Oracle® R Enterprise

User's Guide

Release 1.3 for Windows, Linux, Solaris, and AIX

E36761-08

April 2013

Oracle R Enterprise User's Guide, Release 1.3 for Windows, Linux, Solaris, and AIX

E36761-08

Primary Author: David McDermid

Contributing Author: Margaret Taft

Contents

Preface

This book describes how to use Oracle R Enterprise release 1.3.

Audience

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

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

These manuals describe Oracle R Enterprise:

- *Oracle R Enterprise Installation and Administration Guide*
- *Oracle R Enterprise User's Guide* (this manual)
- *Oracle R Enterprise Release Notes*

For information about Oracle Database, see the O*racle Database Documentation Library 11g Release 2 (11.2)* at `http://www.oracle.com/technetwork/indexes/documentation/index.html?ssSourceSiteId=ocomen`.

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.

Convention	Meaning
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.

What's New in Oracle R Enterprise 1.3?

This section describes new features in releases of Oracle R Enterprise. It includes the following sections:

- New Features for Release 1.3
- New Features for Release 1.1

New Features for Release 1.3

Release 1.3 includes these new features:

- Predicting with R Models using in-database data
- Ordering and indexing, described in Order in Tables
- In-Database Predictive Models in Oracle R Enterprise
- Persist and Manage R Objects in the Database
- Date and Time Data Types
- Sampling and Partitioning
- Long Names for columns
- Automatic Database Connection in Embedded R Scripts
- R neural network for in-database data, described in ore.neural()

Other changes:

- 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.

New Features for Release 1.1

Release 1.1 includes these new features:

- **Support for IBM AIX**: Oracle R Distribution and Oracle R Enterprise are supported on AIX 5.3 and higher.
- **Support for Solaris:** Oracle R Distribution and Oracle R Enterprise are supported on 10 and higher for both 64-bit SPARC and 64-bit x386 (Intel) processors.
- **Use 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.

- **Server runs on Windows**

  The Oracle R Enterprise Server now runs on 64-bit and 32-bit Windows operating systems.

- **Support for Oracle Wallet**

  R scripts no longer need to have database authentication credentials in clear text. Oracle R Enterprise is integrated with Oracle Wallet for that purpose.

- **Improved installation**

  The installation scripts have been improved with more prerequisite checks and detailed error messages. Error messages provide specific instructions on remedial actions.

1 Overview of Oracle R Enterprise

R is an open source statistical programming language and environment. For information about R, see the R Project for Statistical Computing at `http://www.r-project.org`.

R provides an environment for statistical computing, including:

- An easy-to-use language
- A powerful graphical environment for visualization
- Many out-of-the-box statistical techniques
- R packages (An R package is a set of related functions, help files, and data files; as of this writing, there are more than 4000 R packages, but the number grows constantly.)
- The R Console graphical user interface for analyzing data interactively

R's rapid adoption has earned it a reputation as a new statistical software standard.

Oracle R Enterprise is a component of the Oracle Advanced Analytics Option of Oracle Database Enterprise Edition.

For detailed information about Oracle R Enterprise, including links to software downloads, go to **Oracle R Enterprise** at `http://www.oracle.com/technetwork/database/options/advanced-analytics/r-enterprise/index.html`. This site contains links to downloads, the blog, the discussion forum, and the latest documentation. See Oracle R Enterprise Useful Links for information about the blog and the forum.

Oracle R Enterprise allows users to perform statistical analysis on data stored in an Oracle Database. Oracle R Enterprise has these components:

- The Oracle R Enterprise R **transparency layer**. The transparency layer is a collection of packages that support mapping of R data types to Oracle Database objects and generate SQL transparently in response to R expressions on mapped data types. The transparency layer allows an R user to interact directly with database-resident data using R language constructs. One advantage of interacting with database-resident data is that R users can work with data too large to fit into the memory of a user's desktop system.
- The Oracle R Enterprise **statistics engine**, a collection of statistical functions and procedures corresponding to commonly-used statistical libraries. The statistics engine packages execute in Oracle Database.
- **Embedded R** execution enables the database server to manage and control the execution of R scripts by spawning server-side R engines. Embedded R execution enables operationalization of R scripts, that is, running R scripts in a lights-out

fashion as part of an application. Embedded R execution eliminates moving data from Oracle Database. Embedded R execution enables data and task parallel execution, generation of rich XML output and `png` image streams through the SQL API, and provides parallel simulations capability.

Oracle R Enterprise includes many packages; for a list see Oracle R Enterprise and Oracle R Distribution Packages.

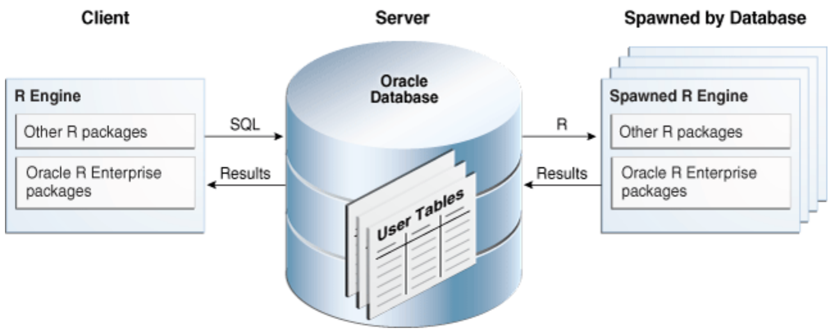
The rest of this chapter describes Oracle R Enterprise Architecture and Oracle R Enterprise Supported Configurations.

Oracle R Enterprise Training is available free from Oracle Learning Library.

Oracle R Enterprise Useful Links describes the blog and the forum.

Oracle R Enterprise Architecture

Oracle R Enterprise has these three components including the connector for Hadoop:



Oracle R Enterprise Components: Client R Engine, Database server Engine, and R Engines spawned by the database.

****

1. The **Client R Engine** (R Engine in Client) is a collection of R packages that allows you to connect to an Oracle Database and to interact with data in that database.

   You can use any R commands from the client. In addition, the client supplies these functions:

   - The R SQL Transparency layer intercepts R functions for scalable in-database execution
   - Functions intercept data transforms, statistical functions, and Oracle R Enterprise-specific functions
   - Interactive display of graphical results and flow control as in open source R
   - Submission of R closures (functions) for execution in Oracle Database

2. The **Server** (in Oracle Database) is a collection of PL/SQL procedures and libraries that augment Oracle Database with the capabilities required to support an Oracle R Enterprise client. The R engine is also installed on Oracle Database to support embedded R execution. Oracle Database spawns R engines, which can provide data parallelism.

   The Oracle R Enterprise Database engine provides this functionality:

- Scale to large datasets
- Access to tables, views, and external tables in the database, as well as those accessible through database links
- Use SQL query parallel execution
- Use in-database statistical and data mining functionality

3. **R Engines spawned by Oracle Database** support database-managed parallelism; provide lights-out scheduled execution of R scripts, that is, scheduling or triggering R scripts packaged inside a PL/SQL or SQL query. Oracle R Enterprise provides efficient transfer to and from the spawned engines. Embedded R execution can be used to emulate MapReduce style programming.

There are several data types specific to Oracle R Enterprise; see Data Types Supported for details.

Oracle R Enterprise Supported Configurations

Oracle R Enterprise consists of a client and a server. The client and the server run on Oracle Linux, Red Hat Linux; the client runs on Microsoft Windows 64-bit. The server is installed in an Oracle Database, to which the client connects. Client and server are not required to run on the same operating system. For example, the client can run on Microsoft Windows with the server installed on Oracle Linux.

Oracle R Enterprise also runs on Oracle Exadata machines with the Linux and Solaris operating systems. For details, see *Oracle R Enterprise Installation and Administration Guide*.

GUIs and IDEs for R

Open source R is distributed through The Comprehensive R Archive Network (CRAN). It can be downloaded, but it is not shipped.

The CRAN distribution contains a Graphical User Interface (GUI) for Windows. There are open source GUIs for R on all operating systems, but they require a download from a separate site and a separate install.

If you require an Integrated Development Environment (IDE) for R, you may wish to use RStudio IDE. For an overview of RStudio IDE installation, see *Oracle R Enterprise Installation and Administration Guide*.

Oracle R Enterprise Training

Oracle R Enterprise Tutorial Series (`https://apex.oracle.com/pls/apex/f?p=44785:24:17534844732288::NO::P24_CONTENT_ID,P24_PREV_PAGE:6528,1`), part of Oracle Learning Library, contains lessons describing Open-source R basics and Oracle R Enterprise functionality. Topics include R basics, graphing in R, the transparency layer, R scripts, and SQL scripts. There is also a lesson about Oracle R Connector for Hadoop. (Oracle Connector for Hadoop is a separate product.)

Lessons in Oracle Learning Library are free.

> **See Also:** The Learning R Series presentations available on the Oracle R Enterprise page on the Oracle Technology Network at `http://www.oracle.com/technetwork/database/options/advanced-analytics/r-enterprise/index.html`

Oracle R Enterprise Useful Links

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

- The Oracle R Enterprise Discussion Forum (`https://forums.oracle.com/forums/forum.jspa?forumID=1397)` supports all aspects of Oracle's R-related offerings, including: Oracle R Enterprise, Oracle R Connector for Hadoop (part of the Big Data Connectors), and Oracle R Distribution. Use the forum to ask questions and make comments about the software.
- The Oracle R Enterprise Blog (`https://blogs.oracle.com/R/`) discusses best practices, tips, and tricks for applying Oracle R Enterprise and Oracle R Connector for Hadoop in both traditional and new Big Data environments.

2

Oracle R Enterprise Transparency Layer

Oracle R Enterprise Transparency Layer performs these functions:

- Traps all R commands and scripts prior to execution and looks for opportunities to ship them to Oracle Database for execution in the database.
- Enables transparent grandparent SQL generation for R expressions that use mapped data types.
- Converts R commands and scripts to SQL equivalents to leverage Oracle Database as a high-performance compute engine, taking advantage of query optimization, tables indexes, deferred evaluation, and parallel execution.

The Oracle R Enterprise transparency layer allows R users to use R syntax to work directly with database-resident objects without having to pull data from Oracle into R's memory on the user's desktop. It thus enables R users to work with data larger than desktop memory allows.

R language constructs and syntax are supported for objects mapped to Oracle Database objects.

This chapter summarizes the functionality provided by the Transparency Layer. These topics are discussed:

- Data Types Supported
- Operators and Functions Supported

Data Types Supported

The following R data types have been overloaded so that they are mapped to database objects and hence enabled for in-database execution:

- Character, Integer, Numeric, and Logical vectors
- Date and Time Data Types
- Factors
- Data Frame
- Matrix is overloaded in two situations:
  - Linear algebra cross-products
  - Creating input matrices for advanced analytics

`class(object)` reports the data type of such mapped objects. For example, if the table NARROW contains the column AGE and AGE is numeric,

```
R> class(NARROW$AGE)
```

```
[1] "ore.numeric"
attr(,"package")
[1] "OREbase"
```

Date and Time Data Types

This section describes how Oracle database supports Date and Time Data Types and illustrates how to use these data types in Oracle R Enterprise.

Date and Time Data Types in Oracle

Oracle Database supports these data and time data types:

- The `DATE` data type stores date and time information. For each `DATE` value, Oracle stores the following fields: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

  The valid date range is January 1, 4712 BC, to December 31, 9999 AD.
- The `TIMESTAMP` data type is an extension of the `DATE` data type. It stores the year, month, and day of the `DATE` data type, plus hour, minute, and second values. Supports an optional `fractional_seconds_precision`, the number of digits in the fractional part of the `SECOND` field in `DATE`. You can specify 0 to 9 digits; the default is 6 digits.

  There are two extensions of `TIMESTAMP`:

  – `TIMESTAMP WITH TIME ZONE` is `TIMESTAMP` as well as time zone displacement value `TIMEZONE_HOUR` and `TIMEZONE_MINUTE`.

  – `TIMESTAMP WITH LOCAL TIME ZONE` is `TIMESTAMP WITH TIME ZONE` with data normalized to the database time zone when it is stored in the database. When the data is retrieved, users see the data in the session time zone.
- `INTERVAL YEAR TO MONTH` stores a period of time using the `YEAR` and `MONTH` fields. This data type is useful for representing the difference between two data time values when only the year and month values are significant.
- `INTERVAL DAY TO SECOND` stores a period of time in terms of days, hours, minutes, and seconds. This data type is useful for representing the precise difference between two date time values.
- `INTERVAL YEAR TO MONTH` stores a period of time in years and months, where optional year_precision, which is the number of digits in the `YEAR` date time field. Accepted values are 0 to 9.
- `INTERVAL DAY TO SECOND` stores a period of time in days, hours, minutes, and seconds. Supports an optional `day_precision`, the maximum number of digits in the `DAY` date time field (value is 0 to 9 with a default of 2.) Also supports optional `fractional_seconds_precision`, the number of digits in the fractional part of the `SECOND` field. (value 0 to 9 with a default of 6).

For detailed information about Oracle Data Types, see "Data Types" in *Oracle Database SQL Language Reference*.

You can perform all expected operations on dates.

Oracle R Enterprise Support for Date and Time

Oracle R Enterprise provides these classes to support date and time calculations:

- `ore.date` (Oracle `DATE`)

- `ore.datetime` (`TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`)
- `ore.difftime` (`INTERVAL DAY TO SECOND`)

Note that `ore.datetime` objects 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.

