OPEN DESKTOP™

Open Desktop®

Report Writer SQL Reference

OPEN DESKTOP™

The Complete Graphical Operating System

ODT-DATA Report Writer Reference

ODT-DATA is based on technology developed by **INGRES CORPORATION**, and includes the following **INGRES** components:

- INGRES/DBMS and SQL Terminal Monitor
- INGRES/User Interfaces
  - Query-by-Forms
  - Report-by-Forms
  - Report Writer
  - Menu
  - Forms Runtime Systems and VIFRED
- INGRES/NET with TCP/IP Support
- INGRES/WindowView
- INGRES/ESQL Preprocessor for C

Document version: 1.0.0C
Date: 15 June 1990

Table of Contents

Preface

This guide describes the ODT-DATA Report-Writer language. It provides an overview of the Report-Writer, documents the Report-Writer statements, and gives examples to help you create your own reports.

In addition to the Report-Writer language described here, ODT-DATA provides other tools for writing reports, such as Report-By-Forms (RBF). For information about RBF, see *Using ODT-DATA Through Forms and Menus.*

This guide can be broken into two sections. Chapters 1 through 4 serve as an overview, while Chapters 5 through 12 provide reference information. The information is presented as follows:

- Chapter 1 introduces ODT-DATA Report-Writer. It discusses the process of creating reports, along with the concepts necessary to understand the ODT-DATA Report-Writer.
- Chapter 2 is an overview of the report-specification statements.
- Chapter 3 covers report parameters, pagination, margins and columns, calculations and summary data.
- Chapter 4 explains print-format statements. This includes the use of aggregates, operations, and template formats for functions, character data, and numeric data.
- Chapters 5 through 12 provide reference material for the Report-Writer statements. For each Report-Writer statement, there are syntax, the parameters associated with the syntax, a description of the functions of the statements, and examples showing how to use the statement.
- Appendix A shows six sample reports. Included with the sample reports are the report format statements for the reports.
- Appendix B list the various error messages available in Report-Writer.

Audience

In a multiuser installation, various database tasks are assigned to various individuals with differing privileges:

- The system administrator manages the ODT-DATA installation
- The database administrator (DBA) creates and manages the database
- The user manipulates data in the database

This guide is for individuals in both types of installations, though at times only the multiuser installation is explicitly addressed. If you are using ODT-DATA in a single-user installation, assume that you are the system administrator, the database administrator and the user.

Conventions Used in This Guide

This guide uses the following conventions:

- Words in **bold** are keywords and must be typed as shown or in its abbreviated form. Keywords preceded by a period (.) are report-formatting statements and must be typed as shown or in its abbreviated form.
- Words in *italic* are variables, the values of which are supplied by the user or the program.
- Clauses in square brackets ([ ]) are optional.
- Clauses in curly braces ({ }) are optional and can be specified zero or more times.
- Vertical bars (|) separate multiple items from which you choose one.

System Notes

In a command line to the UNIX system, a set of parentheses must be preceded and followed by single quotes, for instance: '(...)'.

Associated Publications

The *ODT-DATA Report -Writer Reference Manual* is one of several publications provided for your use of ODT-DATA. The table below lists all the ODT-DATA books available with each Open Desktop product:

Open Desktop:
■ *Administering ODT-DATA*
■ *Using ODT-DATA*
Open Desktop Development System:
■ *ODT-DATA Embedded SQL User's Guide*
■ *ODT-DATA Embedded Open SQL Forms Reference Manual*
■ *ODT-DATA Open SQL Reference Manual*
■ *ODT-DATA Embedded SQL Companion Guide for C*
■ *GCA Application Program Interface*
Open Desktop Server Upgrade:
■ *Administering an ODT-DATA NET Server*
Open Desktop Optional Documentation:
■ *Using ODT-DATA Through Forms and Menus*
■ *ODT-DATA Report-Writer Reference Manual*
■ *ODT-DATA SQL Reference Manual*

Chapter 1
Overview of the Report-Writer

The ODT-DATA Report-Writer provides a language to help you create sophisticated reports without having to write an applications program. The Report-Writer can create regular production reports as well as reports for *ad hoc* applications.

Features of the Report-Writer

The Report-Writer contains the following features:

- **Tools to extract the data you want to print.** For simple reports, you can specify a single table and indicate how you want the information sorted. For a complex report, you can use a query to retrieve selected rows from a database.

- **Support for nulls.** You can include logical operators, null variables, dynamic definition of null strings, and null expressions for specifying how null data should be represented.

- **Control of report appearance.** You can control titles, headings, and the placement of the data on a page. Formatting commands let you specify how numbers and text should be presented. Text formatting includes centering, justification, and automatic pagination.

- **Arithmetic capabilities.** Arithmetic functions make almost any kind of computation possible, including totals and averages over ranges of data.

- **Variables.** Variables are used to assign values in the report specification. You can assign values directly in the report specification, or you can interactively prompt the user to enter the value for a variable.

- **Reports can be run from a file or stored permanently.** You can run reports directly from a file. This lets you test the report interactively. Once the report has been tested, it can be stored in the database.

- **Dynamic report parameters.** Report parameters such as range of data, table names, or any other information can be specified at report time; thus, you can use the same formatting commands for different reports.

Types of ODT-DATA Reports

While the Report-Writer facility permits you to create sophisticated reports, you should also be aware of the alternative report writing facilities available to you. These include:

- **Default reporting facility.** The simplest way to create a report is to access the default report-writing facility through ODT-DATA/MENU. With this facility, ODT-DATA chooses a report format based on the particular table you select.

- **Report-By-Forms.** For more flexibility, you can modify the default report using Report-By-Forms (RBF). This lets you change many of the formatting features of a report.

- **Report-Writer.** The Report-Writer lets you create a custom report specification. You create a text file of Report-Writer statements. The report can be run directly using the **report** command, or it can be stored in a database using the **sreport** command.

This guide describes the Report-Writer facility. For information about the default report or Report-By-Forms, see *Using ODT-DATA* or *Using ODT-DATA Through Forms and Menus.*

Creating the Report Specification

To create a report with the Report-Writer, you follow these steps:

1. Create a report specification.
2. Collect the data.
3. Test the report.
4. Run the report.
5. Store the report specification.

These steps are described in detail next.

Create a Report Specification

To create the report specification, use your text editor to create a file that contains the appropriate Report-Writer statements. This guide documents all the statements that can be included in the report specification file.

Collect the Data

The tables you intend to use for your report must exist on the computer. They must also contain valid data if you want accurate reports. Make sure the tables exist and that you know the names of all the columns.

You may also want to consider queries. Will your report include an entire table or selected rows from a table? If a report is for an entire table, you simply reference the table name. If a report requires a query to extract data, make sure that the query and the tables needed by the query are configured to produce the desired data for the report. For a complex query, you may wish to run the query before you run the report to make sure the query works.

If your report includes a large amount of data, you should use a subset of the data for testing the report. Once you are satisfied with the report, you can use the specified data.

Test the Report

Use a subset of the data to test your report specification. During this phase, you may run the report a number of times on a small segment of data to make adjustments to the specification.

To test the report, use the **report** command with the optional **-i** parameter. This reads the report specifications from the text file and produces the report.

Run the Report

Once the report specification has been tested, you are ready to print reports on the desired tables. Use the **report** command to print the report.

The **report** command reads in the report specifications created by **RBF** or stored by an **sreport** command, performs additional error checking, runs the database query to extract the data (if specified), and writes the formatted report either to a file or to your terminal screen.

Store the Report Specification

When the report has been tested, can store it in the database. Report specifications that are stored in the database are accessible to other users who have access to the database. Reports that are not stored in the database are not accessible to other users.

Use the **sreport** command to store the report specification in the database. **sreport** reads the text file containing report specifications. **sreport** performs rudimentary syntax checking.

If no errors are found, the report specification is added to the database. If the report already exists in the database, **sreport** replaces the old report specification with the new one. If the report does not exist, **sreport** adds the specification into the database and the Report Catalog.

About Queries, Sorts, and Breaks

When you create the report specification, consider how you want the information organized. This includes querying for a subset of data, sorting the data in a logical order, and organizing the printed data by defining breaks. You may want to include summary information such as subtotals. These features are discussed in the following sections.

Queries

Your report can include all the data in a table, or it can include a subset of data that meets a query. The query can contain parameters or variables that are specified when you run the report. The use of parameters in reports is discussed in Chapter 3.

Using Sorted Data

Most reports display sorted data. This makes the report more usable. If you have a report of employees listed by job title within each department, you may need to sort the data in the table. Reports with subtotals require sorting. Sorting is discussed in "Specifying Sorts and Breaks" in Chapter 3.

About Breaks

Breaks are divisions between parts of a report (such as page breaks) or between groups of data in your report (for instance, between data for Employee 1 and Employee 2). You specify breaks between groups of data by designating certain columns in a report as break columns. A break occurs when Report-Writer encounters a change of value in a break column while reading the data.

You can instruct Report-Writer to perform an action after a break has occurred by placing instructions, called break actions, in a header or footer section associated with the break column. For example, you can instruct Report-Writer to print heading information for the next group of data rows, print summary information for the data rows associated with the last break column value, or skip to a new page and print a page header.

Some breaks occur automatically. These include:

- **Start of report.** This break is a change from no data to some data. You can use this break to specify titles and other heading information that appears once at the top of the report.

- **End of report.** This break is a change from some data to no data. You can use this break to specify information that is only printed once, at the end of the report, such as grand totals and footnotes.

- **Detail break.** This break occurs between data rows in a table. This is called a detail break.

You can specify break actions at the tops and bottoms of pages. A page break occurs when the report comes within a specified number of lines of the end of the page. You can define the page size to fit your needs. When a page break occurs, a page footer may be printed, followed by a page header at the top of the next page. You can also print page numbers, the current date or time, values of data items currently being processed by the report, or any number of other items.

Headers and Footers

Headers and footers indicate in your report specification what actions to perform during a break. Headers and footers can be specified at the start and end of the report, at the top and bottom of the page, and at the start and end of a column of data specified as a break column, such as all employees in a department. The footer section can contain instructions for calculating and printing subtotals or other summary information. To calculate this information, you use set functions or aggregates. These are specified in print statements. A header action, if specified, may occur at the start of the report, at the start of a new page, or before the next group of data is processed.

You may specify both footer and header actions for a break column. The footer actions are performed on the previous group of data rows, and the header actions are performed for the group yet to come. At the end of the report, only footer actions are performed, because there is no more data. Similarly, at the start of the report, a break in each of the break columns occurs, and header actions may be performed for each of the major-to-minor break columns.

Detail Section

Report-Writer instructions containing statements used to format, position, and print the data retrieved from the data table are called detail instructions. The detail instructions are grouped together in a detail section.

Summary of a Report Specification

A report specification is a collection of distinct groups of related statements. Some of these statements relate to the overall composition of the report and some relate to action groups. These groups include:

- **The report header.** At the start of the report, you can print text and set up many of the report layout specifications, such as page size and margins.
- **Page headers and footers.** At the top of each page, you can print a page header, and at the bottom, a page footer. These may include titles, page numbers, and the date and time the report was printed.
- **A break header.** Break headers appear at the start of a group of data related to a break column. When a change is detected in a break column, a break occurs. Before a new group of data rows is processed, the break header actions are performed. Break headers can be used to title information in a break.
- **Detail section.** This contains the instructions on how to format and print the report data. The detail break is the only break that does not include a header and a footer.
- **A break footer.** Break footers appear at the end of a group of data related to a break column. Break footers can print subtotals and related information associated with the data rows just processed.
- **The report footer.** The report footer can include text, footnotes, or summary information for the whole report.

Sample Report

The following is a listing for a simple report specification using the ODT-DATA Report-Writer. The Report-Writer specification was created with a text editor, processed with the **sreport** utility, and run with the **report** command.

The report shows a titled listing of data from an existing view in a database. The "jobcat" column is displayed only once for each job category value.

```
                /*   Sample report   */

.NAME sample
.DATA edat
.SORT jobcat, name
.HEADER report
      .NEWLINE2
      .CENTER
      .PRINT 'Sample Report'
      .NL2
.HEADER jobcat
      .TFORMAT jobcat("  zzzz  ")
.DETAIL
      .PRINT jobcat(b8), name(c15), dept,
            code, age, sales(f12.3)
      .NL
```

The statements in this specification work as follows:

- The **.name** statement gives a name to the report. This name is placed in the Reports Catalog by the **sreport** facility, and it is used by the **report** command to locate the report specifications.
- The **.data** statement identifies an existing table or view in the database that contains the data to report.
- The **.sort** statement indicates the order the data is displayed in the report.
- The **.header report** statement indicates that the following Report-Writer statements are part of the report header.
- The **.newline**, **.center**, and **.print** statements position and print a title.
- The **.header jobcat** statement indicates that the following statements are part of a break header associated with the "jobcat" break column. This header is printed any time the value in the "jobcat" column changes.

- The **.tformat** statement temporarily changes the normal print format of the "jobcat" column, but only on the next printing of "jobcat." This occurs in the **.detail** section. Normally, "jobcat" is not printed. Its format is (b8), which means "print 8 blank spaces." The **.tformat** statement makes a "one printing" change to the format so the actual value of the "jobcat" column is printed.

- The **.detail** statement indicates that the following statements are the start of the detail section. The **.print** statement prints out the values of the columns in the formats given after the column names, or the default format for that type of data item, if no format is specified. The format specifications, which appear in the parentheses following the column names, are described in Chapter 7.

The table below shows the data on which the report was run.

Data for the Sample Report

Column Name	Type	Length	Nulls	Defaults
jobcat	integer	4	yes	no
name	c	15	yes	no
dept	c	6	yes	no
code	integer	1	yes	no
age	integer	2	yes	no
sales	money		yes	no

jobcat	name	dept	code	age	sales
10	Adams,Joe	toy	0	22	$ 10,500.00
10	Green,James	toy	0	34	$ 43,645.00
10	Smith,Tony	acct	0	48	$ 8,690.00
20	Davis,Miles	music	0	56	$234,987.00
20	Tanhaus,Karl	music	0	20	$ 18,765.00
30	Jones,Mary	acct	1	34	$ 34,599.00
30	Maney,Sikkim	none	1	51	$ 15,333.00
30	Mellon,Tim	toy	0	44	$ 67,876.00
30	Mellon,Tim	any	0	24	$ 45,098.00
30	Norris,Bill	acct	0	26	$ 23,988.00

Chapter 2
Report Specification Statements

To specify a report, you create a text file that contains Report-Writer statements. These statements define the data, the sort order, the page layout, the position and format of titles and text to be inserted in the report, and the position and format of the data.

Before you begin a report specification file, consider the following:

- What data do you need for the report? If you need to run a database query, design the query and run it independently to make sure it retrieves the correct data.
- Will the report be reproduced with different values each time it is run? If so, you need to assign variables and report parameters.
- Will the data be sorted? If you want headers or footers for subgroups of your data, the data must be sorted on the columns that define the subgroups.
- What will the headers and footers to look like? Do you want titles, subtotals or other aggregates, and extra blank lines? Sketch the report layout on a piece of paper to see how it will look.
- What will be printed for each data row? In what format should the information appear? For numbers, think about the number of significant digits to print, and the number of decimal places.
- What kind of page headers and footers do you want?

Once you have identified these elements, you are ready to begin creating a report specification file.

Types of Report Specification Statements

Report specification statements fall into several groups: report setup statements, page layout and control statements, report structure statements, column and block statements, text-positioning statements, print statements, and conditional and assignment statements. These statements are introduced by group in the following sections.

Report Setup Statements

Statements for setting up the report environment include:

.name	Names the report.
.shortremark	Provides a short description of the report. This is included in the Reports Catalog.
.longremark **.endremark**	Mark the beginning and end of a long description about the report. This description is included in the Reports Catalog.
omments	Comments may be placed in the report specifications file if preceded by /* and followed by */. Comments are ignored in report processing.
.output	Sets up an external file to receive the report.
.data **.query**	Define the data for the report.
.sort	Defines the order in which to sort the data for the report.
.break	Specifies the break columns for the report and the order in which to process breaks.
.declare	The **.declare** statement declares variables.

Page Layout and Control Statements

You can specify the page layout of the report with the following statements:

.pagelength	Defines the page length, in lines.
.formfeeds	Inserts formfeed characters to force a page break at the start of the report and at the end of each page.
.noformfeeds	Suppresses formfeeds within the report.
.leftmargin	Sets up a left margin for the report lines that follow the statement. If the left margin is not specified, Report-Writer determines this default automatically. (See "Automatic Determination of Default Settings" in Chapter 3 for details.)
.rightmargin	Sets the right margin of the report for use with the **.right** and the **.center** statements. If the right margin is not specified, Report-Writer determines this default automatically. (See "Automatic Determination of Default Settings" in Chapter 3 for details.)
.need	Tests for a given number of lines on a page to see if a page break is appropriate.
.newpage	Skips to a new page, and optionally sets a page number.

Report Structure Statements

The statements used to set up the structure of the report include:

.header	Designates a group of formatting statements as a header. This can be a report header, a page header, or a break header.
.footer	Designates a group of formatting statements for the footer.
.detail	Designates a group of formatting statements for each data row.

Column and Block Statements

The following statements specify the print position, column width, and format for the specified database column or for a report block (as defined by a **.block** statement).

.format	Specifies a print format for a column, such as a character string or a decimal notation.
.tformat	Temporarily changes the print format for a column, only for the next value to be printed. This statement has several uses; for example, it can be used to print a value of a column on the first line of a page, or to print a currency symbol at the top of a column of currency values.
.position	Defines the starting position for a column, which can be used with the **.tab**, **.right**, **.left**, or **.center** statement.
.width	Defines the width for a column, to be used with the **.right** or **.center** statements.
.block **.endblock**	Treat sections of the report as blocks, so that you can refer to positions on previous as well as on subsequent lines in the report. These statements can be used with the **.top** and **.bottom** statements to align blocks of data adjacent to each other, rather than in vertical sequence.
.top	Moves the current position to the top line of the current block, used while in block mode.
.bottom	Moves the current position to the bottom line of the current block, used while in block mode.
.within **.endwithin**	Allow you to set the report margins temporarily to the confines of a specific column, using the column position and width.

You should also read the following discussion on "Text Positioning Statements," and "Automatic Determination of Default Settings" in Chapter 3, for additional information.

Text Positioning Statements

The following statements are used to position text. The positioning can be absolute or relative to other positions on the page. See the preceding section "Column and Block Statements."

.tab	Tabs to a specified position before printing. The tab can be in reference to a column name or the default print position for a column.
.newline	Prints the current line and skips to the start of a new line.
.center	Centers text. The position may be the center of the page or the center of a column. For a column, the margins may be either specified or default.
.right	Right justifies text to the right margin or to a specified position, for either the report or a column in the report.
.left	Left justifies text to the left margin or to a specified position, for either the report or a column in the report.
.lineend	Tabs to the end of the text on the current line before continuing to print.
.linestart	Tabs to the left margin before continuing to print.

Print Statements

You use these statements to print text or data values in a report.

.print	Prints text or values at a default position, or at a position that was previously specified with the column and block or text-positioning statements. The text or value to print is specified in an expression in the **.print** statement syntax. Expressions may include any column names from the data retrieval statement, program variables, constants, functions, aggregates, report variables (time, date, day of week, or page number) or run-time parameters. You can optionally indicate the print format within the syntax of the **.print** statement, or you can specify it in a separate **.format** or **.tformat** statement for column values. (See the previous section "Column and Block Statements.")
.nullstring	Specifies a string of characters you want to print in the report, whenever a null value is encountered in the data.
.underline **.nounderline**	Control underlining for sections of text.
.ulcharacter	Sets up a different underline character from the default, for use with the Report-Writer underlining statements.

Conditional and Assignment Statements

.if **.then** **.else**	Specify blocks of statements to execute under specified conditions.
.let	Assigns a value to a variable, which can be used in subsequent computations.

Format for Specification Statements

Every formatting statement is specified with a keyword, preceded by a period (.). The keyword may be followed by parameters. The format for a report specification statement is:

```
.statement {parameters}
```

where

`statement` One of the text formatting statements, such as **.data** or **.tab**. The statements can be upper- or lowercase letters.

`parameters` Parameters may be optional. Their form is dependent on the specific statement. The space between the statement name and parameters is optional if the parameter does not start with a letter. For example, ".nl 2" is the same as ".nl2".

Here are some examples of report-formatting statements; they include a tab example, a **.newline** statement, a heading, a print line, and a **.sort** statement:

```
.tab 10

.newline

.heading report

.pr "This is the value of:",abc(f10.2),
                  " Sum:",sum(def)

.sort a,b,c
```

Report-formatting statements within the text file end with the start of a new statement. Statements can span any number of lines. Except where noted, spaces are used as separators of statements, and commas separate multiple entries for a parameter within statements such as **.sort** or **.print**. The example reports in Appendix A demonstrate the correct specification of statements.

Chapter 3
Using Report-Writer

The initial setup statements must appear at the beginning of your report specification file. The setup statements perform the following tasks:

- Name the report
- Set up a report results file
- Specify the table, view, or query from which data is to be obtained
- Define the order in which the data is to be sorted
- Define the break columns for the report
- Declare any variables used in the report specification
- Enter optional remarks and comments

You use the **.name** statement to name the report, the **.output** statement to set up the output file, and the **.declare** statement to declare variables. These statements are discussed in Chapter 5, "Report Setup Statements."

To obtain the data for your report, you use either the **.data** or the **.query** statement. You cannot use both. The **.data** statement names a table or view from which data is obtained. The **.query** statement retrieves a subset of the available data, based on the results of the query. You can include parameters or variables in the query. This lets the user specify the criteria for the report at runtime. For more information on queries and data retrieval, see "Creating Reports Parameters" and "Creating Reports Using Several Tables" later in this chapter.

To sort the data for your report, you include a **.sort** statement in your report specification. The **.sort** statement lists the columns, in sort order, on which the data is sorted. You also specify the break columns, using the **.break** statement, if you want breaks to occur between data items in columns other than those specified in the **.sort** statement. For more information on sorts and breaks, see "Specifying Sorts and Breaks" later in this chapter.

Your report-specification file can also include descriptive text about your report. The **.shortremark** and **.longremark** statements may be used to include text that appears in the Reports Catalog. Comments may be placed anywhere in the report specification by enclosing them with the comment delimiters /* and */. These are discussed in Chapter 5, "Report Setup Statements."

Creating Reports Parameters

For flexibility, you can design one report specification to be run with different parameters or variables that are specified by the user at runtime. Declared variables can also be assigned a value in the report-specification file.

Using parameters or declared variables in the query lets the user retrieve data that meets particular needs. For instance, the user could obtain a report on a single employee or on all employees in a specified department by entering the employee name(s) or the department name(s) at runtime. The parameters may be entered on the command line when the report is invoked, or they may be entered in response to a prompt.

To create a report parameter, you specify the parameter or declared variable in the query, preceded by a dollar sign ($). This tells the Report-Writer to accept a value entered on the command line, or to prompt the user interactively if a value is not specified on the command line.

For example, suppose you have a banking database in which you keep a table of customer accounts. In this table, you have fields for customer names ("custname"), customer account numbers ("custno"), checking account balances ("checking"), and savings account balances ("savings"). You want to create two reports. They should be identically formatted, but must present different information; one report should provide checking account balances, and the other should give savings account balances. To accomplish this task, you might write a query like this:

```
.query select custno, custname,
     val=$Account_type
     from account
```

As the Report-Writer generates your report, it prompts you to enter an account type (savings or checking). Your response tells Report-Writer which kind of information it should retrieve with the query. If you responded to the prompt with "checking," the completed query would look like this:

```
select custno, custname, val=checking
     from account
```

If you use a parameter in your query, Report-Writer uses a default prompt string when prompting for its value. If you use a declared variable in the query, Report-Writer uses a customized prompt instead. To create the customized prompt, you must use a declared variable rather than a parameter in the query, and use the **with prompt** option in the **.declare** statement to specify the prompt string.

You can use parameters and declared variables in titles and other places within the report. When used outside of the query, declared variables need not be preceded by the dollar sign ($). Parameters, however, must always be preceded by a dollar sign ($).

For more detailed information on using parameters and declared variables in reports, refer to sections on the **.query** and **.declare** statements in Chapter 5.

Creating Reports Using Several Tables

There may be times when you want to use the Report-Writer to produce a report from related information scattered across several tables that share one or more column definitions. The Report-Writer cannot by itself construct such a report, because it does not recognize multiple queries. It can, however, construct a report from a view you define for one or more tables, or from a join of several tables that you specify in an SQL **select** statement.

An example of joining tables for a report is given in Appendix A, "Joining Tables Example."

Specifying Sorts and Breaks

To produce an easy-to-read report, you may want to sort the data on the basis of one or more of the columns. You must sort the data if you want to include subtotals or other summary information in your report. You also specify the break columns to signal Report-Writer to look for subtotaling or other special statements. For example, the first POPULATION sample report in Appendix A is a 1970 U.S. population report by region and state. To generate the regional population subtotals, the states must first be grouped by the value of the "region" column in the database, and breaks must occur at each change of value in the "region" column.

The easiest way to group rows is to sort them on the column that is used as the grouping column, such as "region" in the POPULATION example report. Often, a report is sorted on more than one column. In such cases, the rows are grouped on the basis of the first sort column (called the major sort column) and, within those groups, on the basis of the next sort column (called a minor sort column), and so forth. The sort order is specified by naming the columns in the **.sort** statement in a section containing report setup statements (as discussed in Chapter 5). The **.sort** statement can be used whether or not there is a database query, and provides an alternative to sorting via an **order by** clause in a **.query** statement.

By default, Report-Writer assumes the break columns are the same as the sort columns. In the above example, for instance, no other breaks need be specified. However, the default breaks can be overridden by specifying break columns with the **.break** statement. (See Chapter 5 for details.) The currently active list of break columns (specified by either the **.sort** or the **.break** statement) is known as the break list. The first column in the break list indicates a major break column, while those which follow are considered minor break columns. A break on one break column automatically produces a break on all subsequent break columns in the currently active break list.

In the ACCOUNT example report in Appendix A, break columns are not explicitly specified, so breaks occur on the sort columns. The data is sorted on "acctnum" (the major sort column) and, within "acctnum," on "date." When a change occurs in the value of "date," the "date" break occurs and the system looks for formatting instructions. When a value changes in the "acctnum" column, breaks occur in both "acctnum" and "date."

You do not need to specify actions for every break in your report. You may wish to specify sort columns (which produce breaks) for appearance. In the POPULATION example in Appendix A, breaks in "region" invoke a number of summary and heading actions, whereas breaks in "state" do not.

Under certain conditions, such as with rounded numbers in break columns, the breaks occur when the formatted values change, not when the actual values change. For example, assume a column is rounded to the first decimal place. There is no break between the values of "35.87" and "35.92", since each rounds to "35.9." You have control over how numeric values are rounded through the format specification. (See "Format Specifications" in Chapter 4.) To force breaks to occur on the actual values rather than on the formatted values, use the **-t** flag on the **report** command line, as described in Chapter 12.

Pagination in Reports

Pagination in the report is controlled by a number of statements. The **.pagelength** statement specifies the vertical size of pages, in lines. The statements placed in the page header and footer sections are used to define actions taken at the start and end of pages. The **.newpage** and **.need** statements force page breaks, and the **.formfeeds** statement sends a formfeed character to the printer after printing all lines that fit on the defined page. Line numbering begins at 1 (top line).

Before the Report-Writer begins to print a report, it calculates the number of lines in the page header and footer. After each line is printed, Report-Writer compares the page length with the number of lines printed. If there are only enough blank lines left to write the page footer, the Report-Writer prints the page footer, issues a formfeed (if specified) for a page break, updates the page number, and prints the page header for the next page.

If the **.formfeeds** statement is in effect, the formfeed character is inserted at the start of the report and at the end of each page. In some cases, the **.formfeeds** statement is not needed. For instance, the **.print** statement automatically inserts formfeeds appropriate for 11-inch paper if the default page length (61 lines) is used.

The following shows the commands to create and print a report file. This example assumes the default value of 61 lines per page. It does not require the **.formfeeds** statement.

```
% report -frepfile.lis mydb myreport
% lpr repfile.lis
```

For a format that uses 66 lines per page, you can add a flag of **-v66** at the end of the **report** command line, or you can use the **.pagelength** statement in the report specification.

For special forms and other printers, you can use the **.formfeeds** statement to instruct the Report-Writer to insert formfeeds, or the **.noformfeeds** statement to prevent them.

The **.newpage** statement forces a page break at any point in the report. This statement causes Report-Writer to skip to the bottom of the page and print a page footer, if one is specified, and then skip to the top of the next page.

The **.need** statement forces a page break to occur if the remaining available lines on the page are fewer than the number of lines specified in the **.need** statement. It is used to keep lines of text together on the same page. For instance, this statement may be used prior to a break header to insure that enough lines remain on the current page to print the entire break header.

For detailed information on page control statements, see Chapter 6.

Setting Report Margins

Report-Writer can determine report margins by analyzing your report code. In most cases, the default settings generated by Report-Writer are adequate. In some cases you may want to define these settings explicitly, using the **.leftmargin** and **.rightmargin** page-layout statements. Horizontal character positions start at the left margin (position 0).

In some reports, the right and left margins change dynamically. (See the DICTIONARY example in Appendix A.) In these cases, the margins for the page header and footer are independent of the margins for the rest of the report. These margins may be determined automatically, or they may be specified with the margin-setting statements **.leftmargin** and **.rightmargin** withing the page header statements.

For information on margin setting statements, see Chapter 6.

Positioning, Formatting, and Printing Data

The Report-Writer relies on three different groups of statements to print data in the correct place and format. These are:

- Column and block default setting statements
- Text-positioning statements
- Print statements

These statements are used to:

- Set default print positions and widths for columns
- Position text explicitly, or left justify, right justify or center column values within the margins defined by the column defaults

- Define the print format (character string, decimal, and so on) for the value to be printed
- Print an explicit value or print the next value in a column at the previously defined position, in the designated format

The process of positioning, formatting, and printing data is described below.

Setting Default Print Positions for Columns

Before you can print a value, you must indicate where it should be printed. Report-Writer can automatically determine default column print positions. To set your own defaults, use the following column and block statements:

- **.position**
- **.width**

The **.position** statement lets you set up margins for each column, setting the starting print position for a column and, optionally, the width of the printed column in number of characters. You can also set the width of a column with the **.width** statement. All horizontal print positions start at the left margin (position 0).

To print columns adjacent to each other, you reference the column names within the same **.print** statement, separated by commas. If possible, Report-Writer prints the columns next to each other, at the positions specified in the **.position** statements or at default print positions.

In some cases, you may want to use the following block statements for more control over the printing of adjacent text:

- **.block** and **.endblock**
- **.top**
- **.bottom**
- **.within** and **.endwithin**

The **.block** and **.endblock** statements define a block of formatting and print statements as a unit. You use the **.top** or **.bottom** statement to reset the current line to the top or bottom of the defined block before processing the next statement. The **.within** and **.endwithin** statements temporarily set the report margins to the margins for a referenced column. This enables you to print text (such as the caption "Total") within the column margins without having to calculate the exact print position.

Column and block statements are discussed in Chapter 8.

Positioning Text

In addition to the column and block statements, you can use text-positioning statements to position the text or data. The text positioning statements are:

- **.tab**
- **.newline**
- **.left**
- **.center**
- **.right**
- **.lineend**
- **.linestart**

You may use the **.tab** statement with a column name to tab to the assigned print position for that column before issuing a **.print** statement. In addition to tabbing, text-positioning statements allow you to center or justify text within the column margins, or to position text at the beginning or end of a line, or on another line.

You may use the text-positioning statements with values instead of column names. Explicitly set positions override column defaults. Text-positioning statements are summarized in Chapter 2 and discussed in Chapter 9.

Specifying the Print Format

The appearance of the text or data in your report is controlled by the format specification. For instance, the **c** format indicates a character string format and the **e** format causes a value print in scientific notation. You specify the format with a template such as "**$zz,zzz.nn**," containing characters that define how a value prints.

The print format may be specified in the **.print** statement, or it may be used in a **.format** statement to set a default print format for a column, as in the following:

```
.format emp (c12), sal ("$zz,zzz,zzn.nn")
.print emp,sal
```

The results look like this:

```
Jones              $    109,224.00
Smith              $     32,575.00
```

You can temporarily override a default column format with the **.tformat** statement to print the next value only in a different format. After the value is printed, the format returns to the original default type. This is useful for printing a dollar sign at the start of a page.

You can also override a default format by specifying the format as a parameter in the **.print** statement, such as:

```
.print salary ("zz,zzz,zzn.nn")
```

This prints the "salary" values in the specified format, without the dollar sign, until it encounters another format or print statement for this column. For more information on print formats, see "Format Specifications" in Chapter 4.

You may indicate underlining of text or values using the **.underline** and **.nounderline** statements. Any **.print** statements located between the **.underline** and **.nounderline** statements produce underlined text. By default, the underline character is the hyphen (-) for reports written to a terminal, or the underscore (_) for reports written to a file. You can change the default to any character, using the **.ulcharacter** statement. All underline characters are printed on the line below the text, except for the underscore (_) character, which appears on the same line as the text. For more information on underlining, see Chapter 10.

Specifying What to Print

The actual text or value to print is specified as an expression in the **.print** statement syntax. The expression can be a column name, a constant, a function or an aggregate, a runtime report parameter such as the current date and time, or a variable whose value is specified on the command line with a prompt or a **.let** statement. The use of expressions is discussed in Chapter 4.

By default, Report-Writer prints an empty string when a null value is encountered. If you wish, you may change this default to any string of characters, using the **.nullstring** statement. For instance, you can tell Report-Writer to print the string "none" wherever it finds a null value in the data.

For more information on the **.print** and **.nullstring** statements, see Chapter 10.

Using Conditional and Assignment Statements

You may use the conditional **.if**, **.then**, and **.else** statements to tell Report-Writer to execute blocks of statements, under specific conditions. For example, you could execute alternative **.print** statements to suppress confidential data, based on a user's ID number.

The condition in an **.if** statement is a Boolean expression that returns the value true or false. Each of the following is a condition:

- a clause
- a Boolean function
- **not** *condition*
- *condition* **or** *condition*
- *condition* **and** *condition*
- (*condition*)

Examples of conditions in **.if** statements are:

```
age <= 50
not (age <= 50)
(age <= 50) and (salary >= 40000) and
      (job = "programming")
age > avage
```

The **.let** statement assigns a value to a declared variable. For instance, you could calculate the number of years that have elapsed since an employee was hired, and assign the result to a variable for a report on employee longevity. The **.let** statement can be used with the **.if**, **.then**, and **.else** statements.

For a detailed description of conditional and assignment statements, see Chapter 11.

Calculating and Printing Summary Data

You may use set functions or aggregates such as **sum** or **count**, as well as arithmetic and other built-in functions to calculate subtotals and other summary values to print in a report. An aggregate, arithmetic operation, or function can be specified in the **.print** statement, or an expression containing the operation can be used in a **.let** statement to assign the calculated value to a variable prior to printing.

For a detailed discussion of aggregates, operations, and functions, see Chapter 4.

