevels))
         dat[[j]] <- factor(dat[[j]], levels = xlevels[[j]])
       ore.load(name = datastoreName)
       res <- data.frame(pred = predict(c5mod, dat, type = "class"),
                         actual= dat$churn,
                         state = state)
       res
     }');
END;
/

SELECT * FROM table(rqRowEval(
  cursor(select /*+ parallel(t, 4) */ *
         FROM CHURN_TEST t
         WHERE "state" = 'MA'),
  cursor(SELECT 1 as "ore.connect",
         'myC5.0modelFL' as "datastorePrefix",
         'myXLevels' as "xlevelsDatastore"
         FROM dual),
  'SELECT ''aaa'' "pred",''aaa'' "actual" , ''aa'' "state" FROM dual',
   200, 'myScoringFunction'));
```

In Oracle SQL Developer, the results of the last `SELECT` statement are:

```
pred actual state
---- ------ -----
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   yes    MA
yes  yes    MA
yes  yes    MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
yes  yes    MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   no     MA
no   yes    MA
no   no     MA
no   no     MA
no   no     MA
yes  yes    MA
no   no     MA
no   no     MA
no   no     MA

 38 rows selected
```

A.7 rqTableEval Function

The `rqTableEval` function executes the R function in the script specified by the `EXP_NAM` parameter. You pass data to the R function with the `INP_CUR` parameter. You can pass arguments to the R function with the `PAR_CUR` parameter.

The R function returns an R `data.frame` object, which appears as a SQL table in the database. You define the form of the returned value with the `OUT_QRY` parameter.

Syntax

```
rqTableEval (
     INP_CUR     REF CURSOR     IN
     PAR_CUR     REF CURSOR     IN
     OUT_QRY     VARCHAR2       IN
     EXP_NAM     VARCHAR2       IN)
```

Parameters

Table A-2 Parameters of the rqTableEval Function

Parameter	Description
`INP_CUR`	A cursor that specifies the data to pass to the R function specified by the `EXP_NAME` parameter.
`PAR_CUR`	A cursor that contains argument values to pass to the input function.
`OUT_QRY`	One of the following: • NULL, which returns a serialized object that can contain both data and image objects. • A SQL `SELECT` statement that specifies the column names and data types of the table returned by `rqEval`. Any image data is discarded. You can provide a prototype row using the dual dummy table or you can base the SELECT statement on an existing table or view. The R function must return a `data.frame`. • The string `'XML'`, which specifies that the table returned contains a CLOB that is an XML string. The XML can contain both structured data and images, with structured or semi-structured R objects first, followed by the image or images generated by the R function. Images are returned as a base 64 encoding of the PNG representation. • The string `'PNG'`, which specifies that the table returned contains a BLOB that has the image or images generated by the R function in PNG format.
`EXP_NAM`	The name of a script in the Oracle R Enterprise R script repository.

Return Value

Function `rqTableEval` returns a table that has the structure specified by the `OUT_QRY` parameter value.

Examples

Example A-9 (page A-15) first has a PL/SQL block that drops the script `myNaiveBayesModel` to ensure that the script does not exist in the Oracle R Enterprise R script repository. It then creates a function and stores it as the script `myNaiveBayesModel` in the repository.

The R function accepts two arguments: the data on which to operate and the name of a datastore. The function builds a Naive Bayes model on the `iris` data set. Naive Bayes is found in the e1071 package.

The `myNaiveBayesModel` function loads the e1071 package so that the function body has access to it when the function executes in an R engine on the database server. Because factors in the `data.frame` are converted to character vectors when they are loaded in the user-defined embedded R function, the `myNaiveBayesModel` function explicitly converts the character vector to an R factor.

The `myNaiveBayesModel` function gets the data from the specified column and then creates a model and saves it in a datastore. The R function returns `TRUE` to have a simple value that can appear as the result of the function execution.

Example A-9 (page A-15) next executes a `SELECT` statement that invokes the `rqTableEval` function. In the `INP_CUR` argument of the `rqTableEval` function, the `SELECT` statement specifies the data set to pass to the R function. The data is from the IRIS table that was created by invoking `ore.create(iris, "IRIS")`, which is not shown in the example. The `INP_CUR` argument of the `rqTableEval` function specifies the name of a datastore to pass to the R function and specifies the `ore.connect` control argument to establish an Oracle R Enterprise connection to the database during the embedded R execution of the user-defined R function. The `OUT_QRY` argument specifies returning the value in XML format, and the `EXP_NAM` argument specifies the `myNaiveBayesModel` script in the R script repository as the R function to invoke.

Example A-9 Using the rqTableEval Function