Operators and Functions Supported

Oracle R Enterprise supports data pre-processing functionality extensively so all data preparation and analysis can take place directly in the database.

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).

- **Mathematical transformations**: abs, sign, sqrt, ceiling, floor, trunc, cummax, cummin, cumprod, cumsum, log, loglo, log10, log2, log1p, acos, acosh, asin, asinh, atan, atanh, exp, expm1, cos, cosh, sin, sinh, tan, atan2, tanh, gamma, lgamma, digamma, trigamma, factorial, lfactorial, round, signif, pmin, pmax, zapsmall, rank, diff, besselI, besselJ, besselK, besselY
- **Basic statistics**: mean, summary, min, max, sum, any, all, median, range, IQR, fivenum, mad, quantile, sd, var, table, tabulate, rowSums, colSums, rowMeans, colMeans, cor, cov
- **Arithmetic operators**: +, -, *, /, ^, %%, %/%
- **Comparison operators**: ==, >, <, !=, <=, >=
- **Logical operators**: &, `|`, xor
- **Set operations**: unique, %in%, subset
- **String operations**: tolower, toupper, casefold, toString, chartr, sub, gsub, substr, substring, paste, nchar, grepl
- **Combine Data Frame**: cbind, rbind, merge
- **Combine vectors**: append
- **Vector creation**: ifelse
- **Subset selection**: [, [[, $, head, tail, window, subset, Filter, na.omit, na.exclude, complete.cases
- **Subset replacement**: [<-, [[<-, $<-
- **Data reshaping**: split, unlist
- **Data processing**: eval, with, within, transform
- **Apply variants**: tapply, aggregate, by
- **Special value checks**: is.na, is.finite, is.infinite, is.nan

- **Metadata functions**: nrow, NROW, ncol, NCOL, nlevels, names, names<-, row, col, dimnames, dimnames<-, dim, length, row.names, row.names<-, rownames, rownames<-, colnames, levels, reorder
- **Graphics**: arrows, boxplot, cdplot, co.intervals, coplot, hist, identify, lines, matlines, matplot, matpoints, pairs, plot, points, polygon, polypath, rug, segments, smoothScatter, sunflowerplot, symbols, text, xspline, xy.coords
- **Conversion functions**: as.logical, as.integer, as.numeric, as.character, as.vector, as.factor, as.data.frame
- **Type check functions**: is.logical, is.integer, is.numeric, is.character, is.vector, is.factor, is.data.frame
- **Character manipulation**: nchar, tolower, toupper, casefold, chartr, sub, gsub, substr.
- **Other ore.frame functions**: data.frame, max.col, scale
- **Hypothesis testing**: binom.test, chisq.test, ks.test, prop.test, t.test, var.test, wilcox.test
- **Various Distributions**: Density, cumulative distribution, and quantile functions for standard distributions
- **ore.matrix function**: show, is.matrix, as.matrix, %*% (matrix multiplication), t, crossprod (matrix cross-product), tcrossprod (matrix cross-product A times transpose of B), solve (invert), backsolve, forwardsolve, all appropriate mathematical functions (abs, sign, and so on), summary (max, min, all, and so on), mean

The Oracle R Enterprise sample programs described in Oracle R Enterprise Examples include several examples using each category of these functions with Oracle R Enterprise data types.

3

Using Oracle R Enterprise

This chapter explains how to use Oracle R Enterprise to analyze data stored in tables or views in an Oracle Database.

This chapter discusses these topics:

- Tables in Oracle Database
- View Oracle R Enterprise Documentation
- Oracle R Enterprise Data
- Oracle R Enterprise Database-Embedded R Engine
- Oracle R Enterprise Examples

We assume familiarity with R in the remainder of this section.

For additional examples of using Oracle R Enterprise functionality, see Oracle R Enterprise Statistical Functions. For examples of building statistical models, including models created using Oracle Data Mining algorithm, see In-Database Predictive Models in Oracle R Enterprise.

Tables in Oracle Database

Before you can use Oracle R Enterprise to analyze data stored in database tables, you must install Oracle R Enterprise, start a client, and connect to the database, as described in *Oracle R Enterprise Administrator's Guide*.

By convention, most of the functions and methods defined in Oracle R Enterprise begin with the prefix `ore`. This is done to avoid name collisions with other R software. However, the objects created by those functions and methods can be anything the end user wants them to be. The end user has complete control over object naming.

Pick any object returned by `ore.ls()` and type either `class(OBJECTNAME)` or `class(OBJECTNAME$COLUMN_NAME)`. For example, the following code shows that the class of DF_TABLE is `ore.frame`. The DF_TABLE object is created in Example: Load Data.

```
R> class(DF_TABLE)
[1] "ore.frame"
```

The prefix `ore` indicates that the object is an Oracle R Enterprise created object that holds metadata for the corresponding object in Oracle Database.

ore.frame is the Oracle R Enterprise metadata object that maps to a database table. The `ore.frame` object is the counterpart to an R data.frame.

ore.frame or can be returned by the class() function. For an example of creating ore.frame data, see Load an R Data Frame into the Database.

View Oracle R Enterprise Documentation

Use this command to view the Oracle R Enterprise documentation library:

```
R> OREShowDoc()
```

Oracle R Enterprise Data

Oracle R Enterprise supports this functionality:

- Long Names
- Load an R Data Frame into the Database
- Materialize R Data
- Verify that an ore.frame Exists
- Drop a Database Table
- Pull a Database Table to an R Frame
- Order in Tables
- Persist and Manage R Objects in the Database

Long Names

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

Load an R Data Frame into the Database

Follow these steps to load data from R data frames on your system to the Oracle database:

1. Load contents of the file to an R data frame using read.table() or read.csv() functions documented in R online help.
2. Then use ore.create() to load a data frame to a table:

   ```
   ore.create(data_frame, table="TABLE_NAME")
   ```

Step 2 loads data_frame into the database table TABLE_NAME.

For an example, see Example: Load Data.

Example: Load Data

This example creates an R data frame df consisting of pairs of numbers and letters and then loads the data frame into the table DF_TABLE. The example shows that the data frame and the table have the same dimensions and the same first few elements, but different values for class. The class for DF_TABLE is ore.frame. At the end of the example is a check that DF_TABLE exists in the current schema.

```
R> df <- data.frame(A=1:26, B=letters[1:26])
R> dim(df)
[1] 26  2
R> class(df)
```

```
[1] "data.frame"
R> head(df)
  A B
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
6 6 f
R> ore.create(df, table="DF_TABLE")
R> ore.ls()
[1] "DF_TABLE"
R> class(DF_TABLE)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
R> dim(DF_TABLE)
[1] 26  2
R> head(DF_TABLE)
  A B
0 1 a
1 2 b
2 3 c
3 4 d
4 5 e
5 6 f
R> exists("DF_TABLE")
[1] TRUE
```

If you connect to the database using a tool such as SQL Developer, you can view DF_ TABLE directly in the database.

Materialize R Data

The `ore.push(data.frame)` function stores an R object in the database as a temporary object, and returns a handle to that object. It converts data frame, matrix, and vector objects to a table, and list, model, and other objectss to a serialized object.

The object that you create exists during the R session; to store the data in a permanent way, see Persist and Manage R Objects in the Database

This example pushes the numerical vector created by the R command `c(1,2,3,4,5)` to *v*, an Oracle R Enterprise object:

```
R> v <- ore.push(c(1,2,3,4,5))
R> class(v)
[1] "ore.numeric"
attr(,"package")
[1] "OREbase"
R> head(v)
[1] 1 2 3 4 5
```

Verify that an ore.frame Exists

`ore.exists()` checks for the existence of an `ore.frame` object in the ORE schema environment. For `ore.exists()` to find an `ore.frame` object the object must have been synchronized with `ore.sync()` first.

The objects available in the ORE environment are not necessarily the same as the database objects. One should not use `ore.exists`() to check for table existence.

For an example, see Example: Load Data.

`ore.exists(name, schema)` has these arguments:

- `name`: A character string specifying the name of the `ore.frame` object
- `schema`: A character string specifying the name of database schema to check

`ore.exists()` returns TRUE if the object exists in the ORE schema and FALSE, if it does not exist.

Drop a Database Table

To drop a table in the database use

```
ore.drop(table="NAMEOFTABLE")
```

For example, these commands drop the table `v` and verifies that it does not exist:

```
R> ore.drop(table="v")
R> ore.exists("v")
[1] FALSE
```

If you drop a table that does not exist, there is no error message.

Pull a Database Table to an R Frame

To pull the contents of an Oracle Database table or view to an in-memory R data frame use `ore.pull(OBJECT_NAME)` for the name of an object returned by `ore.ls()`.

Note: You can pull a table or view to an R frame only if the data can fit into R's memory.

Suppose that your Oracle Database contains the table NARROW. Then `ore.pull()` creates the data frame `df_narrow` from the table NARROW. When you verify that `df_narrow` is a data frame. The warning message appears because the table NARROW is not indexed:

```
R> df_narrow <- ore.pull(NARROW)
Warning message:
ORE object has no unique key - using random order
R> class(df_narrow)
[1] "data.frame"
```

Order in Tables

Almost all data in R is a vector or is based on vectors (vectors themselves, lists, matrices, data frames, and so forth). The elements of a vector have an explicit order. Each element has an index. R code actively uses this order of elements.

However, database-backed relational data (tables and views) does not define any order of rows and thus cannot be directly mapped to R data structures. You can define an explicit order on database tables and views via an ORDER BY clause. The order is usually achieved by having a unique identifier (single- or multi- column key). Ordering in this way can be inefficient and slow for some operations that lead to unnecessary sorting.

row.names<- defines ordering but doesn't actually index a table. The assignment option provides a way to specify a unique column. Initially it supports at least one column but may support multi-column specifications as well. When row.names<- is applied to unordered frames, it returns an error.

You can use the integer indexing created by the ordering infrastructure to perform sampling and partitioning, as described in Sampling and Partitioning.

Suppose that the table NARROW is not indexed. The following example illustrates using row.names to create an indexed table:

```
R> row.names(head(NARROW))
Error: ORE object has no unique key
In addition: Warning message:
ORE object has no unique key - using random order
R>
R> row.names(NARROW) <- NARROW$ID
R>
R> row.names(head(NARROW[,1:3]))
[1] "101501" "101502" "101503" "101504" "101505" "101506"
R>
R> head(NARROW[,1:3])
           ID GENDER AGE
101501 101501   <NA>  41
101502 101502   <NA>  27
101503 101503   <NA>  20
101504 101504   <NA>  45
101505 101505   <NA>  34
101506 101506   <NA>  38
```

Sampling and Partitioning

The ordering (indexing) for tables described in Order in Tables can be used to perform sampling and partitioning.

This section provides examples of

- Indexing
- Sampling
- Random Partitioning

Indexing

R supports powerful constructions using vectors as indices. Oracle R Enterprise supports similar functionality with these differences:

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

This example illustrates indexing:

```
R> tmp <- ASTHMA
R> tmp[c(1L, 2L, 1L),]
Error: ORE object has no unique key
R> rownames(tmp) <- tmp
R> tmp[c(1L, 2L, 1L),]
         CITY ASTHMA COUNT
1|0|65      1      0    65
1|0|65.1    1      0    65
```

```
1|1|35      1      1    35
R> tmp[c(1L, 2L, 1L),]@dataQry
```

Sampling

This code illustrates several sampling techniques:

```
# Generate random data
set.seed(123)
N <- 1000000
mydata <- data.frame(x = rnorm(N, mean = 20, sd = 2),
                     group =
                     sample(letters, N, replace = TRUE,
                            prob = (26:1)/sum(26:1)))
mydata$y <-
    rbinom(N, 1,
           1/(1+exp(-(.5 - 0.25 * mydata$x + .1 * as.integer(mydata$group)))))
MYDATA <- ore.push(mydata)
rm(mydata)

# Create a function that creates random row indices from large tables
mysampler <- function(n, size, replace = FALSE)
{
    #' Random Whole Number Sampler
    #' @param n       number of observations in sample
    #' @param size    total number of observations
    #' @param replace indicator for sampling with replacement
    #' @return numeric vector containing the sample indices
    n    <- round(n)
    size <- round(size)
    if    (n < 0) stop("'n' must be a non-negative number")
    if (size < 1) stop("'size' must be a positive number")
    if (!replace && (n > size))
        stop("'n' cannot exceed 'size' when 'replace = FALSE'")
    if (n == 0)
        numeric()
    else if (replace)
        round(runif(n, min = 0.5, max = size + 0.5))
    else
    {
        maxsamp <- seq(size + 0.5, by = -1, length.out = n)
        samp <- round(runif(n, min = 0.5, max = maxsamp))
        while(length(bump1 <- which(duplicated(samp))))
            samp[bump1] <- samp[bump1] + 1
        samp
    }
}

# Data set and sample size
N <- nrow(MYDATA)
sampleSize <- 500

# 1. Simple random sampling
srs <- mysampler(sampleSize, N)
simpleRandomSample <- ore.pull(MYDATA[srs, , drop = FALSE])

# 2. Systematic sampling
systematic <- round(seq(1, N, length.out = sampleSize))
systematicSample <- ore.pull(MYDATA[systematic, , drop = FALSE])

# 3. Stratified sampling
```

```
stratifiedSample <-
    do.call(rbind,
            lapply(split(MYDATA, MYDATA$group),
                   function(y)
                   {
                       ny <- nrow(y)
                       y[mysampler(sampleSize * ny/N, ny), , drop = FALSE]
                   }))

# 4. Cluster sampling
clusterSample <- do.call(rbind, sample(split(MYDATA, MYDATA$group), 2))

# 5a. Accidental/Convenience sampling (via row order access)
convenientSample1 <- head(MYDATA, sampleSize)

# 5b. Accidental/Convenience sampling (via hashing)
maxHash <- 2^32 # maximum allowed in ore.hash
convenient2 <- (ore.hash(rownames(MYDATA), maxHash)/maxHash) <= (sampleSize/N)
convenientSample2 <- ore.pull(MYDATA[convenient2, , drop = FALSE])
Random
```

Random Partitioning

For Oracle R Enterprise random partitions can be generated in the transparency layer by adding a partition or group column to an ore.frame object in the following manner:

```
nrowX <- nrow(x)
x$partition <- sample(rep(1:k, each = nrowX/k, length.out = nrowX), replace =
TRUE)
```

After these partitions have been joined to the original data set, the ore.groupApply function can be used to perform the little bootstraps:

```
results <- ore.groupApply(x, x$partition, function(y) {...}, parallel = TRUE)
```

Persist and Manage R Objects in the Database

R objects exist for the duration of the current session, unless they are explicitly saved. For example, if you build a model in a particular R session, the model is not available when the session is closed, unless the model was explicitly saved.