Automatic Determination of Default Settings

Report-Writer can automatically calculate default settings for the right and left margins of the report, for the starting position and width of each column (for use with the **.tab**, **.right** statements, and so on), and for the formats to use when printing columns. These are only calculated when they have not been specified. The default settings are determined by analyzing the other report-formatting statements. This takes place after the report setup and page layout statements (such as **.leftmargin**) are processed, and before the first printing of the report.

Analysis of Report-Formatting Statements

To determine default values, Report-Writer analyzes the formatting statements in reverse hierarchical order, from the innermost (detail level) statements to the outermost (report level) statements, as shown below:

1. **.detail** section statements
2. **.footer** statements for innermost sort column
3. **.header** section for innermost sort column
4. **.footer** and **.header** sections for next to last sort column, and so on
5. footer and header text for the report

In analyzing the report code, the Report-Writer determines the innermost references to columns in the report, and the leftmost and rightmost print positions indicated by the specified report-formatting statements.

Determining Default Margins

If the margins for the report are specified with the **.leftmargin** and **.rightmargin** statements, these values are used. If not, the minimum and maximum print positions for a line in the report are determined in the scan of the report-formatting statements. If only one of the margins is specified, the other is determined in the scan. The margins are used to determine line positions for the **.center**, **.right**, and **.left** statements, when these statements are used without specified parameters.

Determining Default Column Positions

If no **.position** statement is given for a column, its default position for use with the **.tab**, **.right**, **.left**, or **.center** statement is determined from the analysis of report-formatting statements. Default column positions are determined by the first print position Report-Writer encounters that has been specified for the printing of a value in that column or for an aggregate of that column.

Reports are set up so that the innermost printing of column values occurs in the **.detail** statements of the report. Column headers and aggregates, which print in header or footer text for a break, can then use the **.tab** or another positioning statement in relation to the default position established for the innermost position of a column. If changes are desired in the position of a column and its associated heading or aggregates, only the innermost print position for the column need be changed. Because all references to header, are given in relative terms, their positions are changed automatically.

As an example, see the ACCOUNT example in Appendix A. The default position for the "amt" column is determined by the cumulative aggregate for "amt."

Determining Default Column Formats

If no **.format** statement is given for a column, the default format is determined in a manner similar to that used for determining the default column position. The innermost reference to a format for a column, or to an aggregate for a column, is used as the default format for the column. If no formats are given for a column, the Report-Writer determines defaults from the data type of the column, as described in the discussion entitled "Default Formats" in Chapter 4.

The default format for a column is best used in situations where the format is specified in the reference to a column in the **.detail** formatting statements. Aggregates of that column are then specified in the footers for some of the breaks. The Report-Writer then correctly uses the format specified in the **.detail** section for the aggregates.

However, the **.format** statement is often useful for specifying a series of columns with the same format. See the POPULATION example in Appendix A for a good illustration of the use of the **.format** statement for this purpose.

Determining Default Column Widths

If no **.width** statement or **width** parameter to the **.position** statement is specified for a given column, the default column width is determined by the default format for that column, as specified by the **.format** statement or as determined from the analysis of report-formatting statements. The default width of a column is the width required by the column format to print a value. Report-Writer uses the column width to determine the print positions for the **.right** or **.center** statements.

Chapter 4
Expressions and Formats

Report-Writer accepts a variety of expressions. These may be used in queries, in conditional and assignment statements, and in **.print** statements. Expressions are data elements that may be combined with operators and functions. They may include the following:

- constants
- column names
- parameters
- variables
- aggregates
- arithmetic operators
- comparison operators
- logical operators
- functions

Expressions may be used in the **.query** statement to retrieve a subset of the data. (See Chapter 5 for information on queries.) They may be compared to other expressions with the **.if** statement, or used in the **.let** statement to assign a value to a variable. (See Chapter 11 for conditional and assignment statements.)

Expressions may be printed using the **.print** statement. (See Chapter 10 for details on the **.print** statement.) The format specification determines how the data is printed. It may be as a character string, in decimal or scientific notation, and so forth. Report-Writer uses a default format if you do not specify one in the **.print** statement or with a **.format** or **.tformat** statement. Formats are discussed in "Format Specifications" later in this chapter.

The following example shows several expressions. The example uses a database that has a table of shipments featuring part number, number of defective parts in a shipment, and the total number of parts in a particular shipment. Suppose you want a report of the shipments grouped by part number, with the calculated percentage of defective parts for all the shipments of that part. The following accomplishes this:

```
.sort partno
  .
  .
  .
.footer partno
     .print partno, " IS "
     .print (sum (defective)/sum (total)) * 100, " %
DEFECTIVE "
     .newline
```

In this example, the following are expressions:

```
partno
"IS"
(sum(defective)/sum(total)) * 100
"% DEFECTIVE"
```

Because no print formats are specified in this report code, Report-Writer automatically determines them.

Reserved Words

The following table lists reserved words. They should not be used in any other way. Using reserved words in other ways, particularly as column names, produce unexpected or incorrect results when the Report-Writer prints the report.

abs	**current_date**	**line_number**	**page_length**
and	**current_day**	**locate**	**page_number**
ascii	**current_time**	**log**	**position_number**
atan	**date**	**lowercase**	**report**
average	**detail**	**max**	**right**
averageu	**dow**	**maximum**	**right_margin**
avg	**exp**	**maximumu**	**run**
avgu	**float4**	**maxu**	**shift**
break	**float8**	**min**	**sin**
cnt	**int4**	**minimum**	**smallint**
cntu	**integer**	**minimumu**	**sqrt**
concat	**integer1**	**minu**	**squeeze**
cos	**integer4**	**mod**	**sum**
count	**interval**	**not**	**sumu**
countu	**left**	**null**	**trim**
cum	**left_margin**	**or**	**uppercase**
cumulative	**length**	**page**	**w_column**
			w_name

If you use one of the reserved words in the preceding table as a column name, the Report-Writer does not issue an error message. It supersedes the definition of the built-in function with the column name you specify. All further references to the reserved word is to the column, not to the Report-Writer function. This can produce unexpected results. For example, if you had a column in your retrieval named "page," the built-in definition for the name "page" would be replaced by your definition. After that, when you used a **.page** statement, you would actually get the column name "page."

Types of Data in Expressions

Expressions may contain any of the data elements described below.

String Constants

Many reports have lines of text, or strings that appear in the body of the report. You can specify these string constants by enclosing them in single or double quotation marks. For example:

```
'string'
```

or

"string"

where

string

is any character string.

If you use single quotes as the string delimiter and you wish to include a single quotation mark within the text of the string, you must enter it as two single quotes so that the Report-Writer does not assume it has found the end of a string. Such a pair of single quotes must be placed together on a single line. A backslash (\) within a single-quoted string is automatically interpreted as a literal backslash, unless it precedes a wild card character. (See the following explanation.)

If double quotes are used as the string delimiter, a double quotation mark (") or a backslash (\) within the string must be preceded by a backslash to be interpreted literally.

Examples of valid strings delimited by single quotes are:

```
'This is a string'

'This   has  extra  blanks'

'This has a "quoted" string in it'

'This has one \ backslash in it'
```

Examples of valid strings delimited by double quotes are:

```
"This is a string"

"This   has  extra  blanks"

"This has a \"quoted\" string in it"

"This has one \\ backslash in it"
```

In most cases, you can choose single or double quotes for the string delimiter; you must use single quotes within an SQL **.query** statement. As a convention, this manual uses double quotes to delimit string constants, except within SQL **.query** statements.

Numeric Constants

Numeric constants consist of an integer, a decimal point, and a fraction or scientific notation. Numeric constants may be specified with the following format:

[+|-] {*d*} [.{*d*} [e|E[+|-]*d*[*d*]]]

where *d* is a digit

Examples of valid numeric constants are:

```
23
8.97327
4.7 e-2
```

Numeric constants may range from -10**38 to +10**38 ("**" being interpreted as "to the power of") with precision to 17 decimal places.

Date Constants

Dates are referenced as single- or double-quoted character strings. (Just as with string constants, however, within a **.query** statement, you must use the quotation marks appropriate to your query language.) The Report-Writer accepts formats described below.

Absolute dates. Legal formats for input of the date November 15, 1988, are shown in the following table:

Absolute Date Formats

Format	**Example**
"*mm/dd/yy*"	"11/15/88"
"*dd-mmm-yy*"	"15-nov-88"
"*dd-mmm-yyyy*"	"15-nov-1988"
"*mm-dd-yy*"	"11-15-88"
"*yy.mm.dd*"	"88.11.15"

Format	Example
"*mmddyy*"	"111588"
"*mm/dd*"	"11/15"
"*mm-dd*"	"11-15"
"today"	The string **today** is a legal absolute date with today's date as its value.
"now"	The string **now** is a legal absolute date and time with today's date and the current time as its value.

Absolute times. Legal formats for input of the time 10:30:00 are shown in the following table:

Absolute Time Formats

Format	Example
"*hh:mm:ss*"	"10:30:00"
"*hh:mm:ss xxx*"	"10:30:00 pst"
"*hh:mm*"	"10:30"

Note: ODT-DATA supplies the appropriate time zone designation. Time formats are assumed to be on a 24-hour clock. Times entered with designations of "am" or "pm" are automatically converted to 24-hour internal representation. Any such designation must follow the absolute time and precede the time zone, if included. If you do not specify a date with an absolute time, today's (that is, the current day's) date is supplied.

Absolute date and time. Legal input formats for November 15, 1988, 10:30:00, are shown in the following table:

Absolute Date and Time Formats

Format	Example
"*mm/dd/yy hh:mm:ss*"	"11/15/88 10:30:00"
"*dd-mmm-yy hh:mm:ss*"	"15-nov-88 10:30:00"
"*mm/dd/yy hh:mm:ss xxx*"	"11/15/88 10:30:00 pst"
"*dd-mmm-yy hh:mm:ss xxx*"	"15-nov-88 10:30:00 pst"
"*mm/dd/yy hh:mm*"	"11/15/88 10:30"
"*dd-mmm-yy hh:mm*"	"15-nov-88 10:30"
"*mm/dd/yy hh:mm xxx*"	"11/15/88 10:30 pst"
"*dd-mmm-yy hh:mm xxx*"	"15-nov-88 10:30 pst"

Date intervals. Examples of valid formats for date intervals include the following:

"5 years"
"8 months"
"14 days"
"5 yrs 8 mos 14 days"
"5 years 8 months"
"5 years 14 days"
"8 months 14 days"

Time intervals. Examples of valid time intervals are:

"23 hours"
"38 minutes"
"53 seconds"
"23 hrs 38 mins 53 secs"
"23 hrs 53 seconds"
"28 hrs 38 mins"
"38 mins 53 secs"
"23:38 hours"
"23:38:53 hours"

Columns

To reference a column value in a data row currently being processed, specify the column name. Columns for SQL data types: integer1, smallint (integer2), integer (integer4), float4, and float (float8) are numeric expressions. Columns for the SQL data types c, char, text, and varchar are character expressions. Columns for the SQL data types date and money are abstract expressions.

Parameters

You may specify parameters for runtime substitution in expressions. To indicate parameters, you must precede an alphanumeric name with a dollar sign ($). Examples of parameters are:

```
$myvar
$your_name
$salary
$start_date
```

You can use parameters as substitutes for any part of a query: field names, table names, or even **where** clauses. For example, you may specify a report with the following query:

```
select *
     from emp
     where dept = '$dname'
```

When the report runs, you enter the parameter value on the command line. If you do not enter a value on the command line, the Report-Writer prompts you to enter the value.

Parameters used in a query may be used in other parts of the report specification. Wherever a parameter is used, it must be preceded by the dollar sign ($).

If the parameter is used as a number, its value must be a real number. If the parameter is used as a date, its value must be a legal date. Otherwise the parameter is treated as a character string.

Declared Variables

You may use declared variables in place of, or in addition to, parameters for the runtime substitution of values in an expression. The value must be specified in one of these ways:

- On the command line
- In response to a prompt string you specify with the **.declare** statement
- In a **.let** statement

The advantage of a declared variable over a parameter is that you can create your own prompt. Parameters use a standard Report-Writer prompt. You can specify the data type and null for a declared variable. (See the **.declare** statement in Chapter 5.)

As with parameters, you can use declared variables as substitutes for any part of a query. When used in a query, the declared variable must be preceded by a dollar sign (S). (For more information on the **.query** statement, refer to Chapter 5.)

Declared variables can also be assigned values by means of the **.let** statement for use within the body of the report. When used outside of the query, declared variables should not be preceded by a dollar sign ($). See the **.let** statement in Chapter 11.

Special Report Variables

The following report variables may be used to generate and print such items as page numbers and the date and time a report is run, or to control the report layout.

Special Report Variables

Name	Description
page_number	Current page number in the report. Pages number from 1.
line_number	Current line number on the page. Starts at 1.
position_number	Current column position on the page. Starts at 0.

Name	Description
page_length	Current length of the page.
left_margin	Current left margin column position.
right_margin	Current right margin column position.
current_date	Date when report is run.
current_day	Day of the week when report is run. This is a three-character string (for example, "Mon" or "Fri").
current_time	Time of day when report is run. This is a date.
w_name	Name of the column currently being used in a block. This is a string.
w_column	Value of the **w_name** column in the data row currently being processed.

Aggregates

An aggregate, such as **sum** or **count**, is used to perform a calculation on data read in from one column, up to the occurrence of a break in another column. For instance, in the POPULATION example report shown in Appendix A, the regional population subtotals represent use of the **sum** aggregate on each of the columns "tot," "totwhite," "totblack," and "totother," up to a break in "region." Additionally, the population totals at the end of the report represent use of the **sum** aggregate for the same columns up to a break in "report."

You specify which data should be used in the calculation by naming the column containing that data as a parameter of the aggregate function. In the POPULATION example, the columns containing the relevant data are "tot," "totwhite," "totblack," and "totother.") You indicate the cut-off point for the data to be included in each calculation by placing the aggregate function within the footer section for a particular column or section of the report. The aggregate value is calculated each time a break occurs in the specified footer. SQL users should note that aggregates correspond to the **set functions** of SQL.

Aggregates may be non-unique or unique, simple, or cumulative. A non-unique aggregate performs a calculation based on every value read in from the aggregate column up to a break in the specified footer. A unique aggregate performs a calculation on each break value in the aggregate column, up to a break in the specified footer. Depending on how the data is sorted and where the aggregate is specified, the break values may or may not be the actual unique values in a column. A simple aggregate produces a single value, calculated on all the values in the aggregate column up to a break in the specified footer. A cumulative aggregate calculates a running total for each value in the aggregate column up to the break containing the aggregate instruction. Simple and cumulative aggregates may be either non-unique or unique. Aggregate types are discussed in more detail later in this section.

The following aggregates are allowed:

sum	Sum value of a numeric column up to a break in the specified footer. If the specified footer is a date column, it must have time intervals as values to sum over it.
sumu	Sum unique or break values in a numeric column up to a break in the specified footer. You can specify **sumu** only for break columns. (See additional details in "Unique Aggregates" later in this chapter.)
count	Count the number of rows up to a break in the specified footer.
countu	Count the number of unique or break values up to a break in the specified footer. You can specify **countu** only for break columns. (See additional details in "Unique Aggregates" later in this chapter.)
min	Find the minimum value of a numeric or date column up to a break in the specified footer.
max	Find the maximum value of a numeric or date column up to a break in the specified footer.
avg	Find the average value of a numeric column up to a break in the specified footer.

avgu Find the average value of the unique or break values for a numeric column up to a break in the specified footer. You can specify **agvu** only for a break column. (See additional details in "Unique Aggregates" later in this chapter.)

Syntax of Aggregates

The syntax for an aggregate specification is:

[cum [(*breakname*)]] *aggname* (*columnname* [, *preset*])

breakname The name of a break in the report (either a sort column name, or **report** or **page**). It is optionally used as a parameter to the cumulative function to indicate when to reset the cumulative. The value of a cumulative then represents the aggregate since the last break in *breakname*. The default value for *breakname* is **report** (that is, it represents the cumulative value of an aggregate since the start of the report).

aggname The name of the aggregate to be executed. Valid *aggnames* and synonyms are **sum**, **minimum** (**min**), **maximum** (**max**), **average** (**avg**) and **count** (**cnt**).

columnname A column name in the data being reported. Values of this column are aggregated. Therefore, the column must be of the correct type (for example, numeric or date columns only, for all aggregates except **count**). Note that a *columnname* must be specified for the **count** aggregate, even though all columns result in the same value.

preset Either a constant value or the name of a column that is used for presetting the aggregate before calculations begins. This is used primarily with the cumulative function to set an aggregate to a non-zero value before starting.

For example, if you want to print an account balance next to each transaction in an account, you can use the cumulative **sum** aggregate with a *preset* to the starting balance of the account. See the ACCOUNT example report at the end of the guide for an example of this. If *preset* is a constant, the aggregate is set to that value. It may be a numeric or date constant. If *preset* is a valid numeric or date column name, the aggregate is set to the value in that column at the start of the break over which the aggregate is defined. Also, *preset* is not allowed with the **average** aggregate.

Simple Non-Unique Aggregates

The scope of a simple non-unique aggregate is determined by the context in which it is specified. For example, if "sum (salary)" is specified in the footer for the report, it refers to the sum of "salary" for all rows read in the report. If "sum(salary)" is specified in the page footer, it refers to the sum of "salary" for all rows that were processed during the printing of each page. If specified in the footer for a break in "department," "sum(salary)" refers to the sum of "salary" for all rows in each department.

Simple aggregates can only be specified in the **footer** action for breaks, because these calculations are intended to provide summary information.

Unique Aggregates

You specify a unique aggregate by following the aggregate name with the letter "u," as in **sumu**, **countu**, or **avgu**, respectively. The difference between a unique and a non-unique aggregate is that a unique aggregate performs an operation only when the value in the aggregate column changes, while a non-unique aggregate performs the operation for every value in the aggregate column. Therefore, a unique aggregate performs its calculation only on the break values in the specified column, up to the break containing the aggregate instruction.

For example, if the aggregate "count(region)" were specified in the report footer for the POPULATION example report in Appendix A, the result would be 51 (including the District of Columbia), because there are 51 rows in the report. However, if "countu(region)" were specified instead, the result would be 9, because nine breaks would occur on region.

The number of breaks is not necessarily the same as the actual unique values in the column. This result depends on the break in which the aggregate instruction is placed, and on whether the data in the aggregate column has been sorted or not. For instance, **countu** would produce a result of 3 on the following unsorted data in column 1, even though the data contains only two unique values, because three breaks would occur:

```
Column 1

AAA
BBB
AAA
```

Cumulative Aggregates

Preceding an aggregate name with the keyword **cumulative** or **cum** indicates that the cumulative value of an aggregate is calculated and printed. As such, cumulatives can be specified in any context (for instance, in detail sections) because they are used to provide running totals. A cumulative can be applied to any of the other aggregates. It is particularly useful for applications that need to use running totals, such as account balance applications.

If no *breakname* is specified after the **cumulative** keyword, or if a *breakname* of "report" is specified, the cumulative aggregate is assumed to refer to all data rows processed since the start of the report. If a *breakname* of "page" is specified, the cumulative aggregate refers to all data rows processed since the last page break. If a specified *breakname* is one of the break columns, the cumulative aggregate refers to all data rows processed since the last break in that column.

The *preset* parameter may be specified to set the cumulative function to a constant value or to the value of a column when it is initialized (that is, at the start of the break in *breakname*). For example, in the ACCOUNT example in Appendix A, the "cum(acctnum) sum(amt,balance)" aggregate in the **detail** block indicates a common use of the *preset* parameter. When a break occurs in "acctnum," the cumulative function is set to the value of "balance." As each new transaction is processed, the value of "amt" is added to the cumulative aggregate. Because "deposits" are positive and "withdrawals" are negative, the cumulative aggregate reflects the running balance.

Rounded Versus Actual Values

By default, aggregates utilize the rounded values for any floating-point column whose format has been specified in a **.format** or **.print** statement with a template or as numeric **F**. For additional information about these formats, see "Format Specifications" later in this chapter. That is, the value of the aggregate for such a column is derived from the rounded values for the individual column rows. To force the aggregate to use the actual, rather than the rounded, values, the **-t** flag must be specified on the **report** statement line, as described in Chapter 12.

Examples of Aggregates

Here are some examples of aggregates, with explanations:

```
min(salary)
```

Specified in the footer for "dept," this element gives the minimum value of salary for all data rows in a "dept."

```
average(age)
```

Specified in the footer for "class," this element gives the average age for all data rows in a "class."

```
count(name,200)
```

Specified in the footer for the report, this element gives the count of the number of data rows in the report + 200.

```
sum(transact,oldbal)
```

Specified in the footer for "acct," this element gives the sum of "transact," initialized by the value of "oldbal" at the start of each "acct."

```
cumulative avg(height)
```

Specified in the **detail** text, this element gives the cumulative average of height since the start of the report.

```
cum(acctnum) sum(amt,balance)
```

Specified in the **detail** text, this element gives the cumulative sum of "amt" since the last change in "acctnum" and initialized by the value of "balance" at the last change of value in "acctnum."

Operations

Expressions can include arithmetic, comparison, and logical operators, Boolean and built-in functions, as well as pattern matching with wild cards.The following operators may be used in expressions. These are described in the following sections.

Arithmetic Operators

Numeric expressions may be combined arithmetically to produce other (compound) expressions. The following arithmetic operators are supported (in descending order of precedence):

+, -	plus, minus (unary)
**	exponentiation
*, /	multiplication, division
+, -	addition, subtraction (binary)

Unary operators group from right to left, while binary operators group from left to right. You may force the order of precedence of operations using parentheses. This, for example, is an expression with no ambiguity as to precedence of operations:

```
(salary + 1000) * 12
```

Arithmetic Operations on Dates

The following arithmetic operations are available for date expressions:

Addition:

interval + interval → interval
interval + absolute → absolute

Subtraction:

interval - interval → interval
absolute - absolute → interval
absolute - interval → absolute

An example of the correct use of arithmetic operators in date expressions is:

```
current_date + date("1 days")
```

Another example is:

```
current_date - birthdate
```

The first example returns tomorrow's date. The second example gives a person's age.

Comparison Operators

A comparison operator has two expressions as operands, and returns the result of true or false. Both expressions must have the same type: numeric, string, or date. The following operators are recognized:

=	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

All comparisons have equal precedence. When character strings are compared, blanks are ignored.

Conditional Clauses

A conditional clause has the form:

expr comp_op expr

The *expr* is an expression, and *comp_op* is a comparison operator.

A clause may be enclosed in parentheses without affecting its interpretation, as in the following examples:

```
(age < 50)
((salary * 12) >= 20000)
```

A clause returns the value true or false. It may contain comparison operators and partial match specification characters.

Pattern Matching with Wild Cards

You may indicate partial matches of character strings in a conditional clause, in an **.if** statement, and in the **where** clause of a query by using wild card characters with the comparison operators. The character string must have single or double quotes as delimiters.

Wild Cards in an .if Clause

When used in a string within an **.if** condition, wild card characters can be disabled by preceding them with a backslash (\) character. In this case, the character is interpreted literally. Thus, "*" refers to the asterisk character. When used outside of an **.if** condition, wild card characters have no special meaning and are always interpreted literally.

The following wild card characters may be used in an **.if** statement for comparing character strings:

*	matches any string of zero or more characters
?	matches any single character
[..]	matches any of the characters in the brackets

Examples of the use of wild card characters are:

ename = "*"	matches any value in "ename"
ename = "E*"	matches any value beginning with "E"
ename = "*ein"	matches any value ending with "ein"
ename = "*[aeiou]*"	matches any value with at least one vowel
ename = "Br"	matches any five-character value beginning with "Br"
ename = "[A-J]*"	matches any value beginning with A, B, "C, ..., J."
ename = "[N-Z]???"	"matches any four-character value beginning with N, O, P, ..., Z.

Blanks must be eliminated in bracketed expressions such as "[A-J]*" or "[N-Z]???."

Wild Cards in Queries

When a string appears in the **where** clause of a **.query** statement, the wild card conventions follow those of the SQL database language used to retrieve the data.

Logical Operators

The following logical operators are recognized:

(Boolean operator)

not	(logical not - negation)
and	(logical and - conjunction)
or	(logical or - disjunction)
is null	(test to see if value is null)
is not null	(test to see if value is null)

These operators evaluate clauses or Boolean functions as operands, and return the value of true or false. The **Not** operator has the highest precedence. The **and** and **or** operators have equal precedence. You may use parentheses for arbitrary grouping. Logical operators group from left to right.

Built-in Functions

Functions are denoted by a function name, followed by one or two operands in parentheses. When expressions are substituted for the operands, the function is evaluated and the result is a number, a string, or a date. Functions can be nested to any level.

All of the ODT-DATA conversion, numeric, string, and date functions such as **char**(*expression*), **log**(*n*), **concat**(*string1*,*string2*), and **date_trunc** are available to the Report-Writer user. For descriptions of the functions, see the *ODT-DATA SQL Reference Manual* or the *ODT-DATA Open SQL Reference Manual.*

Boolean Functions

A Boolean function returns a value of *true* or *false*. The result of a Boolean function cannot be printed; it can only be used as a condition. A Boolean function is composed of a function name followed by an operand in parentheses.

The break function is the only Boolean function found in the Report-Writer. (See "The Concept of Breaks" in Chapter 1.) The syntax is:

break (*columnname*)

where *columnname* must either be a break column (that is, in the sort list) or the value **report**.

When a break column is specified, the value true is returned if the current value for that column changes from the previous value or if the current value in any column of higher precedence than column changed. If neither the current value for column nor the current value of any column of higher precedence in the sort list changed, the value of false is returned. When **report** is specified, true is returned if the end of the report is reached; otherwise, false is returned.

Example of Boolean functions include:

```
.sort dept, empno
     /*  Other Report-Writer statements  */
.footer empno
     .if not break(dept) .then
          .newpage
     .endif
```

This generates a new page when the employee number breaks, but only if the department has not changed.

Format Specifications

Expressions in the report may be given special format specifications in the **.print** statement or in a **.format** or **.tformat** statement. The format determines whether the data is printed as a character string, decimal value, date, and so on. Be sure to use the right type of format, depending on the type of expression. As discussed in Chapter 3, if no format is specified, Report-Writer determines a default format from an analysis of your other statements. The following formats are allowed:

- **C format** specifies character strings.
- **T format** specifies character strings like the C format, except that it displays certain unprintable characters in a visible format.

- **F format** specifies numeric expressions. In the **F** format, you can control the placement of the decimal point or suppress it entirely.

- **E format** specifies numeric expressions printed in scientific notation.

- **G format** specifies numeric expressions. This format chooses either **F** or **E** format, depending on what fits in the field width. This format also guarantees that decimal points align, whether printed in **F** or **E** format.

- **N format** specifies numeric expressions like **G** format, but decimal points do not necessarily align.

- **B format** specifies that the value be blanked out. It is a special format used for blanking out a field, for use with temporary formats in conjunction with the **.tformat** statement.

- **Numeric templates** are complex formats for numeric data that allow you to control placement of dollar signs, commas, or other punctuation within the number.

- **Date templates** are formats that allow you very detailed control over the appearance of dates and times in your reports.

A sign character can precede the format specifications to indicate that the print value is right justified, left justified, or centered. The following are valid sign characters:

-	A minus sign indicates tht the data is to be left justified in the specified field width.
*	An asterisk indicates that the data is to be centered in the specified field width.
+	A plus sign indicates that the data is to be right justified in the specified field width.

Examples of each sign character can be seen in the discussions below for each format type. If no sign is given, justification defaults to left for character fields and right for numeric fields.

Character String Format C

Use the C format to print string expressions.

The syntax of is:

[-|*|+] c[f|j]*n*[.*w*]

C Format Parameters

Parameter	Description
f	If specified, this tells the Report-Writer to break the text at words when the text spans lines.
j	If specified, this tells the Report-Writer to right justify text.
n	Specifies the maximum number of characters to print. If there are more than *n* characters in the string, the string is truncated. If there are fewer, it pads with blanks until *n* characters have been printed. Use 0 for *n* to print the entire string, regardless of its length.
w	Specifies the number of characters to print on each line. If *n* is greater than *w*, then more than one line is written in a newspaper column format. By default, *w* is set to *n*.

When specifying these options, you may use upper- or lower-case letters. The field width *n* is optional and may be used to specify an exact width. If *n* is specified and the string is fewer than *n* characters long, blanks are added to make up the *n* characters. If the string is longer than *n* characters, only the leftmost *n* characters are printed.

If you specify a value for *w* and *n*, you can print text in newspaper column format. The **f** and **j** modifiers cause breaks at words for wrapping and right justification of text. If neither is specified, simple wraparound of text occurs, with breaks occurring regardless of words.

To print a visual representation of unprintable characters, use the **T** format statement, discussed below. Tab characters and carriage returns cause tabs and carriage return actions if you are using the **cf** or **cj** format.

Example 1

The following six **.print** statements are for a report that contains a character column called "name" that you want to print, and a value for name is "Jones, J.":

```
.print "First :", name (c15),":First" .nl
.print "Second:", name (c4), ":Second" .nl
.print "Third :", name (c0), ":Third" .nl
.print "Fourth:", name (-c15), ":Fourth" .nl
.print "Fifth :", name (+c15), ":Fifth" .nl
.print "Sixth :", name (*c15), ":Sixth" .nl
```

It produces, respectively, the following six lines of output:

```
First :Jones, J.       :First
Second:Jone:Second
Third :Jones, J.:Third
Fourth:Jones, J.       :Fourth
Fifth :       Jones, J.:Fifth
Sixth :   Jones, J.    :Sixth
```

Example 2

If your data includes the character string "Now is the time for all good people to come to the aid of their country," the following shows the effect of three different format specifications:

```
c0.15                  cf100.15                cj0.15


Now is the time for    Now is the time for     Now is the time for
all good people to     all good people to      all good people to
come to the aid of     come to the aid of      come to the aid of
their country.         their country.          their country.
```

Because the second format specification, "cf100.15," specifies an actual number of character positions to print, Report-Writer prints out two blank lines after the text, to pad to the full 100-character column width.

After a string prints in column format, the current position moves to the top line of the column, at the end of the line. In the example, this is to the right of "time".

Character String Format T

The **T** format is similar to the **C** format. The **T** format translates characters outside the normal character set into visible representations.

This format is useful when you want the output to look like that of the ODT-DATA Terminal Monitor, which expands unprintable characters into visible representations.

The syntax of a **T** format specification is:

[-|*|+] t[f|j]*n*[.*w*]

The **f** and **j** options work the same as they do for the **C** format.

Note that *n* is the width of the field that the expanded output occupies on the page. It does not refer to the number of characters of data that are translated.

Examples

For the character string "John?Smith,\Esq.", where the "?" character stands for a non-printing *formfeed* character:

```
.print "Output:",:Output" .nl
```

This print statement produces:

```
Output:John\fSmith,\\Esq.:Output
```

The following lists the character representation of the **T** format:

- Newline becomes **\n**.
- Horizontal Tab becomes **\t**.
- Backspace becomes **\b**.
- Carriage Return becomes **\r**.
- Form Feed becomes **\f**.

- Backslash becomes \\.
- Null becomes \0.
- Any other unprintable character becomes the character string "\nnn", where *nnn* is the three-digit octal equivalent for character.

Numeric Format F

The **F** format prints numeric expressions in standard decimal notation, with or without a decimal point. Numbers are right justified in the field, unless preceded by a **.left** statement or by the "-" sign in the format designation.

The syntax is:

[-|*|+] **f***w*[.*d*]

F Format Parameters

Parameter	Description
w	The maximum field width.
d	The precision, or the number of digits to print after the decimal point.

The "+" and "-" prefixes specify how the text should appear in the field; as either right or left-justified. They do not have any bearing on the sign of the data.

You may specify this format with upper- or lower-case letters. The field width *w* must be specified, and refers to the maximum number of printing positions in the field. If the value can be printed in fewer than *w* spaces, it is right justified in the field. If the value cannot be printed in *w* or more spaces, the field is filled with asterisks (*).

If *d* is specified, a decimal point is printed, with *d* digits to the right of the decimal point. The number of digits to the left of the decimal point cannot exceed *w* - (*d* + 1), because you must account for the fractional part in the field width specification.

If *d* is not specified, or if the value "0" is specified for *d*, for example "F20.0", then no decimal place is printed.

The following table illustrates the **F** format specification:

```
Format          Value          Output

f10.2           22.3           "      22.30"
F10.2           -.123          "      -0.12"
f10             123.789        "        124"
f4.2            22.34          "****"
+f10.2          22.6           "      22.60"
-f10.2          22.6           "22.60      "
```

Numeric Format E

The **E** format prints numeric expressions in scientific notation. Numbers print with the form [-]"m.*nnnnnn*E[+|-]*pp*". An example is "10.456e+03", which means 10.456 times 10 raised to the 3rd power. Numbers output in **E** format are right justified in the field (unless preceded by a **.left** statement or the "- " sign on the format designation).

The syntax is:

[-|*|+] e*w*[.*d*]

E Format Parameters

Parameter	Description
w	The maximum field width.
d	The precision, or the number of digits to print after the decimal point.

The "e" may be upper- or lower-case. The case specifies the case of the "e" in the output. The field width *w* must be specified, and refers to the maximum number of spaces in the field. Be sure to include four extra spaces for the exponent part of the printout. If the value can fit in fewer than *w* spaces, it is right justified. If the field width is too small for the value, the field is filled with asterisks (*).

If *d* is specified, a decimal point is printed, and *d* digits are printed to the right of the decimal point. If *d* is not specified, or if a value of "0" is specified for *d*, such as "E20.0", then no decimal decimal place prints, although the exponential part prints.

The following shows examples of the **E** format:

```
Format          Value          Output

e10.3           22.3           "22.300e+00"
E10.2           -.123          "-12.30E-02"
e10             123.789        "  1238e-01"
E4.2            22.34          "****"
+E10.2          22.34          " 22.34E+00"
-e10.2          22.34          "22.34e+00 "
```

Numeric Format G

The **G** format uses an **F** format specification if there is enough room in the field, or **E** format if there is not enough room.

The syntax is:

[-|*|+] g*w*[.*d*]

G Format Parameters

Parameter	Description
w	The maximum field width.
d	The precision or number of digits to print after the decimal point.

An upper- or lower-case "g" may be specified. The case determines the case of the "e" if the value is printed in scientific notation. See the **F** and **E** formats for the use of *w* and *d*.

Numbers are right justified if the decimal points are aligned. Numbers that are too long for an **F** format print with **E** format and are right justified. To align **F** format numbers with **E** format numbers, Report-Writer right justifies **F** format numbers four spaces in from the right edge of the field to match up with the exponential designator, ("E+|-pp"). If you specify the optional justification sign ("+" or "-"), the values are forced right or left, according to the sign.

The following are examples of the **G** format:

Format	Value	Output
g10.2	123.456	"123.46 "
G10.2	123456	" 12.35E+04"
g8.2	-134.65	"-.13e+03"
g8	-123	"-123 "
+g10.2	123.45	" 123.45"
-g10.2	123.45	"123.45 "

Numeric Format N

The **N** format is similar to the **G** format specification except that the field is right justified, whether printed with **E** or **F** format. If you specify the optional "-" sign, the value is left justified.

The syntax is:

[-|*|+] **n***w*[.*d*]

N Format Parameters

Parameter	Description
w	The maximum field width.
d	The precision, or number of spaces to print after the decimal.