```
BEGIN
  sys.rqScriptDrop('myNaiveBayesModel');
  sys.rqScriptCreate('myNaiveBayesModel',
    'function(dat, datastoreName) {
       library(e1071)
       dat$Species <- as.factor(dat$Species)
       nbmod <- naiveBayes(Species ~ ., dat)
       ore.save(nbmod, name = datastoreName)
       TRUE
     }');
END;
/

SELECT *
  FROM table(rqTableEval(
    cursor(SELECT * FROM IRIS),
    cursor(SELECT 'myNaiveBayesDatastore' "datastoreName",
           1 as "ore.connect" FROM dual),
    'XML', 'myNaiveBayesModel'));
```

The `SELECT` statement returns from the `rqTableEval` table function an XML string that contains the value `TRUE`.

The `myNaiveBayesDatastore` datastore now exists and contains the object `nbmod`, as shown by the following `SELECT` statement.

```
SQL> SELECT * from rquser_DataStoreContents
  2     WHERE dsname = 'myNaiveBayesDatastore';

DSNAME                  OBJNAME  CLASS       OBJSIZE  LENGTH  NROW  NCOL
---------------------   -------  ----------  -------  ------  ----  ----
myNaiveBayesDatastore   nbmod    naiveBayes     1485       4
```

In a local R session, you could load the model and display it, as in the following:

```
R> ore.load("myNaiveBayesDatastore")
[1] "nbmod"
R> nbmod
$apriori
Y
    setosa versicolor  virginica
        50         50         50
```

```
$tables
$tables$Sepal.Length
            Sepal.Length
Y              [,1]      [,2]
  setosa     5.006 0.3524897
  versicolor 5.936 0.5161711
  virginica  6.588 0.6358796

$tables$Sepal.Width
            Sepal.Width
Y              [,1]      [,2]
  setosa     3.428 0.3790644
  versicolor 2.770 0.3137983
  virginica  2.974 0.3224966

$tables$Petal.Length
            Petal.Length
Y              [,1]      [,2]
  setosa     1.462 0.1736640
  versicolor 4.260 0.4699110
  virginica  5.552 0.5518947

$tables$Petal.Width
            Petal.Width
Y              [,1]      [,2]
  setosa     0.246 0.1053856
  versicolor 1.326 0.1977527
  virginica  2.026 0.2746501


$levels
[1] "setosa"     "versicolor" "virginica"

$call
naiveBayes.default(x = X, y = Y, laplace = laplace)

attr(,"class")
[1] "naiveBayes"
```

A.8 sys.rqScriptCreate Procedure

The `sys.rqScriptCreate` procedure creates a script and adds it to the Oracle R Enterprise R script repository.

Syntax