Oracle R Enterprise supports persistence for R objects onto the database.

Persistence provides these advantages:

- You can access the same R and Oracle R Enterprise object (for example, a model) among different R sessions.
- You can build a model in R and use it for prediction and scoring in embedded Oracle R Enterprise.

Oracle R Enterprise creates **datastores** to contain persisted objects.

Persisted objects reside in a datastore. The following Oracle R Enterprise functionality allows you manage persistence:

- ore.save()
- ore.load()
- ore.delete()
- ore.datastore()

- ore.datastoreSummary()

ore.save()

`ore.save()` saves an R object or a list of R objects to the specified datastore in the connected database in the current user's schema:

```
ore.save({...}, list = character(0), name, envir = parent.frame(), overwrite =
FALSE, append = FALSE, description = character(0)))
```

The parameters for `ore.save()` are as follows:

- `{...}` is the list of R objects to save; the names of the objects to be saved (as symbols or character strings)
- `list` is a character vector containing the names of objects to be saved
- `envir` is the environment to search for objects to be saved
- `overwrite` is a logical value specifying whether to overwrite the datastore if already exists; the default is FALSE (do not overwrite)
- `name` is the name of the datastore; `name` must be specified
- `description` is a comment describing the datastore
- `append` is a logical value specifying whether to append objects to the datastore if already exists; the default is FALSE (do not append)

Examples of ore.save()

Save all objects in the current workspace environment to the datastore `ds_1` in the user's current schema:

```
ore.save(list=ls(), name="ds_1", description = "example datastore")
```

Overwrite existing datastore `ds_2` with objects `x`, `y`, and `z` in the current workspace environment:

```
ore.save(x, y, z, name="ds_2", overwrite=TRUE)
```

Add objects `x`, `y`, and `z` in the current workspace environment to the existing datastore `ds_3` (that is append the objects to the datastore):

```
ore.save(x, y, z, name="ds_3", append=TRUE)
```

ore.load()

`ore.load()` loads all of the R objects stored in a specified datastore in the current user schema in the connected database to R:

```
ore.load(name, list = character(0), envir = parent.frame())
```

The parameters for `ore.load()` are

- `name` is a character string specifying the name of datastore to load the objects from; you must specify a name
- `list` is a character vector containing the names of objects to be loaded
- `envir` is the R environment that objects are loaded to

`ore.load()` returns a character vector containing the names of objects loaded from the datastore.

Examples of ore.load()

Load all objects in the datastore `ds_1`:

```
ore.load("ds_1")
```

Load just the objects `x`, `y`, and `z` from datastore `ds_1`:

```
ore.load("ds_1", list=c("x", "Y", "z"))
```

ore.delete()

`ore.delete()` deletes the specified datastore (and all of the R objects in it) from the current user schema in the connected database:

```
ore.delete(name)
```

The parameter for `ore.delete()` is

- `name` is a character string specifying the name of datastore to delete; you must specify a name

Use ore.datastore() to list the datastores that exist in the user's Oracle Database schema.

Example of ore.delete()

Delete the datastore `ds_1` from the user's current schema:

```
ore.delete("ds_1")
```

ore.datastore()

`ore.datastore()` lists the datastores and basic information about each datastore in the current schema:

```
ore.datastore(name, pattern)
```

The parameters for `ore.datastore()` are

- `name` is a character string specifying the name of datastore to list
- `pattern` is a regular expression character string specifying the names of the datastores to list.

`ore.datastore()` lists information about the datastore with name specified in `name` or information about the datastores whose names match the regular expression specified in `pattern`.

If neither `name` nor `pattern` is provided, `ore.datastore()` returns information about all datastores in user's schema.

Either `name` or `pattern` can be specified but not both.

`ore.datastore()` returns a `data.frame` object with these columns:

- `datastore.name` name of the datastore
- `object.count` number of objects in the datastore identified by `datastore.name`
- `size` size of the datastore in bytes
- `creation.date` date of datastore creation
- `description` comment for datastore (comment is specified in the `description` parameter of `ore.save`)

Each row of the `data.frame` lists one datastore. Rows are sorted by column `datastore.name` in alphabetical order.

Example of ore.datastore()

List all of the datastores in the connected schema:

```
ore.datastore()
```

ore.datastoreSummary()

`ore.datastoreSummary()` returns a `data.frame` that lists the names and summary information for the R objects saved in the specified datastore in the schema in the connected database:

```
ore.datastoreSummary(name)
```

The parameter for `ore.datastoreSummary()` is

- `name` is a character string specifying the name of datastore to summarize; you must specify a name

If the specified datastore does not exist, an error is returned.

`ore.datastoreSummary()` returns a `data.frame` object with these columns:

- `object.name` is the name of the R object
- `class.name` is the class name of the R object
- `size` is the size of the R object in bytes
- `length` is the length of the R object
- `row.count` is the number of rows for the R object
- `col.count` is number of columns of the R object

Each row of the `data.frame` lists one R object. Rows are sorted by column `datastore.name` in alphabetical order.

Example of ore.datastoreSummary()

List summary information for all of the R objects in the datastore `ds_1`:

```
ore.datastoreSummary(name = "ds_1")
```

Using R with Oracle R Enterprise Data Types

The following examples illustrate using R with Oracle R Enterprise data types:

- **Simple column and row selection in R**:

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

```
R> iris_projected = IRIS[, c("PETAL_LENGTH", "SPECIES")]
R> head (iris_projected)
  PETAL_LENGTH SPECIES
0          1.4  setosa
1          1.4  setosa
2          1.3  setosa
3          1.5  setosa
4          1.4  setosa
5          1.7  setosa
```

- **Database JOIN using R**:

```
df1 <- data.frame(x1=1:5, y1=letters[1:5])
df2 <- data.frame(x2=5:1, y2=letters[11:15])
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
# Create database objects to correspond to in-memory R objects df1 and df2
ore.df1 <- ore.create(df1, table="DF1")
ore.df2 <- ore.create(df2, table="DF2")
# Compare results
R> merge (DF1, DF2, by.x="X1", by.y="X2")
  X1 Y1 Y2
0  1  a  o
1  2  b  n
2  3  c  m
3  4  d  l
4  5  e  k
```

- **Database aggregation using R**:

```
# Push built-in data set iris to database
ore.create(iris, table="IRIS")
aggdata <- aggregate(IRIS, by = list(IRIS$SPECIES), FUN = summary)
class(aggdata)
head(aggdata)
```

- **Data formatting and creating derived columns in R**

  Note that adding derived columns does not change the database table. See Derived Columns in Oracle R Enterprise.

```
diverted_fmt <- function (x) {
ifelse(x==0, 'Not Diverted',
ifelse(x==1, 'Diverted',''))
}
cancellationCode_fmt <- function(x) {
ifelse(x=='A', 'A CODE',
ifelse(x=='B', 'B CODE',
ifelse(x=='C', 'C CODE',
ifelse(x=='D', 'D CODE', 'NOT CANCELLED'))))
}
delayCategory_fmt <- function(x) {
ifelse(x>200,'LARGE',
ifelse(x>=30,'MEDIUM','SMALL'))
}
zscore <- function(x) {
(x-mean(x,na.rm=TRUE))/sd(x,na.rm=TRUE)
```

```
# ONTIME_S is a database table
ONTIME_S$DIVERTED <- diverted_fmt(DIVERTED)
ONTIME_S$CANCELLATIONCODE <- cancellationCode_fmt(CANCELLATIONCODE)
ONTIME_S$ARRDELAY <- delayCategory_fmt(ARRDELAY)
ONTIME_S$DEPDELAY <- delayCategory_fmt(DEPDELAY)
ONTIME_S$DISTANCE_ZSCORE <- zscore(DISTANCE)
```

Derived Columns in Oracle R Enterprise

When you add derived columns using Oracle R Enterprise, the derived columns do not affect the underlying table in the database. A SQL query is generated that has the additional derived columns in the select list, but the table is not changed.

Using CRAN Packages with Oracle R Enterprise

This example illustrates using Oracle R Enterprise with a standard R package downloaded from CRAN:

- Build and Use a Regression Model

Build and Use a Regression Model

This example illustrates building a regression model using a CRAN package. You can prepare the data used for training in the database (filtering out observations that are not of interest, selecting attributes, imputing missing values, and so forth). Suppose that the preprocessed data is in the table ONTIME_S_PREPROCESSED_SUBSET. Then pull the prepared training set (which is usually small enough to fit in desktop R memory) into the R client to execute the model build.

You can use the resulting model to score (predict) large numbers of rows, in parallel, in Oracle Database. The data are stored in `ONTIME_S_FINAL_DATA_TO_BE_SCORED`.

Note that scoring is a trivially parallelizable operation because one row can be scored independent of and in parallel with another row. The model built on the desktop is shipped to the database to perform scoring on vast numbers of rows in the database.

The computations are divided into these steps:

1. Build a model in the desktop:

```
dat <- ore.pull(ONTIME_S_PREPROCESSED_SUBSET)
mod <- glm(ARRDELAY ~ DISTANCE + DEPDELAY, dat)
mod
summary(mod)
```

2. Score in-parallel in the database using embedded R:

```
prd <- predict(mod, newdata=ONTIME_S_FINAL_DATA_TO_BE_SCORED)
class(prd)
# Add predictions as a new column
res <- cbind(newdat, PRED = prd)
head(res)
```

R provides many other ways to build regression models, such as `lm()`.

For other ways to build regression models, see Oracle R Enterprise Versions of R Models and In-Database Predictive Models in Oracle R Enterprise.

Oracle R Enterprise Database-Embedded R Engine

The embedded R engine in Oracle Database allows R users to off load desktop calculations that may require either more resources such as those available to Oracle Database or database-driven data parallelism. The embedded R engine also executes R scripts embedded in SQL or PL/SQL programs (lights-out processing).

These examples illustrate using Oracle R Enterprise embedded R engine with standard R packages downloaded from CRAN:

- Perform R Computation in Oracle Database
- Build a Series of Regression Models Using Data Parallelism

Perform R Computation in Oracle Database

This example illustrates **off loading R computation** to execute in the embedded R engine. To off load an R computation, simply include the R code within a closure (that is, `function() {}`) and invoke `ore.doEval()`. `ore.doEval()` schedules execution of the R code with the database-embedded R engine and returns the results back to the desktop for continued analysis:

```
library(biglm)
mod <- ore.doEval(
   function() {
      library(biglm)
      dat <- ore.pull(ore.get("ONTIME_S"))
      mod <- biglm(ARRDELAY ~ DISTANCE + DEPDELAY, dat)
      mod
     }, ore.connect = TRUE);
print(mod)
mod=ore.pull(mod)
print(mod)
```

Build a Series of Regression Models Using Data Parallelism

This example illustrates **database-driven data parallelism** at work in building a series of regression models using a CRAN package. One model is built per unique value of a factor. The database orchestrates the parallel and concurrent building of the models, one per factor and brings the list of all models built to the user desktop for further analysis:

```
modList <- ore.groupApply(
  # Organize input to the R script - This is always an Oracle R Enterprise
  # data frame
   X=ONTIME_S,
  # Specify the grouping column. Here we request one model per unique value of
  # ONTIME_S$DEST
   INDEX=ONTIME_S$DEST,
  # Model building code goes inside the closure. Input and grouping
  # conditions can be referenced as parameters to the function
    function(x) {
     library(biglm)
     biglm(ARRDELAY ~ DISTANCE + DEPDELAY, x)
     });

   modList_local <- ore.pull(modList)
# Print the model for just one destination - BOSTON
summary(modList_local$BOS)
```

Oracle R Enterprise Examples

Oracle R Enterprise is shipped with a collection of demos, examples that illustrate how to use Oracle R Enterprise. These examples are a collection of self-contained R scripts.

Most of the sample programs use the data frame `iris`, which is included in the R distribution. `iris` is loaded into a table as described in Load a Data Frame to a Table.

The rest of this section describes two examples in detail and includes a list of all of the examples:

- Load a Data Frame to a Table
- Handle NULL Values Using airquality
- Oracle R Enterprise Demos

Load a Data Frame to a Table

Start R, load the ORE packages via `library(ORE)`, and then connect to the database.