The "n" may be upper- or lower-case. The case determines the case of the "e" for scientific notation. See the **F** and **E** formats for the use of *w* and *d*.

Numbers printed with **N** format are right justified in the output field. Unlike **G** format, the decimal points are not always aligned.

The following are examples of the **N** format:

Format	Value	Output
n10.2	123.456	" 123.46"
N10.2	123456	" 12.35E+04"
n8.2	-134.65	"-.13e+03"
n8	-123	" -123"
+n10.2	123.79	" 123.79"
-n10.2	123.79	"123.79 "

Blanking Format B

The **B** format, which may be used with any type of data, functions in a special way when used in conjunction with the **.tformat** statement, which temporarily changes a column format. The value of a variable printed with **B** format is not printed but is replaced with blanks.

The syntax of the **B** format is:

b*w*

B Format Parameters

Parameter	Description
w	The desired field width.

The "b" may be upper- or lowercase. This format ignores the value of an expression and inserts *w* spaces in the output.

Numeric Templates

If you need complex numeric formats, you can use a template. A template specifies what the output should look like. You use template characters to indicate what should be printed in the template. For instance, a "**Z**" prints a digit of a number. A comma (,) in the template prints a comma in the specified position. The template "**Z,ZZZ**" prints the value "1000" as "1,000". In addition to the template characters listed below, you may include any other character directly in the numeric template by preceding it with a backslash.

The syntax of for a numeric template is:

[-|*|+] "{*c*}"

Numeric Template Parameters

Parameter	Description
c	One of several special characters that may be repeated any number of times

The numeric template is right justified by default. By specifying the optional "-" sign, you can left justify the template. You must surroundit with double quotes; single quotes are not allowed.

The following special characters are defined:

n or **N**	Prints a digit if unprinted digits remain in the number. If none remain, prints a zero.
z or **Z**	Prints a digit if unprinted digits remain in the number. If none remain, prints a space. This is used for standard blank-padded numeric fields.

$	(Dollar sign) Prints a digit if unprinted digits remain in the number. If none remain, prints a floating dollar sign (at its right-most position only, within a repeated sequence of dollar signs). If a dollar sign has already been printed, prints a space. This can be used to print a dollar sign directly to the left of the number, or to place a dollar sign in a fixed position in the field.
-	(Minus sign–Preceding or Trailing) For preceding: Prints a digit if unprinted digits remain in the number. If none remains and if the number is negative, prints a floating minus sign (at its rightmost position only, within a repeated sequence of minus signs). If a minus sign has already been printed, or if the number is positive, prints a space. For trailing: Prints a minus sign in the position if the number is negative; or if the number is positive, prints a space.
+	(Plus sign–Preceding or Trailing) For preceding: Prints a digit if a digit remains in the number. If none remains, prints a floating sign (+ or -). If one has already been printed, prints a space. For trailing: Prints a plus sign in the position if the number is positive, or a minus sign if the number is negative.
,	(Comma) If followed by a digit, prints a comma in this position. If no digits remain, prints a space.
.	(Decimal point) Prints the decimal point in this position. The template may contain only one decimal point.
*	Prints a digit if unprinted digits remain in the number. If no digits remain, prints an asterisk. This is useful to fill a number on the left with asterisks (such as, for checks).
space	Prints a blank space in this position.

\	(Backslash) Indicates that the character immediately following the backslash is to be printed in that position. This allows dashes, slashes, or other characters to be inserted into the number at runtime. (The backslash itself is not printed.)
CR	(Two characters) Inserts the characters "CR" (for credit) if the number is negative, or two blanks if positive. The letters "CR" appear exactly as specified, in upper- and/or lowercase letters.
DB	(Two characters) Inserts the characters "DB" (for debit) if the number is negative, or two blanks if positive. The letters "DB" will appear exactly as specified, in upper- and/or lowercase letters.
( ) or [ ] or < >	(Parentheses, square brackets or angle brackets) If the number is negative, prints it within the specified symbols.

If a field is specified without any "n"s in the numeric positions, and a value of zero is encountered, nothing is printed in the output field. Also note that where the floating symbols ("$", "-" or "+") are repeated more than once in a template to specify a floating character, the floating character is printed only once in the output field, in its right-most position within the sequence.

The following examples illustrate numeric templates:

Format	Value	Output
"zzzzz"	123	" 123"
"zZzZz.Zz"	0	" "
"zzzzz.nn"	0	" .00"
"+++,+++,+++"	23456	" +23,456"
"---,---,---.NN	23456.789	" 23,456.79"
"---,---,---.zz "	-3142.666	" -3,142.67 "
"zzz,zzz,zzz.zz-"	-3142.666	" 3,142.67-"
"$$$,$$$,$$$.nncr"	235122.21	" $235,122.21 "
"$$$,$$$,$$$.nnDb"	-235122.21	" $235,122.21Db"
"$zz,zzz,zzn.nn"	1234.56	"$ 1,234.56"
"$**,***,***.nn"	12345	"$****12,345.00"
"+$$,$$$,$$$. "	54321	" +$54,321.00"
" nnn\-nn\-nnnn "	023243567	" 023-24-3567 "
-"zzzzz"	123	"123 "
"(zzzzz)"	-123	"(123)"
"[[[[[z]"	-123	" [123]"

Date Format D

The date format specification is a **D**, followed by a double quoted string *template* indicating how to print a date. The date is left justified by default. By specifying the optional "+" sign, you can right justify the date. You must surround the template with double quotes; single quotes are not allowed.

The syntax for a date template is:

[-|*|+] **d** "*template*"

Date Format Parameters

Parameter	Description
template	A string of characters representing a sample absolute date and time.

Specifying Absolute Date and Time Templates

The absolute date and time format is specified by a string containing one of many possible representations of a sample date and time, such as "SUN Feb 3 04:05:06 p.m." or "FEB 03 16:05". The selection and arrangement of the sample date and time elements within the template indicate the way you want all dates and times to be displayed or printed. You must use the following date and time as the basis for your template:

```
Sunday, 1901 February 3 at 4:05:06 p.m.
```

Note: This specific date and time was chosen as the sample for the template because Sunday is the first day of the week, and arguments 1, 2, 3, 4, 5, and 6 are the year, month, day, hour, minute, and second, respectively. This makes it easy to interpret the elements of the template correctly. For instance, in the template **d**"2/3/01" the "2" indicates the month (February), the "3" indicates the day (3), and "01" indicates the year (1901).

You may use all or only some of the arguments in your template, and you may arrange the arguments in any order. The following examples demonstrate the use of absolute date and time templates:

Format	Value	Output
d" 2/ 3/01"	25-oct-1982	"10/25/82"
d" 2/ 3/01"	5-jun-1909	"6/ 5/09"
d"03-02-01"	5-oct-1982 07:24:12	"05-10-82"
d"2/3/1"	25-oct-1982	"10/25/82"
D"2/3/1"	5-jun-1909	"6/5/9"
d"010203"	5-oct-1982	"821005"
d"1|2|3"	5-oct-1982	"82|10|5"
d"FEBRUARY, 1901"	1-sep-2134 09:13:02	"SEPTEMBER, 2134"
d"FEBRUARY, 1901"	7-may-1962 13:08:42	"MAY, 1962"
d"February,,| 1901"	12-sep-1982	"September, 1982"
d"February,,| 1901"	3-may-1982	"May, 1982"
d"Sunday"	5-oct-1983	"Wednesday"
d"SUN Feb 3 16:05 1901"	13-oct-1983 07:24:03	"THU Oct 13 07:24 1983"
d"FEB 03 4:05:06 p.m."	12-dec-[1983 22:13:03	"DEC 12 10:13:03 p.m."
d"04:05:06 PM"	5-oct-1983 14:08:45	"02:08:45 PM"
d"04:05:06 PM"	5-oct-1983 07:29:12	"07:29:12 AM"
d"16:05 pst"	5-oct-1983 14:08:45	"14:08 pst"
d"Sunday, February|"	27-jun-1983	"Monday, June "
+d"Sunday, February|"	27-jun-1983	" Monday, June"
+d"Sunday, February|"	5-oct- 1983	" Wednesday, October"
d"3/01"	5-oct-1983	"278/83"
d"February 3rd"	29-jul-1954	"July 29th"
d"3rd day of 1901",	11-may-1999	"131st day of 1999"

You cannot use month names other than February, days other than Sunday, or the time designations "a" or "am" in the date template. Any other word is printed as it appears.

You can specify 24-hour "military" time by using "16" instead of "4." You cannot use "p" or "pm" with 24-hour time.

The day of the year is printed by specifying the day and year, but leaving out the month (such as, "3/1901").

You can create ordinal numbers from numbers by suffixing them with the appropriate "st", "nd", "rd"or "th" (such as, "3rd day of the 2nd month of 1901").

Numbers requiring more than one digit replace preceding blanks or zeroes in the template. If there are no preceding blanks or zeros left, the number expands to the right. A blank that follows a letter, word, or number in the template is retained in the output; it is not replaced by a succeeding number. Columns of numbers may be aligned by preceding them with an appropriate number of blanks or zeroes (note the first three examples above).

Since full month and weekday names (as well as numbers without preceding blanks or zeros) are of differing lengths, date columns using either of these components in the format will rarely line up. Following "February" or "Sunday" with a vertical bar (|) specifies that for shorter month names or weekdays, an appropriate number of blanks are substituted for the vertical bar to line up the components. Similarly, if you place a vertical bar after a single digit number in your template, Report-Writer prints a blank before each single-digit number it encounters (unless the digit is already preceded by a blank or zero).

Any character preceded by a backslash is printed as it appears.

Specifying Time Interval Templates

The time interval is specified by a string containing one of many possible representations of a sample time interval such as "1 year" or "1 yr 3 day", and so on. The selection and arrangement of the time interval elements within the template indicate the way you want time intervals to be displayed or printed. You must use the following time interval as the basis for your template:

```
1 year 2 months 3 days 4 hours 5 minutes 6 seconds
```

You may use one or more of these units in your template and you may arrange the units in any order. The following examples demonstrate the use of the time interval templates:

Format	Value	Output
D"1 year"	3 yrs 5 mos 16 days	"3 years"
d"2 MONTHS, 3 DAYS"	3 yrs 5 mos 1 days	"41 MONTHS, 1 DAY"
d"3"	3 yrs 5 mos 16 days	"1264"
d"1 yr 3 day"	1 yrs 5 mos 16 days	"1 yr 168 days"
D"4 hours 6 seconds"	23 hrs 8 mins 53 secs	"23 hours 533 seconds"
d"04:05 \hours"	23 hrs 0 mins 53 secs	"23:01 hours"
d"3 days 4 hours"	23 hrs 8 mins 53 secs	"0 days 23 hours"
d" 1 yr 2 mos 3 days"	200 yrs 11 mos 28 days	"200 yr 11 mos 28 days"
d" 1 yr 2 mos 3 days"	5 yrs 1 mos 3 days	" 5 yr 1 mos 3 days"

There are 30.4375 days in a month and 365.25 days in a year. The smallest unit specified is rounded up.

Numbers requiring more than one digit replace preceding blanks or zeroes in the template. If there are no preceding blanks or zeroes left, the number expands to the right. A blank following a letter, word, or number in the template is retained in the output. Columns of numbers may be aligned by preceding them with an appropriate number of blanks or zeros (note the last two examples above).

The word following a number is singular if the number is one; it is made plural if not equal to one. ODT-DATA makes this change only for English-language versions, and only when the on-line word is spelled out. For example, "5 month" would become plural, while "5 mo" would not.

Any character preceded by a backslash is printed as it appears.

Default Formats

If there is no format specified after an expression, the Report-Writer uses a default format.

Default Format for Strings

Any string expression without a specified format is printed in its entirety. That is, the default format for strings is "c0."

Default Format for Columns

If you do not specify a column format with the **.format statement**, the Report-Writer uses the default format for the column. The default format is based on the data type of the column. See "Determining Default Column Formats" in Chapter 3.

The following lists default formats for SQL data types.

Default Column Formats

SQL Data Type	Default Column Format
c1 - c35	c1 - c35
c36 - c2000	cj0.35

SQL Data Type	Default Column Format
char(1) - char(35)	c1 - c35
char(36) - char(2000)*	cj0.35
text(1) - text(35)	c1 - c35
text(36) - text(2000)*	cj0.35
varchar(1) - varchar(35)	c1 - c35
varchar(36) - varchar(2000)*	cj0.35
integer1	f6
smallint (integer2)	f6
integer (integer4)	f13
float4	n10.3
float (float8)	n10.3
date	c25
money	"$----------.nn"

* All character data types are fully supported in non-ODT-DATA databases accessed via gateways, in which case the column size limit may be greater than 2000 bytes.

Default Format for Special Report Variables

The following non-string report variables have the corresponding default formats:

Default Formats for Report Variables

Report Variable	Default Format
page_number	f6
line_number	f6
position_number	f6
left_margin	f6
right_margin	f6
page_length	f6
current_date	d"3-feb-1901"
current_time	d"16:05:06"
w_column	The default format for the column currently being used in a **within** block (see the default column formats in the previous table).

Default Format for Aggregates

The default format for all the aggregates except **count(u)** is the format of the column being aggregated. For **count(u)**, the default format N*w* is used, where *w* is the width of the column being counted.

Default Format for Numbers

Any other numeric expressions such as numeric constants, numeric functions, numeric parameters, and arithmetic operations have a default format of n12.2.

Default Format for Dates

Any other date expressions such as the date function, date parameters and date arithmetic operations have a default format of c0, which appears in the report as d" 3-feb-1901" for an absolute date, d"3-feb-1901 16:05:06" for an absolute date and time, or the portion needed of the template d"1 yrs 2 mos 3 days 4 hrs 5 mins 6 secs" for a time interval.

Chapter 5
Report Setup Statements

This chapter documents the report setup commands. These commands are used at the beginning of a report specifications file to identify report parameters.

.name

The **.name**statement names a report.

Syntax

.name |**.nam** *reportname*

Description

The **.name** statement is required and must be the first statement specified for a report. The report specification program **sreport** stores the report in the database under the report name.

You may store specifications for several reports in one text file by using several **.name** statements. Each occurrence of a **.name** statement signals the end of the previous report's specification statements and the beginning of a new report.

Parameters for .name

Parameter	Description
reportname	The name of a report to which the next set of formatting statements apply. The *reportname* is a standard ODT-DATA object.

Examples

The following denotes the start of report "abc":

```
.name abc
```

The following denotes the start of report "my_rep":

```
.name my_rep
```

comments

The comment delimiters include documentation in the report specification file.

Syntax

/* {*any_text*} */

Description

You can include comments in the report specification file by bracketing between the "/*" and "*/" characters. All text between these characters is ignored in report processing and can be used as your own documentation.

Comments may be nested (that is, you can have a set of comments within another set of comments). Comments may be placed anywhere within your file.

Parameters for comments

Parameter	Description
any_text	Any text, except the characters "*/", which close the comment.

Example

```
/*  this is an example
                of a comment...
                                                    */
```

.shortremark

The **.shortremark** identifies a one-line remark describing the report.

Syntax

.shortremark|.srem *remark-text*

Description

The **.shortremark** statement is an optional statement that specifies a one-line description of the report. You can use this short description to help document your report specifications if you wish, but its primary purpose is to provide information that appears on the Catalog and Save frames of the RBF application.

Use only one **.sremark** statement in a program. A second **.sremark** statement is flagged as a syntax error. Only the first 60 characters of the descriptive text are stored in the database.

Parameters for .shortremark

Parameter	Description
remark-text	A string of characters on the same line as the statement keyword.

Examples

```
.shortremark Monthly Accounts Receivables

.srem customized emp & dept report tables
```

.longremark and .endremark

The **.longremark** statements mark the begining and the end of a block of text that describes the report.

Syntax

.longremark|.lrem
remark_text
.endremark|.endrem

Description

The **.longremark** and **.endremark** statements are an optional pair that specify a lengthy description of the report. The start of the block of descriptive text is denoted with the **.longremark** statement, and the end is denoted by the **.endremark** statement. This long description appears in the Catalog and Save frames of the Report-by-Forms utility.

The descriptive text is stored in the database and is available to other ODT-DATA application development tools.

Use only one **.longremark** statement in a report specification. A second statement produces a syntax error.

Parameters for .longremark

Parameter	Description
remark-text	Any number of characters or lines of text.

You may enter as much remark text as you like. Only the first 600 characters are saved in the database. Leading spaces that separate the **.longremark** statement from the first character of text are ignored. Tab characters are converted to blank characters.

Examples

```
.longremark
This report correlates information from the sales
order header, the sales order detail, and the
inventory files, to produce the customer backlog by
part number report.
.endremark

.lrem
Stock Analysis Report
8 1/2" x 11" output
10 minutes runtime
Input:  Begin/End date
.endrem
```

.data

The **.data** statement specifies the table or view in the database that is the source of the data for the report.

Syntax

.data| .dat | .table| .view *tablename*

Description

The **.data** statement identifies a table in the database that is used in its entirety in the report. The four synonyms above can be used interchangeably. All of the data in the table are available for use in the report specification.

Either the **.data** or the **.query** statement is required. The **.data** and **.query** statements are mutually exclusive. Only one may appear in a report specification.

Parameters for .data

Parameter	Description
tablename	The name of a table or view in the database. All rows and columns in the table are read each time the report is run. *Tablename*, because it is the name of a table in your database, follows the same rules for table names as the rest of ODT-DATA.

Examples

Use table "repdat" for the report.

```
.data repdat
```

Use view "myview" for the report.

```
.table myview
```

.declare

The **.declare** statement declares variables that can be assigned values and used in expressions.

Syntax

.declare *variablename = datatype*

[with null|not null]
[with prompt "]
{, *variablename = datatype...*}

Description

The **.declare** statement declares variables that may be assigned runtime values on the command line or through a prompt, or that may be assigned values in **.let** assignment statements, for later use in expressions. Only one **.declare** statement may be specified. Declared variables may also be used in a query block to specify runtime substitution of text in the query.

Parameters for .declare

Parameter	Description
variablename	A valid name up to 32 characters long. It must begin with an alphabetic or underscore (_) character. Following characters must be alphanumeric or underscore.
datatype	A legal ODT-DATA data type.
promptstring	A string constant up to 100 characters in length. See String Constants in Chapter 4, "Expressions and Formats."

Note: When declared variables are referenced within a query block, they must be preceded by a dollar sign ($). The dollar sign ($) is not be used in the **.declare** statement, it is used in a query.

The **.declare** statement declares each variable to be the given data type. You may include the **with null** or **not null** option.

- If the variable is declared with the **with null** option, it is initialized to the null value.
- If the variable is declared with the **not null** option, it is initialized to the default value for the data type.

If neither option is specified, the variable data type defaults to null or not null, depending on the query language (SQL or QUEL) used in the **.query** statement. If a **.data** statement is specified instead of the **.query** statement, the installation default language determines default nullability.

A declared variable may be assigned a value in any of these ways:

- With the **.let** statement, placed anywhere in the Report-Writer code .
- Alternatively, the initial value of any declared variable may be specified on the command line with the runtime parameters.
- You may use the **with prompt** option to instruct Report-Writer to prompt for the initial value of the variable, using the specified prompt string.

If no initial value or prompt is specified and the variable is referenced outside of a query block, the initial value is **null** (or the default value for that data type if **not null** was used). When a declared variable is referenced within a query block, its initial value must be entered either on the command line or in response to a prompt string, which was specified in the **.declare** statement.

Example

```
.declare
      counter = integer,
      salary = money with prompt
            "Please enter the salary:",
      spouse = c30 with null,
      dept = i4 not null with prompt
            "What department?"
```

.output

The **.output** statement specifies the filename to which the report is written.

Syntax

.output/ .out *filename*

Description

The **.output** statement is an optional statement that specifies the name of a file where the report will be written. If you do not use the **.output** statement in your report specification, the Report-Writer either directs the output to the terminal or to a filename specified on the command line for the **report** command with the **-f** flag. If the **.output** statement is not specified, and no file is specified with the **-f** flag, the report is written to your terminal.

Parameters for .output

Parameter	Description
filename	A file to which the formatted report is written each time the report is run. *Filename* must follow all conventions for valid filenames in the operating system.

Write to file in current directory.

```
.output myreport.lis
```

Write to file with full pathname.

```
.out /direct/subdirect/otherrep.out
```

.query

The **.query** statement specifies an SQL query used to generate data for a report.

Syntax

.query

select [**all|distinct**] *column_list*
from *table* [*corr_name*] {, *table* [*corr_name*]}
[**where** *search_condition*]
[**group by** *column* {, *column*}]
[**having** *search_condition*]
{**union select** ...}
[**order by** ...]

See the *ODT-DATA SQL Reference Manual* for a complete explanation of the syntax of the **select** statement.

Description

The **.query** statement indicates the start of a valid SQL query that creates the data to be reported. This query follows the same rules as any other SQL **select** statement, although it may also contain parameters. You may use as many lines as you need to specify the query. The end of the query is detected by the start of a new report formatter statement.

Either the **.query** or the **.data** statement (but not both) must be specified for every report. Only one **.query** statement is permitted for a report, and only one data retrieval statement is permitted within the **.query** statement. There may not be both a **.query** with an **order by** clause and a **.sort** statement in the same report specification, because their functions are mutually exclusive.

Because the **.query** statement generates a standard ODT-DATA query, the standard limits apply to any report's query. For ODT-DATA databases, these limits are 127 columns and 2008 bytes per row. These limits are extended on some gateways--please refer to your ODT-DATA/Gateway manual if accessing your databases through a gateway.

String constants must be enclosed by the standard SQL string delimiter, the single quote. Note that the single quote string delimiter is required only within the **.query** statement; within other Report-Writer statements, either the single or double quote may be used as the string delimiter.

Parameters and declared variable names:

- Can be up to 32 characters long. Valid characters are letters, digits, and underscore (_).
- Must begin with a letter.
- Cannot match any of the reserved words listed under "Reserved Words" in Chapter 4.

Parameters and Declared Variables in Queries

Parameters and declared variables may be specified for runtime substitution of text into the query. You indicate parameters and declared variables in a query by preceding the name with a dollar sign ($). For example, you can specify a query as follows:

```
.query
     select empname, salary, manager
          from emp
          where salary > $minsal
```

Subsequently, you can invoke the report with a statement like the following:

```
report mydb myrep (minsal = 20000)
```

in which case the query is converted to:

```
... where salary > 20000
```

If the value of a parameter is not specified on the command line, the Report-Writer prompts you for the value, using a default prompt. If you wish to use a different prompt, you may use a declared variable in the query and specify a prompt string using the **with prompt** option in the variable declaration.

You can specify as many parameters or declared variables as you wish in a query, differentiated by name. If the same parameter or declared variable is to be substituted more than once within the query, simply specify the name, prefixed by a dollar sign, at each place where substitution is to be done.

Parameters and declared variables may be specified anywhere in the query. They may even be specified within quoted strings, or within the column list for a **select** statement. For example, the following query phrases are legal:

```
... where name = '$Employee_name' ...
... select $var, ... from emp ...
```

If you actually want to include the dollar sign ($) as a constant part of the query, simply precede it with a backslash (\). For example:

```
... where symbol = '\$' ...
```

Parameters and declared variables specified in the **report** command may also be used in the body of the report to indicate text to be printed. The value of the parameter or variable is printed when the parameter or variable name is used in a text printing statement. See the POPULATION example in Appendix A for an example of this.

Examples

The query:

```
.query
      select *
            from emp
            where salary > $sal
            and   dept = '$dept'
```

when invoked with the command:

```
report mydb myrep (sal = 50000,dept = CAE)
```

executes the following query:

```
select *
      from emp
      where salary > 50000
      and dept = 'CAE'
```

For another example, consider a table called "account" with fields including "custno," "custname," "checking," and "savings". You have separate fields for checking and savings accounts on one row because most customers have both a savings and a checking account with the bank. If you want to write one report specification that prints either the savings or checking account balances with a single query, you could code a **.query** statement similar to the following:

```
.declare Account_type = ...
.query
     select custno, custname, val=$Account_type
          from account
```

The above query can be invoked with the command:

```
report otherdb repname
```

At execution, the Report-Writer issues the following prompt:

```
$_Enter 'Account_type':
```

If you were to respond with:

```
savings
```

the following query would be executed:

```
select custno, custname, val=savings
     from account
```

Note that this query selects values from the database column "savings"; it does not select the string constant "savings".

.sort

The **.sort** statement specifies the ordering of rows to be reported.

Syntax

.sort|.srt {*columnname*[:*sortorder*] {,*columnname*[:*sortorder*]}}

Description

The optional **.sort** statement specifies the ordering that applies to the rows of data to be reported. Rows are first sorted on the first column in the list, and if several rows have the same value for that column, they are sorted on the second column in the list, and so forth. If there is exactly one sort column, and there are duplicate values for the sort column, all rows with that value appear together, but in an undetermined order relative to each other.

The **.sort** statement also specifies the columns used as break columns in the report (unless overridden by a **.break** statement). A break on one column in the sort list produces a break on all subsequent columns in the list.

Each column specified in the **.sort** statement can have header and/or footer formatting statements specified (with the **.header** or **.footer** statement.) Of course, columns specified on the **.sort** statement do not have to be break columns as well. You can use the **.sort** statement simply to order rows for appearance in the report.

Note that using a **.sort** statement eliminates duplicate rows from your query, leaving only one instance of each different set of data values. Depending on the storage structure of the table, the table may have duplicate rows stored in it. When that table is sorted using the **.sort** statement, the duplicate rows are eliminated, which could result in fewer records than you initially expected.

You may have either a **.sort** statement or an **order by** clause in a **.query** statement but not both in a report specification.

Parameters for .sort

Parameter	Description
columnname	The name of a column in the table to be reported, or the label for a column in the result column list of the specified query.
sortorder	Either **ascending** (or **a**) or **descending** (or **d**), depending on how you want the rows to be ordered. If neither is specified, the default is **ascending**.

Examples

Sort two columns of a table, both in ascending order:

```
.sort sex,name
```

Sort three columns of a table, each with different orders.

```
.srt dept:descending, jobcode, name:d
```

.break

The **.break** statement specifies the break columns for the report and the order in which they should break.

Syntax

.break|.brk *columnname* {, *columnname*}

Description

The optional **.break** statement can specify the break columns if no **.sort** statement has been specified, or to override the default break columns created by the **.sort** statement. The order in which the break statements are processed is the order in which they appear in the specified break list. A break on one column in the list produces a break on all subsequent columns in the list.

The columns that have **.header** or **.footer** statements must be included in this break list if no **.sort** statement is specified. However, you do not have to specify a **.sort** statement to use the **.break** statement.

If you specify an **order by** clause in a **.query** statement, you must also specify a **.break** statement that lists the columns in the **order by** clause. The **.query** statement does not create default column breaks as does the **.sort** statement.

If a **.sort** statement is specified in addition to a **.break** statement, the break columns in the **.break** statement completely supersede the list of break columns declared implicitly in the **.sort** statement. The sort still takes place in the order requested in the **.sort** statement. However, the columns named in the **.sort** statement are no longer assumed to be the break columns by default. This feature is useful in situations where you want to disable the break action on one column of a report, but you still want to print the report in sorted order on that column.

Parameters for .break

Parameter	Description
columnname	The name of a column in the table to be reported, or the label for a column in the result column list of the specified query.

Examples

The first example breaks on two columns. The order to sort the rows retrieved from the database appears in the **.query** statement:

```
.query
     select *
          from emp
          order by state, city
.break state, city
```

In this example, the **.break** statement was required to identify the sort columns to the Report-Writer.

Chapter 6
Page Layout and Control Statements

This chapter explains the page layout and control statements. These include:

.leftmargin
.rightmargin
.pagelength
.formfeeds/.noformfeeds
.newpage
.need

.leftmargin

The **.leftmargin** statement sets a specific left margin to the report.

Syntax

.leftmargin| .lm [+|-] *n*

Description

The **.leftmargin** statement sets the left margin of the report. Subsequent to this statement, new lines begin at the new left margin position. To set the left margin for the entire report, place the statement in the **.header report** section. The **.leftmargin** position is used by the **.left** and **.center** statements to determine the default position for those statements.

If not specified, a default value is determined for your report. See "Automatic Determination of Default Values" in Chapter 3.

Parameters for .leftmargin

Parameter	Description
n	The position of the new left margin of the report. If signed, the new position is calculated relative to the current position. If unsigned, it is set to absolute position *n*. The default value is discussed in "Automatic Determination of Default Values" in Chapter 3.

The value specified for the **.leftmargin** statement must be greater than or equal to zero (0), less than the specification for the right margin and less than the line size (as specified with the **-l** flag on the **report** command).

Example

The following sets the left margin to 5; printing begins at the sixth character position.

```
.lm 5
```

.rightmargin

The **.rightmargin** statement sets a specific right margin to the report.

Syntax

.rightmargin| .rm [+|-] *n*

Description

The **.rightmargin** statement sets the right margin of the report. To set the right margin for the entire report, place the statement in the **.header report** section. The **.rightmargin** value is used by the **.right** and **.center** statements to determine the default position for those statements. Text is wrapped around within the right margin.

If not specified, a default is determined for the report. See "Automatic Determination of Default Values" in Chapter 3.

Parameters for .rightmargin

Parameter	Description
n	The position of the new right margin of the report. If signed, the new position is calculated relative to the current position. If unsigned, it is set to absolute position *n*. The default value is discussed in the section titled "Automatic Determination of Default Values" in Chapter 3.

The value specified for the **.rightmargin** statement must be greater than the specification for the left margin and less than the line length (as set by the **-l** flag on the **report** command).

Examples

The following statements specify margins that produce a default **.center** position of 50.

```
.lm 10
.rm 90
        ...
.center
.print "This is a title in position 50"
```

.pagelength

The **.pagelength** statement sets a new default page length, in number of lines per page.

Syntax

.pagelength|.pl *nlines*

Description

The **.pagelength** statement sets the page length. As the report is written, the report processor checks the number of lines remaining on the current report page. If all the body text lines have been written, a page footer is printed followed by a page header, assuming that headers and footers have been specified.

If the **.formfeeds** statement is in effect, the pages are given formfeeds at the end of each page footer.

This statement can be overridden at runtime by specifying the **-v** flag on the **report** command line, as described in Chapter 11.

Parameters for .pagelength

Parameter	Description
nlines	The number of lines per page. The default is 61 lines per page if the report is written to a file, and 23 lines per page if written to a terminal.

The value used in the **.pagelength** statement must be greater than the combined number of lines specified in the heading and footing for the page.

Example

Set a new page length for terminals.

```
.pl 24
```

.formfeeds and .noformfeeds

The **.formfeeds** and **.noformfeeds** statements force or suppress the addition of formfeed characters to the end of each page in the report.

Syntax

.formfeeds|.ffs| .ff

.noformfeeds|.noffs| .noff

There are no parameters to either statement.

Description

For printers that support formfeeds, use these statements to embed ASCII formfeeds in your report files for pagination. The **.formfeeds** statement can be used to force formfeeds at the start of the report, and at the end of each page in the report. The page size is determined with the **.pagelength** statement or as a default value. When writing to a terminal, the **.formfeeds** statement is ignored.

The formfeed character is sent at the end of the page footer formatting statements, if specified. If not specified, it is sent after the last line of the page, as determined from the page size.

Specify the **.formfeeds** statement at the start of your report specification statements, before any header or footer statements are specified.

These statements can be overridden at runtime with the **-b|+b** flag on the **report** command line, as described in Chapter 12.

Example

The default is:

```
.noformfeeds
```

To turn on **.formfeeds**:

```
.formfeeds
```

.newpage

The **.newpage** statement forces a page break, with an optional change in the page number.

Syntax

.newpage|.np [[+|-] *pagenumber*]

Description

The **.newpage** statement can appear anywhere in your report specifications. It forces a page break by skipping to the end of the page and printing a page footer. Then a new page begins by incrementing the page number (or setting the page number to the specified value) and writing out a page header.

At the end of the report, a **.newpage** statement is automatically performed if a page footer is specified (in this case, no page header appears on the next page). If a **.newpage** statement is encountered as the first printing action of the report, no page footer is printed.

Parameters for .newpage

Parameter	Description
pagenumber	The page number to be assigned to the next page in the report. If signed, the next page number is calculated relative to the current page number. If unsigned, the next page number is set to the specified value. If not specified, the default page number is determined by incrementing the current page number by one.

Examples

The following skips to a new page, incrementing the page number by 1.

```
.newpage
```

The following skips to a new page, and numbers the new page as page 22.

```
.np22
```

.need

The **.need** statement keeps a specified number of text lines together on a page.

Syntax

.need| **.ne** nlines

Description

The **.need** statement insures that a number of text lines are kept together on a page. Page breaks are conditionally made to keep the text blocks together. This statement can be used to make sure that all lines of text in the headers, and so on, are kept on the same page. See Appendix A for multiple examples of their placement.

Parameters for .need

Parameter	Description
nlines	is the number of lines in the text block to remain together.

Example

The following keeps the break header together on one page.

```
.need 3
.print "Header for account:",acct .nl
.print "------ --- -------" .nl 2
```

Chapter 7
Report Structure Statements

This chapter explains the report structure statements. These include:

.header
.footer
.detail

.header

The **.header** statement identifies the beginning of a block of formatting statements to execute at the top of a break.

Syntax

.header|.heading| .head

report | page | *column_name*

Description

The **.header** statement starts the block of text formatting statements that define the action at the start of a break in the report. The statements that appear after the **.header** statement are executed before a new value of a break column (if the *column_name* is specified in the **.header** statement), before the start of the report (if the keyword **report** is specified), or at the top of all pages but the first (if the keyword **page** is specified).

All statements between one **.header** statement and any subsequent **.header**, **.footer**, or **.detail** statement are considered as part of the first header action.

Parameters for .header

Parameter	Description
column_name	A break column name specified in the **.sort** or **.break** statements.

Example

The following statements start a page header.

```
.header page
     .tab 10 .print "Accounts Receivable Aging
          Report by Client"
     .newline
```

.footer

The **.footer** statement identifies the end of a block of formatting statements to execute at the end of a break.

Syntax

.footer| .footing|.foot

report | page | *column_name*

Description

The **.footer** statement starts the block of text formatting statements that define the footer action at the end of a break in the report. The statements that follow the **.footer** statement execute at the end of a group of data rows with the same value for a break column (if the *column_name* is specified on the **.footer** statement), at the end of the report (if the keyword **report** is specified), or at the bottom of each page (if the keyword **page** is specified).

All statements between one **.footer** statement and any subsequent **.header**, **.footer**, or **.detail** statement part of the first footer action.

Parameters for .footer

Parameter	Description
column_name	A break column name specified in the list of the **.sort** or **.break** statement.

Example

The following starts the footer for a report. It is followed by a header for abc.

```
.footer report
     .tab 10 .pr "This is the report footer"  .nl
.head abc
```

.detail

The **.detail** statement specifies the start of the detail section of the report specification.

Syntax

.detail|.det

The **.detail** statement has no parameters.

Description

The **.detail** statement starts the group of formatting statements execute each time a data row for the report is processed. These formatting statements execute after any break headers and before any break footers.

The formatting statements specified in the **.detail** block are also used for determining the default margins and the default positions of columns. See "Automatic Determination of Default Values" in Chapter 3 for more information on how this is accomplished.