```
sys.rqScriptCreate (
    V_NAME         VARCHAR2    IN
    V_SCRIPT       CLOB        IN
    V_GLOBAL       BOOLEAN     IN     DEFAULT
    V_OVERWRITE    BOOLEAN     IN     DEFAULT)
```

Parameter	Description
`V_NAME`	A name for the script in the Oracle R Enterprise R script repository.
`V_SCRIPT`	The R function definition to store in the script.
`V_GLOBAL`	`TRUE` specifies that the script is public; `FALSE` specifies that the script is private.

Parameter	Description
V_OVERWRITE	If the R script repository already has a script with the same name as V_NAME, then TRUE replaces the content of that script with V_SCRIPT and FALSE does not replace it.

Related Topics:

Creating a Script with the SQL APIs (page 6-47)

Manage Scripts in SQL (page 6-46)

A.9 sys.rqScriptDrop Procedure

The sys.rqScriptDrop procedure removes a script from the Oracle R Enterprise R script repository.

Syntax

```
sys.rqScriptCreate (
    V_NAME           VARCHAR2     IN
    V_GLOBAL         BOOLEAN      IN      DEFAULT
    V_SILENT         BOOLEAN      IN      DEFAULT)
```

Parameter	Description
V_NAME	A name for the script in the Oracle R Enterprise R script repository.
V_GLOBAL	TRUE specifies that the script is public; FALSE specifies that the script is private.
V_SILENT	If the R script repository already has a script with the same name as V_NAME, then TRUE replaces the content of that script with V_SCRIPT and FALSE does not replace it.

Related Topics:

Creating a Script with the SQL APIs (page 6-47)

B

Oracle Database Views for Oracle R Enterprise

Oracle Database has several data dictionary views that contain information about Oracle R Enterprise datastores and scripts in the Oracle R Enterprise script repository.

Views for Datastores

- ALL_RQ_DATASTORES (page B-1)
- RQUSER_DATASTORECONTENTS (page B-2)
- RQUSER_DATASTORELIST (page B-3)
- USER_RQ_DATASTORE_PRIVS (page B-3)
- USER_RQ_DATASTORES (page B-4)

Views for Scripts

- ALL_RQ_SCRIPTS (page B-2)
- USER_RQ_SCRIPT_PRIVS (page B-4)
- USER_RQ_SCRIPTS (page B-5)

Related Topics:

Manage Scripts in SQL (page 6-46)

SQL APIs for Oracle R Enterprise (page A-1)

B.1 ALL_RQ_DATASTORES

ALL_RQ_DATASTORES describes the datastores available to the current user.

Column	Datatype	Null	Description
DSOWNER	VARCHAR2(256)	NOT NULL	The owner of the datastore.
DSNAME	VARCHAR2(128)	NOT NULL	The name of the datastore.
NOBJ	NUMBER	NOT NULL	The number of objects in the datastore.
DSSIZE	NUMBER	NOT NULL	The size of the datastore.

Column	Datatype	Null	Description
CDATE	DATE	NOT NULL	The creation date of the datastore.
DESCRIPTION	VARCHAR2(2000)	NULL allowed	A description of the datastore.
GRANTABLE	VARCHAR2(1)	NOT NULL	Whether read privilege access to the datastore can be granted by the owner to another user.

Related Topics:

About Oracle R Enterprise Datastores (page 2-19)

Manage Datastores in SQL (page 6-48)

B.2 ALL_RQ_SCRIPTS

ALL_RQ_SCRIPTS describes the scripts in the Oracle Database R script repository that are available to the current user.

Column	Datatype	Null	Description
OWNER	VARCHAR2(256)	NOT NULL	The owner of the script.
NAME	VARCHAR2(128)	NOT NULL	The name of the script.
SCRIPT	CLOB	NOT NULL	The R function of the script.

Related Topics:

USER_RQ_SCRIPT_PRIVS (page B-4)

USER_RQ_SCRIPTS (page B-5)

B.3 RQUSER_DATASTORECONTENTS

RQUSER_DATASTORECONTENTS contains information about the contents of Oracle R Enterprise datastores.