Follow these steps to load an R data frame to a database table:

1. This example uses the R data set `iris`.

   The `iris` data set is located in the datasets package that is part of the R distribution:

   ```
   R> find("iris")
   [1] "package:datasets"
   ```

   Use the R command `class` to verify that `iris` is an R data frame:

   ```
   R> class(iris)
   [1] "data.frame"
   ```

   `iris` consist of measurements of parts of iris flowers. Use the R command `head` to see a small sample of the data in `iris`.

   ```
   R> head(iris)
         Sepal.Length Sepal.Width Petal.Length Petal.Width Species
   1          5.1         3.5          1.4         0.2  setosa
   2          4.9         3.0          1.4         0.2  setosa
   3          4.7         3.2          1.3         0.2  setosa
   4          4.6         3.1          1.5         0.2  setosa
   5          5.0         3.6          1.4         0.2  setosa
   6          5.4         3.9          1.7         0.4  setosa
   ```

2. Now load the data frame `iris` into the database that you are connected to.

   Suppose that the database table version of `iris` is named IRIS_TABLE. Drop IRIS_TABLE to make sure that no table of this name exists in the connected schema:

   ```
   ore.drop(table = "IRIS_TABLE")
   ```

   If IRIS_TABLE doesn't exist, you do not get a message.

3. Now create a database table with the data contained in `iris`:

   ```
   ore.create(iris, table = "IRIS_TABLE")
   ```

   Use `ore.ls()` to verify that the table was created:

   ```
   R> ore.ls()
   [1] "IRIS_TABLE" "NARROW"     "ONTIME_S"
   ```

4. IRIS_TABLE is a database-resident table with just metadata on the R side:

```
R> class(IRIS_TABLE)
[1] "ore.frame"
attr(,"package")
[1] "OREbase"
```

5. Use `head` to see the column names and the first few values in IRIS_TABLE:

```
R> head(IRIS_TABLE)
  SEPAL_LENGTH SEPAL_WIDTH PETAL_LENGTH PETAL_WIDTH SPECIES
0          5.1         3.5          1.4         0.2  setosa
1          4.9         3.0          1.4         0.2  setosa
2          4.7         3.2          1.3         0.2  setosa
3          4.6         3.1          1.5         0.2  setosa
4          5.0         3.6          1.4         0.2  setosa
5          5.4         3.9          1.7         0.4  setosa
```

6. Use `class` to see the data type of the column SPECIES.

```
R > class(IRIS_TABLE$SPECIES)
[1] "raw"
```

You can use R functions to analyze the data in the table. Here are some simple examples taken from the example `basic.R`:

- Use `unique` to get a list of the unique entries in a column. This example finds the unique SPECIES:

```
R> unique(IRIS_TABLE$SPECIES)
[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
```

- Find the minimum, maximum, and mean of PETAL_LENGTH:

```
R> min(IRIS_TABLE$PETAL_LENGTH)
[1] 1
R> max(IRIS_TABLE$PETAL_LENGTH)
[1] 6.9
R> mean(IRIS_TABLE$PETAL_LENGTH)
[1] 3.758
```

If you need information about an R function, use the command `help(function-name).`

Handle NULL Values Using airquality

Oracle Database has logical values: TRUE, FALSE, NULL. There is a 3x3 table that defines truth values for propositions with AND and OR. NULL is treated as an unknown value. For some operations the result is either deterministic (for example TRUE OR NULL) or unknown (TRUE AND NULL). If logical values are used in a WHERE clause, only rows with the condition TRUE are selected; FALSE and NULL are ignored. R, on the other hand, keeps TRUE and NA. Rows with NA are selected with value NA.

The option `ore.na.extract` controls whether NAs are selected or not. The default is to treat NA as SQL treats FALSE.

The demo `nulls.R` is the only sample that does not use `iris` as data. `nulls.R` compares the handling of NULLs in SQL with the handling of NAs in R.

In R, NA is a logical constant of length 1 which contains a missing value indicator. In the database, null refers to the absence of a value in a column of a row. Nulls indicate missing, unknown, or inapplicable data.

Follow these steps to understand the demo `nulls.R`:

1. This demo uses the data frame `airquality`. Verify that the data set is a data frame and look at the few rows of the data frame:

   ```
   R> class(airquality)
   [1] "data.frame"
   R> head(airquality)
     Ozone Solar.R Wind Temp Month Day
   1    41     190  7.4   67     5   1
   2    36     118  8.0   72     5   2
   3    12     149 12.6   74     5   3
   4    18     313 11.5   62     5   4
   5    NA      NA 14.3   56     5   5
   6    28      NA 14.9   66     5   6
   ```

2. Load `airquality` into the database as "AIRQUALITY":

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

   Use `ore.ls()` to verify that the table was created. If you wish, use `class(AIRQUALITY)` to verify that `AIRQUALITY` is a database-resident table with just metadata on the R side.

3. Examine how R handles NAs. Return all observations where ozone < 30 is TRUE:

   ```
   R> nrow(airquality[airquality$Ozone < 30,])
   [1] 92
   ```

   Compare this with the results when NAs are explicitly excluded:

   ```
   R> nrow(airquality[airquality$Ozone < 30 & !is.na(airquality$Ozone),])
   [1] 55
   ```

4. The default behavior for SQL tables is to exclude NULLS in output:

   ```
   nrow(AIRQUALITY[AIRQUALITY$OZONE < 30,])
   [1] 55
   ```

   To handle NULLs the same way that R handles NA, request the behavior explicitly:

   ```
   options(ore.na.extract = TRUE)
   nrow(AIRQUALITY[AIRQUALITY$OZONE < 30,])
   [1] 92
   ```

Oracle R Enterprise Demos

These scripts have been added as demos to the ORE package.

To list all of the demos included with Oracle R Enterprise, type

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

To run one of these scripts, specify the name of the demo in a `demo` function call. For example, to run `aggregate.R`, type

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

These demos are shipped with Oracle R Enterprise:

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

4 Oracle R Enterprise Statistical Functions

This chapter describes Oracle R Enterprise functions that perform most common or base statistical procedures. These functions are designed to help users who are converting from commercially available products to Oracle R Enterprise.

Oracle R Enterprise provides these collections of functions:

- ore.corr
- ore.crosstab
- ore.extend
- ore.freq
- ore.rank
- ore.sort
- ore.summary
- ore.univariate

Also of interest are `ore.lm()`, `ore.stepwise()`, and `ore.neural()` described in Oracle R Enterprise Versions of R Models.

The use of the functions is illustrated with examples. Most of the examples use the same data, described in Data for Examples.

Data for Examples

Most of the examples use the table NARROW.

NARROW is an `ore.frame` with 9 columns:

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

Only some of the columns are numeric.

ore.corr

`ore.corr` has these characteristics:

- Performs correlation analysis across numeric columns in an ore.frame
- Supports partial correlations with a control column
- Enables aggregations prior to correlations
- Allows post-processing of results and integration into an R code flow

The output of `ore.corr` can be made to conform to output of the R `cor()` function; this allows the output of ore.corr to be post-processed by any R function or graphics.

See ore.corr Parameters for syntax and output and ore.corr Examples for examples.

ore.corr Parameters

`ore.corr` has these parameters:

- **data**: The data for which to compute correlation coefficients as an ore.frame.
- **var**: The numeric column(s) of **data** for which to build correlation matrix
- **group.by**: Indicates the correlation matrices to calculate; `ore.corr` calculates as many correlation matrices as unique values in **group.by** columns; default value is NULL
- **weight**: A column of the data whose numeric values provide a multiplicative factor for **var** columns; default value is NULL
- **partial**: Columns of data to use as control variables for partial correlation; default value is NULL
- **stats**: The method of calculating correlations; one of pearson (default), spearman, kendall

`ore.corr` returns an `ore.frame` as output in all cases except when **group.by** is used. If **group.by** is used, returns an Oracle R Enterprise `list` object.

ore.corr Examples

These examples show how to use `ore.corr`:

- Basic Correlation Calculations
- Partial Correlation
- Create Several Correlation Matrices
- Visualization of Correlations

These examples use the NARROW data set; for more information, see Data for Examples.

Basic Correlation Calculations

Before you can use `ore.corr`, you must project out all non-numerical values:

```
R> names(NARROW)
 [1] "ID"             "GENDER"      "AGE"     "MARITAL_STATUS"
"COUNTRY"        "EDUCATION"      "OCCUPATION"
 [8] "YRS_RESIDENCE"  "CLASS"           "AGEBINS"
R> NARROW=NARROW[,c(3,8,9)]
```

Now calculate correlation in several ways:

```
R> x=ore.corr(NARROW,var='AGE,YRS_RESIDENCE,CLASS')
#Calculate using Spearman
```

```
R> x=ore.corr(NARROW,var='AGE,YRS_RESIDENCE,CLASS', stats='spearman')
# Calculate using Kendall
R> x=ore.corr(NARROW,var='AGE,YRS_RESIDENCE,CLASS', stats='kendall')
```

Partial Correlation

Use the version of NARROW with non-numeric values that was created in Basic Correlation Calculations.

Calculate partial correlation using Spearman's methods:

```
R> x=ore.corr(NARROW,var='AGE,YRS_RESIDENCE,CLASS', stats='spearman',
partial='GENDER')
```

Create Several Correlation Matrices

Use the version of NARROW with non-numeric values that was created in Basic Correlation Calculations.

Create several correlation matrices and then convert the output so that it is compatible with R output:

```
x <- ore.corr(IRIS, var = "Sepal.Length, Sepal.Width, Petal.Length", partial =
"Petal.Width", group.by = "Species")
R> class(x)
[1] "list"
```

Visualization of Correlations

If you calculate several matrices, you can use R packages to visualize them.

ore.crosstab

Cross tabulation is a statistical technique that finds an interdependent relationship between two tables of values.

`ore.crosstab` enables cross column analysis of an ore.frame. This function is a sophisticated variant of the R `table()` function.

`ore.crosstab` must be performed before frequency analysis is done using ore.freq.

You can extend the cross tab calculation with various sums as described in ore.extend.

`ore.crosstab` is written in R. The function is mapped to SQL that gets executed at the database server.

See ore.crosstab Parameters for syntax and output and ore.crosstab Examples for examples.

You can use ore.extend to augment cross tabulation.

ore.crosstab Parameters

`ore.crosstab` has these parameters:

- **expr**: The cross tabulation definition

```
[COLUMN_SPEC] ~ COLUMN_SPEC [*<WEIGHTING COLUMN>] [/<GROUPING COLUMN>]
 [^<STRATIFICATION COLUMN>] [|ORDER_SPECIFICATION]
COLUMN_SPEC is <column-name>[+COLUMN_SET][+COLUMN_RANGE]
COLUMN_SET is <column_name>[+COLUMN_SET]
```

```
COLUMN_RANGE is <FROM COLUMN>-<TO COLUMN>
```

where

```
COLUMN_SPEC is <column>[+COLUMN_SET][+COLUMN_RANGE]
COLUMN_SET is <column>[+COLUMN_SET]
COLUMN_RANGE is (<from column>-<to column>)
ORDER_SPECIFICATION is one of [-]NAME, [-]DATA, [-]FREQ, or INTERNAL
```

The stratification column is used to cluster, or group, data. When used, the values contribute to the ORE$STRATA column of the resulting cross-tabulated table.

- **data**: The `ore.frame` containing the data to cross tabulate
- **grouping column**: Calculate as many cross tabulations as unique values in grouping columns; default value is NULL
- **order**: Define optional sorting of output data. Specify [-]NAME to sort by tabulation columns, [-]FREQ to sort by frequency counts in table. Unspecified order is the most efficient. The optional '-' reverses the order direction.
- **weights**: Column of the data that indicates the frequency of the corresponding row; default value is NULL
- **partial**: Columns of data used as control variables for partial correlation; default value is NULL

`ore.crosstab` returns an `ore.frame` as output in all cases except when multiple tables are created. If multiple tables are created, ore.crosstab returns an Oracle R Enterprise `list` object.

ore.crosstab Examples

These examples illustrate use of `ore.crosstab`:

- Single-Column Frequency Table
- Analyze Two Columns
- Weighting Rows
- Order Rows in the Cross Tabulated Table
- Analyze Three or More Columns
- Specify a Range of Columns
- Produce One Cross Table for Each Value of Another Column
- Augment Cross Tabulation with Stratification
- Custom Binning Followed by Cross Tabulation
- ore.extend

These examples use the NARROW data set; for more information, see Data for Examples.