Example

```
.detail
      .PR acctnum(b16), tdate(b16), .T+8 .P
            transnum("nnnn"), deposit,
      withdrawal .T+5 .P cum(acctnum)
            sum(amt.balance) .NL
```

This example illustrates how the **.detail** statement works, you need not understand the contents of each line in this example.

Chapter 8
Column and Block Statements

This chapter explains the column and block statements. These include:

.format
.tformat
.position
.width
.block and **.endblock**
.top
.bottom
.within and **.endwithin**

.format

The **.format** statement sets up a default printing format for a column or set of columns.

Syntax

.format| **.fmt** *columnname* {, *columnname*} (*format*)

{, *columnname* {, *columnname*} (*format*) }

Description

The **.format** statement sets up a default format associated with a column to be used whenever the column or an aggregation of a column is printed. You can use the **.format** statement to control the default width of a column. It is used in determining the default width only if the **.width** or **.position** statements are not used to specify the default width for a column.

If a **.format** statement is not specified for a column, the Report-Writer determines it. See "Setting Up Default Values." If a default format cannot be determined, the Report-Writer uses the default values that are listed in "Automatic Determination of Default Values" in Chapter 3.

Breaks occur on the formatted values not on the actual values. To force the Report-Writer to use the actual, rather than the formatted, values to determine breaks, you must specify the **-t** flag on the **report** command line, as described in Chapter 12.

Parameters for .format

Parameter	Description
columnname	The name of a column (or columns) in the data being reported.
format	A valid format specification, as described in the section titled "Overview of Format Specifications." The format must be the correct type for the column(s).

Example

This example shows a **.format** statement that declares formats for several columns, followed by a **.print** statement that uses the formats specified in the **.format** statement to print the information.

```
.format trans, balance ("$$$,$$$,$$$.nn"),
     charvar (c20), a,b,c,d (f10.2)
       ..
.print trans,balance,charval,a,b,c,d
```

.tformat

The **.tformat** statement changes the format temporarily for the output of a column.

Syntax

.tformat| .tfmt *columnname* {, *columnname*} (*format*)
{, *columnname* {, *columnname*} (*format*) }

Description

The **.tformat** statement temporarily changes the format used to print a value of a column. After the column is printed using this format, the temporary format is discarded, and the next printing of the column uses the default format.

To print a leading dollar sign for the currency the first time it appears on a page, you could specify a **.tformat** statement. For example, put "$$$,$$$,$$n.nn" in the "header" action for page breaks. If the normal format for the column is "zzz,zzz,zzn.nn," the column prints with a leading dollar sign the first time it prints on each page.

Another common use of the **.tformat** statement is for blanking out the unchanged values of break columns in the "detail" action for a report. The **B** type format (described in Chapter 4) is used to accomplish this. By specifying a **B** format with the appropriate field width as the standard format for printing a column in the detail section, the default action blanks out and does not print the value of that column. If a printing format in a **.tformat** statement is specified in the heading for a break in the column, you can print the column whenever a new value is encountered for that column. See the use of the **.tformat** statement for the "date" column in the ACCOUNT report example in Appendix B or the examples below for more details.

Parameters for .tformat

Parameter	Description
columnname	The name of a column in the report data.
format	A printing format, as described in Chapter 2, "Overview of Report Specification Statements". It should be of the right type for this column.

Examples

The following prints a dollar sign at the top of a page:

```
Top of page

Jones, A.   $23,145
Jones, B.    16,145
Jost, C.     32,143

     ...

.header page
     .print "Top of page" .nl 2,
     .tformat salary("$$$,$$n")
.detail
     .print name(c14), salary("zzz,zzn")
       ...
```

The following prints the value of a break column when it changes:

```
01-34567-8                $345.21
                           $14.10
                         $1,143.23
04-35999-2                   $1.99
                           $177.00

                         ...*/

.format acctnum(b10), transact("$$$,$$$,$$$.nn")
       ...
.heading acctnum
     .tformat acctnum(c10)
.detail
     .print acctnum .tab +2 .print transact
       ...
```

.position

The **.position** statement sets a default output position and optional width associated with a column.

Syntax

.position| .pos *columnname* {, *columnname*} (*position* [,*width*])

{, *columnname* {, *columnname*} (*position* [,*width*]) }

Description

The **.position** statement sets a default position in the output line associated with a column name for use with statements such as:

.left
.right
.center
.tab.

It can also be used to set an optional default width of a column when calculating positions in the **.center** and **.right** statements.

Normally, this statement is not needed, as default positions and widths are determined from the formatting statements. See "Automatic Determination of Default Values" in Chapter 3. However, if the determined default position for a column is not convenient, or you would like a different position associated with a *columnname*, you can override the default with this statement. Subsequently, you can use the **.tab**, **.right**, **.left**, or **.center** statements with a *columnname* to refer to this position.

If you do not specify a **.position** statement for a column, and *columnname* is not printed in the report, the default position is zero (0). If a position is specified, but no width is specified for a column, the default width is determined by looking at the default format for the column. You can optionally use the **.width** statement to specify the width of a column.

Parameters for .position

Parameter	Description
columnname	The name of a column in the report.
position	Specifies the location on the output line where the default column position should be. This value must be less than the maximum line size (as set by the **-l** flag on the **report** command) and greater than or equal to 0.
width	The default width of the column to be used when calculating the positioning for **.center** and **.right** statements. If not specified, this value is determined by looking at the default format for this column.

Examples

The following sets up a default position for columns, and prints out the data:

```
.position acct(5), transact(20), balance(35)
.format transact, balance ("$,$$$,$$$.nn")
.format acct("nn-nnnnn-n")
        ..
.tab acct .print acct
.tab transact .print transact
.tab balance .print balance
```

The resulting printout looks like this:

```
01-02234-4  $1,345.24   $11,429.32
02-41989-1    $876.24   $10,553.08
```

An easier way to set up the default positions is shown below:

```
.format transact, balance ("$,$$$,$$$.nn")
.format acct("nn-nnnnn-n")

.detail
      .t5 .p acct .t20 .p transact
      .t35 .p balance .nl
```

.width

The **.width** statement sets a default output width associated with a column.

Syntax

.width *columnname* {, *columnname*} (*width*) {, *columnname* ... }

Description

The **.width** sets the default width of a column when calculating positions in the **.center** and **.right** statements. Alternately, you can specify the default width for a column as a parameter to the **.position** statement, and this statement is provided for convenience and documentation only. Normally, this statement is not needed, as default widths are determined by an analysis of the report-formatting statements. See the section titled "Automatic Determination of Default Value" in Chapter 3 for a full description of how the default values are determined. However, if the determined default width for a column is not convenient, or you would like a different width associated with a *columnname*, you can override the default with this statement. Subsequently, you can use the **.right** or **.center** statements with a *columnname* to use this width, in conjunction with the default position for this column in calculating the placement of text.

If no width is specified for a column, the default width is determined by looking at the default format for the column.

Parameters for .width

Parameter	Description
columnname	The name of a column in the report.
width	The width of the column to be used when calculating the positioning for the **.center** and **.right** statements.

Example

Set up default position and widths for columns to print out the following:

```
        SAL1              SAL2   |
|   $1,234.24   |    $11,429.32|
|     $876.24   |    $10,553.08|

      ....

.position sal1(3), sal2(18)
.format sal1, sal2 ("$$$,$$$.nn")
.width sal1(14), sal2(16)
        ..
.head ...
.ce sal1 .pr "SAL1"
.rt sal2 .pr "SAL2"
.detail
      .tab sal1 .pr "|", sal1
      .rt sal1 .pr "|"
      .tab sal2 .pr sal2
      .rt sal2 .pr "|"
```

.block; and .endblock

The **.block** and **.endblock** statements set the Report-Writer into, and out of, **block** mode. This lets you refer to positions on previous as well as subsequent lines in the report.

Syntax

.block| .blk
Other formatting statements
.endblock|.endblk|.end block

There are no parameters to either statement.

Description

The **.block** and **.endblock** statements switch the Report-Writer into and out of block mode. This gives you advanced formatting capabilities. In block mode, you can move across the page (through the **.tab** statement), down the page (through the **.newline** statement), and back up the page (through the **.top** statement). Block mode gives you the capability of printing information in your report, and then putting summary information ahead of the information. This can be accomplished by switching the Report-Writer into block mode, printing out some number of lines, moving to the top of the block to add summary information, and then printing out the entire block by leaving block mode.

When used in conjunction with the **.within** and **.endwithin** statements, described later in this chapter, you can describe column headings and subtotalling in a more natural and convenient fashion than is possible if you had to describe each line completely before going to the next line.

All formatting statements are allowed within block mode, except for the **.newpage** and **.need** statements. You can use the **.top** and **.bottom** statements only while in block mode to move the current position within the block.

The length of a block written in block mode is limited by the **-w** flag on the **report** command. By default, a block can be up to 100 lines long, though by setting the value of *mxwrap* on the **report** command line, you can increase this value.

Examples

The following example illustrates a block of statements followed by an example of the output.

```
.block
      .pr "Line 1" .newline
      .pr "Line 2" .newline
      .top
      .tab 10 .pr "more line 1" .newline
.endblock
```

These statements produce the following:

```
 Line 1  more line 1
 Line 2
```

The following statements are from the POPULATION report:

```
.header region
      .need 4
      .block
            .pr "Region: ", region .nl
.detail
            .t5 .pr state(c15) ...
                  totother("n,nnn,nnn") .nl
.footer region
            .top .lineend .tab+5
            .pr "Count of states: ", count(state) (f3)
      .end block
```

They produce the following output:

```
Region: Mountain          Count of states:    8
      Arizona                    1,770,900  . . .
      Colorado                   2,207,259  . . .
      Idaho                        712,567  . . .
      Montana                      694,409  . . .
      Nevada                       488,738  . . .
      New Mexico                 1,016,000  . . .
      Utah                       1,059,273  . . .
      Wyoming                      332,416  . . .
```

.top

The **.top** statement changes the current output line to the top line in the current block.

Syntax

.top| .tp

There are no parameters to this statement.

Description

The **.top** statement is used in block mode (after a **.block** statement and before the corresponding **.endblock** statement). It moves the current output line to the first (topmost) line in the block.

The character position on that line is the position at which the line was when the last **.newline** statement affected the topmost line. To get to the left margin of the top line, you can use the **.tab** statement with no parameters. To get to the last nonblank character on the line, you can use the **.lineend** statement.

Examples

The following is an example of the statements and the output they produce:

```
.block
      .pr "Line 1" .newline
      .pr "Line 2" .newline
      .top
      .tab+2 .pr "more line 1" .newline
.endblock
```

These statements produce the following output:

```
Line 1  more line 1
Line 2
```

.bottom

The **.bottom** statement changes the current output line to the bottom line in the current block.

Syntax

.bottom |.bot

There are no parameters to this statement.

Description

The **.bottom** statement can be used only while block mode is in effect (that is, after a **.block** statement, but before the corresponding **.endblock** statement). It moves the current output line to the current bottom line in the block. The character position on that line is one space beyond the last character printed on that line.

Example

By using the following sequence of Report-Writer statements:

```
.block
      .pr "Line 1" .newline
      .pr "Line 2" .newline
      .top
      .tab+2 .pr "more line 1" .newline
      .bottom .lineend
      .pr "Last line in block" .newline
.endblock
```

You would get the following output:

```
Line 1  more line 1
Line 2Last line in block
```

.within and .endwithin

The **.within** and **.endwithin** statements set the Report-Writer into, and out of, column formatting mode.

Syntax

.within|.wi *columnname* {, *columnname*} | ***all***

Other formatting statements:

.endwithin
.endwi
.end within

Description

The **.within** and **.endwithin** statements switch the Report-Writer into and out of column formatting mode, in which the right and left margins of the report are temporarily set to correspond to a column position within the report. This allows you to set margins of the report temporarily to the left and right margins for a given column, determined either by default (as described in Chapter 3), or through the use of the **.position**, **.width**, and **.format** statements. All statements between the **.within** and the corresponding **.endwithin** statement is processed using the margins for that column, rather than the margins for the report. If more than one column is specified on the **.within** statement, or if the keyword **all** is used, the set of statements is applied to each of the columns in turn.

Parameters for .within and .endwithin

Parameter	Description
columnname	The name of a column (or columns) in the data being reported within which the "other formatting statements" are to be used.
all	Means that all columns in the report are to be used.

When using the **.within** and **.endwithin** block of statements for a set of columns, it is often useful to be able to invoke a slightly different set of formatting statements within each column, differing only in the column referenced by a formatting statement. To help accomplish this, two special names are available for use in formatting statements while in column formatting mode. These can be used to refer to the column that is currently being used. These names are:

w_column	Can be used anywhere *columnname* would normally be used on a formatting statement, such as in ".print w_column" or ".print sum(w_column)."
w_name	The name of the column currently being used in the within block. It can be used to print out the actual column names.

See the following examples or the reports in Appendix A for the use of these special names.

Because the margins of the report are temporarily changed to the margins for a column while the **.within** statement is in effect, the positions referred to by the default values for the **.left**, **.right**, and **.center** statements are those of the column, rather than the full width of the report.

In most cases where the formatting statements inside the **.within** block include one or more **.newline** statements, you should precede the **.within** statement with a **.block** statement (and follow the **.endwithin** with an **.endblock**), because you probably want to move down the page within one column and go back up to the line on which you started before proceeding to the next column. In fact, a **.top** statement is automatically executed immediately before the **.endwithin** statement to simplify this type of specification.

Once you start to use the **.within** and **.endwithin** statements, you may find that the **.position**, **.width**, and **.format** statements take on additional usefulness.

Examples

Here is a sequence of Report-Writer statements:

```
.position col1(0), col2(8), col3(16)
         ..
.within col1, col2, col3
       .pr w_name
.end within
```

The sequence results in the following output:

```
col1        col2          col3
```

Here is another example:

```
.position totpop(2,15), totwhite(20,15)
.block
      .within totpop
            .ce      .pr    "Total" .nl
            .ul .ce .pr "Population" .nl .nou
      .end within
      .within totwhite
            .nl
            .ul .ce .pr "White Pop" .nl .nou
      .end within
.end block
```

This sequence results in the following output:

```
  Total
Population     White Pop
----------     ----- ---
```

The following statements in POPULATION can be used to print out the subtotals for each of two columns using the same set of statements.

```
.position totpop(10), totwhite(25)
.format totpop,totwhite("nnn,nnn,nnn")
        ..
.foot report
      .block
            .within totpop, totwhite
                  .rt .prline "-----------"
                  .rt .prline sum(w_column)
            .end within
      .end block
```

This would result in the following:

```
          . . . . .
        (detail lines)
          . . . . .

-----------    -----------
203,165,699    177,612,309
```

Chapter 9
Text Positioning Statements

This chapter explains the text positioning statements. These include:

.tab
.linestart
.lineend
.newline
.left
.center
.right

.tab

The **.tab** statement specifies the position on the line to print text.

Syntax

.tab [+|-] *n* | *columnname*

Description

The **.tab** statement moves the current position marker to the specified position.

Parameters for .tab

Parameter	Description
n	The next print position on the line. If signed, *n* represents a relative change from the last position output. If unsigned, *n* represents an absolute position in the line.
columnname	The name of a column being included. The position on the line to which a column refers is determined either explicitly through the use of the **.position** statement, or implicitly as described in the section titled "Automatic Determination of Default Values" in Chapter 3. If *columnname* is specified, the next output text begins at the position associated with the named column.

If the **.tab** statement is not followed by either *n* or *columnname*, then the **.tab** statement works like a **.linestart** statement, with the next text beginning at the left margin of the report. The **.linestart** statement is described later in this chapter.

The **.tab** statement takes on a slightly different meaning when executed in column mode sections (when the **.within** statement is in effect and default column widths and positions are assumed). When the **.tab** statement is executed without a parameter in column formatting mode, the current position moves to the left margin of the current line. The left margin is determined by the **.within** statement.

For more details on column formatting mode, see the **.within** and **.endwithin** statements in Chapter 8.

Example

Suppose you want the following output:

```
abc     def
```

Use these statements:

```
.print "abc" .tab +7 .print "def"
```

To output "HERE" in character position 12 on a line, use:

```
.tab12 .print "HERE"
```

To output the value of column "bal" in position 30 use:

```
.position bal(30)
          ...
     .tab bal .print bal("+++++.NN")
```

.linestart

The **.linestart** statement moves the current position to the left margin.

Syntax

.linestart |.lnstart | .linebegin

There are no parameters to this statement.

Description

The **.linestart** statement changes the position of the current marker for the output line so that the next text printed by the **.print** statement appears at the current left margin. The left margin is set either by the **.lm** statement, by default, or by the left edge of the column currently in use while in a **.within** block. The **.linestart** statement is useful in reports that use the **.tab** statement extensively. The **.linestart** statement always restores the current position marker to a known position, at the beginning of the line.

Example

Suppose you want the following output:

```
abc    def
```

You could use these statements:

```
.tab 14   .pr "def"
.linestart   .pr   "abc"
```

.lineend

The **.lineend** statement moves the current position to the end of the current line.

Syntax

.lineend| .lnend

There are no parameters to this statement.

Description

The **.lineend** statement changes the current position in the output line so that the text printed in the next **.print** statement is placed immediately after the last non-blank character on the line. This is useful in some advanced reports that use the **.tab** statement extensively. The **.lineend** statement always moves the current position marker to a position within the current margins of the report.

Example

Suppose you want the following output:

```
abc     def:xyz
```

You could use these statements:

```
.tab 14 .pr "def "
.tab 5 .pr "abc"
.lineend .pr ":xyz" .linestart .pr "abc"
```

.newline

The **.newline** statement writes out the current line and optionally advances a number of lines on the output page.

Syntax

.newline|.nl [*nlines*]

Description

You must specify the **.newline** statement to advance to a new line on the output page. A **.print** statement does not imply a new line at its completion. You can use the **.println** statement for this purpose.

After **.newline** executes, text output begins at the left margin, unless another text positioning statement overrides the default.

Parameters for .newline

Parameter	Description
nlines	The number of lines to advance. The default value of *nlines* is one, advance to the next line.

If the output of a new line reaches the end of the current page, or if there are fewer than *nlines* left on the current page, the page footer and page header are printed, if so specified.

If the current line includes column type (Cn.w) strings, the **.newline** statement advances to the bottom of the longest column printed during the formation of the line. For the Dictionary example in Appendix A, the **.newline** statement in the footer for "word" causes an advance to the line following the end of the definition.

When you invoke column formatting mode, **.newline** causes an advance to the next line at the left margin determined by the **.within** statement. For more information on column-formatting mode, see the **.within/.endwithin** statements in Chapter 8.

Examples

The following prints one line of text:

```
.print "This is a line" .newline
```

The following is one way you can print an end-of-page (note that the excess newlines are ignored):

```
.print "bye bye page.".nl 10000
```

.left

Left justifies the next text to be printed.

Syntax

.left| .lft [[+|-] *n* | *columnname*]

Description

The **.left** statement left justifies the text printed in the next **.print** statement to either the specified position (relative to the last output or absolute) or to a default position for a column. All leading and trailing blanks are removed from the text before it is placed in the output line.

The **.left** statement is the same as the **.tab** statement for all output except text that contains leading blanks, such as formatted numbers.

Note that the meaning of the **.left** statement is slightly changed when executed in column-formatting mode (when the **.within** statement is in effect and default column widths and positions are assumed). When executed under these circumstances, the **.left** statement positions text at the left margin of the column indicated in the **.within** statement. For more details, see the **.within** and **.endwithin** statements in Chapter 8.

Parameters for .left

Parameter	Description
n	The position to which the next text is left justified. If signed, the position is moved *n* positions relative to the last position output. If unsigned, the position is the absolute position in the output line. The default value is the left margin of the report (set by the **.lm** statement).

Parameter	Description
columnname	The name of a column being included. The position on the line to which a column refers is determined either explicitly through the use of the **.position** statement, or implicitly as described in the section titled "Automatic Determination of Default Values" in Chapter 3. If *columnname* is specified, the next output text is left justified and placed at the position associated with the named column.

Examples

This example outputs the value of "balance," left justifies to the default column.

```
.detail
      ... .t50 .pr balance  ...
.left balance .print balance(f20.2)
```

This example outputs the value of "abc" at the left margin.

```
.left
.print abc("+++++++")
```

.center

Centers the next text to be printed.

Syntax

.center|.cen|.ce [[+|-] *n* | *columnname*]

Description

The **.center** statement centers the text printed in the next **.print** statement. All leading and trailing blanks are removed from the text before it is placed in the output line.

Parameters for .center

Parameter	Description
n	The position around which the next block of text is centered. If *n* is signed, the position is moved *n* positions relative to the last output position. If *n* is unsigned, the position is the absolute position in the output line. The default value is the halfway point between the left and right margins of the report.
columnname	The name of a column being reported. The position on the line to which a column refers is determined either explicitly through the use of the **.position** statement, or implicitly as described in the section titled "Automatic Determination of Default Values" in Chapter 3. The text printed in the next **.print** statement is centered around the "center" for the column (see below).

If you specify *n* (either relative or absolute), the text is centered around that position. If you specify nothing, the Report-Writer calculates the center of the page as the halfway point between the left and right margins of the report. If you specify **.lm** and **.rm** statements, you can calculate the center by the same method. However, if you are using the default values for the right and left margins (the right in particular), read the section titled "Automatic Determination of Default Values" in Chapter 3 for a discussion of how the margins are determined.

If you specify centering with the *columnname* parameter, the text is centered in that column. The center of the column is determined through the default position of the following:

- The column, determined either by the **.position** statement or by default.
- The width of the column, either the default width, the width as specified in the **.width** or the **.position** statements, or the width of the format specified in the **.format** statement.

The **.center** statement centers around a position calculated as:

centering position = default column position + (default format width / 2)

The position is rounded up if there is any fraction.

Note that the **.center** statement has a somewhat different meaning when executed in column-formatting mode (that is, inside a **.within** statement with default column widths and positions assumed). Because the **.within** statement temporarily resets the report margins to the left and right margins of a specified column's width and position, a **.center** statement so executed centers a text string within the column width, not within the report page margins. For more information about column-formatting mode, see Chapter 8.

Examples

This example outputs a title centered on the page.

```
.center
.print "Report Title:",
      current_date
```

This example outputs a heading for column "bal" centered above the default for that column (here, this centers around position 25).

```
.format bal("+++,+++.nn")
            ...
.center bal .print "Balance"
            ...
.detail
            ...
      .tab 20 .print bal ...
```

.right

.right justifies the next text to be printed.

Syntax

.right|.rt [[+|-] *n* | *columnname*]

Description

The **.right** statement right justifies the text printed in the next **.print** statement. All leading and trailing blanks are removed from the text before it is placed in the output line.

Parameters for .right

Parameter	Description
n	The position to which the next block of text is right justified. If signed, the position is moved *n* positions relative to the last output position. If unsigned, the position is the absolute position in the output line. The default value is the right margin of the report.
columnname	The name of a column being included. The position to which this column refers is either determined explicitly, through the use of the **.position** statement, or implicitly as described in the section titled "Automatic Determination of Default Values" in Chapter 3. The text printed in the next **.print** statement is right justified to the right edge of that column, as determined from the default position and width of that column (see below).

If you specify *n* (either relative or absolute), the text is right justified to that position. If you specify nothing, the Report-Writer right justifies the text to the right margin of the report. The right margin is either specified with the **.rm** statement or determined by default as described in the section titled "Automatic Determination of Default Values" in Chapter 3.

If you specify right justification with the *columnname* parameter, the text is right justified to the right edge of that column, as determined from the following:

- The default position of the column, determined either from the **.position** statement or by default.
- The width of the column, either the default width, the width specified in **.width** or the **.position** statements, or the width of the format specified in a **.format** statement for that column.

The **.right** statement justifies to a position calculated as:

justification position =
default column position + default width

The meaning of the **.right** statement is slightly changed when the **.right** statement is executed within column-formatting mode (that is, when the **.within** statement is in effect and default column widths and positions are assumed). When the **.right** statement is so executed without a parameter, the current position becomes the right margin as defined by the **.within** statement, not the right margin of the report. For more information about column-formatting mode, see the **.within** and **.endwithin** statements in Chapter 8.

Examples

This example outputs a page number, right justified on the line.

```
.right
.print "Page", page_number("zn")
```

This example outputs a heading for column "bal," right justified to the right edge of that column (in this example, to position 50).

```
.format bal("+++,+++.nn")
                ...
.detail
               ...
      .t 40  .print bal ...
               ...
.right bal  .print "Balance"
```

Chapter 10
Print Statements

This chapter explains the print statements. These include:

.print and **.println**
.underline and **.nounderline**
.ulcharacter
.nullstring

.print and .println

The **.print** and **.println** statements print literal text strings, columns from the database, or expressions on the report.

Syntax

.print |**.pr**|**.p** *expression* [(*format*)] {, *expression* [(*format*)]}

.println |**.prln**|**.pln** *expression* [(*format*)] {, *expression* [(*format*)]}

Description

The **.print** statement specifies text to include in the body of the report. Text can be character strings printed directly, data items from the data table, program variables, parameters, aggregations or a combination of these. The text is included at the place in the report where the **.print** statement is encountered. By preceding the **.print** statement with the positioning statements such as **.newline**, **.tab**, **.center**, **.right** or **.left**, you may specify the location of the text. By default, the text is included immediately after the last text output with the **.print** statement.

If you use the optional **.println** form of the statement, the current print position advances to the next line after the specified text is printed.

Parameters for .print

Parameter	Description
expression	Any legal expression.
format	An optional format specification for the expression. The form of the specification depends on the expression type. If you do not specify a format, the Report-Writer uses one of the default formats (see "Format Specifications" in Chapter 4).

The **.print** statement can include as many expressions as you wish.

Examples

The following statements use literals:

```
.print "This is some text"
     " which can be included on several lines."
```

The statements print the following:

```
This is some text which can be included on several
lines.
```

Note that the two text strings printed next to each other on the same line, because no specification statement separates the fields.

In another example, if "page_number" were equal to 3, here are the statements:

```
.pr "Page number:", page_number(f2)
```

It prints the following:

```
Page number: 3
```

The following example shows the specifications needed to print a data value and an aggregate, using a numeric template for the aggregate:

```
.p bal, sum(bal)("nnn,nnn,nnn")
```

A complex **.print** statement that displays a large number of data items might look like the following:

```
.print "Values of the data are: ", var1,
var2(e20.4) cvar1(c40), " and finally",
lastvar ("   $$$,$$$,$$$.nnCR")
```

Note that in the above example, the field *var1* was listed without a specification. The Report-Writer prints the value with the default format for the data type, according to the table in the section titled "Default Format for Columns." You can mix the default data formats with complex templates.

.underline and .nounderline

The **.underline** and **.nounderline** statements turn underlining of text on and off.

Syntax

.underline|.ul|.u
any printing statements
.nounderline/.noul|.nou

There are no parameters to either statement.

Description

To underline text in a report, put an **.underline** statement immediately before the spot where underlining begins, and **.nounderline** at the spot where it stops. You can underline anything that can be printed—including character strings, column values, parameter values, or aggregate values. By default, the underlining character is a hyphen (-) if the report is written to a terminal, and an underscore (_) if written to a file. This can be changed with the **.ulcharacter** statement (described in the next section).

When underlining mode is in effect, only letters and digits are underlined. All other characters, such as blanks, commas, periods, and so on, are ignored. If the underlining character is anything other than an underscore, the underlining is printed on the line below that containing the text to be underlined. Underscores are printed on the same line.

Example

Suppose you want to produce the following line:

```
Here is 123,456 underlining
----    --- --- -----------
```

Use the following specifications:

```
.u .pr "Here" .nou .print "is" .u .pr

  "123,456 underlining"
```

.ulcharacter

The **.ulcharacter** statement sets the underlining character to any single character.

Syntax

.ulcharacter|.ulchar|.ulc "*c*"

Description

You can specify an alternate underlining character with the **.ulcharacter** statement.

Parameters for .ulcharacter

Parameter	Description
c	Any single character, within single or double quotes, subsequently used as the underlining character. The default underlining character is the hyphen (-) for reports written to a terminal and the underscore (_) for reports written to a file.

The character *c* must be a single character enclosed in quotes. That character remains in effect until another **.ulcharacter** statement is encountered in the report.

If underscoring (_) is specified with the **.ulcharacter** statement, underlining is printed on the same line as the text. If any other character is specified, or the default character of hyphen (-) is used, underlining is printed as a second line immediately below the underlined text.

Example

To produce the following:

```
Underline me
--------- --
and me
=== ==
```

Use the following specifications:

```
.underline
     .ulc "-" .pr "Underline me" .nl
     .ulc "=" .pr "and me" .nl
.nounderline
```

.nullstring

The **.nullstring** statement specifies an alternate null string.

Syntax

.nullstring|.nullstr *'null_string' | "null_string"*

Description

The **.nullstring** statement specifies a string to print when a null value appears on the report. Because a data value of *null* means that there is really no data present to print, you can use the **.nullstring** to print a designated string that signifies the absence of the data.

Parameters for .nullstring

Parameter	Description
null_string	Any string of characters. You must enclose the string in single or double quotes, so the computer can properly handle leading and trailing blanks, which are important in some format specifications.

The column must be large enough to print the designated *null_string*; if it is not, the empty string is printed in that instance.

If you do not specify a **.nullstring** statement, the Report-Writer uses a default of the empty string (a string with no characters) to print a null value. You can specify several **.nullstring** statements in a report specification. The system uses the current **.nullstring** until another **.nullstring** statement is executed.

Example

Suppose "phone_number" is an integer column with a null value. If you issued the following print statements:

```
.nullstring       "N/A"
.print            "Phone number = ",
                  phone_number .nl
.nullstr          '?'
.print            'Phone number = ',
                  phone_number .nl
```

Report-Writer prints the following:

```
Phone number = N/A
Phone number = ?
```

Chapter 11
Conditional and Assignment Statements

This chapter explains the conditional and assignment statements. These include:

.if
.then
.else
.let

.if

The **.if** statement specifies alternative statements to execute under specific conditions.

Syntax

.if *condition* **.then** {*statement*}
  {**.elseif** *condition* **.then** {*statement*}}
  [**.else** {*statement*}]
.endif

Description

The **.if** statement specifies alternative statements to execute depending upon the specific condition.

Conditions are evaluated one after another. As soon as one condition is met, the statements following **.then** are executed. If *none* of the specified conditions are met, nothing is done. If, when none of the conditions are met, there is an **.else** included, the statements following the **.else** statement are executed.

Parameters for .if

Parameter	Description
condition	A Boolean expression that returns the value true or false.
statement	Any action statement, including the **.if** statement (this excludes the setup and structure statements in Chapters 5 and 7).

Examples

This example illustrates the use of the **.if** statement to evaluate the current condition of the Report-Writer environment. It tests the current character position, and starts a new line if the current position is past the end of a line:

```
 .if position_number > 80 .then
      .newline
.endif
```

This example tests the data, executes, and prints different print statements depending on the result:

```
.if balance > 0 .then
      .print "(",-balance, ")"
.else
      .tab +1
      .print balance
.endif
```

This example tests a column value and uses **.if** statements to translate a numeric code to a text string:

```
.if deptcode = 1 .then
      .print "books"
.elseif deptcode = 2 .then
      .print "furniture"
.elseif deptcode = 3 .then
      .print "jewelry"
.else
      .print "misc"
.endif
```

.let

Assigns the value of an expression to a declared variable.

Syntax

.let *variablename* [:]= *expression*

Description

The **.let** statement evaluates an expression and assigns the value to a declared variable. The type of the expression must be compatible with the type of the variable. For example, an integer expression can be assigned to a floating-point variable, and a date expression string can be assigned to a date variable, but a date expression cannot be assigned to an integer variable.

Parameters for .let

Parameter	Description
variablename	A variable name declared in a **.declare** statement (see Chapter 5). The *variablename* cannot be a special report variable or column.
expression	An expression. The *expression* cannot be a Boolean or conditional one,because there cannot be variables of data type Boolean.

Example

The following **.declare** statement defines two data variables, age and birthday. These are then used in **.let** statements showing a number of possible assignments that could be made.

```
.declare
     age = integer,
     birthday = date

.let age := 6.2
.let age = age+1 /* one year older*/
.print age .nl
.let birthday := "29-jul-1954"
.let age =
     date ("9-jul-1987") - birthday      /*ERROR!*/
.let age := interval ("years",
     date ("9-jul-1987") - birthday)
.print age .nl
```

If these assignments were made in sequence as shown, the output would be as follows:

```
 7
32
```

Chapter 12
The sreport, report, and copyrep Commands

This chapter explains the following commands:

sreport
report
copyrep

sreport

The **sreport** command checks a report specification file and stores the specification in a database.

Syntax

sreport [**-s**] [**-u***username*] *dbname filename*

Description

The **sreport** command reads a report specification file of Report-Writer statements, performs basic syntax error checking, and, if error-free, stores the report specification in the Reports Catalog of the database you specify. If the report specification contains syntax errors, **sreport** prints error messages. If a report in the text file has the same name as an existing report in the Reports Catalog, the older report definition is replaced. If no prior report exists, the report is added to the Reports Catalog. You can then use the specifications to run a report using either the **report** command or the Reports option of ODT-DATA/MENU.

Parameters for sreport

Parameter	Description
-s	If specified, requests that the status messages normally printed by **sreport** be suppressed.
-u*username*	If specified, requests that **sreport** act as if you are a user with login name *username*. This may only be used by the DBA for the database or by the ODT-DATA system manager. No spaces appears between the "u" in the flag and the first character of the username. A space is interpreted as a username of blank characters.
dbname	The database in which the report specification is to be stored.

Parameter	Description
filename	The name of the report specifications file for one or more reports. You may specify the full pathname for the file. The full path is not required. If you do not explicitly specify an extension for the file, the system assumes the default extension of **.rw**.

The **sreport** command requires valid values for both *filename* and *dbname*. If you do not enter these parameters, **sreport** prompts you for them.

The **-u** flag can be used by the database administrator (DBA) for a database or by the ODT-DATA system manager to temporarily assume another user's account identity.

Examples

The following **sreport** command stores report specifications in file **repspec.txt** in the Reports Catalog of a database named "mydb":

```
sreport mydb repspec.txt
```

The following **sreport** command specifies a database name of "myowndb" and uses **sreport**'s prompting facility to store the report specification located in the file **myrep.rw**:

```
sreport myowndb
```

You are prompted for the filename, as follows:

```
Report File?
```

You enter the name of a file containing report specifications:

```
myrep.rw
```

This third example uses **sreport**'s prompting facility to store the report specification located in the file **myrep.rw**, in the Reports Catalog for the database "myowndb". Enter:

```
sreport
```

At the database prompt, enter a database name:

```
Database?  myowndb
```

At the "Report File?" prompt, enter a filename:

```
Filename?  myrep
```

report

The **report** command executes a report specification.

Syntax

report [**-c***numactions*] [**-f***filename*] [**-s**] [**-u***username*]
[**-r**]|[**-m**[*mode*]]|[**-i***filename*] [**-l***mxline*] [**-q***mxquer*]
[**-w***mxwrap*] [**+t**|**-t**] [**+b**|**-b**] [**-h**] [**-5**] [**-v***pagelength*]
dbname reportname|*tablename* [({*parameter*=*value*})]

Description

The **report** command executes the report specifications that correspond to the *reportname* parameter or a default report for a table in the database.