Column	Datatype	Null	Description
DSNAME	VARCHAR2(128)	NOT NULL	The name of the datastore.
OBJNAME	VARCHAR2(128)	NOT NULL	The names of the objects in the datastore.
CLASS	VARCHAR2(128)	NOT NULL	The R class of an object.
DSSIZE	NUMBER	NOT NULL	The size of an object.
LENGTH	NUMBER	NOT NULL	The size of an object.

Column	Datatype	Null	Description
NROW	NUMBER	NULL allowed	The number of rows in an object.
NCOL	NUMBER	NULL allowed	The number of columns in an object.

Related Topics:

ALL_RQ_DATASTORES (page B-1)

RQUSER_DATASTORELIST (page B-3)

B.4 RQUSER_DATASTORELIST

RQUSER_DATASTORELIST contains information about Oracle R Enterprise datastores.

Column	Datatype	Null	Description
DSNAME	VARCHAR2(128)	NOT NULL	The name of the datastore.
NOBJ	NUMBER	NOT NULL	The number of objects in a datastore.
DSSIZE	NUMBER	NOT NULL	The size of the datastore.
CDATE	DATE	NOT NULL	The date the datastore was created.
DESCRIPTION	VARCHAR2(2000)	NULL allowed	The description of the datastore.

Related Topics:

Manage Datastores in SQL (page 6-48)

Oracle Database Views for Oracle R Enterprise (page B-1)

ALL_RQ_DATASTORES (page B-1)

B.5 USER_RQ_DATASTORE_PRIVS

USER_RQ_DATASTORE_PRIVS describes the datastores and the users to whom the current user has granted read privilege access.

Column	Datatype	Null	Description
DSNAME	VARCHAR2(128)	NOT NULL	The name of a datastore.
GRANTEE	VARCHAR2(30)	NOT NULL	The user to whom read privilege access has been granted.

Related Topics:

About Oracle R Enterprise Datastores (page 2-19)

Manage Datastores in SQL (page 6-48)

ALL_RQ_DATASTORES (page B-1)

USER_RQ_DATASTORES (page B-4)

B.6 USER_RQ_DATASTORES

USER_RQ_DATASTORES describes datastores created by the current user.

Column	Datatype	Null	Description
DSNAME	VARCHAR2(128)	NOT NULL	The name of a datastore.
NOBJ	NUMBER	NOT NULL	The number of objects in the datastore.
DSSIZE	NUMBER	NOT NULL	The size of the datastore.
CDATE	DATE	NOT NULL	The creation date of the datastore.
DESCRIPTION	VARCHAR2(2000)	NULL allowed	A description of the datastore.
GRANTABLE	VARCHAR2(1)	NOT NULL	Whether read privilege access to the datastore can be granted by the owner to another user.

Related Topics:

About Oracle R Enterprise Datastores (page 2-19)

Manage Datastores in SQL (page 6-48)

ALL_RQ_DATASTORES (page B-1)

USER_RQ_DATASTORE_PRIVS (page B-3)

B.7 USER_RQ_SCRIPT_PRIVS

USER_RQ_SCRIPT_PRIVS describes the scripts in the Oracle Database R script repository to which the current user has granted read access and the users to whom access has been granted.

Column	Datatype	Null	Description
NAME	VARCHAR2(128)	NOT NULL	The name of the script to which read access has been granted.
GRANTEE	VARCHAR2(128)	NOT NULL	The user to whom read access has been granted.

Related Topics:

ALL_RQ_SCRIPTS (page B-2)

USER_RQ_SCRIPTS (page B-5)

B.8 USER_RQ_SCRIPTS

`USER_RQ_SCRIPTS` describes the scripts in the Oracle Database R script repository that are owned by the current user.

Column	Datatype	Null	Description
`NAME`	`VARCHAR2(128)`	`NOT NULL`	The name of the script.
`SCRIPT`	`CLOB`	`NOT NULL`	The R function of the script.

Related Topics:

ALL_RQ_SCRIPTS (page B-2)

USER_RQ_SCRIPT_PRIVS (page B-4)

C

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 (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 (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, 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

E

F

G

H

I

K

P

Q

R

S

T

U

V

W