Single-Column Frequency Table

The most basic use case is to create a single column frequency table. The following command filters NARROW grouping by GENDER:

```
R> ct = ore.crosstab(~AGE, data=NARROW)
R> ct
```

Analyze Two Columns

This command analyses AGE by GENDER and AGE by CLASS:

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

Weighting Rows

To weight rows, include a count based on another column; this example weights values in AGE and GENDER using values in YRS_RESIDENCE:

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

Order Rows in the Cross Tabulated Table

There are several possibilities:

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

Here are two examples:

```
R> ct = ore.crosstab(AGE~GENDER|FREQ, data=NARROW)
R> head(ct)
  AGE GENDER ORE$FREQ ORE$STRATA ORE$GROUP

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

Analyze Three or More Columns

This is similar to what the SQL GROUPING SETS clause accomplishes:

```
 R> ct = ore.crosstab(AGE+COUNTRY~GENDER, NARROW)
```

Specify a Range of Columns

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

```
R> names(NARROW)
[1] "ID"            "GENDER"         "AGE"            "MARITAL_STATUS"
[5] "COUNTRY"       "EDUCATION"      "OCCUPATION"     "YRS_RESIDENCE"
[9] "CLASS"
```

Since AGE, MARITAL_STATUS and COUNTRY are successive columns, you can simply use

```
ct = ore.crosstab(AGE-COUNTRY~GENDER, NARROW)
```

An equivalent version is

```
ct = ore.crosstab(AGE+MARITAL_STATUS+COUNTRY~GENDER, NARROW)
```

Produce One Cross Table for Each Value of Another Column

This command produces one cross table (AGE, GENDER) for *each* unique value of another column COUNTRY:

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

You can extend this to more than one column. For example, this command produces one (AGE, EDUCATION) table for each unique combination of (COUNTRY, GENDER):

```
R> ct = ore.crosstab(AGE~EDUCATION/COUNTRY+GENDER, data=NARROW)
```

Augment Cross Tabulation with Stratification

All of the above cross tabs can be augmented with stratification. For example,

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

The command in this example is the same as

```
ct = ore.crosstab(AGE~GENDER, NARROW, strata="CLASS")
```

Custom Binning Followed by Cross Tabulation

First bin AGE, then calculate cross tabulation for GENDER and the bins:

```
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)
```

ore.extend

The cross tabulation produced using ore.crosstab can be further augmented with these three basic statistics:

- Row and column sums

  ```
  crosstab = ore.extend.sum(crosstab)
  ```

- Cumulative sums for each cell of the table

  ```
  crosstab = ore.extend.cumsum(crosstab)
  ```

- Total for the entire table

  ```
  crosstab = ore.extend.total(crosstab)
  ```

The following example illustrates `ore.extend`:

```
R> ct <- ore.crosstab(GENDER~CLASS, NARROW)
R> ore.freq(ct)
  METHOD    FREQ DF PVALUE              DESCR GROUP
1 PCHISQ 72.4241  1      0 Pearson Chi-Square     1
```

ore.freq

`ore.crosstab` must be performed before frequency analysis is done using ore.freq.

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

- 1-way cross tables

  Goodness-of-fit tests for equal proportions or specified null proportions, confidence limits and tests for equivalence.
- 2-way cross 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 tables
  - N 2-way cross tables
  - Statistics across and within strata

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

See ore.freq Parameters for syntax and output and ore.freq Examples for examples.

ore.freq Parameters

`ore.freq` supports these parameters:

- **crosstab**: The `ore.frame` object that is output from `ore.crosstab()`
- **stats**: List of statistics required; these statistics are supported:
  - Chi Square: AJCHI, LRCHI, MHCHI, PCHISQ
  - Kappa: KAPPA, WTKAP
  - Lambda: LAMCR, LAMRC, LAMDAS
  - Correlation: KENTB,PCORR, SCORR
  - Stuart's Tau, Somers: D|C, STUTC, SMDCR,SMDRC
  - Fisher's, Cochran's Q, FISHER, COCHQ
  - Odds Ratio: OR, MHOR, LGOR
  - Relative Risk: RR,MHRR,ALRR
  - Others: MCNEM, PHI, CRAMV, CONTGY, TSYM, TREND, GAMMA

  The default value is NULL.
- **Params**: Control parameters specific to the statistical function specified in **stats**:
  - SCORE: TABLE | RANK | RIDIT | MODRIDIT
  - ALPHA: *number*
  - WEIGHTS: *number*

  The default value is NULL.
- **skip.missing**: Skip cells with missing values in the cross table (TRUE or FALSE); default value is FALSE
- **skip.failed**: Return immediately if a statistical test required fails on the cross table because it is found to be in-applicable to the table (TRUE or FALSE); default value is FALSE

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

ore.freq Examples

These examples use the NARROW data set; for more information, see Data for Examples.

Before you use `ore.freq`, you must calculate cross tabs.

For example:

```
R> ct = ore.crosstab(~GENDER, NARROW)
R> ore.freq(ct)
  METHOD     FREQ DF PVALUE      DESCR GROUP
0   PCHI 161.9377  1      0 Chi-Square     1
```

ore.rank

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

`ore.rank` 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

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

See ore.rank Parameters for syntax and ore.rank Examples for examples.

ore.rank Parameters

`ore.rank` supports these parameters:

- **data**: The ore.frame containing the data to rank
- **var**: The numeric columns in **data** to rank
- **desc**: If `desc=TRUE`, rank in descending order; otherwise, rank in ascending order. (The default is to rank in ascending order.)
- **groups**: Partition rows into #groups based on ranks. For percentiles, `groups=100`, For deciles, `groups=10`, For quartiles, `groups=4`.

  The default value is NULL.
- **group.by**: Rank each group identified by `group.by` columns separately

  The default value is NULL.
- **ties**: Specify how to treat ties. Ways to treat ties are assign the largest of, or smallest of, or mean of corresponding ranks to tied values

  The default value is NULL.
- **fraction**: The rank of a column value divided by the number of non-missing column values; the default value is FALSE.

  Use with `nplus1` to estimate the cumulative distribution function
- **nplus1**: `fraction` plus 1, that is, 1 plus the rank of a column value divided by the number of non-missing column values; the default value is FALSE.

Use with `fraction` to estimate the cumulative distribution function.

- **percent**: **fraction** converted to a percent value, that is f**raction** * 100.

`ore.rank` 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).

ore.rank Examples

These examples illustrate using `ore.rank`:

- Rank Two Columns
- Handle Ties
- Rank Within Groups
- Partition into Deciles
- Estimate Cumulative Distribution Function

These examples use the NARROW data set; for more information, see Data for Examples.

Rank 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 (ascending):

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

Handle Ties

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

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

Rank Within Groups

This example ranks the two columns AGE and CLASS and ranks the values according to COUNTRY:

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

Partition into Deciles

This example ranks the two columns AGE and CLASS and partitions the columns into deciles (10 partitions):

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

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

Estimate Cumulative Distribution Function

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

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

Score 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). `savage` calculates exponential scores and `blom` calculates normal scores:

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

ore.sort

`ore.sort` enables flexible sorting of a data frame along one or more columns specified in a **by** clause.

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

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

See ore.sort Parameters for syntax and ore.sort Examples for examples.

ore.sort Parameters

`ore.sort` supports these parameters:

- **data**: An ore.frame containing the data to be sorted; **required**
- **by**: The column(s) in **data** by which to sort the data; **required**
- **stable**: Maintains relative order within sorted group (TRUE or FALSE); default value is FALSE
- **reverse**: Reverses collation order for character variables (TRUE or FALSE); default value is FALSE
- **unique.keys**: Deletes observations with duplicate values in the columns being sorted (TRUE or FALSE); default value is FALSE
- **unique.data**: Deletes observations with duplicate values in all columns (TRUE or FALSE); default value is FALSE

data and **by** are required parameters; all other parameters are optional.

`ore.sort` returns an `ore.frame`.

ore.sort Examples

The following examples illustrate using `ore.sort`:

- Sort Columns in Descending Order
- Sort Different Columns in Different Orders
- Sort and Return One Row per Unique Value

- Remove Duplicate Columns
- Remove Duplicate Columns and Return One Row per Unique Value
- Preserve Relative Order in Output

Most of these examples use the NARROW data set; for more information, see Data for Examples. There are also Examples Using ONTIME_S.

Sort Columns in Descending Order

Sort the columns AGE and GENDER in descending order:

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

Sort Different Columns in Different Orders

Sort AGE in descending order and GENDER in ascending order:

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

Sort and Return One Row per Unique Value

Sort by AGE and keep one row per unique value of AGE:

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

Remove Duplicate Columns

Sort by AGE and remove duplicate rows:

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

Remove Duplicate Columns and Return One Row per Unique Value

Sort by AGE. Also remove duplicate rows, and return one row per unique value of AGE:

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

Preserve Relative Order in Output

Maintain the relative order in the sorted output:

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

Examples Using ONTIME_S

These examples use the ONTIME_S airline data that is installed when you install Oracle R Enterprise:

- Sort ONTIME_S by airline name in descending order and departure delay in ascending order:

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

- Sort ONTIME_S by airline name and departure delay and select one of each combination (that is, return a unique key):

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

```
    unique.key=TRUE)
```

ore.summary

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

`ore.summary` supports these statistics:

- Mean, min, max, 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

`ore.summary` provides a relatively simple syntax compared with SQL queries for the same results.

See ore.summary Parameters for syntax and ore.summary Examples for examples.

ore.summary Parameters

`ore.summary` supports these parameters:

- **data**: Data to aggregate as an `ore.frame`
- **class**: Column(s) of **data** to aggregate (that is, SQL GROUP BY); default value is NULL
- **var**: Column(s) of **data** on which to apply statistics functions (SQL SELECT list)
- **stats**: List of statistics functions to be applied on **var** columns

  mean, min, max, cnt, n, nmiss, css, uss, cv, sum, sumwgt, range, stddev, stderr, var, t, kurt, skew, p1, p5, p10, p25, p50, p75, p90, p95, p99, qrange, lclm, rclm, clm, mode that can be requested on **var** columns.

  The default values are n, mean, min, max.
- **weight**: A column of **data** whose numeric values provide a multiplicative factor for **var** columns
- **maxid**, **minid**: For each group lists maximum or minimum value from other columns in **data**; default value is NULL
- **ways**: Restricts output to only certain grouping levels of the **class** variables; default value is NULL
- **group.by**: Column(s) of data to stratify summary results across; default value is NULL
- **order**: Defines optional sorting of output data. Specify [-]NAME to sort by tabulation columns, [-]FREQ to sort by frequency counts in table. Unspecified order is the most efficient. The optional '-' reverses the order direction
- **_FREQ**: Frequency, number of observations in a group

- **_TYPE**: Identifies the grouping, binary code based
- **_LEVEL**: Identifies number of variables used in grouping

`ore.summary` returns an `ore.frame` as output in all cases except when a **group.by** clause is used. If a **group.by** clause is used, `ore.summary` returns a list of `ore.frame` objects, one frame per stratum.

ore.summary Examples

These examples illustrate the use of `ore.summary`:

- Calculate Default Statistics
- Skew and t Test
- Weighted Sum
- Two Separate Group By Columns

These examples use the NARROW data set; for more information, see Data for Examples.

Calculate Default Statistics

This example calculates mean, min, max for columns AGE and CLASS and rolls up (aggregates) GENDER:

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

Skew and t Test

This example calculates skew for skew of AGE as column A and the t-test for CLASS as column B:

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

Weighted Sum

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

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

Two Separate Group By Columns

Group CLASS by GENDER and MARITAL_STATUS:

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

All Possible Group By

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

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

ore.univariate

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

`ore.univariate` provides these statistics:

- All statistics reported by ore.summary
- Signed rank test, Student's t-test
- Extreme values reporting

See ore.univariate Parameters for syntax and ore.univariate Examples for examples.

ore.univariate Parameters

`ore.univariate` supports these parameters:

- **data**: The data to aggregate as an ore.frame
- **var**: Numerical column(s) of **data** to analyze
- **weight**: A column of the *data* whose numeric values provide a multiplicative factor for **var** columns; the default value is NULL
- **stats**: Optional specification of a subset of statistics to calculate and display:
  - moments: n, sumwgt, mean, sum, stddev, var, skew, kurt., uss.css.cv, stderr
  - measures: mean, stddev, median, var, mode, range, iqr
  - quantiles: p100, p99, p95, p90, p75, p50, p25, p10, p5, p1, p0
  - location: studentt, studentp, signt, signp, srankt, srankp
  - normality
  - loccount: loc<,loc>,loc!
  - extremes

  The default value is NULL.

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

ore.univariate Examples

These examples illustrate the use of `ore.univariate`:

- Default Univariate Statistics
- Location Statistics
- Complete Quantile Statistics

These examples use the NARROW data set; for more information, see Data for Examples.

Default Univariate Statistics

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

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

Location Statistics

This example calculates location statistics for YRS_RESIDENCE:

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

Complete Quantile Statistics

This example calculates complete quantile statistics for AGE and YRS_RESIDENCE:

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

5 Predicting with R Models

Predictive models allow you to predict future behavior based on past behavior. After you build a model, you use it to score new data, that is, make predictions.

R allows you to build many kinds of models. When you predict new results (score data) using an R model, the data must be in an R frame. The `ore.predict` package, included with Oracle R Enterprise, allows you to use an R model to score data that is in an `ore.frame`, that is, database resident- data.

`ore.predict()` allows you to make predictions only using `ore.frame` objects; you cannot rebuild the model.

If you need to build models with data in a database table, consider building an Oracle Data Mining model using the OREdm package, described in In-Database Predictive Models in Oracle R Enterprise.

For more information, see the R help associated with `ore.predict`().

ore.predict for R Models

`ore.predict()` allows you to score (predict using) these R models:

- `lm()`Linear regression models
- `glm()` Generalized linear models
- `hclust()` Hierarchical clustering models
- `kmeans()` (*k*-Means clustering)
- `negbin()` (`glm.nb`) Negative binomial generalized binomial models
- `nnet::multinom` Multinomial log-linear model
- `nnet::nnet` neural network models
- `rpart::rpart` Recursive partitioning and regression tree models

Examples

This code builds a linear regression model `irisModel` (built using `lm`) on the `iris` data and then scores IRIS (a table that could be created by pushing `iris` to the database):