This command produces the report. The **report** command produces the following actions. The report catalogs are checked to see if the report has been stored in the database. If found, the specifications for the report are read and checked for errors. If errors occur, the report is not run. If the specification is error-free, parameters are replaced with their specified values. Data is extracted and the query, if specified, is run, the data are sorted, if required, and the report is formatted and output. If no report with the given name is found, the name is assumed to be a table name. A default report for that table is formatted and run. If the table or report is not found, an error message results.

The Report-Writer prompts you for anything that you do not specify on the **report** command line, including the *reportname*, the *dbname*, and values for any parameters encountered in the report specifications.

If specified, the *dbname*, *reportname*, or *tablename*, and *parameter* should be placed at the end of the command line, in the order shown earlier under "Syntax."

The recommended use is to let the Report-Writer prompt you if you have parameters to enter for your report. When prompted, you may enter embedded blanks or commas as part of the parameter value.

Parameters for report

Parameter	Description
-c*numactions*	If specified, this sets the number of Report-Writer action statements to be processed within one buffer to "numactions." This minimizes real memory usage on systems where this is a concern. The default is 32,000, which is large enough to cover all known cases. If the value is set too large, only the actual number of statements is used in computing the value.
-f*filename*	Directs the formatted report to *filename* for subsequent output. If this option is not specified, the report may be written to the file specified in the **.output** statement in the report specification file or to the default output file (normally your terminal).
-s	If specified, requests that status messages, including prompts, be suppressed.
-u*username*	If specified, requests that the Report-Writer pretend you are the user with login name *username*. This can be used only by the DBA for a database, or by the ODT-DATA system manager.
-r	If specified, tells the Report-Writer that a report is being specified, rather than a table. This gives an error if no report with the given name is found. By default, the Report-Writer looks for a report of the given name, and if one is not found, and a table of the given name exists, a default report for that table is set up.

Parameter	Description
-m[*mode*]	If specified, tells the Report-Writer that a table has been specified, rather than a report. This instructs the Report-Writer to format a default report for the specified table. If the optional *mode* value of **wrap, column** or **block** is specified after the **-m** flag, that style of default report is used.
-i*filename*	If specified, tells the Report-Writer to read report specifications from a source file outside the Reports Catalog. This lets you run a report without first executing the **sreport** command. You cannot use this flag in conjunction with the **-c**, **-m**, or **-r** flags.
-l*mxline*	If specified, sets the maximum output line size to *mxline* characters. By default, the maximum output line size is 132 characters if output is to a file; otherwise, the default maximum line size is the width of the terminal. This option is needed only if reports are written that contain unusually long lines.
-q*mxquer*	If specified, sets the maximum length of the query specified in the **.query** statement, after all substitutions for runtime parameters have been made. By default, the maximum query size is 1000 characters. This option is needed only for particularly long queries.

Parameter	Description
-w*mxwrap*	If specified, sets *mxwrap* as the maximum number of lines to wrap with one of the column **C** formats, or the maximum number of lines that can be used within any block. By default, the maximum value is 100 lines. This means that a column written with a format such as "c0.20" (which writes a character string in a column 20 characters wide) contains a maximum of 100 lines, or the maximum number of **.newline** statements within an invocation of "block" mode is 100. This maximum is provided as a protection against misspecified columns, and is rarely needed.
-t|+t	If turned on (+t), causes aggregates to occur over rounded values for any floating-point column whose format has been specified in a **.format** statement as numeric **F** or template. Each value in the column is rounded to the precision given by its format. If this flag is turned off (-t), aggregates utilize the underlying values, not the rounded values. **+t** is the default.
-b|+b	If turned on (**+b**), forces formfeeds at the end of each page. If turned off (**-b**), this flag suppresses formfeeds for the end of each page. The flag overrides any **.formfeed** or **.noformfeed** statement occurring in the report specification file.
-h	If specified, a report that retrieves no rows is provided a null set of data. All header and footer sections are executed. The detail section is suppressed. This feature allows you to include the following **.if** statement in the report footer to output a positive acknowledgement that no rows were found:

Parameter	Description

```
.if count(column) = 0 .then
   .println
   "No data matched the query specifications."
.endif
```

-5

If specified, the report is made compatible with version 5.0 of INGRES. The default is that the flag is not specified. To ensure compatibility, the following assumptions are made:

+t option is the default.

Only floating-point arithmetic is used. Integer columns are converted to floating-point before use in computation.

The month part of the *current_date* function is displayed in capitals if no format is specified. Normally, the system displays the month names in lowercase letters. For example, what is now displayed as "01-feb-1985" would, with the **-5** flag set, be displayed as "01-FEB-1985."

-v*pagelength*

If specified, sets **pagelength** as the number of lines for each page of output. *pagelength* must be a positive integer. This flag overrides any **.pagelength** statement in the report specification file. The default is 61 lines per page if the report is written to a file, and 23 lines per page if written to a terminal.

dbname

The name of the ODT-DATA database containing the report data.

Parameter	Description
reportname	The name of a report that appears in a **.name** statement in a report specification that has been stored in the Reports Catalog. Do not enter a *reportname* if you specify a *tablename* in this command.
tablename	The name of a table or view in your database for which you want a default report. Do not enter a *tablename* if you have specified a *reportname* in this command.
parameter	The name of a parameter used in the report specification. This parameter may either be used in the specified query as a declared variable or simply referred to in a Report-Writer statement. Parameter or value combinations on the command line must be separated by blanks, commas, or tabs. Note that you must specify a space (or tab) before the opening parenthesis of the parameter or value list.
value	The value placed in every occurrence of the corresponding *parameter* reference in the report specifications. *value* should be surrounded by quotes (which are removed when it is processed) if you want to pass through a string or date value.

If you specify neither the **-f** option on the command line nor an **.output** file in your report specifications, the report is written to the standard output file. If this is a terminal, the default page size set for the report is 23 lines, rather than the normal 61. At the end of each page written to your terminal, the following prompt appears:

```
ENTER C, S, HELP OR RETURN:
```

You should respond:

C	Or "c" to request that printing of the report be continuous to the end of the report.
S	Or "s" to stop printing the report.
RETURN	Or ENTER to request the printing of the next page of the report.
H	Or "h"or "HELP" to print a description of these options.

If you specify the **-u** flag, the Report-Writer acts as if you were another user. It allows the DBA for a database or the ODT-DATA system manager to run cataloged or default reports that are owned by others.

The **-r** flag can be used to force the Report-Writer to only check for reports with the given name, and the **-m** flag can be used to force the Report-Writer to only check for tables with the given name. These are sometimes useful if you have reports and tables with the same name, and need to be more specific about what you want. Additionally, the **-m** flag can be used to specify the style of default report to be produced. See Chapter 2 for more information on these styles, or see *Using ODT-DATA Through Forms and Menus*.

Examples

In this example, run a report that has no parameters:

```
report mydb myreport
```

This example uses the **report** prompt facility to enter a report name and a database name, and also have the Report-Writer write the report to an alternate file:

```
report -faltout.rep
```

At the "Database" prompt, enter a database name:

```
Database name?  mydb
```

At the "Report or Table" prompt, enter a report or table name:

```
Report or Table   myrep
```

In this example, specify all parameters to a report with a query specification such as the following:

```
.query
     select *
          from emp
          where name = '$name'

report persdb namerep '(name =
          "Smith,     T.")'
```

The same report prompts you with:

```
Enter 'name'?
```

You would respond:

```
Smith, T.
```

This also requests a default line size of 200 characters.

```
report persdb namerep -l200
```

In this example, report with parameters to a query and also for printing within the body of the report. The query might be:

```
.query
     select transact, name
          from trans
          where transact > $minval
          and  transact < $maxval
```

A print statement in the report might be:

```
.print $date(c20)   .nl
```

Note that this prompts for some things (that is, the values of report name and "maxval".

```
report mydb '(minval=+123.45
     date=06/20/81)'
```

In this example, write out a default report for MYTAB. Also, write out a default report for YOURTAB, which forces the block style of format for the default report.

```
report mydb mytab
report -mblock mydb yourtab
```

copyrep

Allows you to copy report specifications from one database to another.

Syntax

copyrep [**-s**] [**-u***username*] [**-f**] [**-c***numactions*]
dbname txtfile report {*report*}

Description

The **copyrep** command copies a report specification, or set of report specifications, from a database to a text file. You can then use the report specification with a different database in the **sreport** command.

This command works much like the **Archive** operation accessed through the Reports Catalog form of RBF. However, reports created with RBF may also be copied using the **copyrep** command, retaining the knowledge that they are RBF reports.

As a useful side effect, the command also provides a method for externally storing the definitions of reports in simple files in much the same way as the **copydb** command works. The command allows you to copy any number of reports to a single text file. The reports are named within the file, but contain no owner tied to a report. Therefore, you may copy out a (set of) report(s) owned by one user, and then copy them back in as another user, effectively changing their owner.

If you omit any of the parameters not preceded by a dash, the **copyrep** command prompts you for the missing values. If no reports are specified, you are prompted for reports to be entered one per line.

You end the list with a Ctrl D.

Parameters for copyrep

Parameter	Description
-s	If specified, means to suppress status messages.
-c*numactions*	If specified, this sets the number of Report-Writer action statements to be processed within one buffer to "numactions." This can be useful to minimize real memory usage on systems where this is a concern. Default value is 32,000, which is large enough to cover all known cases. If the value is set too large, only the actual number of statements is used in computing the value.
-u*username*	Uses reports owned by the specified user. This command can only be used by the DBA for a database, or an ODT-DATA superuser.
-f	If specified, writes the reports out in the same format as is done with the **Archive** operation accessed through the Reports Catalog form of RBF. For reports created with RBF, this will strips out many of the statements.
dbname	The name of the database.
txtfile	The name of a text file in which to write the report definitions.
report	The name of one or more reports that are to be written to the text file.

The file created by this command is almost the same as the file created through the use of the **Archive** operation accessed in the Reports Catalog form of RBF. (See *Using ODT-DATA Through Forms and Menus* for more information.) For reports originally created outside of RBF and entered by using the **sreport** command, the output to the file created by the **copyrep** is identical (except that comments are stripped out). However, for reports created by RBF, all information pertaining to RBF is retained in the report output, thus eliminating the restriction

in **Archive** that the reports cannot be copied directly back into a database as RBF reports. The **-f** flag can be used to mimic the **Archive** method, which strips many of the RBF statements out of the report, making it easier to edit. However, you should be warned not to edit the RBF reports created by the **copyrep** command before copying them back into a database, as you could easily make a report unusable in additional RBF sessions.

Examples

Suppose you want to move a report called "emphours" from the "emp" database into the "newemp" database. The following statement performs the first part of the task, copying the report into a text file called **emphours.txt**:

```
copyrep emp emphours.txt emphours
```

To copy report files created through the copyrep command back into a database, perhaps under a different owner, you can use the **sreport** command. To continue the example above, the report in the text file **emphours.txt** can be copied into the database "newemp" simply by executing the following command:

```
sreport newdb emphours.txt
```

Appendix A
Report Examples

This chapter contains five sample reports, including both input and output. Two of the reports have two alternative sets of specifications. The reports are as follows:

- The POPULATION report demonstrates a common type of report with subtotalling. POP2 shows an alternative set of formatting statements for producing the same output.
- The ACCOUNT report gives a complex report that might be used in accounting applications.
- The DICTIONARY report shows the use of character printing options within the Report-Writer. DICT2 shows an alternative set of formatting statements for producing the same output.
- The LABEL report demonstrates the formatting of mailing labels, three across on a page, from a list of names and addresses. It features the use of the **.if** statement (see Chapter 11).
- The BOOKS report illustrates the use of joining tables for producing a report.

Each example is organized as a set of explanatory texts, followed by a listing of the report formatting statements, followed by a listing of the report itself. For the sake of clarity, the formatting statements are indicated in the examples by uppercase letters, although they can actually be specified in either upper- or lowercase letters.

Population Example

The POPULATION example demonstrates the use of the Report-Writer in formatting a report of census data, by region and state, for the United States. The base tables for this report are the following:

- "Region" contains region names associated with region abbreviations.
- "State" contains state names, as well as state abbreviations, and associated region abbreviations.
- "Pop" contains population data for each state for different census years.

The report formatter statements are discussed below. Notice that the output for this report is followed by a description and listing of a slightly different set of statements that can be used to produce the same output.

- The **.query** statement shows the database query needed to set up the data in the form required to write the report. Essentially, the query sets up a table with one row for each state, including the columns "region" (name of region), "state" (name of state), "tot" (the total population of the state), "totwhite," "totblack," and "totother" (populations of three racial groups).

- The query contains a parameter, "$Year," which is used in the **where** clause to select data for only one census year. In the example shown, you can select the data for 1970 by running the report with the command:

```
report rwsqldb pop '(year=1970)'
```

You can also run the report with the following:

```
report
```

In this case the Report-Writer prompts you for the report name, database name, and value for "year."

- The **.sort** statement specifies a sorting of the data by "region," and within region, by "state." This also defines potential break actions for changes in value of "region" and "state."

- The **.format** statement sets up a default format for a set of columns in the report. These are used not only for the printing of the actual data but also for the printing of subtotals based on that data. Note that the four numeric columns ("tot," "totwhite," "totblack," and "totother") are given the same format specification. Actually, the **.format** statement is not strictly needed, but provides a convenient way to specify the same format for a number of columns.

- The **.header report** statement is followed by a set of formatting statements that are run at the start of the report. There is nothing particularly elegant about the formatting statements, which write out the centered title seen at the top of the report. Note the value of parameter "year" is preceded by a dollar sign to indicate that it is a parameter. Also, underlined column headings are printed in this section. The locations of the headings are based on the positions of the column names given as parameters to the **.rt** (right justify) statements. These positions are determined by the location at which the associated column is printed in the **.detail** statements.

- The **.header region** statement is followed by a set of formatting statements that are run at the start of each region. The **.need** statement insures that at least four lines are available on a page before printing the heading for "region." This assures that the heading and the detail lines for at least two states are printed on a page.

- The **.detail** statement is followed by formatting statements that are processed for every row created by the query. In this case, rows are created for each state, and the statements specify printing of the actual population data. By analyzing these statements, the Report-Writer determines the "positions" of the columns used throughout the report in the **.rt** statements.

- The **.footer** statement is followed by formatting statements, which are processed after the last state in each region is read and the requisite **.detail** formatting statements are processed. A **.need** statement is included simply to assure that the two lines in the footer are printed on the same page. A region heading is printed and followed on the same line with the values of some subtotals for the region. The formats used in printing the subtotals are those specified in the **.format** statement at the start of the report.

- The statements following the **.footer** report statement are almost identical to those following the **.footer region** statement, except for the heading and length of the dashed line separators. The values of the subtotals, however, are different because of the different context.

- The statements following the **.header page** statement specify the title seen at the top of the second page of the report, as well as a respecification of the column headings. (A macro facility is planned to simplify this respecification in the future.)

- The **footer page** statement starts the block of statements printed at the bottom of each page, including the current page number. Because the **.right** statement has no parameters, the text is justified to the right margin (determined as the rightmost position printed in the formatting statements in the report).

```
        /*      POPULATION   -   Population Report              */

.NAME pop
.QUERY
        select  region.region,  state.state,
                tot = pop.totwhite + pop.totblack + pop.totother,
                        pop.totwhite,  pop.totblack,  pop.totothe
                        from region,  state,  pop
                        where state.statabbrev = pop.statabbrev
                        and state.regabbrev = region.regabbrev
                        and pop.year = $Year

.SORT region,  state

.FORMAT tot,  totwhite,  totblack,  totother ("      z,zzz,zzz,zzz")

.HEADER report
        .NEWLINE 3
        .UL .CE .PR "Population of the United States, by Race" .NOU .NEWLINE
                .CE .PR "Data for the Year - ", $Year(c4)  .NL2
        .U .RT tot .PR "Total Pop"   .RT totwhite .PR "White Pop"
                .RT totblack .PR "Black Pop"   .RT totother .P "Other Pop"
        .NOU .NL2

.HEADER region
        .NEED 4
        .PR "Region: ", region .NL

.DETAIL
        .NEED 2   .T5
        .PR state(c20)  .T+11  .PR tot,  totwhite,  totblack,  totother .NL
```

```
.FOOTER region
        .NEED 2  .RT tot .PR "----------"  .RT totwhite .P "----------"
               .RT totblack .P "----------" .RT totother .P "----------" .NL
        .PR "Totals: ", region (c0) .T tot
        .PR sum(tot), sum(totwhite), sum(totblack), sum(totother) .NL2
```

```
.FOOTER report
        .NEED 2 .RT tot .PR "-------------" .RT totwhite .P "-------------"
               .RT totblack .P "-------------" .RT totother .P "-------------" .NL
        .PR "USA Totals" .T tot
        .PR sum(tot), sum(totwhite), sum(totblack), sum(totother) .NL
```

```
.HEADER page
        .NL3 .PR "Population by State and Region: ", $Year .NL2
        .U .RT tot .P "Total Pop .RT totwhite .P "White Pop"
               .RT totblack .P "Black Pop" .RT totother .P "Other Pop"
        .NOU .NL2
```

```
.FOOTER page
        .NL
        .PR "Source: US Department of the Interior, Bureau of the Census."
        .RIGHT .PR "Page", page_number("zN") .NL4
```

Report output:

```
                    Population of the United States, by Race
                    ---------- -- --- ------ ------- -- ----
                           Data for the Year - 1970

                                 Total Pop       White Pop       Black Pop       Other Pop
                                 ----- ---       ----- ---       ----- ---       ----- ---
Region: East North Central
     Illinois                   11,113,976       9,600,381       1,425,674          87,921
     Indiana                     5,193,669       4,820,324         357,464          15,881
     Michigan                    8,875,083       7,833,474         991,066          50,543
     Ohio                       10,652,017       9,646,997         970,477          34,543
     Wisconsin                   4,417,731       4,258,959         128,224          30,548
                                 ----------      ----------      ----------      ----------
Totals: East North Central      40,252,476      36,160,135       3,872,905         219,436
```

Population Example

Region: East South Central				
Alabama	3,444,165	2,528,983	908,247	6,935
Kentucky	3,218,706	2,971,232	241,292	6,182
Mississippi	2,216,912	1,393,283	815,770	7,859
Tennessee	3,923,687	3,283,432	631,696	8,559
Totals: East South Central	12,803,470	10,176,930	2,597,005	29,535
Region: Middle Atlantic				
New Jersey	7,168,164	6,349,908	770,292	47,964
New York	18,190,740	15,790,307	2,166,933	233,500
Pennsylvania	11,793,909	10,737,732	1,016,514	39,663
Totals: Middle Atlantic	37,152,813	32,877,947	3,953,739	321,127
Region: Mountain				
Arizona	1,770,900	1,604,948	53,344	112,608
Colorado	2,207,259	2,112,352	66,411	28,496
Idaho	712,567	698,802	2,130	11,635
Montana	694,409	663,043	1,995	29,371
Nevada	488,738	448,177	27,762	12,799
New Mexico	1,016,000	915,815	19,555	80,630
Utah	1,059,273	1,031,926	6,617	20,730
Wyoming	332,416	323,024	2,568	6,824
Totals: Mountain	6,281,562	7,798,087	180,382	303,093
Region: New England				
Connecticut	3,031,709	2,835,458	181,177	15,074
Maine	992,048	985,276	2,800	3,972
Massachusetts	5,689,170	5,477,624	175,817	35,729
New Hampshire	737,681	733,106	2,505	2,070
Rhode Island	946,725	914,757	25,338	6,630
Vermont	444,330	442,553	761	1,016
Totals: New England	11,841,663	11,388,774	388,398	64,491

Source: US Department of the interior, Bureau of the Census. Page 1

Population by State and Region: 1970

	Total Pop	White Pop	Black Pop	Other Pop
Region: Pacific				
Alaska	300,382	236,767	8,911	54,704
California	19,953,134	17,761,032	1,400,143	791,959
Hawaii	768,561	298,160	7,573	462,828
Oregon	2,091,385	2,032,079	26,308	32,998
Washington	3,409,169	3,251,055	71,308	86,806
Totals: Pacific	26,522,631	23,579,093	1,514,243	1,429,295

Region: South Atlantic				
Delaware	548,104	466,459	78,276	3,369
District of Columbia	756,510	209,272	537,712	9,526
Florida	6,789,443	5,711,411	1,049,578	28,454
Georgia	4,589,575	3,387,516	1,190,779	11,280
Maryland	3,922,399	3,193,021	701,341	28,037
North Carolina	5,082,059	3,891,510	1,137,664	52,885
South Carolina	2,590,516	1,794,430	789,041	7,045
Virginia	4,648,494	3,757,478	865,388	25,628
West Virginia	1,744,237	1,666,870	73,931	3,436
Totals: South Atlantic	30,671,337	24,077,967	6,423,710	169,660
Region: West North Central				
Iowa	2,824,376	2,782,762	32,596	9,018
Kansas	2,246,578	2,122,068	106,977	17,533
Minnesota	3,804,971	3,736,038	34,868	34,065
Missouri	4,676,501	4,177,495	480,172	18,834
Nebraska	1,483,493	1,4323,867	39,911	10,715
North Dakota	617,761	599,485	2,494	15,782
South Dakota	665,507	630,333	1,627	33,547
Totals: West North Central	16,319,187	15,481,048	698,645	139,491
Region: West South Central				
Arkansas	1,923,295	1,561,108	357,225	4,962
Louisiana	3,641,306	2,539,547	1,088,734	13,025
Oklahoma	2,559,229	2,275,104	177,910	106,218
Texas	11,196,730	9,696,569	1,419,677	80,484
Totals: West South Central	19,320,560	16,072,328	3,043,543	204,689
USA Totals	203,165,699	177,612,309	22,672,570	2,880,820

Source: US Department of the Interior, Bureau of the Census. Page 2

Pop2 Example

The POP2 example shows an alternative set of formatting statements for producing the same output as POPULATION. This report makes use of the **.block** and **.endblock** statements, as well as the **.within** and **.endwithin** statements in producing the report. These statements are useful for reports that contain several columns for which the same set of statements is repeated, as is the case with the "totpop," "totwhite," "totblack" and "totother" columns in POPULATION.

All of the statements in POP2 are identical to the statements in POPULATION with the exception of those in the **.foot** region and **.foot** report sections. In these sections, instead of spelling out the format of the subtotals line by line, the block and column formatting statements can be used to duplicate the same set of statements for each of several columns.

In detail, the statements are:

- The **.block** statement sets the Report-Writer into block mode, which allows you to write a two-dimensional block of text, in which you can write text on several lines, return to the first line in the block, and then write more text on the first lines in the block.

- The **.within** statement sets the Report-Writer into column-formatting mode. Because the statement is followed by four column names, "tot," "totwhite," "totblack," and "totother," all statements between the **.within** and its corresponding **.endwithin** statement are executed four times, using the margins for each of the columns in turn.

- The string "----------" is printed, right justified, within each of the four columns in the first line of the block. Because this is a **.printline** statement, the current output line is moved down one line in the block after the string is printed. On the second line of the block, right justified within each of the columns, a sum is printed. Because this sum uses the special name **w_column**, a separate sum is calculated and printed for each of the columns in turn.

- The **.end within** statement ends the set of formatting statements to be done within each column. Note that because the block mode of the Report-Writer is in effect, a **.top** statement is automatically executed immediately before the **.end within** statement. This assures that the statements for each of the columns prints across the page, rather than stairstepping down the page.

- Immediately following the **.end within** statement is the **.top** statement, which moves the current output line back to the first line in the current block (which contains all of the "----------" strings). The **.newline** statement moves the current position to the second line in the block (because block mode is still in effect). There, the "Totals: region" message is printed. Block mode is then left, by specifying the **.endblock** statement, which causes the block, consisting of two lines, to be printed. The last **.newline** statement merely inserts a blank line.

- The statements within the **.foot** report are identical, except for the message "USA Totals." Because of the context, the "sum(w_column)" refers to totals over the report, rather than the region.

The statements in this example are not quite as intuitive as those in the first report, but they show an important capability in formatting reports with column-oriented statements.

```
    /*    POP2  -  Population Report using .WITHIN     */

.NAME pop2
.QUERY
        select region.region, state.state,
               tot = pop.totwhite + pop.totblack + pop.totother,
               pop.totwhite, pop.totblack, pop.totother
                       from region, state, pop
                       where state.statabbrev = pop.statabbrev
                       and state.regabbrev = region.regabbrev
                       and pop.year = $Year

.SORT region, state

.FORMAT tot, totwhite, totblack, totother ("   z,zzz,zzz,zzz")

.FORMFEEDS

.HEADER report
        .NEWLINE 3
        .UL .CE .PR "Population of the United States, by Race" .NOU .NEWLINE
               .CE .PR "Data for the Year - ", $Year(c4) .NL2
        .U .RT tot .PR "Total Pop"  .RT totwhite .PR "White Pop"
               .RT totblack .PR "Black Pop"  .RT totother .P "Other Pop"
        .NOU .NL2

.HEADER region
        .NEED 4
        .PR "Region: ", region .NL

.DETAIL
        .NEED 2  .T5
        .PR state(c20) .T+11 .PR tot, totwhite, totblack, totother .NL

.FOOTER region
        .NEED2
        .BLOCK .WITHIN tot, totwhite, totblack, totother
                       .RT .PRINTLINE "----------"
                       .RT .PRINTLN sum(w_column)
               .ENDWITHIN
               .TOP .NEWLINE .PR "Totals: ", region(c0)
        .ENDBLOCK .NEWLINE
```

```
.FOOTER report
        .NEED2
        .BLOCK .WITHIN tot, totwhite, totblack, totother
                        .RT .PRINTLINE "-------------"
                        .RT .PRINTLN sum(w_column)
                .END WITHIN
                .TOP .NEWLINE .PR "USA Totals"
        .ENDBLOCK .NEWLINE

.HEADER page
        .NL3 .PR "Population by State and Region: ", $Year .NL2
        .U .RT tot .P "Total Pop .RT totwhite .P "White Pop"
                .RT totblack .P "Black Pop" .RT totother .P "Other Pop"
        .NOU .NL2

        .FOOTER page
                .NL
                .PR "Source: US Department of the Interior, Bureau of the Census."
                .RIGHT .PR "Page", page_number("zN") .NL4
```

Account Example

The ACCOUNT example shows a fairly complex report that could be written from some accounting data. For each account, the report prints the name and address of a customer, followed by a listing of transactions in an account. Deposits are listed in one column, withdrawals in another, and a running balance is listed in a third. The base tables used are the following:

- The "customer" table contains the name and address of a customer.
- The "account" table associates an account number with a customer name and address (because a customer may have more than one account). It also contains the balance of an account as of an arbitrary date. In actual accounting applications, this balance must be updated outside of the Report-Writer.
- The "transact " table contains a description of all transactions for an account. This contains columns "transnum" (the transaction number), "acctnum" (the account number), "date" (the date of the transaction), "amount" (the dollar amount of transaction), and "type" (the type of transaction: 0 for deposits, 1 for withdrawals).

The report-formatting statements are discussed below:

- After the report is named, the query used to provide data for the report is shown. This essentially retrieves the "transact" table, with data from the other tables joined in. The calculation of the columns "amt," "withdrawal," and "deposit" is also shown. "Withdrawal" is set to "amount" if type is 1, and set to zero if not. "Deposit" is set to "amount" if type equals 0, and set to zero otherwise. "Amt" is calculated as a signed value of "amount," which is negative for withdrawals and positive for deposits. "Amt" is used in calculating the running balance.

- The order of the data is described in the **.sort** statement. In the example output shown, only one account is shown for a name, though this type of sort order does not limit the report to that type of content.

- The **.formfeeds** statement tells the Report-Writer to insert formfeed characters at the start of the report and at the end of each page of the report. Because no **.pagelength** statement is specified, a default page size of 61 lines is assumed.

- The **.format** statements provide default formats for some of the output columns. Note the use of "-" in the format for "acctnum" to force hyphens in specific places in the output.

- The **.head name** statement begins the set of formatting statements done at the start of each new name. The **.newpage** statement tells the Report-Writer to skip to the top of a new page and to set the page number to the value 1 at the start of each new name. Next, the address is printed and lines are skipped.

- The **.head acctnum** block prints the opening balance, column headings, and sets a temporary format for "acctnum" (so that it is printed for the first transaction only). Notice that the positions associated with the columns are determined from a scan of the formatting statements in the **.detail** section. Even a position for the "amt" column is determined, even though it is somewhat hidden in the cumulative sum function.

- The **.head tdate** block simply sets a temporary format for "date," so that the date is printed only the first time it is encountered.

- The **.detail** block simply prints out the lines in the report. It also determines the default margins and column positions from an analysis of these statements. Notice that the formats for "date" and "acctnum," which specify nonprinting formats, may be overridden by the **.tformat** statements specified in the header text for "date" and "acctnum."

The "cum(acctnum) sum(amt,balance)" aggregate specifies the calculation and printing of the running balance. The first part, "cum(acctnum)" specifies that the running balance is a cumulative aggregate, which is initialized at the most recent break in "acctnum." The rest, "sum(amt,balance)," specifies that the cumulative aggregate is a sum of "amt," and that the cumulative is to be initialized to the value of "balance" when the report starts (at the most recent break in "acctnum"). The format to be used is specified as the default for "amt" because the aggregate specification is not followed by a parenthesized format.

- The **.foot accnum** block prints out summations of the "withdrawal" and "deposit" columns and the closing balance of the account. This figure, the "sum(amt,balance)" aggregate, is calculated as the sum of "amt" for a specific "acctnum" (because of the context), and is then initialized to the value of "balance" at the start of "acctnum." Remember that the figure is negative for withdrawals and positive for deposits. Again, because the aggregate specification is not followed by a parenthesized format specification, the **.format** statement for "amt" at the beginning of the report is used as the default format for the aggregate.

- The **.foot name** block specifies the printing of an ending statement.

- The **.head page** block describes the heading shown at the top of each page. Note that the **.newpage** statement in the **.head name** statements forces the printing of the page header on the first page (which normally does not happen).

- The **.foot page** block simply specifies a few lines to force at the end of each page.

```
/*        ACCOUNT   -   example of bank statement report.        */

.NAME  account
.QUERY
        select c.name, c.address, c.city, c.state, c.zip,
               a.acctnum, a.balance, t.transnum, tdate = t.date,
               withdrawal = t.amount * t.type,
               deposit = t.amount * (1 - t.type),
               amt = (t.amount * (1 - t.type)) - (t.amount * t.type)
                       from transact t, account a, customer c
                       where a.acctnum = t.acctnum and c.name = a.name

.SORT name, acctnum, tdate

.FORMAT acctnum(" nn\e-nnnnnn\e-n "), tdate(d"01/02/03"),
        withdrawal, deposit, amt, balance ("   $$$,$$$,$$$.zz")

.HEAD name
        .NEWPAGE 1 .NL 3
        .PR name .NL
        .PR address .NL
        .PR city(c0), " ", state(c0), " ", zip("nnnnn") .NL 4

.FOOT name
        .NL3 .PR "End of accounts for: ", name .NL

.HEAD acctnum
        .NL 3
        .P "Account: ", acctnum .RT amt .P "Opening balance:", balance .NL2
        .UL .CE acctnum .P "Account" .CE tdate .P "Date" .CE transnum .P "Transaction"
          .RT deposit .P "Deposit" .RT withdrawal .P "Withdrawal" .RT amt .P "Balance" .NL
        .NOU
        .TFORMAT acctnum(" nn\e-nnnnnn\e-n  ")

.FOOT acctnum
        .NL 2
        .PR "Account", acctnum, "totals." .T deposit .P sum(deposit) .T withdrawal
              .P sum(withdrawal)
        .NL 2 .RT amt .P "Closing balance:", sum(amt, balance) .NL

.HEAD tdate
        .TFORMAT tdate(d"01/02/03       ")
```

Account Example

```
.DETAIL
        .PR acctnum(b16), tdate(b16), .T+8 .P transnum("nnnn"), deposit,
                withdrawal .T+5 .P cum(acctnum) sum(amt, balance) .NL

.HEAD page
        .NL 2
        .PR "Customer: ", name .CE .P "Date: ", current_date(d"February 3, 1901"),
                .RT .PR "Page ", page_number .NL4

.FOOT page
        .NL 3
```

```
Customer: Big Daddy                    Date: 24-JUN-1982                          Page      1

Big Daddy
1 Bestview Lane
Topofthehill NJ 05436

 Account:   74-902543-6                                    Opening balance:      $234,657.00

    Account       Date          Transaction      Deposit      Withdrawal             Balance
    -------       ----          -----------      -------      ----------             -------

  74-902543-6   81/07/01            0101       $100,000.00                        $334,657.00
                                    0102                     $50,500.00           $284,157.00
                                    0103                        $24.56            $284,132.44
                                    0104                     $10,100.00           $274,032.44
                81/07/15            0105        $50,000.00                        $324,032.44
                81/07/17            0106                     $10,143.54           $313,888.90
                                    0107                        $243.56           $313,645.34
                81/07/22            0108                        $100.00           $313,545.34
                81/07/23            0109                     $25.000.00           $288,545.34
                                    0110       $100,000.00                        $388,545.34

Account  74-902543-6 totals.                   $250,000.00    $96,111.66

                                                           Closing balance:      $388,545.34

End of accounts for: Big Daddy
```

Customer: Fast Sally Date: 24-JUN-1982 Page 1

Fast Sally
1234 71st St
Big City NY 01234

Account: 48-821908-2 Opening balance: $1,245.00

Account	Date	Transaction	Deposit	Withdrawal	Balance
48-821908-2	81/05/25	0101		$200.00	$1,045.00
	81/07/03	0102	$250.00		$1,295.00
	81/07/05	0103		$320.34	$974.66
		0104	$65.23		$1,039.89
	81/07/08	0105		$100.00	$939.89
	81/07/10	0106		$56.32	$883.57
	81/07/16	0107		$24.71	$858.86
	81/07/20	0108		$120.00	$738.86
	81/07/25	0109		$31.16	$707.70
Account 48-821908-2 totals.			$315.23	$852.53	
				Closing balance:	$707.70

End of accounts for: Fast Sally

Dictionary Example

The DICTIONARY example shows an example of a report that lists a glossary of ODT-DATA terms, with a listing of related keywords. This demonstrates the use of some of the word-processing functions available in the Report-Writer. The base tables used are the following:

- The "ddef" table contains names of terms and definitions of those terms.
- The "dref" table contains a list of terms and their related keywords.

Additional details on these table layouts are given in the following table.

ddef Table

Column information:

Column Name	Type	Length	Key	Sequence
word	c	20		
definition	c	250		

Word	Definition
aggregate function	An aggregate operator that first groups rows on the basis of the value of a (list of) column (called the "by-list"), before computing the aggregate for each value of the "by-list."
aggregate operator	An aggregate operator is a computation performed on a column across all rows in a table. Common aggregate operators are SUM, COUNT, and AVG. Aggregate operators can have qualifications to limit the number of rows used in the calculation.
attribute	Another term for a "column" in a table.
buffer	Another term for the ODT-DATA "workspace."
word	Definition

column	All data in ODT-DATA is saved in the form of tables made up of rows and columns. In traditional database terminology, a "column" is a "field" in a record.
comparison operator	A symbol that specifies the kind of comparison to make in a qualification, such as ">" (for greater than), or "=" (for equality check).
compressed	Any of the ODT-DATA internal storage structures can be compressed. Compression reduces the storage required for a table, by deleting all trailing blanks in character columns.

dref Table

Column information:

Column Name	Type	Length	Key Sequence
word	c	20	
ref	c	20	

Word	Ref
aggregate function	aggregate operator
aggregate function	aggregation
aggregate function	by list
aggregate function	computation
aggregate operator	aggregate function

Column information:

Column Name	Type	Length	Key Sequence
word	c	20	
ref	c	20	

Word	Ref
aggregate operator	aggregation
aggregate operator	computation
attribute	column
buffer	workspace
column	attribute
column	domain
column	field
comparison operator	qualification
comparison operator	restriction
compressed	character strings
compressed	compression
compressed	storage structures

The report-formatting statements are discussed below:

- The query used to create the data for the report simply joins words and definitions with a list of the related keywords. Therefore, the data returned to the report contains one row per related keyword. The **.detail** statements deals with the keywords, whereas the **.heading word** deals with the definition.

- The data are sorted by word, and within word, by related keyword.

- The left and right margins are set to specific values because the default margins calculated for the report do not reflect the required margins of the report.

- The report header simply puts out a page header.

- The header for "word" prints out the underlined word and the newspaper style printing of the definition. The "cj0.50" format specifies a column format 50 spaces wide, with right justification, with printing occurring until the end of the string. The **.t80** statement then moves to position 80 (5 spaces to the right of the edge of the definition), and sets the left margin of the report to that position. This causes all printing to wrap around between the left margin (80) and the right margin (100). Notice that no **.newline** statement is given, so that the next printing occurs at column 80 of the top line of the definition.

- The **.detail** statements simply print out the next related keyword for "word," until the next word is found. Because the format specified for "ref" is "c20," it exactly fits within the temporary margins, and wraparound causes each keyword to be placed on a separate line. Remember that the **.lm0** statement in the header text for "word" resets the left margin for printing a new word and definition.

- The footer for "word" simply finishes off the text for one "word" by printing out all the lines in the definition and related keyword list, and an extra line as well.

- The page header simply prints out a title, page number, and column headings.

Dictionary Example

```
    /*        DICTIONARY - text example        */

.NAME dict
.QUERY
        select ddef.word, ddef.definition, dref.ref
               from ddef, dref
               where ddef.word = dref.word

.SORT word, ref
.LM 0
.RM 100

.HEAD report
        .NEWPAGE 1
.HEAD word
        .NE 3 .LM 0
        .UL .PR word(c25) .NOU
               .P definition(cj0.50) .T80 .LM80
               .DETAIL
        .P ref(c20)
.FOOT word
        .NL2
.HEAD page
        .NL2
        .P "Dictionary of ODT-DATA Terms"
        .RT .P "Page", page_number .NL2
        .UL .P "Word" .T definition .P "Definition"
        .T80 .P "Related Term" .NOU .NL2
.FOOT page
        .NL 3
```

Dictionary of ODT-DATA Terms

Word	Definition	Related Term
aggregate function	An aggregate operator that first groups rows on the basis of the value of a (list of) column (called the "by-list"), before computing the aggregate for each value of the "by-list".	aggregate operator aggregation by list computation
aggregate operator	An aggregate operator is a computation performed on a column across all rows in a table. Common aggregate operators are SUM, COUNT, and AVG. Aggregate operators can have qualifications to limit the number of rows used in the calculation.	aggregate function aggregation computation
attribute	Another term for "column" in a table.	column
buffer	Another term for the ODT-DATA "workspace."	workspace
column	All data in ODT-DATA is saved in the form of tables made up of rows and columns. In traditional database terminology, a "column" is a "field" in a record.	attribute domain field
comparison operator	A symbol that specifies the kind of comparison to make in a qualification, such as ">" (for greater than), or "=" (for equality check).	qualification restriction
compressed	Any of the ODT-DATA internal storage structures can be compressed. Compression reduces the storage required for a table, by deleting all trailing blanks in character columns.	character strings compression storage structure

Dict2 Example

The DICTIONARY report uses some margin tricks to accomplish what can perhaps more easily be accomplished with the block mode of the Report-Writer. Instead of letting the margins and wraparound accomplish the task of moving down the page, within block mode, you can use the more natural **.newline** statement to do this. The DICT2 report is the same as the DICTIONARY report, except for differences in the **.head** and **.foot** for "word," and a slight change in the **.detail** section.

The changes in statements are:

- In the header for "word," the Report-Writer is first set into block mode. This allows you to move down the page in a more orderly fashion than would otherwise be possible. The underlined word is printed on the first line of the block. The newspaper style printing of the definition causes some number of lines within the block to be written, depending on the length of the definition. However, when it has finished printing, the current output line is the top line in the block. You are now ready to print the detail lines, which contain the keywords for a term.

- Within the detail section of the report, you tab to column 80, and print the next value of "ref." The **.newline** statement moves the current output line down one line in preparation for the next value of "ref." Because the Report-Writer is in block mode, all text since the header for "word" is saved until the **.endblock** statement is encountered.

- In the footer section for "word," the **.end block** statement is specified, which prints out the current block containing the word, its definition, and a list of related keywords. A **.newline** statement is given to add another blank line.

The DICT2 report accomplishes the same output as the DICTIONARY report, but in a somewhat more natural fashion.

```
    /*        DICT2 - text example, using .BLOCK        */

.NAME dict2
.QUERY
        select ddef.word, ddef.definition, dref.ref
               from ddef, dref
where ddef.word = dref.word

.SORT word, ref
.LM 0
.RM 100

.HEAD report
        .NEWPAGE 1
.HEAD word
        .NEED 3
        .BLOCK
        .UL .PR word(c25) .NOU
               .PR definition(cj0.50)
.DETAIL
        .T80 .PR ref(c20) .NL
.FOOT word
        .END BLOCK
        .NL
.HEAD page
        .NL2
        .P "Dictionary of ODT-DATA Terms"
        .RT .P "Page", page_number .NL2
        .UL .P "Word" .T definition .P "Definition"
        .T80 .P "Related Term" .NOU .NL2
.FOOT page
        .NL 3
```

Label Example

The LABEL example shows a report that prints mailing labels three across the page. The base table "subscriber" is a mailing list containing the name, post office box, address, city, state, and zip code for each label. If there is no post office box for the label, the field is left blank.

The report-formatting statements are discussed below:

- The data are sorted by zip code.
- The labels are assumed to be 8 lines long and 30 columns wide. Therefore, the page length is set to 8 and the right margin is set to 90 (3 * 30), as the labels are 3 across.
- The report first begins a block so that the labels may be printed across the page.
- A label is created by printing all fields of the table across 4 lines. If the field for the post office box is blank, the corresponding line is not printed.
- The left margin for the next label is moved one label's width (30) to the right of the previous left margin if doing so does not cause the label to move beyond the right margin of 90 (that is, only 1 or 2 labels have so far been formatted for the line). When no more room exists on the line, the block of three labels ends, **.newpage** moves the report to the top of the next block of labels, the left margin is reset to 0, and a new block of labels begins. When the report finishes, the last block is ended because there may be less than 3 labels left in the block buffer.

```
/*
**     LABEL. Write out three across mailing labels
**          with suppression of blank PO boxes.  This
**          assumes printing to mailing labels that
**          are 8 lines tall and 30 columns wide
*/

.NAME label
.QUERY
         select name, po_box, address, city, state, zip
         from subscriber

.SORT zip

.HEAD report
         .PL 8
         .LEFTMARGIN 0
         .RIGHTMARGIN 90
         .BLOCK

.DETAIL
         .TOP .LINESTART
         .PRINTLINE name
         .IF po_box != " " .THEN
                .PRINTLINE "PO Box ", po_box
         .ENDIF
         .PRINTLINE address(cf0.30)
         .PRINTLINE city(c0), ", ", state(c0), "   ", zip

    /* Now move the mailing label over if it fits */

    .IF left_margin + 30 < right_margin .THEN
                .LEFTMARGIN +30
         .ELSE
                .ENDBLOCK
                .NEWPAGE
                .LEFTMARGIN 0
                .BLOCK

  .ENDIF

.FOOTER report
         .ENDBLOCK
         .NEWPAGE
```

Betty Clark
2556 Carey Rd.
Boston, MA 01002

Pat McTigue
Route 146
Trumbell, CT 04239

Ming Ho
1020 The Parkway
Mamaroneck, NY 10012

T. Shigio
PO Box 1234
201 Emperor Lane
Rye, NY 10101

Marvin Blumbert
17 Saville Row
Carmel, CA 93001

Carlos Ramos
2459 39th Ave
San Francisco, CA 94121

Anastassios Vasos
722 Fourth St.
Gualala, CA 95035

Mario Verducci
PO Box X-207
General Delivery
Middletown, WA 98112

Joining Tables for a Report

The following example demonstrates the technique of joining tables for a report. Suppose you want to assemble a report from a database of the books in your personal library. You decide upon a report design to present title, author, and subject information like this:

```
TITLE OF BOOK
=============
        Author1          Subject1
        Author2          Subject2
                         Subject3
```

In your database, you have designated three separate tables to hold this information. If you do not recall the tablenames you assigned, take a moment to retrieve the table information. In this case, you use one table for titles ("title"), one table for authors ("name"), and one table for subject information ("subject").

Here is the "title" table:

```
Book Table
----------
Columns:        id      integer
                title   varchar(20)
Data:
      |id            |title                                    |
      |-----------------------------------------------------|
      |          1001|The C Programming Language              |
      |-----------------------------------------------------|
```

Here is the "name" table:

```
Author Table
------------
Columns:        id      integer
                name    varchar(15)
Data:
        |id           |name           |
        |-----------------------------|
        |         1001|Ritchie        |
        |         1001|Kernighan      |
        |-----------------------------|
```

And here is the "subject" table:

```
Subject Table
-------------
Columns:        id      integer
                subject varchar(15)
Data:
        |id           |subject        |
        |-----------------------------|
        |         1001|C              |
        |         1001|programming    |
        |         1001|language       |
        |-----------------------------|
```

Now you must combine these tables, establishing a Master or Detail relationship between them. Taking "title" as your master table, you must join to it each detail table ("name" and "subject") without joining the detail tables to each other. You must also include a code in the new joined table to tell Report-Writer when to make its breaks.

Here is one method, using a UNION:

```
select title.id, title, name, '', 1
      from title, author
      where title.id = author.id
union
select title.id, '', subject, 2
      from title, subject
      where title.id = subject.id;
```

Without the UNION, you would use these queries to join the tables:

```
create table tempreport (
      id integer,
      title varchar(20),
      name varchar(15) not null with default,
      subject varchar(15) not null with default,
      code integer1
);


insert into tempreport(id,title,name,
      code)
      select b.id, b.title, a.name, code=1
      from book b, author a
      where b.id = a.id;


insert into tempreport (id,title,
      subject,code)
      select b.id, b.title, s.subject, code=2
      from book b, subject s
      where b.id = s.id;
```

Your new joined table should look like this:

```
|id          |title                          |name       |subject      |code  |
|-------------------------------------------|-----------|-------------|------|
|         1001|The C Programming Language    |Kernighan  |             |     1|
|         1001|The C Programming Language    |Ritchie    |             |     1|
|         1001|The C Programming Language    |           |C            |     2|
|         1001|The C Programming Language    |           |language     |     2|
|         1001|The C Programming Language    |           |programming  |     2|
|-------------------------------------------|-----------|-------------|------|
```

In this example, we have labeled the new table "tempreport" because it represents only a temporary arrangement of data, strictly for use in the Report-Writer. Although you may wish to use such a join or view in a report more than once, the table data may change between reports. Therefore, when you use this technique, you should make it a habit to define anew the join or view each time you produce a report.

Putting Joined Tables in a Report

The final step in creating your library report is to write a report specifications file for your joined table, "tempreport". (Note that the report is also to be named "tempreport".) Here are the specifications:

```
.NAME tempreport
.DATA tempreport
.SORT title,code
.BREAK title,code  /* title and code will
     be break columns */
.RIGHTMARGIN 80 /* it is important to set
     the right margin here*/


.HEADER title
      .LEFTMARGIN 0
      .ULCHARACTER "="
      .UNDERLINE
      .PRINT title .NEWLINE
      .NOUNDERLINE
      .BLOCK      /* start a block after printing
            the master info */
```

```
.HEADER code
      .TOP  /* goto the top of the block
            each time code changes  */
      .ULCHARACTER "-"
/* test the value of code and set the margin
      appropriately */
      .IF code = 1 .THEN
            .LEFTMARGIN 5
            .UNDERLINE
            .PRINT "Authors" .NEWLINE
            .NOUNDERLINE
      .ELSEIF code = 2 .THEN
            .LEFTMARGIN 20
            .UNDERLINE
            .PRINT "Subjects" .NEWLINE
            .NOUNDERLINE
      .ENDIF


.DETAIL
/* test the value of code to see which
      column to print */
      .IF code = 1 .THEN
            .PRINT name .NEWLINE
      .ELSEIF code = 2 .THEN
            .PRINT subject .NEWLINE
      .ENDIF


.FOOTER title
      .ENDBLOCK   /* end the block at the
            end of the master info */
      .NEWLINE
```

Here is the completed report:

```
The C Programming Language
=== = =========== ========
     Authors          Subjects
     -------          --------
     Kernighan        C
     Ritchie          language
                      programming
```

Avoiding Awkward Page Breaks

Let's say that you have invested in a new bookcase and have expanded the size of your personal library by many volumes. Now when you combine your three tables, you create a much larger joined table than before:

```
|id          |title                          |name       |subject      |code  |
|------------------------------------------|-----------|-------------|------|
|        1001|The C Programming Language     |Kernighan  |             |     1|
|        1001|The C Programming Language     |Ritchie    |             |     1|
|        1002|Computer Programming and Arch.|Eckhouse   |             |     1|
|        1002|Computer Programming and Arch.|Levy       |             |     1|
|        1003|The ODT-DATA Papers            |Stonebraker|             |     1|
|        1004|Database Systems               |Ullman     |             |     1|
|        1005|The Quiet American             |Greene     |             |     1|
|        1001|The C Programming Language     |           |C            |     2|
|        1001|The C Programming Language     |           |language     |     2|
|        1001|The C Programming Language     |           |programming  |     2|
|        1002|Computer Programming and Arch.|           |architecture |     2|
|        1002|Computer Programming and Arch.|           |assembler    |     2|
|        1002|Computer Programming and Arch.|           |computer     |     2|
|        1002|Computer Programming and Arch.|           |programming  |     2|
|        1003|The ODT-DATA Papers            |           |Database     |     2|
|        1003|The ODT-DATA Papers            |           |ODT-DATA     |     2|
|        1003|The ODT-DATA Papers            |           |computer     |     2|
|        1004|Database Systems               |           |Database     |     2|
|        1004|Database Systems               |           |management   |     2|
|        1005|The Quiet American             |           |Vietnam      |     2|
|------------------------------------------|-----------|-------------|------|
```

If you create a report from such variable blocks of data, you should issue very specific instructions to the Report-Writer about where and where not to place page breaks; otherwise, you may find that some of your data has been incongruously parceled across two pages. In cases where you use the Report-Writer to generate a report from a single, unjoined table, you would use the **.need** statement to establish proper page breaks (see Chapter 6 for a full explanation of the **.need** statement).

In this case, when you generate a report from a joined table, you must simulate the **.need** statement to assure proper page breaks. You must first create two additional tables before executing your report.

Using SQL, simulating the **.need** statement is a three-step process:

1. First, you create two new tables.
2. Next, you join the two new tables to the "tempreport" table you have already made.
3. Finally, you add a **.query** section to your report specifications file, and alter some of the file's formatting statements .

The reason for the three-step process is that SQL requires a special method for the calculation of *num_sub* and *num_auth*. In SQL, when you perform a set function on a set of data and group rows together, you cannot place in the **select** clause any column not also listed in the **group by** clause, except as an argument to a set function. When a **select** statement includes a **group by** clause, any columns listed in the **select** clause must be single-valued per group.

To solve this problem, you must create two new tables, and join them to the "tempreport" table:

```
create table sub as
        select id,num_sub=count(subject)
        from tempreport
        where subject != ''
        group by id;


create table auth as
      select id.num_auth=count(name)
      from tempreport
      where name != ''
      group by id;
```

```
select * from sub;
```

```
|id            |num_sub       |
|--------------------------|
|          1001|             3|
|          1002|             4|
|          1003|             3|
|          1004|             2|
|          1005|             1|
|--------------------------|
```

```
select * from auth;
```

```
|id            |num_auth      |
|--------------------------|
|          1001|             2|
|          1002|             2|
|          1003|             1|
|          1004|             1|
|          1005|             1|
|--------------------------|
```

Now that you have created and joined the necessary tables, you must add a **.query** section to your report specifications file. Here is the revised specification file for your report, with the new **.query** section:

```
.NAME tempreport
.QUERY
     select t.id,t.title,t.subject,t.name,
          t.code,s.num_sub,a.num_auth
          from tempreport t,sub s,auth a
          where t.id = s.id
          and t.id = a.id
.SORT title,code
```

```
.BREAK title,code/* title and code will be
     break columns */
.RIGHTMARGIN 80   /* it is important to set
     the right margin here*/


.HEADER title
/* The calculation below contains the
     following variables and constants:*/
/* line_number is the current line number
     on the page   (RW variable) */
/* page_length is the current page length
     in the report (RW variable) */
/* num_sub is the number of subjects for
     this book (calculated above) */
/* num_auth is the number of authors for
     this book (calculated above) */
/* 4 is the number of lines taken up by
     headers in each block */
/* These variables are used to determine if
     there is enough */
/* room left on the page to print the next
     block of data */
.IF line_number + num_sub + 4 > page_length
     .THEN
                .NEWPAGE
     .ELSEIF line_number + num_auth + 4 >
          page_length .THEN
                .NEWPAGE
     .ENDIF
          .LEFTMARGIN 0
          .ULCHARACTER "="
          .UNDERLINE
          .PRINT title .NEWLINE
          .NOUNDERLINE
          .BLOCK  /* start a block after printing the
                master info  */
```

```
.HEADER code
      .TOP  /* goto the top of the block each time
            code changes  */
      .ULCHARACTER "-"
/* test the value of code and set the
            margin appropriately  */
      .IF code = 1 .THEN
            .LEFTMARGIN 5
            .UNDERLINE
            .PRINT "Authors" .NEWLINE
            .NOUNDERLINE
      .ELSEIF code = 2 .THEN
            .LEFTMARGIN 20
            .UNDERLINE
            .PRINT "Subjects" .NEWLINE
            .NOUNDERLINE
      .ENDIF


.DETAIL
/* test the value of code to see which
            column to print  */
      .IF code = 1 .THEN
            .PRINT name .NEWLINE
      .ELSEIF code = 2 .THEN
            .PRINT subject .NEWLINE
      .ENDIF


.FOOTER title
      .ENDBLOCK   /* end the block at the end of the
            master info */
      .NEWLINE
```

Appendix B
Report-Writer Error Messages

Number	Message
7000	Bad flag '%0' specified in REPORT command. Correct usage of REPORT command is: REPORT [-f..] [-m..l-r] [-qn] [-ln] [-wn] [+tl-t] [-s] [-uuser] [-h] [-a] [-vn] [+bl-b] db rep ['({param=val})'].
7001	Bad parameters specified at or near '%0'. Correct usage of REPORT command is: REPORT [-f..] [-m..l-r] [-qn] [-ln] [-wn] [+tl-t] [-s] [-uuser] [-h] [-a] [-vn] [+bl-b] db rep ['({param=val})'].
7002	Not enough information specified on the REPORT command. You must specify "repname," "dbname," and any parameters used in your query, unless you specify the "-p" flag on the REPORT command, which prompts you for any missing information. Correct usage of REPORT is: REPORT [-f..] [-m..l-r] [-qn] [-ln] [-wn] [+tl-t] [-s] [-uuser] [-h] [-a] [-vn] [+bl-b] db rep ['({param=val})'].
7004	Report or table '%0' does not exist or is not owned by you.
7005	Data table '%0' in Report '%1' does not exist or is not owned by you.
7006	Sort column '%0' does not exist.
7007	Database '%0' does not exist.

Number	Message
7008	Bad numeric value specified for flag '%0' on REPORT command. Only positive values allowed. Value set to default. Correct syntax of REPORT is: REPORT [-f..] [-m..l-r] [-qn] [-ln] [-wn] [+tl-t] [-s] [-uuser] [-h] [-a] [-vn] [+bl-b] db rep [({param=val})]. Error check continues...
7009	Bad value '%0' specified for the -m flag on REPORT command. Legal values are -mdefault (or -m), -mcolumn, -mwrap or -mblock. Value set to default. Error check continues...
7010	'%0' nonfatal errors have occurred in setting up the report. No report is written. Correct errors and rerun.
7011	Error opening report file '%0'. No report is written.
7012	Because of errors found in the .SORT list, the report writer stops. Fix .SORT list and rerun SREPORT. Processing stops.
7020	Query specified in .QUERY command is too long. Maximum allowable size is %0 characters. You can use the -q flag on the REPORT command to extend the max.
7021	Error in .QUERY command RANGE statement. Problem at or near RANGE of %0 is %1. Error check continues...
7022	Parameter '%0' not specified on command line. You can have the Report-Writer prompt you for values of parameters by specifying the -p flag on the REPORT command. Correct syntax is: REPORT [-f..] [-m..l-r] [-qn] [-ln] [-wn] [+tl-t] [-s] [-uuser] [-h] [-a] [-vn] [+bl-b] db rep ['({param=val})']. Error check continues...

Number	Message
7023	Bad parameter name in .QUERY at or near '%0'. Error check continues...
7025	Either you had errors in the .QUERY command, or you did not specify either a .QUERY or .DATA command. Processing stops.
7026	Because of errors in running report, the Report-Writer stops. Fix errors in report or data table and rerun.
7027	Cannot open temporary filenamed %0, used internally. May not be able to provide complete information on future occurrences of errors 7510-7521. Processing continues.
7100	Unrecognizable command in %0 text for '%1'. Line with error: '%2 %3'. Error check continues...
7101	Cannot interpret printing command in %0 text for '%1'. Problem occurs at or near '%2'. Line with error: '%3 %4'. Error check continues...
7102	Unknown parameter name '%0' in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...
7103	Unrecognized numeric value in %0 text for '%1'. Line with error: '%2 %3'. Error check continues...
7104	Extra characters ignored in %0 text for '%1'. Did you forget to specify the .PRINT command? Line with error: '%2 %3'. Error check continues...
7105	Bad format for '%0' in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...

Number	Message
7106	Character format specified for numeric item '%0' in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...
7107	Numeric format specified for character or date item '%0' in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...
7108	No column specified for aggregate '%0' in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...
7109	Bad column '%0' specified for aggregate '%1'. Error occurred in %2 text for '%3'. Line with error: '%4 %5'. Error check continues...
7110	Aggregate '%0' in %1 text for '%2' not in footer text. Line with error: '%3 %4'. Error check continues...
7111	Cannot aggregate a column with a higher sort order. Aggregate '%0' in '%1' caused the error. Line with error: '%2 %3'. Error check continues...
7112	Trying to aggregate character column '%0' in aggregate '%1' in text for '%2'. Line with error: '%3 %4'. Error check continues...
7114	Column '%0' not a break column in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...
7115	Bad name '%0' found in cumulative in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...
7116	Bad name found in %0 text for '%1'. Line with error: '%2 %3'. Error check continues...

Number	Message
7117	Bad or no primitive aggregate found for cumulative in %0 text for '%1'. Line with error: '%2 %3'. Error check continues...
7118	Bad preset value found in '%0' in %1 text for '%2'. Preset can be a numeric, date constant, or a column name and must match the type of the aggregated column. Line with error: '%3 %4'. Error check continues...
7119	Cannot preset average aggregate in %0 text for '%1'. Line with error: '%2 %4'. Error check continues...
7120	Bad parameter name at or near '%0' in %1 text for '%2'. Rest of print command skipped. Line with error: '%3 %4'. Error check continues...
7121	Parameter '%0' in %1 text for '%2' not on command line. Use -p option on REPORT command to have the report writer prompt you for values of parameters. Rest of command skipped. Line with error: '%3 %4'. Error check continues...
7123	You can only specify a COUNTU, SUMU or AVGU aggregate on a column which is in the sort list. Aggregate '%0' in '%1' caused the error. Line with error: '%2 %3'. Error check continues...
7129	Date format specified for numeric or character item '%0' in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...
7130	Bad column name '%0' found in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...

Number	Message
7131	Bad format of FORMAT or TFORMAT command at or near '%0'. Error found in %1 text for '%2'. Correct format is: Line with error: '%3 %4'. Rest of command skipped. Error check continues...
7133	A .PRINT command does not follow justification in %0 text for '%1'. Line with error: '%2 %3'. Error check continues...
7140	Bad format of .POSITION command at or near '%0'. Error found in %1 text for '%2'. Correct format is: Line with error: '%3 %4'. Rest of command skipped. Error check continues...
7141	Value given for .POSITION command too large or too small. Error found in %0 text for '%1'. Maximum value is 255, minimum is 0. Line with error: '%2 %3'. Error check continues...
7143	Bad format of .WIDTH command at or near '%0'. Error found in %1 text for '%2'. Correct format is: Line with error: '%3 %4'. Rest of command skipped. Error check continues...
7144	Value given for column width too large or too small. Error found in %0 text for '%1'. Maximum value is 255, minimum is 1. Line with error: '%2 %3'. Error check continues...
7160	Expected quoted character for .ULCHAR command at or near '%0'. Error found in %1 text for '%2'. Line with error: '%3 %4'. Error check continues...
7162	No column name specified for .WITHIN command. Error occurred in %0 text for '%1'. Error check continues...

Number	Message
7163	Bad column name '%0' found in .WITHIN command. Error occurred in %1 text for '%2'. Error check continues...
7164	Bad format of .WITHIN command at or near '%0'. Error occurred in %1 text for '%2'. Correct format of command is: Line with error: '%3 %4'. Rest of command skipped. Error check continues...
7165	Nested .WITHIN blocks in %0 text for '%1'. Make sure you .ENDWITHIN the previous block first. Line with error: '%2 %3'. Error check continues...
7166	Tried to .END a .WITHIN command in %0 text for '%1'. No .WITHIN command precedes the .END. Error check continues...
7170	Bad name '%0' found after .END command. Error occurred in %1 text for '%2'. Only .END WITHIN or .END BLOCK or .END allowed. Error check continues...
7175	Bad name '%0' found after .BEGIN command. Error occurred in %1 text for '%2'. Error check continues...
7181	Command out of place in %0 text for '%1'. Command only allowed in a .WITHIN block of commands. Line with error: '%2 %3'. Error check continues...
7182	Command out of place in %0 text for '%1'. Command not allowed in a .WITHIN block of commands. Line with error: '%2 %3'. Error check continues...

Number	Message
7183	Column name specified in positioning command in %0 text for '%1'. Because you are currently in a .WITHIN block, you cannot specify a column name on the positioning command. Line with error: '%2 %3'. Error check continues...
7184	Specified W_COLUMN constant outside of .WITHIN block. Error occurred in %0 text for '%1'. Line with error: '%2 %3'. Error check continues...
7185	Cannot specify W_COLUMN as the CUM column or the PRESET column. Error occurred in %0 text for '%1'. Line with error: '%2 %3'. Error check continues...
7201	Left margin set to too small a value in %0 text for '%1'. Value set to minimum allowable value. Processing continues...
7202	Left margin set to too big a value in %0 text for '%1'. Value set to default. Processing continues...
7203	Page length set to too small a value in %0 text for '%1'. Must be greater than the combined size of the page header and page footer. Set to minimum allowable value. Processing continues...
7204	Non-character format found in %0 text for '%1'. Field skipped. Error check continues...
7206	Right margin set to too small a value in %0 text for '%1'. Value set to default. Processing continues...
7207	Right margin set to too large a value in %0 text for '%1'. Value set to maximum allowable value. Processing continues...

Number	Message
7208	Tab set to negative column number in %0 text for '%1'. Value set to zero. Processing continues...
7209	Tab set to position beyond end of line in %0 text for '%1'. Value set to maximum allowable value. Processing continues...
7210	Left margin set to value greater than right margin in %0 text for '%1'. Right margin set greater than this value. Processing continues...
7211	Right margin set to value less than left margin in %0 text for '%1'. Value set greater than left margin. Processing continues...
7220	Positioning for .CENTER, .LEFT, or .RIGHT too small. Error found in %0 text for '%1'. Position set to default. Processing continues...
7221	Positioning for .CENTER, .LEFT, or .RIGHT too big. Error found in %0 text for '%1'. Position set to default. Processing continues...
7230	Value for .POSITION in %0 text for '%1' ignored. Value is below minimum(0) or above maximum(255). Processing continues...
7231	Value for .WIDTH in %0 text for '%1' ignored. Value is below minimum(1) or above maximum(255). Processing continues...
7301	Exceeded maximum number of lines that can be written in one block (using the wraparound, filling, or adjusting formats). Maximum value allowed is %0. You can change this maximum by using the -w flag on the REPORT command. Processing continues...

Number	Message
7305	Tried to write beyond end of line while centering or justifying. The offending line has been truncated. Maximum line length is %0. You can change this maximum by using the -l flag on the report command. Processing continues...
7500	Comparison done between expressions of different kind (numeric, string, or date) in %0 text for '%1'. Line with error: '%2 %3'. Processing continues...
7501	Expected numeric expression in %0 text for '%1'. Line with error: '%2 %3'. Processing continues...
7502	Expected numeric or date expression in %0 text for '%1'. Line with error: '%2 %3'. Processing continues...
7503	Arithmetic operands not the same type (numeric or date) in %0 text for '%1'. Line with error: '%2 %3'. Processing continues...
7504	Wrong type of argument for function in %0 text for '%1'. Line with error: '%2 %3'. Processing continues...
7510	Comparison done between expressions of different kind (numeric, string, or date) in %0 text for '%1'. Line with error: '%2 %3'. Processing stops.
7511	Expected numeric expression in %0 text for '%1'. Line with error: '%2 %3'. Processing stops.
7512	Expected numeric or date expression in %0 text for '%1'. Line with error: '%2 %3'. Processing stops.
7513	Arithmetic operands not the same type (numeric or date) in %0 text for '%1'. Line with error: '%2 %3'. Processing stops.

Number	Message
7514	Wrong type of argument for function in %0 text for '%1'. Line with error: '%2 %3'. Processing stops.
7515	Divided by zero in expression in %0 text for '%1'. Line with error: '%2 %3'. Processing continues...
7516	Result overflowed while evaluating expression in %0 text for '%1'. Line with error: '%2 %3'. Processing continues...
7517	Error occurred evaluating numeric expression (for example, divided by zero or result overflowed) in %0 text for '%1'. Line with error: '%2 %3'. Processing continues...
7518	Error occurred in %0 text for '%1'. Line with error: '%2 %3'. Processing continues...
7520	Absolute date encountered while doing sum or avg on %0. This value is skipped. Processing continues...
7521	Can't obtain diagnostic information for next error:
7700	Bad flag '%0' specified on RBF command. Correct usage of RBF command is: RBF [-m..l-r] [-ln] [-s] [-uuser] db rep. Processing stops.
7702	Not enough information specified on the RBF command. You must specify "repname" and "dbname," unless you specify the -p flag on the RBF command, which prompts you for any missing information. Correct usage of RBF is: RBF [-m..l-r] [-ln] [-s] [-uuser] db rep. Processing stops.

Number	Message
7708	Bad numeric value specified for flag '%0' on RBF command. Only positive values allowed. Value set to default. Correct usage of RBF command is: RBF [-m..l-r] [-ln] [-s] [-uuser] db rep Error check continues...
7709	Bad value '%0' specified for the -m flag on RBF command. Legal values are -mdefault (or -m), -mcolumn, -mwrap, or -mblock. Value set to default. Correct usage of RBF command is: RBF [-m..l-r] [-ln] [-s] [-uuser] db rep. Error check continues...
7710	Report '%0' created with SREPORT rather than RBF. You can only use RBF to edit reports created by default or in a previous RBF session.
7711	Report '%0' is too wide for your terminal.
7800	Bad flag specified on COPYREP command. Legal syntax is: COPYREP [-f] [-s] [-uuser] db file {report(s)} Use SREPORT to load reports into a database. Processing stops.
7801	No reports specified for COPYREP command. Legal syntax is: COPYREP [-f] [-s] [-uuser] db file {report(s)}. Processing stops.
7803	Error opening or writing file '%0' in COPYREP. Processing stops.
7900	Bad flag '%0' specified on SREPORT command. Correct format is: SREPORT [-s] [-uuser] dbname filename. Processing stops.

Number	Message
7901	Not enough information specified on the SREPORT command. You must specify a filename and a dbname, unless you specify the -p flag, which prompts you for any missing information. Correct format is: SREPORT [-s] [-uuser] dbname filename. Processing stops.
7902	Text file '%0' does not exist, is not readable, or is a bad name. Processing stops.
7903	File '%0' Line %1: Expected command at '%2'.
7904	File '%0' Line %1: Bad command name '%2' found.
7905	File '%0' Line %1: Bad name '%2' specified for report. Processing stops.
7906	File '%0' Line %1: No .NAME command encountered. It must be the first command in the file. Processing stops.
7907	File '%0' Line %1: Parameters starting at '%2' in command '%3' won't fit. Break up the command into two or more commands so that it fits.
7908	File '%0' Line %1: .HEADER or .FOOTER specified for '%2', which is not a break column (specified in a .SORT command), or 'report' or 'page'. Command ignored.
7909	File '%0' Line %1: Bad format of .HEADER or .FOOTER command. Correct format is: .HEADER column|report|page or .FOOTER column|report|page. Command ignored.
7910	%0 errors encountered in report specification. The report(s) is not added to your database. Rerun SREPORT with errors corrected. Processing stops.

Number	Message
7911	File '%0' Line %1: Bad format of .DATA command. Correct format is: Command ignored.
7912	File '%0' Line %1: Bad format of .OUTPUT command. Correct format is: Command ignored.
7913	File '%0' Line %1: Bad format of .SORT command. Correct format is:
7914	File '%0' Line %1: '%2' is a reserved name. It cannot be used as a column name in your data table.
7915	File '%0' Line %1: Bad sort direction '%2' specified. Correct values are 'ascending' or 'descending' (or 'a' or 'd'). 7916 File '%0' Line %1: Column name '%2' already specified in .SORT. You can only specify a column name once.
7917	File '%0' Line %1: Bad format of cumulative or aggregate. Correct format is: [CUM [(break)]] aggname (colname [,preset])
7920	Cannot create file '%0' for writing. Processing stops. This file is used internally by the SREPORT program. If you don't know why this error occurs, see the ODT-DATA system manager.
7921	File '%0' Line %1: Header or Footer already specified for %2. You can only specify one header and one footer for a break.
7922	File '%0' Line %1: .OUTPUT already specified for report. You can only specify one output file for a report.