```
R> irisModel <- lm(Sepal.Length ~ ., data = iris)
R> IRIS <- ore.push(iris)
R> IRISpred <- ore.predict(irisModel, IRIS, se.fit = TRUE, interval =
"prediction")
R> IRIS <- cbind(IRIS, IRISpred)
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
```

6 Oracle R Enterprise Versions of R Models

Oracle R Enterprise includes several functions that create R models with data in Database tables.

These functions are available at this time:

- `ore.lm()`
- ore.stepwise()
- ore.neural()

This approach has several advantages, as described in ore.lm() and ore.stepwise() Advantages.

ore.lm()

`ore.lm()` performs least squares regression on data represented in an ore.frame object. The model creates a model matrix using the `model.matrix` method from the `OREstats` package. The model matrix and the response variable are then represented in SQL and passed to an in-database algorithm. The in-database algorithm estimates the model using an algorithm involving a block update QR decomposition with column pivoting. After the in-database algorithm estimates the coefficients, it does a second pass of the data to estimate the model-level statistics. Finally, the model is returned as an `ore.lm` object.

The implementation of `ore.lm()` and `ore.stepwise()` provides several advantages, as described in ore.lm() and ore.stepwise() Advantages.

`ore.lm` will not estimate the coefficient values for a set of collinear terms.

After the model is created, use `summary` to create a summary of the model.

For an example, see Linear Regression Example.

ore.lm() and ore.stepwise() Advantages

These are important advantages of the way that `ore.lm()` and `ore.stepwise()` are implemented:

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

  QR-based estimates are often are substantially more accurate than alternative techniques.

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

- You can process data that does not fit into machine's memory, that is, out-of-core data. QR factors a matrix into two matrices, one of which fit into memory with he other stored on disk.

  `ore.lm()` and `ore.stepwise()` can solve data sets with more than one billion rows.

- `ore.lm()` and `ore.stepwise()` allow fast implementations of forward, backward, and stepwise model selection techniques.

`ore.neural` has similar advantages.

Linear Regression Example

This example pushes`longley` to a table and builds a regression model:

```
# longley consiste of employment statistics:
head(longley)
     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
#Push longley to a table
LONGLEY <- ore.push(longley)
# Fit full model
  oreFit1 <- ore.lm(Employed ~ ., data = LONGLEY)
  summary(oreFit1)
```

For more information, see the R help associated with `ore.lm` invoked by `help(ore.lm)`.

ore.stepwise()

`ore.stepwise()` performs stepwise least squares regression on data represented in an `ore.frame` object. The model creates a model matrix using the `model.matrix` method from the `OREstats` package. The model matrix and the response variable are then represented in SQL and passed to an in-database algorithm. The in-database algorithm estimates the model using an algorithm involving a block update QR decomposition with column pivoting. After the in-database algorithm estimates the coefficients, it does a second pass of the data to estimate the model-level statistics. Finally, the model is returned as an `ore.stepwise` object.

`ore.stepwise()` excludes collinear terms throughout the computation.

After the model is created, use `summary` to view a summary of the model.

For an example, see Stepwise Regression Example.

Stepwise Regression Example

This example pushes `longley` to a table and builds a stepwise model.

```
LONGLEY <- ore.push(longley)

 # Two stepwise alternatives
  oreStep1 <-
    ore.stepwise(Employed ~ .^2, data = LONGLEY, add.p = 0.1, drop.p = 0.1)
  oreStep2 <-
```

```
    step(ore.lm(Employed ~ 1, data = LONGLEY),
         scope = terms(Employed ~ .^2, data = LONGLEY))
```

For more information, see the R help associated with `ore.lm` invoked by `help(ore.lm)`.

ore.neural()

Neural network models can be used to capture intricate nonlinear relationships between inputs and outputs, or to find patterns in data.

`ore.neural()` builds a single layer feedforward neural network on `ore.frame` data.

`ore.neural()` uses the Broyden–Fletcher–Goldfarb–Shanno (BFGS) method to solve the underlying unconstrained nonlinear optimization problem that results from fitting a neural network.

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

For detailed information about parameters and output, see the R help for `ore.neural()`. For an example, see Neural Network Example.

Neural Network Example

This example builds a neural network with default values, including hidden size 1.

The `longley` data set consists of statistics related to employment. This example pushes `longley` to a table. Note that the example creates a model that uses a subset of `longley` and then predicts results for a different subset of `longley`.

```
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)
```

7 In-Database Predictive Models in Oracle R Enterprise

The Oracle Advanced Analytics option consists of both Oracle Data Mining and Oracle R Enterprise. Oracle R Enterprise provides a familiar R interface for predictive analytics and data mining functions available in Oracle Data Mining. This is exposed through the `OREdm` package within Oracle R Enterprise.

Data mining uses sophisticated mathematical algorithms to segment data and evaluate the probability of future events. Oracle Data Mining can mine tables, views, star schemas, transactional data, and unstructured data.

For more information about Oracle Data Mining and the algorithms that it supports, see *Oracle Data Mining Concepts 11g Release 2 (11.2)* `http://www.oracle.com/technetwork/database/options/advanced-analytics/odm/index.html`.

See OREdm Models for a complete list of supported algorithms and brief descriptions of the algorithms.

Note: The CRAN package `RODM` also supports many Oracle Data Mining algorithms. `RODM` is different from `OREdm`.

The `OREdm` interface is designed to provide a standard R interface for corresponding predictive analytics and data mining functions.

This section provides an overview of the algorithms supported by `OREdm`. For detailed information about a specific model, see the R help associated with the specific `OREdm` function.

In order to build a model, you must have build (training) data that satisfies OREdm Requirements.

Oracle Data Mining models are somewhat different from `OREdm` models; see OREdm Models and Oracle Data Mining Models.

For list of the models available at this release and brief overview information, see OREdm Models.

Examples of using `OREdm` to build models are included in the descriptions of each function. For example, Attribute Importance Example shows how to build an AI model.

OREdm Requirements

`OREdm` requires that the data used to train (build) models exists in a single table or view that contains columns of the following types only: VARCHAR2, CHAR, NUMBER, and FLOAT.

All privileges required by Oracle Data Mining are automatically grant during Oracle R Enterprise installation.

Oracle Data Mining must be enabled for the database that you connect to.

OREdm Models and Oracle Data Mining Models

Within `OREdm`, Oracle Data Mining models are given generated names. As long as the `OREdm` R model object exists, these model names can be used to access Oracle Data Mining models 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.

Oracle Data Miner can be useful in a number of ways:

- 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 will not be carried over into Oracle Data Miner or SQL scoring.

Similarly, SQL can be used to get a list of models, inspect model details, and score appropriately transformed data with these models.

Models created using `OREdm` are transient objects; they usually are not persisted past the R session that created them. Oracle Data Mining models created using Data Miner or SQL, on the other hand, exist until they are explicitly dropped.

Model objects can be saved or persisted, as described in Persist and Manage R Objects in the Database. This allows `OREdm`-generated model objects to exist across R sessions and keeps the ODM 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.

OREdm Models

`OREdm` supports these Oracle Data Mining models:

- Attribute Importance
- Decision Tree
- Generalized Linear Models
- k-Means
- Naive Bayes

- Support Vector Machine

Oracle Data Mining and Open-Source R uses different terminology; see Data Mining Terminology.

Note that there are several Overloaded Functions that perform common actions such as predict (score), summary, and print summary.

Data Mining Terminology

Oracle Data Mining and the Oracle R Enterprise `OREdm` package that creates statistical models use somewhat different terminology. These are the most important differences

- Oracle R Enterprise *fits* models, whereas Oracle Data Mining *builds* or *trains* models.
- Oracle R Enterprise *predicts* using new data, whereas Oracle Data Mining *scores* new data, or *applies* a model to new data.
- Oracle R Enterprise uses formula, as described in Formula, in the API calls; Oracle Data Mining does not support formula.

Formula

R model definitions require a **formula** that expresses relationships between variables. The `formula` class is included in the R `stats` package. For more information, see the R help associated with `?formula`. A `formula` provides a symbolic description of the model to be fitted.

The `[stats]{formula}` specification has the form `(response ~ terms)` where

- `response` is the numeric or character response vector.
- `terms` is a series of terms, that is, the column names to include in the model. Multiple terms are specified using `+` between column names.

Use `{response ~ .}` if all columns in data should be used for model building

Functions can be applied to response and terms to realize transformations.

To exclude columns, use `-` before the name of each column to exclude.

The examples of model builds in this document and in the R help all contain sample formulas. There is no equivalent of `formula` in the Oracle Data Mining API.

Overloaded Functions

`predict()`, `summary()`, and `print()` are defined across all `OREdm` algorithms, for example, as illustrated in GLM Examples.

`summary()` returns detailed information about the model created, such as details of the generated decision tree.

Attribute Importance

Oracle Data Mining uses the Minimum Descriptor 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.

`ore.odmAI` 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 details about parameters, see the R help associated with `ore.odmAI`.

For an example, see Attribute Importance Example.

Attribute Importance Example

This example creates a table by pushing the data frame `iris` to the table IRIS and then builds an attribute importance model:

```
IRIS <- ore.push(iris)
ore.odmAI(Species ~ ., IRIS) # Analyse the column Species
```

Decision Tree

The Decision Tree algorithm is based on conditional probabilities. Decision trees generate rules. A rule is a conditional statement that can easily be understood by humans and easily 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. `ore.odmDT` offers two homogeneity metrics, gini and entropy, for calculating the splits. The default metric is gini.

`OREdm` includes these functions for Decision Tree (DT):

- `ore.odmDT` creates (builds) a DT model.
- `predict` predicts classifications on new data using the DT model.
- `summary` provides a summary of the DT model. The summary includes node details that describe the tree that the model generates, and a symbolic description of the model. Returns an instance of `summary.ore.odmDT`.
- `print.ore.odmDT` prints select components of the `ore.odmDT` model.

For details about parameters, see the R help associated with `ore.odmDT`.

For an example, see Decision Tree Example.

Decision Tree Example

This example creates an input table, builds a model, makes predictions, and generates a confusion matrix.

```
# Create MTCARS, the input data
  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 <- ore.push(m)
  row.names(MTCARS) <- MTCARS
# Build the model
  dt.mod  <- ore.odmDT(gear ~ ., MTCARS)
  summary(dt.mod)
 # Make predictions and generate a confusion matrix
  dt.res  <- predict (dt.mod, MTCARS,"gear")
  with(dt.res, table(gear,PREDICTION))  # generate confusion matrix
```

Generalized Linear Models

Generalized Linear Models (GLM) 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.

Oracle Data Mining's 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 via 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 (This approach is similar to the approach in Komarek's paper of 2004.) The choice of algorithm is handled internally and is transparent to the user.

GLM can be used to create 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.

  For an example, see GLM Examples.

- **Regression:** Linear regression is the GLM regression algorithm. The algorithm assumes no target transformation and constant variance over the range of target values.

  For an example, see GLM Examples.

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

`OREdm` provides these functions for Generalized Linear Models (GLM):

- `ore.odmGLM` creates (builds) a GLM model; note that some arguments apply to classification models only, and some to regression models only.

- residuals is an ore.frame containing three types of residuals: deviance, pearson, and response.
- fitted is fitted.values: an ore.vector containing the fitted values:
  - rank: The numeric rank of the fitted model
  - type: The type of model fit
- predict.ore.odmGLM predicts new data using the GLM model.
- confint is logical indicator for whether to produce confidence intervals for the predicted values.
- deviance is minus twice the maximized log-likelihood, up to a constant.
- coef.ore.odmGLM retrieves coefficients for GLM models with linear kernel.
- extractAIC.ore.odmGLM extracts Akaike's *An Information Criterion* (AIC) from the global details of the GLM model.
- logLik extracts Log-Likelihood for an OREdm GLM model.
- nobs extracts the number of observations from a model fit. nobs is used in computing BIC.

  BIC is defined as AIC(object, ..., k = log(nobs(object))).
- summary creates a summary of the GLM model. The summary includes fit details for the model. Also returns formula, a symbolic description of the model. Returns an object of type summary.ore.odmGLM
- print prints selected components of the GLM model.

For details about parameters and methods, see the R help associated with ore.odmGLM.

GLM Examples

These examples build several models using GLM. The input tables are R data sets pushed to the database.

- Linear regression using the longley data set:

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

- Ridge regression using the longley data set:

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

- Logistic regression (classification) using the infert data set:

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

- Changing the reference value to 1 for infit1:

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

k-Means

The *k*-Means (KM) algorithm, a distance-based clustering algorithm that partitions data into a specified number of clusters, is an enhanced version with these 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 numerical attributes.

`OREdm` includes these functions for *k*-Means (KM) models:

- `ore.odmKMeans` creates (builds) a KM model.
- `predict` predicts new data using the KM model.
- `rules.ore.odmKMeans` extracts rules generated by the KM model.
- `clusterhists.ore.odmKMeans` generates a `data.frame` with histogram data for each cluster and variable combination in the model. Numerical variables are binned.
- `histograms.ore.odmKMeans` produces lattice-based histograms from a clustering model.
- `summary` returns a summary of the KM model, including rules. Also returns `formula`, a symbolic description of the model. Returns an object of type `summary.ore.KMeans`.
- `print` prints selected components of the KM model.

For details about parameters, see the R help associated with `ore.odmKM`().

For an example, see k-Means Example.

k-Means Example

This example creates the table X, builds a cluster model, plots the clusters via `histogram`(), and makes predictions:

```
# Create input table X
  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)
 # Build clustering mode; plot results
  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)
# Make predictions
  head(predict(km.mod1,X))
  head(predict(km.mod1,X,type=c("class","raw")),3)
  head(predict(km.mod1,X,type=c("class","raw"),supplemental.cols=c("x","y")),3)
  head(predict(km.mod1,X,type="class"),3)
  head(predict(km.mod1,X,type="class",supplemental.cols=c("x","y")),3)
  head(predict(km.mod1,X,type="raw"),3)
  head(predict(km.mod1,X,type="raw",supplemental.cols=c("x","y")),3)
```

Naive Bayes

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.)

`OREdm` includes these functions for Naive Bayes (NB) models:

- `ore.odmNB` creates (builds) an NB model.
- `predict` scores new data using the NB model.
- `summary` provides a summary of the NB model. Also returns `formula`, a symbolic description of the model. Returns an instance of `summary.ore.odmNB`.
- `print` prints select components of the `NB` model.

For details about parameters, see the R help associated with `ore.odmNB`.

For an example, see Naive Bayes Example.

Naive Bayes Example

This example creates MTCARS, builds a Naive Bayes model, and then uses the model to make predictions:

```
# Create MTCARS
  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 <- ore.push(m)
  row.names(MTCARS) <- MTCARS
 # Build model
  nb.mod  <- ore.odmNB(gear ~ ., MTCARS)
  summary(nb.mod)
 # Make predictions
  nb.res  <- predict (nb.mod, MTCARS,"gear")
  with(nb.res, table(gear,PREDICTION))  # generate confusion matrix
```

Support Vector Machine

Support Vector Machine (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 a set of objects having different class memberships. SVM finds the vectors ("support vectors") that define the separators giving the widest separation of classes.

  SVM classification supports both binary and multiclass targets.

  For an example, see SVM Classification.

- **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.

  For an example, see SVM Regression.

- **Anomaly Detection**: Anomaly detection identifies 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.

  For an example, see SVM Anomaly Detection.

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.

`OREdm` provides these functions for SVM models:

- `ore.odmSVM` creates (builds) SVM model.
- `predict` predicts (scores) new data using the SVM model.
- `coef` retrieves the coefficient of an SVM model.

  SVM has two kernels, Linear and Gaussian; the Linear Kernel generates coefficients.

- `summary` creates a summary of the SVM model.Also returns `formula`, a symbolic description of the model. Returns an object of type `summary.ore.odmSVM`.
- `print` print selected components of the SVM model.

For details about parameters, see the R help associated with `ore.odmSVM`.

Support Vector Machine Examples

These examples build three models:

- SVM Classification
- SVM Regression
- SVM Anomaly Detection

SVM Classification This example creates `mtcars` in the database from the R `mtcars` dataset., 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 <- ore.push(m)

  svm.mod  <- ore.odmSVM(gear ~ .-ID, MTCARS,"classification")
  summary(svm.mod)
  coef(svm.mod)
  svm.res  <- predict (svm.mod, MTCARS,"gear")
  with(svm.res, table(gear,PREDICTION))  # generate confusion matrix
```

SVM Regression This 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)
```

SVM Anomaly Detection This example uses MTCARS created in the classification example and builds an anomaly detection model:

```
  svm.mod  <- ore.odmSVM(~ .-ID, MTCARS,"anomaly.detection")
  summary(svm.mod)
  svm.res  <- predict (svm.mod, MTCARS, "ID")
  head(svm.res)
  table(svm.res$PREDICTION)
```

8 Oracle R Enterprise Embedded Execution

This chapter describes these topics:

Security Considerations for Scripts

Both R scripts and SQL scripts allow access to the database server. For this reason, creation of scripts must be controlled. The RQADMIN Role is required for those users who create and drop scripts.

RQADMIN Role

Oracle R Enterprise creates the RQADMIN role.

The RQADMIN role must be explicitly granted to a user.

The RQADMIN role is required in these instances:

- Calling `ore.doEval()` with `FUN` argument
- Creating and dropping scripts with `ore.scriptCreate` and `ore.scriptDrop`

The RQADMIN role is *not* required when calling `ore.rowApply`, `ore.groupApply`, `ore.tableApply`, `ore.indexApply`, and `ore.doEval` with the `FUN.NAME` argument.

To grant RQADMIN to RQUSER, start SQL*Plus as `sysdba` and type

```
grant rqadmin to RQUSER
```

Note: You should grant RQADMIN only to those users who need it.

Support for Database Parallelism

Parallel processing is not restricted to Oracle R Enterprise functions only; it can be enabled for Open Source R packages that are not part of Oracle R Enterprise. For such packages data-parallelism can be leveraged through Oracle R Enterprise embedded R execution.

On the R side, Oracle R Enterprise provides `ore.groupApply()`, `ore.rowApply()`, and ore.indexApply() for data-parallel processing. Data-parallel processing consists of dividing a data set into multiple subsets that can be processed in parallel (independently). Oracle R Enterprise also provides SQL-equivalent functionality for group apply and row apply as described in Oracle R Enterprise Embedded SQL Scripts.

Open Source packages (CRAN packages) can generally not leverage the Oracle R Enterprise transparency layer (because they are not written using base R exclusively or include callouts to functionality such as C functions) and execute on data in the R address space. This means that their use is subject to memory and parallelism constraints of R and the way the CRAN package was written. Oracle R Enterprise does not automatically parallelize the internal code of CRAN packages.

Embedded R execution enables leveraging what is likely a larger server (a Database server, such as Oracle Exadata) in terms of memory and number of processors to expand what a typical R client may be able to achieve. In addition, embedded R execution provides for more efficient transfer of data between the database and the R engine (since they are on the same machine). Embedded R execution also allows for data parallel execution of user R functions that may leverage CRAN packages, both from Oracle R Enterprise R and SQL APIs.

R Interface for Embedded Oracle R Enterprise Scripts

These Oracle R Enterprise functions permit R-based applications to embed Oracle R Enterprise functionality in the scripts. For example, they allow R scripts to perform operations on database objects.

An R script contains a single function definition. R scripts reside in the Oracle R Enterprise in-database R script archive.

Embedded R scripts provide several advantages:

- You can execute R scripts in the database where the data resides; you do not have to move data out of the database. The scripts may contain custom techniques or include functions from CRAN packages.
- You can run existing R scripts within R-based applications and operational SQL-based applications.
- You can leverage distributed data flow parallelism in Oracle Database; the parallelism is user controlled but database managed.
- You can use the security provided by Oracle Database. See Security Issues for Embedded R Scripts for information about how to register scripts so that they are secure.

Oracle R Enterprise provides these functions that support running R scripts in the database:

- ore.doEval()
- ore.tableApply()
- ore.groupApply()
- ore.rowApply()
- ore.indexApply()
- ore.scriptCreate()
- ore.scriptDrop()

There are example scripts in ore.doEval() and ore.indexApply().

Security Issues for Embedded R Scripts

All of these scripts require an argument `FUN` or `FUN.NAME`. For security reasons, use of the argument `FUN` requires the RQADMIN role, a collection of Oracle Database privileges. Since creation of the script represented by the argument `FUN.NAME` must be published by a user with RQADMIN credentials, it can be used by anyone authorized to use Oracle R Enterprise.

Input for ore.*Apply() and ore.doEval()

The functions `ore.tableApply()`, `ore.groupApply()`, `ore.rowApply()`, and `ore.indexApply()`, and `ore.doEval()` all take either a `FUN.NAME` parameter (for a function that has been loaded into the R script repository) or `FUN`, which is an R function (closure).

All functions can return anything. However, when you specify the FUN.VALUE argument the output should be a matching `data.frame`.

All functions take parameters that are passed as optional arguments (`... arguments`). They can named or not.

All functions take the `FUN.NAME` parameter, which is the name of a function in the R script repository in the database, or an actual R function in the `FUN` parameter.

ore.doEval()

`ore.doEval()` invokes a stand-alone R script in the database without input data; parameters are allowed. It returns an `ore.frame` object or serialized R objects.

Input for `ore.doEval()` is internally generated data. You can load data from a file or a table using `ore.pull()`.

Input data is one of the following:

- Internally generated
- Loaded from a file or pulled from the database by using `ore.pull()`
- Made available through the Transparency Layer

`ore.doEval()` takes the `FUN.NAME` parameter, which is the name of a function in the R script repository in the database, or an actual R function in the `FUN` parameter.

The following additional arguments to the `FUN` parameter starting with `ore.` are special control arguments. They are not passed to the function specified by the `FUN` or `FUN.NAME` arguments, but instead control what happens before or after the execution of the closure. The following control arguments are supported:

- `ore.drop` controls the input data. If TRUE, one column `data.frame` will be converted to a vector. The default value is TRUE.
- `ore.connect` controls whether to automatically connect to Oracle R Enterprise inside the closure. This is equivalent to doing an `ore.connect` call with the same credentials as the client session. 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 parameters 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.

This example scales the first *n* integers by the value provided. The result is a serialized R object (`data.frame`):

Oracle R Enterprise comes with a number of predefined graphical scripts. All predefined scripts have a reserved name that start with `RQG$` followed by a function name from the graphics package that the script wraps. Depending on the function it either takes the first, the first and second, or all of the columns of the input `data.frame`. Thus, predefined scripts can only be used with `ore.tableApply`, ore.`groupApply`, or `ore.rowApply`. Each function also has `...` so that it can pass any parameter to the function that it wraps.

```
res <-
   ore.doEval(function (num = 10, scale = 100) {
          ID <- seq(num)
          data.frame(ID = ID, RES = ID / scale)
          })
class(res)
res
local_res <- ore.pull(res)
class(local_res)
local_res
```

For more examples, see the R help for `ore.doEval()`.

ore.tableApply()

`ore.tableApply()` invokes an R script with an entire table (`ore.frame`) as input. The input is provided all at once to the function. As with `ore.doEval()`, it can return an `ore.frame` object or serialized R objects.

Input data is an `ore.frame` object.

Returns a data frame signature as an `ore.frame` object.

Takes NULL or `<variable>=<value>` as an argument.

ore.groupApply()

`ore.groupApply()` partitions the data according to a specified column's values and invokes the R script on each partition in parallel, when possible. The return value is a list of each group's execution results.

Input data is an `ore.frame` object.

Returns either a NULL value as an `ore.object` or a data frame signature as an `ore.frame` object.

Takes NULL or `<variable>=<value>` as an argument.

You must specify the partition column for `ore.groupApply()`.

Takes the `FUN.NAME` parameter, which is the name of a function in the R script repository in the database, or an actual R function in the `FUN` parameter.

ore.rowApply()

`ore.rowApply()` enables you to specify a chunk size, which is the number of rows that the function should act upon. The function is invoked multiple times in parallel, if multiple R engines can be invoked at the database server, until all data is processed. The return value is a list of each chunk's execution results.

Input data is an `ore.frame` object.

Returns either a NULL value as an `ore.object` or a data frame signature as an `ore.frame` object.

Takes NULL or `<variable>=<value>` as an argument.

You can specify the chunk size for `ore.rowApply()`.

Takes the `FUN.NAME` parameter, which is the name of a function in the R script repository in the database, or an actual R function in the `FUN` parameter.

ore.indexApply()

`ore.indexApply()` invokes an R script *n* times, where *n* is a positive integer. The return value is a list of each execution's results.

Input data is one of the following:

- Internally generated
- Loaded from a file or pulled from the database by using `ore.pull()`
- Made available through the Transparency Layer

`ore.indexApply()` can take NULL or `<variable>=<value>` as arguments.

You must specify *n*, the number of times to invoke the R function.

Takes the `FUN.NAME` parameter, which is the name of a function in the R script repository in the database, or an actual R function in the `FUN` parameter

For example, this code applies the function 10 times:

```
res<-ore.indexApply(10,function (x, scale = 100) x / scale)
```

ore.scriptCreate()

`ore.scriptCreate()` creates an R script in the database. The script can be used by name in other embedded R script functions.

Note: `ore.scriptCreate()` requires the RQADMIN role.

`ore.scriptCreate()` has this syntax:

```
ore.scriptCreate(name, FUN)
```

where

- `name` is a character string specifying the name of the R script in Oracle Database.
- `FUN` is a function definition to be used with functions `ore.doEval(`, `ore.groupApply()`, `ore.indexApply()`, `ore.rowApply()`, or `ore.tableApply()`.

The function returns an invisible NULL value if it succeeds; if it does not succeed in creating the script, it returns an error.

ore.scriptCreate() Example

This example creates a script and then drops it:

```
ore.scriptCreate("MYLM",function(data, formula, ...) lm(formula, data, ...))
       IRIS <- ore.push(iris)
       ore.tableApply(IRIS[1:4], FUN.NAME = "MYLM" formula = Sepal.Length ~ .)
       ore.scriptDrop("MYLM")
```

ore.scriptDrop()

`ore.scriptDrop()` drops a named R script from the database repository. Requires the RQADMIN role.

Note: `ore.scriptDrop()` requires the RQADMIN role.

`ore.scriptDrop()` has this syntax:

```
ore.scriptDrop(name)
```

where

- `name` is a character string specifying the name of the R script in Oracle Database.

The function returns an invisible NULL value if it succeeds; if it does not succeed in dropping the script, it returns an error.

For an example, see ore.scriptCreate() Example.

Automatic Database Connection in Embedded R Scripts

An embedded R script can automatically connect to an Oracle database.

If automatic connections are enabled, the following functionality occurs:

- Embedded R scripts are automatically connected to the database.
- The automatic connection has the same credentials as the session that invokes the embedded R SQL functions.
- The script runs in an autonomous transaction.
- ROracle queries work with the automatic connection.
- Oracle R Enterprise transparency is enabled in the embedded script.
- User and site-wide R profile loading is disabled in embedded R.

  Profile loading was supported in earlier Oracle R Enterprise releases. An automatic connection provides a more secure connection.

Automatic connections are disabled by default. You can specify whether automatic connections are enabled or disabled by using the `ore.connect` control argument. Control arguments are documented in R help for `ore.doEval()`.

To enable automatic connections, ROracle was extended by adding a new driver `ExtDriver` with the constructor `Extproc` that is initialized by passing an external pointer wrapping the `extproc` context. Similarly to `OraDriver`, `ExtDriver` is a singleton. Both drivers can exist simultaneously in a session since these are represented by two distinct singletons. This setup allows working with `extproc` and explicit `OraDriver` connections in the same R script as shown by the following example.