Number	Message
7930	Premature end-of-file found. This is probably caused by unmatched quotes ("), parentheses, or .IF/.ENDIF. Processing stops.
7950	File '%0' Line %1: Error in RANGE statement in .query command. Correct format of RANGE is: range of range_var is tablename. Rest of query skipped.
7951	File '%0' Line %1: Unrecognized QUEL command in .query. Problem at or near '%2'. You can only specify RANGE and RETRIEVE commands in query. Rest of query skipped.
7952	File '%0' Line %1: Cannot use RETRIEVE INTO in .query. Only a simple RETRIEVE statement is allowed in query. Rest of query skipped.
7953	File '%0' Line %1: Error in RETRIEVE target list in .query. You probably have a problem with nesting of parentheses. Rest of query skipped.
7954	File '%0' Line %1: Error in RETRIEVE "where" clause in .query. You probably have a problem with nesting of parentheses. Rest of query skipped.
7955	File '%0' Line %1: Premature end of query found. Check the syntax of your query. Correct syntax is: Range of x is y retrieve (target_list) [where ...]
7956	File '%0' Line %1: You have already specified .query or .data. You must specify .query or .data for each report, but you can't specify both, or one of them twice.
7960	File '%0' Line %1: .%2 cannot be within an .IF statement.
7961	File '%0' Line %1: .%2 must follow .IF ...

Number	Message
7962	File '%0' Line %1: .%2 cannot follow .ELSE.
7963	File '%0' Line %1: .THEN expected.
7970	File '%0' Line %1: Boolean expression expected.
7971	File '%0' Line %1: Numeric, string, or date expression expected.
7972	File '%0' Line %1: Comparisons must be done between expressions of the same kind (numeric, string, or date).
7973	File '%0' Line %1: Expected numeric operand.
7974	File '%0' Line %1: Expected closing parentheses.
7975	File '%0' Line %1: Cannot have logical operator here.
7976	File '%0' Line %1: Expected comma.
7977	File '%0' Line %1: Expected open parentheses.
7978	File '%0' Line %1: No such function or aggregate.
7979	File '%0' Line %1: Alphanumeric name must follow S.
7980	File '%0' Line %1: Logical operators must have Boolean expressions as operands.
7981	File '%0' Line %1: Numeric operators must have numeric expressions as operands.
7982	File '%0' Line %1: CUM must precede aggregate outside of .FOOTER.

Number	Message
7983	File '%0' Line %1: + or - must have numeric or date expressions as operands.
7984	File '%0' Line %1: Wrong type of argument for function.
7991	Report '%0': Query specified in '%1'; '%1' is not a valid query language at this installation.
7992	Report '%0': The "order by" clause and "distinct" keyword cannot be used in the query specified with the .QUERY command. To specify ordering use the .SORT command.

Special Characters

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

ODT-DATA SQL Reference

ODT-DATA is based on technology developed by **INGRES CORPORATION**, and includes the following **INGRES** components:

- INGRES/DBMS and SQL Terminal Monitor
- INGRES/User Interfaces
  - Query-by-Forms
  - Report-by-Forms
  - Report Writer
  - Menu
  - Forms Runtime Systems and VIFRED
- INGRES/NET with TCP/IP Support
- INGRES/WindowView
- INGRES/ESQL Preprocessor for C

Document version: 1.0.0C
Date: 15 June 1990

Contents

Preface

The primary objective of this manual is to provide the ODT-DATA user with a complete description of the ODT-DATA relational database system, including ODT-DATA SQL (Structured Query Language). It is not intended to serve as a tutorial on the use of ODT-DATA or on relational database systems. Rather, it serves as the primary reference to the current syntax and function of ODT-DATA commands and files.

Intended Audience

The *ODT-DATA SQL Reference Manual* is intended for Open Desktop users who have a basic understanding of how ODT-DATA or other relational database systems work. The reader is not required to have a detailed understanding of the computer's operating system. However, readers should be familiar with logging on and off, as well as the computer's filesystem, if advanced features are to be used.

In a multiuser installation, various database-related tasks are assigned to various individuals with different privileges:

- The system administrator manages the ODT-DATA installation
- The database administrator (DBA) creates and manages a database
- The user manipulates data in the database

This manual is addressed to both types of installations, though at times the multiuser type is addressed explicitly. If you are a single user, assume that you are the system administrator and the database administrator as well as the user.

Structure of This Manual

The manual is divided into four chapters and two appendixes, each of which contains a number of subsections. Each chapter contains an introductory explanation, followed by detailed descriptions of applicable commands and files.

The contents include:

- Chapter 1 which explains SQL's syntactic elements.
- Chapter 2 which describes the SQL statements.
- Chapter 3 which describes the ODT-DATA SQL Terminal Monitor, the interactive system that allows you to enter SQL commands, edit the query and perform other useful tasks.
- Chapter 4 which describes the operating system's commands that apply to ODT-DATA. At the operating system command level, a database can be created or destroyed, the Terminal Monitor and Embedded SQL programs can be executed, and some database maintenance functions can be performed.
- Appendix A which lists the keywords in the Interactive and Embedded SQL environments.
- Appendix B which gives an in-depth description of the system catalogs required to operate an ODT-DATA environment.

The reference sections for statements and files contain various subsections, which are described and discussed in the following parts:

Purpose	A one-line summary of what the statement does or what the file is.
Syntax	The statement's syntax and usage (only included in sections that describe statements).
Description	*For commands:* A detailed description of command parameters and function. *For files:* The file format. *For others:* Applicable descriptions.
Examples	*For commands:* Specific examples. *For files:* Sample formats.
Files	For the operating system commands, the names of files referenced or affected by the command.

Conventions

The following conventions are used for describing the syntax of statements in this manual:

- Words in **boldface** are keywords and must be typed as shown when used. Required symbols and punctuation are also indicated by **boldface**.
- Words in *italics* are program-supplied elements.
- Clauses enclosed in square brackets ([ ]) are optional.
- Clauses enclosed in curly braces ({ }) are optional and can be repeated zero or more times.
- Keywords or clauses separated by vertical bars (| |) indicate lists from which one element is chosen.

Associated Publications

The *ODT-DATA SQL Reference Manual* is one of several publications provided for your use of ODT-DATA. The table below lists all the ODT-DATA books available with each Open Desktop product:

Open Desktop:
■ *Administering ODT-DATA*
■ *Using ODT-DATA*
Open Desktop Development System:
■ *ODT-DATA Embedded SQL User's Guide*
■ *ODT-DATA Embedded Open SQL Forms Reference Manual*
■ *ODT-DATA Open SQL Reference Manual*
■ *ODT-DATA Embedded SQL Companion Guide for C*
■ *GCA Application Program Interface*
Open Desktop Server Upgrade:
■ *Administering an ODT-DATA NET Server*
Open Desktop Optional Documentation:
■ *Using ODT-DATA Through Forms and Menus*
■ *ODT-DATA Report-Writer Reference Manual*
■ *ODT-DATA SQL Reference Manual*

Chapter 1
SQL Syntax

SQL (Structured Query Language) is the database language supported by ODT-DATA. SQL allows you to retrieve, manage, and maintain data in an existing ODT-DATA database. SQL statements are high-level descriptions of what needs to be done rather than how it should be done. In relational database terminology, SQL provides "automatic navigation" to the data in the database.

SQL statements can be used in any of several contexts. They can be:

- Entered directly through the ODT-DATA Terminal Monitor
- Embedded within programs written in high-level languages by using Embedded SQL
- Included in report specifications for the ODT-DATA Report-Writer

Consult Chapter 3 of this manual for information about the ODT-DATA Terminal Monitor. For information about Embedded SQL, consult the *ODT-DATA Embedded SQL Companion Guide for C*. The *ODT-DATA Report-Writer Reference Manual* describes the Report-Writer. There are four major SQL statements, each beginning with one of the following keywords:

select
insert
delete
update

As the keywords suggest, the statements are used, respectively, for selecting data, inserting data, deleting data, and updating data values. The syntactical forms of the four statements are similar, with **select** statements being the most general. For that reason, **select** statements are used in this chapter to illustrate SQL syntax.

Notation and Terminology

Keywords

A list of all keywords in ODT-DATA SQL is included in Appendix A of this manual. There you will also find the keywords of ODT-DATA Embedded SQL and ANSI standard SQL.

Names

Names in SQL are sequences of no more than 24 alphanumeric characters, starting with an alphabetic character. The underscore (_) is considered an alphabetic character. The "#", "@", and "$" signs are considered part of the alphanumeric character set. Thus, a name may begin with "a" through "z" (upper- or lowercase) or underscore (_), and the rest of the name may contain those characters, as well as "0" through "9" and "#", "@", and "$". A name may not begin with "ii". Names beginning with "ii" are reserved for use by ODT-DATA.

All uppercase letters in a name are converted to lowercase.

Comments

A comment is an arbitrary sequence of characters bounded by "/*" on the left and by "*/" on the right. For example:

```
/* This is a comment */
```

A comment so bounded is ignored in query processing.

Statement Separator

No statement terminator is required by the SQL language. For this reason, no semicolon is included in the syntax description for each statement.

The semicolon (;) as a statement separator is required in using the Terminal Monitor when more than one statement precedes a "\g". (Chapter 3 discusses the Terminal Monitor.) A group of statements followed by "\g" is called a "go block."

Examples showing modules of code in this manual include optional semicolons (;) as statement separators. Because SQL is used in a variety of contexts, the optional statement separator helps avoid unwanted side effects that could result if the context were to change.

Data Types

There are three classes of data type: character, numeric, and abstract. Character strings can be fixed length (**c** and **char**) or variable length (**vchar** and **varchar**). Numeric strings may be exact numeric (**integer**, **smallint**, or **integer1**) or approximate numeric (**float** and **float4**). The abstract data types are **date** and **money**.

Numeric	Approximate numeric	**float (float8)** **float4**
	Exact numeric	**integer (integer4)** **smallint (integer2)** **integer1**
Character	Fixed length	**c** **char**
	Variable-length	**vchar (text)** **varchar**
Abstract		**date** **money**

Fixed-Length Character Strings

Fixed-length character strings are sequences of no more than 2000 ASCII characters. Upper- and lowercase alphabetic characters within strings are accepted literally.

Two types of fixed-length character strings are supported in ODT-DATA: **char** and **c**. **Char** strings may contain any character, printing or non-printing. Blanks are significant when comparing **char** strings. **Char** is the preferred fixed-length character type. The **C** type is supported for compatibility with previous ODT-DATA versions.

Only printing characters are allowed within **c** strings. Non-printing characters (for example, control characters) are converted to blanks.

Blanks are ignored when comparing **c** strings. For example, the next two **c** strings are treated identically:

```
the house is around the corner

thehouseisaroundthecorner
```

Variable-Length Character Strings

Variable-length character-string constants are sequences of no more than 2000 ASCII characters. Upper- and lowercase alphabetic characters within variable-length strings are accepted literally.

To include a quotation mark within a variable length character string, double it, as in:

```
the ''dog'' is black
```

This evaluates to:

```
the 'dog' is black
```

There are two types of variable-length character-strings in ODT-DATA: **vchar** and **varchar**. **Varchar** is the preferred variable-length character-string type. **Vchar** is supported for compatibility with previous ODT-DATA versions. All ASCII characters except the NULL character are allowed within **vchar** strings. NULL characters are converted to blanks. **varchar** strings may contain any character, including non-printing characters and the NULL character.

Blanks are not ignored in comparisons by either **vchar** or **varchar**. For example, the next two character strings are treated identically:

```
the house is around the corner

thehouseisaroundthecorner
```

However, the way blanks are handled by the two data types is different. In comparing strings of unequal length, **varchar** effectively adds blanks to the end of the shorter string to bring it up to the same length as the longer string. **Vchar** does not add blanks; it considers a shorter string as "less than" a longer string if all characters up to the length of the shorter string are equal.

As an example of how this affects comparisons, consider the two strings (a) abcd\001 and (b) abcd. (Assume \001 represents one ASCII character, **ControlA**.) If these are compared as **vchar**, then (a) > (b). However, if compared as **varchar**, then (a) < (b), because the **varchar** has a higher ASCII value than 001.

Integer Numbers

Integer values range from -2,147,483,648 to +2,147,483,647, and they contain no fractional part. Integer values that exceed that range are converted to floating-point. If an integer is less than +32,767 and greater than -32,768, it is treated as a 2-byte integer. Otherwise, it is converted to a 4-byte integer.

The three integer data types are **integer1** (1 byte), **smallint** (2 bytes), and **integer** (4 bytes).

Floating-point Numeric Data Types

Floating-point values consist of an integer part, a decimal point and a fraction part or scientific notation of the following format:

[+|-] {*dig*} [.*dig*{*dig*}][**e**|**E** [+|-] {*dig*}]

where *dig* is a digit. An example is:

```
2.3 e-02
```

A mantissa with a missing exponent has an exponent of one (1) inserted. Floating-point numbers are double-precision quantities with a range of approximately -10**38 to +10**38 and a precision of approximately 16 significant figures.

The character used to indicate the decimal point, by default a period (.), can be changed by means of the **II_DECIMAL** environment variable, described in *Administering ODT-DATA*.

The two approximate numeric data types are **float4** (4 bytes) and **float** (8 bytes).

Dates

Date Output

ODT-DATA supports date values that constitute either absolute dates and times or time intervals. ODT-DATA outputs such values as strings of 25 characters with trailing blanks inserted.

ODT-DATA uses one of the following output formats for an absolute date or time:

Format	Example
dd-mmm-yyyy	15-nov-1982
dd-mmm-yyyy hh:mm:ss	15-nov-1982 12:32:48

ODT-DATA displays 24-hour times for the current time zone, which is determined when ODT-DATA is installed. Dates are stored in Greenwich Mean Time and adjusted for your time zone when they are displayed.

For a time interval, ODT-DATA displays the most significant portions of the interval that fit in the 25 character string. If necessary, ODT-DATA inserts trailing blanks to fill out the string. The format appears as follows:

```
yy yrs mm mos dd days hh hrs mm mins ss secs
```

Significance is a function of the size of any component of the time interval. For instance, consider the following time interval:

```
5 yrs 4 mos 3 days 12 hrs 32 min 14 secs
```

ODT-DATA displays such an interval as follows:

```
5 yrs 4 mos 3 days 12 hrs
```

Date Input

Dates are input as quoted character strings. ODT-DATA accepts the following valid input formats:

- These are the legal formats for input of November 15, 1982:

Format	Example
'*mm/dd/yy* '	'11/15/82'
'*dd-mmm-yy* '	'15-nov-82'
'*dd-mmm-yyyy* '	'15-nov-1982'
'*mm-dd-yy* '	'11-15-82'
'*yy.mm.dd* '	'82.11.15'
'*mmddyy* '	'111582'
'*mm/dd* '	'11/15'
'*mm-dd* '	'11-15'
'today '	The string 'today' is a legal absolute date with today's date as its value.
'now '	The string 'now' is a legal absolute date and time with today's date and the current time as its value.

NOTE: The date formats described here are the default formats, also known as US format. See the following section titled "International Date Formats" for information about changing the date format conventions to accommodate international conventions.

- These are the legal formats for input of 10:30:00:

Format	Example
'*hh:mm:ss* '	'10:30:00'
'*hh:mm:ss xxx* '	'10:30:00 pst'
'*hh:mm* '	'10:30'

NOTE: ODT-DATA supplies the appropriate time zone designation. Time formats are assumed to be on a 24-hour clock. However, times entered with a designation of "am" or "pm" are automatically converted to 24-hour internal representation. Any such designation must follow the absolute time and precede the time zone, if included. If you do not specify a date with an absolute time, today's (that is, the current day's) date is supplied.

- These are the legal input formats for November 15, 1982, 10:30:00:

Format	Example
'*mm/dd/yy hh:mm:ss* '	'11/15/82 10:30:00'
'*dd-mmm-yy hh:mm:ss* '	'15-nov-82 10:30:00'
'*mm/dd/yy hh:mm:ss xxx* '	'11/15/82 10:30:00 pst'
'*dd-mmm-yy hh:mm:ss xxx* '	'15-nov-82 10:30:00 pst'
'*mm/dd/yy hh:mm* '	'11/15/82 10:30'
'*dd-mmm-yy hh:mm* '	'15-nov-82 10:30'
'*mm/dd/yy hh:mm xxx* '	'11/15/82 10:30 pst'
'*dd-mmm-yy hh:mm xxx* '	'15-nov-82 10:30 pst'

- These are the legal formats for date intervals which include the following designations:

Examples:

```
'5 years'
'8 months'
'14 days'
'5 yrs 8 mos 14 days'
'5 years 8 months'
'5 years 14 days'
'8 months 14 days'
```

- These are the legal formats for time intervals which include the following designations.

Examples:

```
'23 hours'
'38 minutes'
'53 seconds'
'23 hrs 38 mins 53 secs'
'23 hrs 53 seconds'
'28 hrs 38 mins'
'38 mins 53 secs'
'23:38 hours'
'23:38:53 hours'
```

International Date Formats

The database may be set to one of five date formats (modes) for the interpretation of dates. This mode is set on a session basis. The II_DATE_FORMAT environment variable described in *Administering ODT-DATA* can be used to change the date format conventions to accommodate the international date conventions shown here. The modes are:

Mode	Input	Interpreted as
US	default	(as above)
MULTINATIONAL	mm/dd/yyyy	dd/mm/yyyy
ISO (Multinational)	mmddyy	yymmdd
SWEDEN/FINLAND	mm-dd-yyyy	yyyy-mm-dd
GERMAN	xxxxx	dmmyy
	xxxxxx	ddmmyy
	xxxxxxx	dmmyyyy
	xxxxxxxx	ddmmyyyy

Money

ODT-DATA stores money values as their actual money amount, significant to exactly two decimal places. Thus, ODT-DATA rounds all money values to their amounts in dollars and cents on input and output. Arithmetic operations on the **money** data type retain two-decimal place precision.

ODT-DATA supports the following range of money values:

$-99999999999999.99 <= *m* <= $99999999999999.99

ODT-DATA displays money values as strings of 20 characters. The display format is:

$*sddddddddddddddd.dd*

where *s* is the sign (- for negative and no sign for positive) and *d* is a digit from 0 to 9.

ODT-DATA accepts money values on input either as character strings or as numbers, as follows:

Character string input — '$*sddddddddddddddd.dd*'

The dollar sign is optional. The sign defaults to + if not specified. A cents value of zero (".00") need not be specified.

Numeric input — ODT-DATA accepts any valid integer or floating-point number on input as a money value, and conversion to the **money** data type occurs automatically.

Note that several environment variables described in *Administering ODT-DATA* affect the display of money values. The II_MONEY_FORMAT environment variable can be used to set the currency symbol. As indicated above, the default currency symbol is the dollar sign ($). The II_MONEY_PREC environment variable sets the precision with which money values are displayed. The default precision is two decimal digits. The II_DECIMAL environment variable sets the character used to indicate the decimal point, by default a period (.).

Storage Formats for Data Types

Every data item in an ODT-DATA database is stored in one of the following storage formats:

Notation	Type	Range
char(1) - char(2000)	character	a string of 1 to 2000 characters
c1 - c2000	character	a string of 1 to 2000 characters
varchar(1) - varchar(2000)	character	a string of 1 to 2008 characters
vchar(1) - vchar(2008)	character	a string of 1 to 2008 characters
integer1	1-byte integer	-128 to +127
smallint (integer2)	2-byte integer	-32,768 to +32,767
integer (integer4)	4-byte integer	-2,147,483,648 to +2,147,483,647

Notation	Type	Range
float4	4-byte floating	-10**38 to +10**38 (7 decimal precision)
float (float8)	8-byte floating	-10**38 to +10**38 (16 decimal precision)
date	date (12 bytes)	1-jan-1582 to 31-dec-2382 (for absolute dates) and -800 years to 800 years (for time intervals)
money	money (8 bytes)	$-99999999999999.99 to $99999999999999.99

NOTE: **c** and **vchar** are supported for compatibility with previous ODT-DATA versions. **Char** and **varchar** are now preferred.

NOTE: If your hardware supports the IEEE standard for floating-point numbers, then the **float** type is accurate to 15 decimal precision, and **money** type is accurate to 14 decimal precision (that is, -$ddddddddddddd.dd to +$ddddddddddddd.dd). Also, floating-point numbers range from -10**256 to +10**256.

The designations **integer2**, **integer4**, and **float8** may be used in place of **smallint**, **integer**, and **float**, respectively.

Constants

There are two basic types of constants: string and numeric. In addition, there is a special constant, NULL. SQL also provides system constants to provide data that can help improve query performance. Constants are also known as "literals."

Each type of constant is assigned a default data type, but you can assign them another data type if you wish.

String Constants

String constants are represented by a sequence of characters enclosed in apostrophes (' '). Printing characters are represented literally. To represent a non-printing character you must use the hex constant. (The hex constant is only necessary in the Terminal Monitor; in Embedded SQL, any sequence of characters that can be assigned to a host program variable may be assigned to a character string.)

A hex constant is a special kind of string constant. It is represented by an "X" followed by a string enclosed by apostrophes that contains an even number of characters from the set {['A'-'F'],['a'-'f'],['0'-'9']}. For example, this represents the ASCII string "ABC<carriage return>":

```
X'4142430D'
```

SQL string constants do not support the octal representation of ASCII.

The default data type for string constants is **varchar**, but they may be assigned without explicit conversion to any of the character data types or the **money** data type.

Numeric Constants

Numeric constants are represented by a sequence of digits, an optional decimal point, and an optional exponent representation. If no decimal point is specified, and if the value of the constant is within the legal range, the default is **integer**. Otherwise, the default is **float**. Numeric constants may be assigned without explicit conversion to any of the numeric data types or the **money** data type.

Null Constant

The NULL constant may be assigned to any nullable data type.

Structured Data

Tables

All data in ODT-DATA is stored as tables. A table is a named array of values. The array is composed of columns (sometimes called fields or attributes) and rows (sometimes called records or tuples). Table names may not begin with "ii".Here is an example of a table:

Name	Party	Age	Funding
Robbins	Republican	42	1250000
Capetti	Democrat	52	946000
Greenberg	Citizens	48	766000
Hernandez	Democrat	38	987000
Johnson	Independent	46	854000
Chang	Republican	55	1540000

Columns

Each column of a table has a name, which must be a legal ODT-DATA name. All values in any given column have the same storage format (that is, data type and width in bytes). The maximum number of columns in a table is 127.

Rows

A row represents an individual record in a table. All rows in a table are of the same width in bytes, and they each maintain the same column types. The maximum length of a row is 2000 bytes.

A Sample Database

A sample database is used for examples throughout this reference manual. The name of the database is "empdata," and its description appears in the following table:

Table Name	Column Name	Data Type
employee	eno	smallint
	ename	char(10)
	age	integer1
	job	smallint
	salary	float4
	dept	smallint
dept	dno	smallint
	dname	char(10)
	mgr	smallint
	floor	integer1
job	jid	smallint
	jtitle	char(10)
	lowsal	float4
	highsal	float4

Correlation Name

Consider the following **select** statement:

```
select      employee.eno, employee.ename
from        employee
where       employee.dept = 23;
```

This statement retrieves employee numbers and names for all employees in department 23. Its **select-from-where** structure is typical of retrieval statements in SQL.

Now consider the following alternative formulation of the same query:

```
select      e.eno, e.ename
from        employee e
where       e.dept = 23;
```

In the second query example, "e" is a correlation name. A correlation name is also known as a range variable because it is used in an SQL statement to "range over" some table. It is specified as shown, by its appearance following the table name in a **from** clause (or an **update** clause, in the case of an **update** statement). At any particular point during execution of the statement in question, the correlation name serves to mark a particular row of the specified table as the current row for processing. Statement execution completes when every row of the table has been marked and processed in this way. Thus, in the earlier example, "e" marks each employee record in turn, and the query is complete when all employee records have been processed.

It is not always necessary to introduce a correlation name explicitly; the first formulation shown above is perfectly legal SQL. However, the correlation name is still present there implicitly. The symbol "employee" in that version is actually being used to play two roles: (1) it serves to identify the employee table and, (2) it also serves as a correlation name ranging over that table. Note that it is never wrong, and sometimes it is necessary, to introduce correlation names explicitly.

A correlation name can be any sequence of alphanumeric characters acceptable as a name (see "Names" earlier in this chapter).

Finally, it is not always necessary to qualify column names explicitly with the correlation name. An unqualified column name (appearing in, for example, a **select** or a **where** clause) is assumed to be implicitly qualified by a table or correlation name appearing in the **from** clause (or **update** clause) on the same syntactic level as that unqualified reference (see "Subqueries" later in this chapter). Thus, for example, the first query could be simplified to the following:

```
select      eno, ename
from        employee
where       dept = 23;
```

"Eno," "ename," and "dept" are all implicitly qualified by "employee." Likewise, the second query could be simplified to the following:

```
select      eno, ename
from        employee e
where       dept = 23;
```

"Eno," "ename," and "dept" are now all implicitly qualified by "e."

Note that, to prevent ambiguity, column names must be qualified explicitly when it is not clear which table the column comes from.

The maximum limit to the number of correlation and table names that can be referenced in a single statement is 30. Under certain circumstances, the limit may be less.

Groups

It is sometimes convenient to think of the rows of a table as being divided up into groups or partitions by the value(s) of some column(s) of that table. For example, the candidates table presented in the "Tables" section might be grouped by party, to yield the result shown in the following table:

Name	Party	Age	Funding
Greenberg	Citizens	48	766000
Capetti	Democrat	52	946000
Hernandez	Democrat	38	987000
Johnson	Independent	46	854000
Chang	Republican	55	1540000
Robbins	Republican	42	1250000

Note that such grouping is purely conceptual; the table is not really rearranged in the database. The grouping is specified dynamically by means of a **group by** clause, as follows:

```
select      ...
from        candidates
group by    party;
```

The purpose of such grouping is generally to allow some set function to be computed for each group. For example:

```
select      party, avg (funding)
from        candidates
group by    party;
```

This statement will retrieve each party name, together with the average funding for that party, from the candidates table.

Expressions

Expressions are used in SQL in many contexts; for example, to denote values to be retrieved (in a **select** clause) or compared (in a **where** clause). SQL expressions fall into two broad classes: those that involve set functions and those that do not. Most of the rules for forming expressions apply equally to each of the two classes, with the following exceptions:

- The argument to a set function is an expression, but that expression cannot in turn involve any set functions. In other words, no nesting of set functions is permitted.
- Expressions involving set functions can appear only in certain specific contexts.
- Constants are considered expressions.

The sections later in this chapter titled "Scalar Functions" and "Set Functions" provide more information about functions and expressions.

Columns

A column name, explicitly or implicitly qualified, is an expression. For example, each of the following names in an expression:

```
employee.ename

e.ename

ename
```

Parentheses

An expression can be enclosed in parentheses, such as ('J. J. Jones'), without affecting its meaning.

Arithmetic Operations

Expressions of numeric types can be combined arithmetically to produce other expressions. ODT-DATA supports the following arithmetic operators (in descending order of precedence):

+,-	plus, minus (unary)
**	exponentiation
*,/	multiplication, division
+,-	addition, subtraction (binary)

Unary operators group from right to left, and binary operator group from left to right.

Parentheses can force the desired order of precedence. For example:

```
(job.lowsal + 1000) * 12
```

In this expression, the "+" operator is forced to take precedence over the "*" operator.

A variety of arithmetic checks, such as integer overflow, integer divide by zero, floating-point underflow, floating-point overflow, and floating-point divide by zero, can be enabled by specifying the **-x** flag on the **sql** command line. Refer to the **sql** command in Chapter 4, "ODT-DATA Operating System Commands."

The + operator can also be used to concatenate strings. For example

```
'This ' + 'is ' + 'a ' + 'test.'
```

This gives the value:

```
'This is a test.'
```

When used in this fashion, the + operator behaves exactly like the **concat** function.

Arithmetic Operations on Dates

ODT-DATA supports a limited set of arithmetic operations on items of the **date** data type:

Addition:

interval	+	interval	->	interval
interval	+	absolute	->	absolute

Subtraction:

interval	-	interval	->	interval
absolute	-	absolute	->	interval
absolute	-	interval	->	absolute

ODT-DATA does not support multiplication or division of date values.

ODT-DATA also enables you to convert date constants into numbers of days relative to an absolute date. For example, to convert today's date to the number of days since January 1, 1900, use the expression:

```
num_days = int4(interval('days', 'today' -
     date('1/1/00')))
```

To convert back, use:

```
(date('1/1/00') + concat(char(num_days), ' days'))
```

where "num_days" is the number of days added to the date constant.

Note that for comparisons, a blank (default) date is less than any interval date. All interval dates are less than all absolute dates. Intervals are converted to comparable units before they are compared. For instance, date ("5 hours") is greater than date ("200 minutes"). Note also that dates are stored internally in an absolute format. For this reason, "5:00 pm pst" compares as equal to "8:00 pm est."

Note also that this expression yields March 1:

```
date("1-feb") + "1 month"
```

Adding a month always yields the same date in the next month unless there are fewer days in the next month, in which case it yields the last day of the next month. For instance, adding a month to May 31 yields June 30. Similar rules hold for subtraction. Moreover, similar rules apply for adding and subtracting years.

When adding intervals, each of the units is added. For example:

```
date("6 days") + date("5 hours")
```

This yields "6 days 5 hours." To yield "4 years 6 months 1 hour 40 minutes," use the expression:

```
date("4 years 20 minutes") + date("6 months 80
minutes")
```

When adding or subtracting intervals, or when subtracting absolute dates, overflow or underflow are propagated upward, except that neither passes from days to months.

Type Conversion

When two numeric expressions are combined, ODT-DATA converts as necessary to make the storage formats (that is, data types and widths) identical. The resulting expression then has the same storage format.

When ODT-DATA operates on an integer and a floating-point number, the integer is converted to a floating-point number before the operation. When ODT-DATA operates on two integers of different sizes, the smaller is converted to the size of the larger. When operating on two floating point numbers of different sizes, ODT-DATA converts the larger to the size of the smaller number.

When multiplying or dividing a **money** data item by a non-money item (that is, integer or floating-point), ODT-DATA converts the non-money multiplier or divisor to the **money** type prior to calculation.

The following table summarizes the possible results of numeric combinations:

	integer1	smallint	integer	float4	float	money
integer1	integer1	smallint	integer	float4	float	money
smallint	smallint	smallint	integer	float4	float	money
integer	integer	integer	integer	float4	float	money
float4	float4	float4	float4	float4	float4	money
float	float	float	float	float4	float	money
money	money	money	money	money	money	money

For example:

```
(job.lowsal + 1000) * 12
```

For this expression, the first operator (+) combines a **float4** expression (job.lowsal) with a **smallint** constant (1000). The result is **float4**. The second operator (*) combines the **float4** expression with a **smallint** constant (12), resulting in a **float4** expression.

Note that this produces a **float4** expression:

```
(job.lowsal + 1000) * 12
```

On the other hand, this produces a **float** expression:

```
float8((job.lowsal+1000)*12)
```

ODT-DATA also provides specific type converison functions. These are discussed later in "Explicit Type Conversion Functions."

Numeric Overflow

Numeric overflow can occur when the results of a computation are larger than can be held by the data type the computation is performed in. For example, in the following statement the calculation on the right-hand side is done in **integer2** arithmetic. If the **integer2** arithmetic results in a value greater than 32767, the largest possible **integer2** value, then overflow occurs.

```
update emp
set integer4col = integer2col * integer2col ;
```

You can avoid many common types of overflow by converting to a higher precision before performing the calculation. For example

```
update emp
set integer4col=int4(integer2col) * int4(integer2col);
```

(For more information on the **int4** function, see "Explicit-Type Conversion Functions" later in this chapter.)

Numeric overflow, underflow (for floating-point calculations), and division by zero are controlled by the **-x** command-line flag on the sql database start-up statement. ODT-DATA either continues as if no error occurred, signals an error and aborts the query, or signals a warning and continues, depending on how the **-x** flag is set. See the **sql** statement in Chapter 4, "ODT-DATA Operating System Commands," for more information.

Default Character-Type Conversion

Whenever a string of type **c** or **char** is put into a column defined as type **vchar** or **varchar**, all the string's trailing blanks are removed. Conversely, whenever a string of type **vchar** or **varchar** is put into a column defined as type **c** or **char**, the string is padded with blanks to fill out the column's defined width, if necessary.

Explicit-Type Conversion Functions

In addition to ODT-DATA's default type conversions, many explicit-type conversion functions are available. The following explicit-type conversion functions can be used:

Name	Operand Type	Result (Format)	Description
c(*expr*)	any	**c**	Converts any value to **c** string.
char(*expr*)	any	**char**	Converts any value to **char** string.
date(*expr*)	**c, vchar, char, varchar**	**date**	Converts **c**, **char**, **varchar**, or **vchar** string to internal date representation.
dow(*expr*)	**date**	**c**	Converts absolute date into its day of week (for example, 'Mon„' 'Tue').
float4(*expr*)	any except **date**	**float4**	Converts non-date expression to **float4**.
float8(*expr*)	anyexcept**date**	**float**	Converts non-date expression to **float**.
hex(*expr*)	**varchar, char, c, vchar**	varchar	Returns the hex representation of the argument string. The result length is 2 times the input string length. For example: hex('A') - '61' (ascii) or 'C1' (ebcdic).
int1(*expr*)	anyexcept**date**	integer1	Converts non-date expression to **integer1**.
int2(*expr*)	anyexcept**date**	**smallint**	Converts non-date expression to **smallint**.
int4(*expr*)	anyexcept**date**	**integer**	Converts non-date expression to **integer**.
money(*expr*)	anyexcept**date**	**money**	Converts non-date expression to internal money representation.
vchar(*expr*)	any	**vchar**	Converts any value to a **vchar** string. This function removes trailing blanks, if any, from **c** or **char** string expressions.

Name	Operand Type	Result (Format)	Description
varchar (*expr*)	any	**varchar**	Converts any value to a **varchar** string. This function also removes trailing blanks, if any, in **c** or **char** string expressions.

Scalar Functions

Two kinds of functions are provided: scalar functions and set functions. Scalar functions take as their argument a single-valued expression (or, in some cases, two such expressions). Set functions take as their argument an entire set of scalar values. This section is concerned only with scalar functions; set functions are described in a following section.

A scalar function reference consists of the function name, followed by a parenthesized expression (or pair of expressions) representing the function argument(s). A scalar function reference is an expression. Scalar function references can be nested to any level.

The explicit-type conversion functions discussed earlier are scalar functions; the other available scalar functions are described next.

Numeric Functions

In addition to the type conversion functions described above, the following numeric functions are available:

Name	Format(Result)	Description
abs(*n*)	all numeric types and **money**	absolute value of *n*
atan(*n*)	**float**	arctangent of *n*
cos(*n*)	**float**	cosine of *n*
exp(*n*)	**float**	exponential of *n*
log(*n*)	**float**	natural logarithm of *n*

Name	Format(Result)	Description
mod(*n*,*b*)	**integer, smallint, integer1**	*n*, modulo *b*. *n*, and *b* must be integers
sin(*n*)	**float**	sine of *n*
sqrt(*n*)	**float**	square root of *n*

For example:

```
exp(job.lowsal)
```

This gives the exponential of "job.lowsal" as a float expression.

String Functions

The following functions operate on **c**, **char**, **vchar**, or **varchar** data. The expressions *c1* and *c2* represent arguments for the various functions. They can represent any of the string types, except where noted. The expressions *len* and *nshift* represent integer arguments.

Name	Format(Result)	Description
concat(*c1*,*c2*)	**c**, **vchar** or **varchar**	Concatenates one string to another. The result size is the sum of the sizes of the two arguments. If the result is a **c** or **char** string, it is padded to achieve the proper length. To determine the characteristics of concatenating one string to another, see the following chart.