```
ore.doEval(function() {
  ore.disconnect()
  con1 <- dbConnect(Extproc())
  res1 <- dbGetQuery(con1, "select * from grade order by name")
  con2 <- dbConnect(Oracle(), "scott", "tiger")
  res2 <- dbGetQuery(con2, "select * from emp order by empno")
  dbDisconnect(con1)
  dbDisconnect(con2)
  cbind(head(res1)[,1:3], head(res2)[,1:3])
}, ore.connect = TRUE)
```

Examples of Embedded R Scripts

For a detailed example of an embedded R script, see the Oracle R Enterprise Blog "Introduction to ORE Embedded R Script Execution" at `https://blogs.oracle.com/R/entry/analyzing_big_data_using_the1`.

Part 6: ORE Embedded R Scripts: R Interface in the free **Oracle R Enterprise Tutorial Series** describes embedded R scripts and contains several examples. See Oracle R Enterprise Training for information about the Tutorial Series.

Several of the Oracle R Enterprise Demos illustrate embedded execution.

Oracle R Enterprise Embedded SQL Scripts

The SQL interface allows you to embed R script execution in production database applications.

The functions associated with the SQL interface must be stored in the database R repository, and referenced by name in SQL API functions. See Registering and Managing SQL Scripts for a description of how to add scripts to the repository, remove scripts from the repository, and list and use scripts in the repository.

For descriptions of the SQL functions, see Oracle R Enterprise SQL Functions.

Registering and Managing SQL Scripts

For security purposes, you must first register the R script under some system unique name and use the new name instead of the actual script in calls to `rq*Eval` table functions.

There are two administrative functions that create and drop scripts and a view that lists scripts:

- `sys.rqScriptCreate()`
- `sys.rqScriptDrop()`
- view allows you to list and use scripts that were created

The scripts require the RQADMIN role described in RQADMIN Role.

When using `sys.rqScriptCreate()`, you must specify a corresponding R Closure of the function string.

Here is an example of registering the scripts and of using the registered scripts:

```
begin
  sys.rqScriptCreate('tmrqfun2',
'function() {
ID <- 1:10
res <- data.frame(ID = ID, RES = ID / 100)
res
}');
end;
```

```
/

select *
  from table(rqEval(
        NULL,
       'select 1 id, 1 res from dual',
       'tmrqfun2'));

begin
  sys.rqScriptDrop('tmrqfun2');
end;
/
```

Oracle R Enterprise SQL Functions

The `rq*Eval` functions result in one or more new R engines being started at the database depending on database parallelism settings.

To enable execution of an R script in the database (lights-out processing), Oracle R Enterprise provides variants of `ore.doEval()`, `ore.tableApply()`, `ore.groupApply()`, and `ore.rowApply()` in SQL. (`ore.doEval()`, `ore.tableApply()`, `ore.groupApply()`, and `ore.rowApply()` are described in R Interface for Embedded Oracle R Enterprise Scripts.)

The SQL functions are

- `rqTableEval()`
- `rqEval()`
- `rqRowEval()`
- `rqGroupEval()`

`rqGroupEval()` requires additional SQL specification and is provided here as a virtual function, which partitions the data according to a specified column's values and invokes the R script on each partition. For more information, see rqGroupEval() Function.

You can also use these functions with objects in a datastore, as described in rq*Eval() and Objects in a Datastore.

The `rq*Eval()` functions (`rqEval()`, `rqTableEval()`, `rqGroupEval()`, and `rqRowEval()`) have similar syntax:

```
rq*Eval(
     cursor(select * from table-1),
     cursor(select * from table-2),
     'select <column list> from table-3 t',
     <grouping col-name from table-1 or num_rows>,
     <R closure name of registered-R-code>
     )
```

where

- The first cursor is the input cursor: Input is passed as a whole table, group, or N rows at a time to the R closure described in the fourth parameter.

  `rqEval()` does *not* have this cursor argument.

- The second cursor is the parameters cursor: One row of scalar values (string, numeric, or both) can be passed; for example, the name of the model and several numeric scalar values for model setting.
- The query specifies the output table definition; output can be 'SELECT statement', 'XML', or 'PNG'.
- `grouping col-name` applies to `rqGroupEval()`; it provides the name of the grouping column.
- `num_rows` applies to `rqRowEval()`; it provides the number of rows to provide to the functions at one time.
- <R closure name of registered-R-code> is a registered version of the R function to execute. See Registering and Managing SQL Scripts for details.

The return values for all of the SQL functions specify one of these values:

- A table signature that is specified in a SELECT statement, which returns results as a table from the `rq` function.
- XML, returned as a CLOB which returns both structured and graph images in an XML string. The structured components are provided first, followed by the base 64 encoding of the `png` representation of the image.
- PNG, returned as a BLOB which returns graph images in PNG format.

`rqEval()`, `rqTableEval()`, `rqGroupEval()`, and `rqRowEval()` must specify an R script by the name that is stored in the R script repository. See Registering and Managing SQL Scripts for information about the `sys.rq_scripts` view provides a list of registered scripts.

The following examples illustrate using these functions:

- This example uses all rows from the table `fish` as input to the R function that takes no other parameters and produces `output` that contains all input data plus the ROWSUM of values.

  Note that parameters (`param`) to the R function is optional.

```
begin
sys.rqScriptCreate('tmrqfun2',
'function(x, param) {
dat <- data.frame(x, stringsAsFactors=F)
cbind(dat, ROWSUM = apply(dat,1,sum)+10)
}');
end;
/

select * from table(rqTableEval(
   cursor(select * from fish),
   NULL,
   'select t.*, 1 rowsum from fish t',
   'tmrqfun2' ));

begin
sys.rqScriptDrop('tmrqfun2');
end;
/
```

- This example illustrates passing n=1 (4th parameter) row at a time from the table `fish` to the R function. No parameters are required by the function. The function generates ROWSUM which is added as an extra column to `fish` in the output.

```
begin
```

```
sys.rqScriptCreate('tmrqfun2',
'function(x, param) {
dat <- data.frame(x, stringsAsFactors=F)
cbind(dat, ROWSUM = apply(dat,1,sum)+10)
}');
end;
/

select * from table(rqRowEval(
   cursor(select * from fish),
   NULL,
   'select t.*, 1 rowsum from fish t',
    1,
   'tmrqfun2' ));

begin
sys.rqScriptDrop('tmrqfun2');
end;
/
```

rqGroupEval() Function

`rqGroupEval()` invokes an R script on data that is partitioned by a grouping column.

`rqGroupEval()` requires the creation of two PL/SQL objects, a package and a pipelined table function:

1. Create a PL/SQL package that specifies the types of result to be returned.
2. Create a function that takes the return value of the package and uses the return value with PIPELINED_PARALLEL_ENABLE set to indicate the column on which to partition data.

Suppose that ONTIME_S is a table that stores information about arrival of airplanes. The data cursor uses all data, but you could also define cursors that use some columns using PL/SQL records. Then you must define as many PL?SQL table functions as the number of grouping columns that you are interested in using for a particular data cursor.

```
CREATE PACKAGE ontimePkg AS
  TYPE cur IS REF CURSOR RETURN ontime_s%ROWTYPE;
END ontimePkg;
/

CREATE FUNCTION ontimeGroupEval(
  inp_cur  ontimePkg.cur,
  par_cur  SYS_REFCURSOR,
  out_qry  VARCHAR2,
  grp_col  VARCHAR2,
  exp_txt  CLOB)
RETURN SYS.AnyDataSet
PIPELINED PARALLEL_ENABLE (PARTITION inp_cur BY HASH (month))
CLUSTER inp_cur BY (month)
USING rqGroupEvalImpl;
/
```

At this time, only one grouping column is supported. If you have multiple columns combine the columns into one column and use the new column as a grouping column. PARALLEL_ENABLE clause is optional but CLUSTER BY is not.

rq*Eval() and Objects in a Datastore

rq*Eval()and related functions allow you to use serialized R objects saved in a datastore using a parameter cursor. You can specify the association of object and datastore names of the serialized R objects with the R function parameter names in that parameter cursor.

Here is an example of how to use rq*Eval()this way. Suppose that user scott has saved a model in the datastore ontime_model as the object lm.mod,. Suppose scott wants to use this model in SQL for embedded Oracle R Enterprise scoring. This code shows how to use the model for embedded scoring. See Automatic Database Connection in Embedded R Scripts for the configuration parameters for ore.connect().

```
begin
  sys.rqScriptCreate('tmrqmodelscore',
    'function(dat, in.dsname, in.objname) {
       ore.load(name=in.dsname, list=in.objname)
       mod <- get(in.objname)
       prd <- predict(mod, newdata=dat)
       prd[as.integer(rownames(prd))] <- prd
       res <- cbind(dat, PRED = prd)
       res
    }');
end;
/ -- score model

select * from table(rqTableEval(
             cursor(select ARRDELAY, DISTANCE, DEPDELAY from ontime_s
                where year = 2003 and month = 5 and dayofmonth = 2),
             cursor(select 'ontime_model' as "in.dsname",
            'lm.mod' as "in.objname", 1 as "ore.connect" from dual),
             'select ARRDELAY, DISTANCE, DEPDELAY, 1 PRED from ontime_s',
            'tmrqmodelscore'))
order by 1, 2, 3;
```

Datastore Management in SQL

Oracle R Enterprise provides basic management for datastores in SQL. Basic datastore management includes show, search, and drop. The following functions and views are provided:

- rqDropDataStore() deletes a datastore and all of the objects in the datastore.

  Syntax: rqDropDataStore('<ds_name>'), where <ds_name> is the name of the datastore to delete.

  The following example deletes the datastore ds_model from current user schema:

  ```
  rqDropDataStore('ds_model')
  ```

- rquser_DataStoreList is a view containing datastore-level information for all datastores in the current user schema. The information consists of datastore name, number of objects, size, creation date, and description.

  These examples illustrate using the view:

  ```
  select * from rquser_DataStoreList
  select dsname, nobj, size from rquser_datastorelist where dsname = 'ds_1'
  ```

- `rquser_DataStoreContents` is a view containing object-level information about all datastores in the current user schema. The information consists of object name, size, class, length, number of rows and columns.

  This example lists the datastore contents for datastore `ds_1`:

  ```
  select * from rquser_DataStoreContents where dsname = 'ds_1';
  ```

A

Oracle R Enterprise and Oracle R Distribution Packages

This appendix lists the R packages supported by Oracle R Distribution and Oracle R Enterprise. R functions included in these packages are supported on R versions 2.13.2 and 2.15.1.

Packages Related to Oracle R Distribution

These packages related to Oracle R Distribution are supported by Oracle R Enterprise:

```
KernSmooth - Functions for kernel smoothing for Wand & Jones (1995)
MASS - Support Functions and Datasets for Venables and Ripley's MASS
Matrix - Sparse and Dense Matrix Classes and Methods
base - The R Base Package
boot - Bootstrap Functions (originally by Angelo Canty for S)
class - Functions for Classification
cluster - Cluster Analysis Extended Rousseeuw et al.
codetools - Code Analysis Tools for R
compiler - The R Compiler Package
datasets - The R Datasets Package
foreign - Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ...
grDevices - The R Graphics Devices and Support for Colours and Fonts
graphics - The R Graphics Package
grid - The Grid Graphics Package
lattice - Lattice Graphics
methods - Formal Methods and Classes
mgcv - GAMs with GCV/AIC/REML smoothness estimation and GAMMs by PQL
nlme - Linear and Nonlinear Mixed Effects Models
nnet - Feed-forward Neural Networks and Multinomial Log-Linear Models
rpart - Recursive Partitioning
spatial - Functions for Kriging and Point Pattern Analysis
splines - Regression Spline Functions and Classes
stats - The R Stats Package
stats4 - Statistical Functions using S4 Classes
survival - Survival analysis, including penalised likelihood.
tcltk - Tcl/Tk Interface
tools - Tools for Package Development
utils - The R Utils Package
```

Packages Related to Oracle R Enterprise

These packages are installed when Oracle R Enterprise.

The following R command loads the libraries and any required packages if they are not already installed:

```
> library(ORE)
```

These packages related to Oracle R Enterprise are supported by Oracle R Enterprise:

```
DBI - R Database Interface
ORE - Oracle R Enterprise
OREbase - Oracle R Enterprise - base
OREdm - Oracle R Enterprise - dm
OREeda - Oracle R Enterprise - exploratory data analysis
OREgraphics - Oracle R Enterprise - graphics
OREpredict - Oracle R Enterprise - model predictions
OREstats - Oracle R Enterprise - stats
OREexml - Oracle R Enterprise - R objects to XML
ROracle - OCI based Oracle database interface for R
XML - Tools for parsing and generating XML within R and S-Plus.
bitops - Functions for Bitwise operations
png - Read and write PNG images
```

Index