1st String	2nd String	Trim Blanks		Result Type
		from 1st?	from 2nd?	
c	**c**	Yes	—	c
c	**vchar**	Yes	—	c

1st String	2nd String	Trim Blanks		Result Type
		from 1st?	from 2nd?	
c	**char**	Yes	—	**c**
c	**varchar**	Yes	—	**c**
vchar	**c**	No	—	**c**
char	**c**	Yes	—	**c**
varchar	**c**	No	—	**c**
vchar	**vchar**	No	No	**vchar**
vchar	**char**	No	Yes	**vchar**
vchar	**varchar**	No	No	**vchar**
char	**vchar**	Yes	No	**vchar**
varchar	**vchar**	No	No	**vchar**
char	**char**	No	—	**char**
char	**varchar**	No	—	**char**
varchar	**char**	No	—	**char**
varchar	**varchar**	No	No	**varchar**

Name	Format(Result)	Description
left(*c1*,*len*)	any character data type	Returns the left-most *len* characters of *c1*. If the result is a fixed-length **c** or **char** string, it is the same length as *c1*, padded with blanks. The result format is the same as *c1*.

Name	Format(Result)	Description
length(*c1*)	**smallint**	If *c1* is a fixed-length **c** or **char** string, returns the length of *c1* without trailing blanks. If *c1* is a variable-length string, returns the number of characters actually in *c1*.
locate(*c1*,*c2*)	**smallint**	Returns the location of the first occurrence of *c2* within *c1*, including trailing blanks from *c2*. The location is in the range 1 to size(*c1*). If *c2* is not found, the function returns size(*c1*) + 1.
lowercase(*c1*)	any character data type	Converts all uppercase characters in *c1* to lowercase.
pad(*c1*)	**vchar** or **varchar**	Returns *c1* with trailing blanks appended to *c1*; for instance, if *c1* is a **varchar** string that could hold 50 characters but only has two characters, then "pad(*c1*)" appends 48 trailing blanks to *c1* to form the result.
right(*c1*,*len*)	any character data type	Returns the right-most *len* characters of *c1*. Trailing blanks are not removed first. If *c1* is a fixed-length character string, the result is padded to the same length as *c1*. If *c1* is a variable-length character string, no padding occurs. The result format is the same as *c1*.
shift(*c1*,*nshift*)	any character data type	Shifts the string *nshift* places to the right if *nshift* > 0 and to the left if *nshift* < 0. If *c1* is a fixed-length character string, the result is padded with blanks to the length of *c1*. If *c1* is a variable-length character string, no padding occurs. The result format is the same as *c1*.
size(*c1*)	**smallint**	Returns the declared size of *c1* without removal of trailing blanks.

Name	Format(Result)	Description
squeeze(*c1*)	**vchar** or **varchar**	Compresses white space. White space is defined as any sequence of blanks, NULL characters, newlines (line feeds), carriage returns, horizontal tabs, and form feeds (vertical tabs). Trims white space from the beginning and end of the string, and replaces all other white space with single blanks. This function is useful for comparisons. The value for *c1* must be a string of variable-length character string data type (not fixed-length character data type). The result is the same length as the argument.
trim(*c1*)	**vchar** or **varchar**	Returns *c1* without trailing blanks. The result has the same length as *c1*.
uppercase(*c1*)	any character data type	Converts all lowercase characters in *c1* to uppercase.

The string functions can be arbitrarily nested to achieve other string functions. For example:

```
left(right(x.name, size(x.name) - 1), 3)
```

This returns the substring of "x.name" from character positions 2 through 4.

You can also nest string functions within themselves. For example:

```
concat(concat(x.lastname, ','), x.firstname)
```

This concatenates "x.lastname" with a comma and then concatenates "x.firstname" with the first concatenation result. Note, however, that the same result can be achieved with the + operator:

```
x.lastname + ',' + x.firstname
```

Date Functions

ODT-DATA supports two functions that derive values from absolute dates and one function that derives a value from interval dates. These functions operate on rows that contain date values. The unit expression is a quoted string that represents the part of the date to use in the calculation. Legal values are:

Unit	Value
second	**seconds sec secs**
minute	**minutes min mins**
hour	**hours hr hrs**
day	**days**
week	**weeks wk wks**
month	**months mo mos**
quarter	**quarters qtr qtrs**
year	**years yr yrs**

The date expression must be an absolute date and not a date interval.

Name	Format (Result)	Description

date_trunc (*unit*,*date*)	**date**	Returns a date value that represents the input *date* truncated to the level of granularity expressed in the *unit*. By using the **date_trunc** function you can group all the dates within the same month or year, and so forth.
		
		For example:
		
		`date_trunc('month',date('23-oct-1985 12:33'))`
		
		This returns "1-oct-1985" as its value. Another example is:
		
		`date_trunc('year',date('23-oct-1985'))`
		
		This returns "1-jan-1985" as its value.
		
		All truncation takes place in terms of calendar years and quarters ("1-jan," "1-apr," "1-jun," and "1-oct"). If you need to truncate in terms of a fiscal year, simply offset the calendar date by the number of months between the beginning of your fiscal year and the beginning of the next calendar year ('6 mos' for a fiscal year beginning July 1, or '4 mos' for a fiscal year beginning September 1). For example:
		
		`date_trunc('year',date+'4 mos') - '4 mcs'`
		
		Monday constitutes the starting day for weeks. Note that the beginning of a week for an early January date may fall into the previous year.

Name	Format (Result)	Description

date_part (*unit,date*)	**integer**	Returns an integer representing one component of the input date. The *unit* parameter represents the desired component. This function is useful in set functions and in assuring correct ordering in complex date manipulation.
		
		For example, if *date_field* contains the value "23-oct-1985," then this returns a value of 10:
		
		`date_part('month',date(date_field))`
		
		This returns a value of 23:
		
		`date_part('day',date(date_field))`
		
		Months are ordered with January set to month 1. Hours are set to a 24-hour clock. Quarters are numbered 1 through 4. Weeks return a number representing the number of the week since the beginning of the year in which the input date falls. Week 1 begins on the first Monday of the year. Dates before the first Monday of the year are considered to be in week 0.

interval (*unit,date*)	**float**	Converts a date interval into a floating-point constant in user-specified units. Allowable values for *unit* are **seconds, secs, minutes, mins, hours, hrs, days, weeks, wks, months, mos, quarters, qtrs, years** or **yrs.**

NOTE: The **interval** function assumes that there are 30.4375 days per month and 365.25 days per year when using the **mos, qtrs,** and **yrs** specifications.

The Ifnull Function

The **ifnull** function allows users to return a fixed value instead of a null value when a null is encountered. The **ifnull** function is defined as **ifnull***(v1,v2)*. The function takes two input arguments of the same data type, *v1* and *v2*. The resulting value is the same type as the type specified in *v1* and *v2*. If the value of *v1* is **not null**, *v1* is returned. If the value of *v1* is **null**, *v2* is returned. For example:

ifnull (i2, i4) results in i4
ifnull (i4, i2) results in i4

The result is the "larger" of the data types where

f8 > f4 > i4 > i2 > i1

and

varchar > text > char > c

and the length is taken from the longest value. Therefore:

ifnull (varchar (5), c 10)

results in varchar (10).

The result is nullable if either argument is nullable. The *v1* value is not required to be nullable, although in most applications it would be nullable.

Dbmsinfo( ) Function

The **dbmsinfo**( ) function is used to request information from a database. This function queries the database from SQL.

The **dbmsinfo**( ) function takes the place of **_username**. This function has the syntax:

dbmsinfo (*request_name*)

The following request names can be used with **dbmsinfo**( ).

Request Name	Response Description
transaction_state	'1' means in a transaction; '0' means not in a transaction
autocommit_state	'1' means autocommit is on; '0' means off
_bintim	Returns the current time and date in an internal format, represented as the number of seconds since January 1, 1970 00:00:00 GMT
_cpu_ms	CPU time for session, in milliseconds
_et_sec	Elapsed time for session, in seconds
_dio_cnt	Direct I/O requests for session
_bio_cnt	Buffered I/O requests for session
_pfault_cnt	Page faults for server
dba	ODT-DATA username of the database owner
username	ODT-DATA username of the user currently running ODT-DATA (like **user**)
_version	ODT-DATA version number (for example, '6.0/01')
database	Database name
terminal	Terminal address
query_language	SQL

These request names are case insensitive, and **dbmsinfo( )** always returns a **varchar(32)** as the result. If **dbmsinfo** is given a request name it does not recognize, it returns an empty string.

The following query returns a variable-length string containing the answer, that is, '1':

```
select dbmsinfo('transaction_state')
```

Set Functions

A set function is a function that operates on an entire column of values, not just a single value. Consider the following example:

```
select      sum (employee.salary)
from        employee
where       employee.dept = 23;
```

This statement retrieves the total salary for employees in department 23. The argument to the function is the set (column) of employee salary values where the employee department is equal to 23.

The following set functions are supported:

Name	Format(Result)	Description
count	**integer**	Count of occurrences
sum	**integer, float, money**	Summation
avg	**float, money**	Average (sum/count)
max	same as argument	Maximum value
min	same as argument	Minimum value

The general syntax of a set function reference takes the form:

set_fun ([**distinct** | **all**] *expr*)

where set_fun denotes a set function, expr denotes any expression that does not itself include a set function reference (at any level of nesting), and the optional **distinct** keyword indicates that duplicate values are to be eliminated from the argument before the set function is performed. The optional keyword **all** indicates the default condition, in which duplicate values are not eliminated. Note that it makes no sense to use **distinct** in conjunction with the functions **min** and **max**.

The **count** function includes a special case. The set function reference

count(*)

may be used to count the number of rows in the result table. For example:

```
select      count(*)
from        employee
where       dept = 23;
```

This statement counts the number of employees in department 23. The argument "*" cannot be qualified by **all** or **distinct**.

NULL values are ignored by the set function. Here again, **count(*)** is the exception, because it counts rows rather than columns. Consider the following table:

Name	Exemptions
Smith	0
Jones	2
Tanghetti	4
Fong	NULL
Stevens	NULL

Running

```
count(c1)
```

returns the value "3" whereas

```
count(*)
```

returns "5".

The following restrictions apply to the use of set functions:

- First, as already mentioned, they cannot be nested.
- Second, set function references, or expressions that include such a reference as

```
sum (employee.salary) / 25
```

  are permitted only in the context of a **select** or **having** clause. Furthermore, any column names appearing (in such a **select** or **having** clause) outside such a set function reference must have been specified as one of the operands in a **group by** clause at the same syntactic level as that **select** or **having** clause.

If the argument to a set function evaluates to an empty set, then the value returned is as follows:

count	zero
sum, avg	the NULL value
max, min	the NULL value

The **group by** clause allows set functions to be performed on groups of rows, according to the values in specified columns of the rows.

Set functions are also available in the ODT-DATA REPORTS subsystems (Report-Writer and Report-By-Forms). There they are known as aggregate functions.

IFNULL and Set Functions

As stated above, the **sum, avg, max,** and **min** set functions can return a null value, when the argument to a set function evaluates to an empty set. This can occur even when the column the set function is operating on is **not null**able. To assure that a set function never returns a **null**, use the **ifnull** function. **Ifnull** returns the normal set function result unless that result is **null**, in which case it returns the second argument to the **ifnull** function.

The following returns **-1** if **sum(employee.salary)/25** is **null**:

```
ifnull ( sum(employee.salary)/25, -1 )
```

The following returns **0** if **max(s.empno)** is **null**:

```
ifnull ( max(s.empno), 0 )
```

Search Conditions

Search conditions are used in **where** and **having** clauses to qualify the selection of data. Search conditions are composed of predicates of various kinds, optionally combined together by means of parentheses and the logical operators **and**, **or** and **not**. Thus, any of the following is a legal search condition:

predicate
not *search_condition*
search_condition **or** *search_condition*
search_condition **and** *search_condition*
(*search_condition*)

where *search_condition* stands for an arbitrary search condition.

Of the three logical operators, **not** has the highest precedence, followed by **and**, with **or** having the lowest precedence. They group from left to right. The parentheses may be used for arbitrary grouping.

There are seven kinds of predicates, each described in its own section below:

comparison predicate
like predicate
between predicate
in predicate
any-or-all predicate
exists predicate
is NULL predicate

Predicates evaluate to "true," "false," or "unknown." They evaluate to "unknown" if one or both operands are the NULL value (the "is NULL" predicate is the exception). When predicates are combined using logical operators (**and**, **or**, or **not**) to form a search condition, the search condition evaluates to "true," "false," or "unknown" as determined by the following tables:

and	true	false	unknown
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

or	true	false	unknown
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

Not(true) is false, **not**(false) is true, **not**(unknown) is unknown.

After all search conditions are evaluated, the value of the **where** or **having** clause is determined. The **where** or **having** clause can be "true" or "false" only; "unknown" values are considered "false."

Subqueries

Nesting of queries is accomplished in SQL by means of a search condition feature known as the subquery. A subquery is a subselect used in a predicate of a search condition. (See the section called "Select" in this chapter for more information about subselects.) The search condition containing the subquery can be part of another subquery, or of any data manipulation statement permitting search conditions. Multiple levels of nesting are permitted. Here is an example of a subquery:

```
select      ename
from        employee
where       dept in
            (select dno
            from   dept
            where  floor = 3);
```

The expression in parentheses is the subquery; it evaluates to the set of department numbers for departments on the third floor. The outer query then retrieves the names of employees whose department number is in that set, that is, names of employees who work on the third floor.

Subqueries often take the place of expressions in predicates. Note that subqueries can be used in place of expressions only in the specific instances outlined in the following sections on predicate types.

The previous example serves to illustrate the concept of syntactic level. Briefly, the **select**, **from**, and **where** clauses in the subquery are considered to be at a different syntactic level from the **select**, **from**, and **where** clauses in the outer subselect. More generally, two syntactic units within the same statement are considered to be at the same syntactic level if and only if there exists a subselect within that statement such that the two syntactic units are both immediately contained within that subselect (that is, neither one is contained within a subselect (subquery) nested within that subselect).

The syntax of the subquery is identical to that of the subselect, except for one restriction: expressions in the **select** clause cannot be assigned result column names.

A subquery may include references to correlation names defined (explicitly or implicitly) outside the subquery. For example:

```
select      ename
from        employee empx
where       salary >
            (select avg (salary)    from
            employee empy     where
            empy.dept = empx.dept);
```

("Select names of employees with salary greater than the average for their department.")

Here the subquery includes a reference to a correlation name (empx) defined in an outer query; that is, at a different syntactic level. Note that the reference must be explicitly qualified here; otherwise, it would be assumed to be implicitly qualified by "empy." The overall query is evaluated by letting "empx" take each of its permitted values in turn (that is, letting it range over the employee table), and for each such value of "empx," evaluating the subquery. Note that at least one of the correlation names must be explicit in this example (either "empx" or "empy," but not both, could be allowed to default to simply "employee").

Comparison Predicate

A comparison predicate takes the form:

expression_1 comparison_operator expression_2

where *comparison_operator* is one of the following:

=	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Note that the comparison operator "not equal to" may also be indicated by "< >" or "^=".

All comparison operators are of equal precedence.

NOTE: If a subquery is in the right-hand argument of a comparison predicate, the subquery may return at most one row. If the subquery returns zero rows, the comparison predicate evaluates to "false."

If there is a NULL value on either or both sides of any comparison operator, it evaluates to "false."

Like Predicate

The **like** predicate provides the only pattern-matching capability in SQL for the character data types (**char, varchar, c,** and **vchar**). It takes the following form:

columnname [**not**] **like** *pattern* [**escape** *escape_character*]

where *pattern* is a string constant, not a column. The pattern-matching characters are the percent sign (%) to denote zero or more arbitrary characters, and the underscore (_) to denote exactly one arbitrary character.

If the **escape** clause is specified, the escape character 'escapes' the pattern matching characters mentioned previously. In addition, the *escape_character*, if specified, also escapes the bracket characters ([and]) with a twist. When the escape character escapes the pattern match characters, percent and underscore, it means to treat these characters like normal characters in the string; do not use their pattern-matching meanings. When the escape character escapes a bracket, it means to treat this character like a pattern-matching character. If a bracket is not escaped, it is treated like a normal character in a string.

Between Predicate

The operators **between** and **not between** have the following meanings:

Operator	Meaning
y **between** *x* **and** *z*	$x < y$,**and** $y < z$
y **not between** *x* **and** *z*	**not** (*y* **between** *x* **and** *z*)

In the foregoing, *x*, *y*, and *z* are expressions. Subqueries may not be substituted for any of the expressions.

In Predicate

The operators **in** and **not in** (followed by a parenthesized list of expressions) are defined as follows:

Operator	Meaning
y **in** (*x*, ..., *z*)	*y* = *x* **or** ... **or** *y* = *z*
y **not in** (*x*, ..., *z*)	**not** (*y* **in** (*x*, ..., *z*))

In the foregoing, *x*, *y*, and *z* are expressions and may not be subqueries. If there is only one expression in the list, the parentheses are optional.

Another version of the **in** predicate takes the form:

expression [**not**] **in** (*subquery*)

The subquery must contain a reference to exactly one column in its **select** clause.

Any-or-All Predicate

An **any-or-all** predicate takes the form:

any-or-all-operator (*subquery*)

The subquery must have exactly one expression in its **select** clause (so that it evaluates to a set of scalar values, not a set of rows). The **any-or-all** operator is one of the following:

=any	**=all**
!=any	**!=all**
<any	**<all**
<=any	**<=all**
>any	**>all**
>=any	**>=all**

It is permissible to include a space between the comparison operator and the keyword **any** or **all**.

Let "$" denote any one of the comparison operators =, !=, <, <=, >, >=. Then the predicate

x **$any** (*subquery*)

evaluates to "true" if and only if the comparison predicate

x $ *y*

is true for at least one value *y* in the set of values represented by *subquery*. If the subquery is empty, the **$any** comparison fails (evaluates to "false"). Likewise, the predicate

x **$all** (*subquery*)

is true if and only if the comparison predicate

x $ *y*

is true for all values *y* in the set of values represented by *subquery*. If the subquery is empty, the **$all** comparison succeeds (evaluates to TRUE).

The operator =**any** is equivalent to the operator **in**. For example:

```
select      ename
from        employee
where       dept in
            (select     dno
            from  dept
            where floor = 3);
```

This may be rewritten as:

```
select      ename
from        employee
where       dept = any
            (select     dno
            from  dept
            where floor = 3);
```

The operator **some** is a synonym for operator **any** and would appear as:

```
select      ename
from        employee
where       dept = some
            (select     dno
            from  dept
            where floor = 3);
```

Exists Predicate

An **exists** predicate takes the form:

exists (*subquery*)

It evaluates to "true" if and only if the set represented by *subquery* is nonempty. For example:

```
select      ename
from        employee
where       exists
            (select     *
            from  dept
            where dno = employee.dept
            and   floor = 3);
```

("Names of employees who work on the third floor.")

It is typical, but not required, for the subquery argument to **exists** to be of the form "select *".

Is NULL Predicate

The **is NULL** predicate takes the form:

is [not] null

where *x* **is null** is true if and only if *x* is the NULL value. Because you cannot test for NULL by using the comparison operator "=", you must use this predicate to find out whether an expression is NULL.

Data Manipulation Statements

The SQL data manipulation statements are **select**, **update**, **delete**, and **insert**.

Select

The general syntax of **select** is:

subselect
{**union [all]** *subselect*}
[**order by** *result_column* [**asc** | **desc**]{, *result_column* [**asc** | **desc**]}]

where:

- *subselect* **union** *subselect* yields all results that either subselect run individually would yield.
- Corresponding data types across subselects must be coercible into a common data type. They must be either all character types or all numeric types.
- All subselects in a **select** have the same number of columns in their result table.
- Each *result_column* in the **order by** clause consists of either a result column name or an integer constant in the range 1 - *n*, where *n* is the number of columns in the result table of each of the subselects.
- The optional keywords **asc** and **desc** specify ascending and descending sort sequence, respectively. If neither is specified for a particular column, **asc** is assumed by default.
- Each subselect has the syntax shown below.

The syntax for subselect is:

select [**all**|**distinct**] *expression* [**as** *result_column*] {, *expression* [**as** *result_column*]}
from *table* [*corr_name*] {, *table* [*corr_name*]}
[**where** *search_condition*]
[**group by** *column* {, *column*}]
[**having** *search_condition*]

The keyword **distinct** indicates that duplicate rows are to be eliminated. The keyword **all**, the default condition, causes duplicate rows to remain.

The expressions in the **select** clause can be any expressions constructed in accordance with their rules (refer to the earlier section titled "Expressions" in this chapter). They may also take one of the following forms:

correlation_name.*	meaning all the columns of the table denoted by *correlation_name*.
table.*	meaning all the columns of *table*.
*	meaning all the columns of all the tables named in the **from** clause. This cannot be part of a comma-separated list; it must be the only statement in the select list.

A result_column may be assigned to any expression that denotes a single column in the result table (that is, where *expression* does not use the "*" syntax). The result column then appears in the result table as the column heading for the expression.

The ability to assign a result column name to an expression is of particular benefit when the expression is not simply a column from a database table. If the expression is such a column, the column heading in the result table is by default, the name of that column. However, when the expression is, for example, a scalar or set function or involves a computation, ODT-DATA returns blanks for the column heading. To override this default, assign the expression an appropriate result column. The result column, whether default or explicit, is also used in the **order by** clause.

The columns in the **group by** clause are names of columns from the table(s) identified in the **from** clause. They may be qualified by a **having** clause.

From a conceptual standpoint, the subselect is evaluated in the following manner:

- The Cartesian product of all tables identified in the **from** clause is formed. (Cartesian products are defined later in the section titled "Cartesian Product.")
- From that product, rows not satisfying the search condition specified in the **where** clause are eliminated.
- Next, the remaining rows are grouped in accordance with the specifications of the **group by** clause.
- Groups not satisfying the search condition in the **having** clause are then eliminated.
- Finally, the expressions specified in the **select** clause are evaluated. If the keyword **distinct** has been specified, any duplicate rows are eliminated from the result table.

NOTE: Bear in mind that the foregoing explanation is purely conceptual in nature. Actual evaluation normally does *not* proceed in precisely the manner described but instead uses some more efficient method, as determined by the ODT-DATA query optimizer.

If the subselect includes a **group by** clause, each expression in the **select** clause must be single-valued per group. That is, the only data items permitted in such an expression are the following:

- constants
- the grouping columns
- set function references

As usual, however, terms that can be combined by the use of arithmetic operations, can also be the arguments to scalar functions, and so on.

If the subselect includes a **having** clause, each expression in that clause must also be single-valued per group. If the **group by** clause is omitted in a subselect with a **having** clause, the entire table is considered to be a single group.

The result of a **select** statement is the union of the results of all subselects in that statement, ordered in accordance with the specifications of the optional **order by** clause. Duplicate rows are always eliminated if either **union** or **distinct** is specified. If **order by** is not specified, the rows of the result appear in unpredictable order.

The following is an example of a **select** statement:

```
select      eno
from        employee
where       age  45
union
select      mgr
from        dept
where       floor = 3
order       by 1;
```

Update

The general syntax of **update** is as follows:

update *table* [*corr_name*]
set *column* = *expression*{, *column* = *expression*}
[**where** *search_condition*]

Here is an example:

```
update      employee
set         job = 27,
            salary = salary * 1.1
where       job = 25;
```

Delete

The general syntax of **delete** is:

delete
from *table* [*corr_name*]
[**where** *search_condition*]

Here is an example:

```
delete      from employee
where       job = 0;
```

Insert

The general syntax of **insert** is:

insert
into *table* [(*column* {, *column*})] *source*

where *source* is either a subselect or takes the form:

values (*expression* {, *expression*})

Expressions used in the **values** clause can be only constants, scalar functions on constants, or arithmetic operations on constants.

Here are two examples:

```
insert
into   dept (dno, dname, mgr)
values (38, 'Purchasing', 21458);

insert
into   employee (eno)
       select     mgr
       from dept
       where      dname = 'newdept';
```

Relational Concepts

One of the first query languages proposed for use in relational systems was based on relational algebra. Even though no purely algebraic language is in current use, some of the algebraic operators have become a standard part of the terminology of relational systems. The most familiar of these are:

- Projection
- Restriction
- Cartesian product
- Join

This section shows how these operators are expressed in SQL. For illustration, this section uses the tables "employee," "dept," and "job" defined in the earlier section, "A Sample Database."

Projection

Projection is an operator that constructs a "vertical section" of an existing table by taking a subset of its columns. For example:

```
project employee on (ename, age)
```

This theoretical statement specifies a table consisting of the "ename" and "age" columns of the "employee" table.

The **select** clause in SQL corresponds to projection. For example, the statement "project employee on (ename, age)" is expressed in SQL as:

```
select      ename, age
from        employee;
```

Restriction

Restriction constructs a "horizontal section" of a table by taking those rows that satisfy a specified condition. For example

```
restrict employee on (age = 40)
```

This theoretical statement defines a table consisting of all rows in "employee" for which the value in "age" is greater than 40.

The **where** clause of an SQL statement corresponds to restriction. For example, "restrict employee on (age >40)" is expressed in SQL as:

```
select      *
from        employee
where       age >40;
```

Cartesian Product

The Cartesian product of two tables, for example, A and B, is a table (denoted, for example, by A*B) consisting of all concatenations of rows from A with rows from B. That is, each row *t* in A*B is of the form:

$$t = ab$$

where *a* is a row from A and *b* is a row from B, and every distinct pair (*a*,*b*) produces a row in A*B.

For example, "employee*job" is a table consisting of all concatenations *ej*, where *e* is a row from "employee" and *j* a row from "job."

The Cartesian product is easily expressed in SQL with the **select** statement. For example, the theoretical "employee*job" is expressed in SQL as:

```
select      *
from        employee, job;
```

Join

The join operator constructs a table out of two existing tables by collecting all pairs of rows such that each pair satisfies some condition. When the condition is equality between columns from the rows, the operator is called an equijoin. For example:

```
join employee with job on (job of employee = jid of job)
```

This theoretical statement would be an equijoin. By contrast, the following theoretical statement is a join, but not an equijoin:

```
join employee with job on (100*(age of employee)
     lowsal of job
```

A join is equivalent to a combination of Cartesian product followed by a restriction. For example, the second join in the previous paragraph is equivalent to a theoretical formulation:

```
restrict (employee*job) on (100*(age of employee)
     lowsal of job
```

Joins are easily expressed because SQL allows Cartesian product and restriction to be combined in a single query. The two theoretical examples of joins that were given are expressed in SQL as follows:

```
select      employee.*, job.*
from        employee, job
where       employee.job = job.jid;

select      employee.*, job.*
from        employee, job
where       100*employee.age  job.lowsal;
```

Nulls and Defaults

NULL is a data value that represents an unknown or inapplicable value. ODT-DATA gives you the option of having NULL values assigned automatically in a given column when no other value is specifically assigned. NULL is not the same as a zero, a blank, or an empty string.

NULLs are useful if you want to take an aggregate on a column, but do not want unknown or inapplicable values to affect the aggregate. For example, if there is a column "age" in the "employee" table, and you want to run an aggregate on that column to determine the average age of the employees, you want to make sure that any ages that have not been entered do not count as zeros. If ages that have not been entered are given the value NULL rather than zero, they aer not counted when the aggregate is run.

If you choose not to allow a column to contain the NULL value, ODT-DATA also lets you choose whether you want a default value (zero, blank, or empty) assigned to that column. If you do not allow either a NULL or a default value to be assigned, then the user is forced to enter a value in the column to avoid an error message. Disallowing NULLs and defaults is a good way to make sure that all columns are filled in, in cases where this is appropriate.

SQL returns NULL for an aggregate over an empty set, even when the aggregate includes columns that are not nullable. In the following example, **select** returns NULL, because there are no rows in **tbl**.

```
create table tbl (coll integer NOT NULL);
select max(coll) as x from tbl;
```

To eliminate this condition, you could use the IFNULL function. For example:

```
select IFNULL(max(coll),0) as x from tbl;
```

This returns zero (0).

You determine whether to allow NULLs and defaults in a column at the time you create the table, either with the **create table** command or the **table** facility of ODT-DATA/MENU. Please refer to *Using ODT-DATA Through Forms and Menus* or Chapter 2, "SQL Commands" of this manual for more details.

Transactions

A transaction in ODT-DATA is defined as one or more SQL statements that are to be processed as a single, indivisible database action. Transactions are atomic units of consistency and concurrency in the ODT-DATA multiuser database environment. None of the effects on a database of one user's transaction is visible to other users' transactions until the transaction is committed. When the transaction is committed, all of its effects are written permanently to the database, and they become available to the transactions of other users.

Concurrency control in ODT-DATA insures that simultaneously executing transactions do not interfere with each other in ways that could compromise the atomic status of a transaction. Deadlock is a possible consequence of transaction concurrency control, and deadlock is handled by the ODT-DATA transaction processing system. (See "Transaction Rollback" later in this chapter for a definition of deadlock.)

Transactions are committed or rolled back under user control. Transactions can also be rolled back under system control in cases of deadlock. Users can also declare savepoints within a transaction and subsequently roll back parts of a transaction to a declared savepoint.

Single statements, both inside and outside a transaction, can be rolled back under system control in cases of deadlock, timeout or error conditions (for example, a replace that generates a duplicate key in a table that has unique keys). Single statements can be rolled back under user control in the case of interrupts.

Transaction Control Statements

The transaction-controlling statements are as follows:

Command	Function
commit	Ends a transaction block and commits the transaction's effects to the database.
rollback	Terminates a transaction in progress, undoing the effects of all processed statements.
savepoint *savepoint_name*	Declares a savepoint.
rollback to *savepoint_name*	Rolls back all statements of a transaction subsequent to the named savepoint.

Committing Transactions

A transaction is committed when its updates to the database are written. Committing a transaction occurs at the end of the transaction. Before ODT-DATA commits a transaction, none of its updates to the database are available to other users, and the transaction can be rolled back without causing inconsistency or propagating undesirable rollbacks of other transactions. After the transaction is committed, however, its effects in the database are considered permanent and are visible to other transactions.

A transaction is committed explicitly with the **commit** statement. If a user **rollback** command or system-generated rollback on deadlock terminates the transaction before a **commit** command is processed, then the transaction is rolled back, and all its effects on the database are backed out.

A user may specify that queries should be committed implicitly by using the **set autocommit on** statement.

Transaction Rollback

At any time before a **rollback** statement commits a transaction, the transaction can be rolled back under user or system control. All effects of the transaction on the database are "undone," and no other transactions in progress are adversely affected.

Transactions can be rolled back in any of the following ways:

- User Rollback—The **rollback** statement causes immediate termination of a transaction in progress.

- System Abort—Deadlock is a situation that may arise during concurrent execution of transactions. Briefly described, deadlock can occur when transactions must "wait" to perform updates on a part of a database (for example, a table or a data page) because other transactions are currently updating the same part of a database. Deadlock occurs when two transactions are waiting for each other to release a part of the database that is being updated so it can perform its update. One transaction requires what the other transaction owns, and vice versa. Neither transaction releases the part of the database it has until it gets the other part, which it needs. Because of this standoff, neither transaction can proceed.

  ODT-DATA detects this situation when it occurs and chooses one transaction to roll back to end the deadlock. An error message (4700) is returned to the user to indicate rollback on deadlock. The user may then restart the transaction, if desired.

- Quitting ODT-DATA (\q) within a transaction—Exiting the Terminal Monitor (described in Chapter 3, "ODT-DATA Terminal Monitor") by typing \q while in the midst of a transaction causes the Terminal Monitor to ask whether you want your transaction committed or rolled back, and then it waits for your reply. Remember to commit transactions with the **commit** command before exiting ODT-DATA.

Savepoints and Partial Transaction Aborts

Savepoints are marker statements within a transaction. The **savepoint** command allows users to establish savepoints within a transaction. Within a transaction, statements already executed can be backed out as far as any specified savepoint included in the transaction. Savepoint names are character strings that conform to the rules for valid ODT-DATA names, with one exception. This exception is that savepoint names may begin with a numeric character; this allows integers (for example, 1, 2, 44) to be used as savepoint names. If the same savepoint name is used in multiple savepoint declarations within a single transaction, only the latest savepoint with that name is available for rollbacks. There is no limit to the number of savepoint declarations allowed within a single transaction.

It is worth noting that the **savepoint** command does not commit the partial transaction. None of the changes to a database affected by the transaction are committed until a **commit** statement commits the entire transaction. Savepoints provide useful constructs that facilitate conditional processing within a transaction by allowing partial or total transaction rollback.

At any time within a transaction a user can roll back to a pre-declared savepoint. All database changes affected by the transaction appearing after the savepoint are "undone," and all effects of transaction statement preceding the savepoint remain. The transaction can then continue executing other statements, including the declaration of other savepoints.

It is permissible to roll back to the same savepoint repeatedly within a transaction.

Interrupt and Timeout Handling in Transactions

The transaction-processing system in ODT-DATA recognizes the interrupt signal **Ctrl C**. This has a distinct effect on transaction processing.

A **Ctrl C** received by the Terminal Monitor during Multi-statement Transaction processing causes ODT-DATA to abort automatically the latest statement of the transaction. The transaction remains uncommitted and can be continued in normal fashion. This action takes place only once for a given transaction; subsequent **Ctrl C** characters are ignored unless a new statement is added to the MST since the last **Ctrl C**. The transaction must eventually be terminated in normal fashion, either with **end transaction** or **abort**.

A timeout condition detected while waiting for a lock (see the **set lockmode** statement) causes an error status (4702) to be returned to the user and otherwise behaves as if a **Ctrl C** had been received from the front-end.

SQL Transaction Semantics

Every SQL database query either begins or is added to an existing "Multi Query Transaction." An SQL transaction is started at the execution of the first SQL statement, and subsequent statements (for example, **select/insert/update/delete**) accumulate as part of that transaction. The transaction is not committed until a **commit** (or **end transaction**, which is supported for compatibility with previous versions of ODT-DATA) statement is issued. Statements that cannot be issued within a transaction, for example the **set lockmode** and **set autocommit** statements, can be executed if no other SQL statements have been executed since the last **commit**.

Queries issued between **commits** accumulate as part of the transaction and locks on data touched by each query held until the next **commit** statement. Even read locks, associated with **select** statements, accumulate and are held until **commit** time.

It is possible to change these standard SQL transaction semantics so that every SQL statement becomes a "Single Query" transaction and an implicit **commit** statement happens after every successful statement. This is referred to as **autocommit**, and can be turned on by the statement:

```
set autocommit on ;
```

See the **set** command in Chapter 2, "SQL Commands," for more information on **set autocommit**.

Database Procedures

Database procedures are a collection of statements managed as objects by ODT-DATA as part of the database definition. Procedures provide strong benefits for the user. They enhance performance by reducing the amount of communication between the application and the DBMS. They provide the database administrator (DBA) with an extra level of control over data access and modification. Additionally, one procedure can be used in many applications in a database, which reduces coding time.

Using Database Procedures

Procedures can be created or dropped in the SQL Terminal Monitor or within Embedded SQL. Procedures can only be executed from within Embedded SQL.

A procedure may include data manipulation statements, such as **select** or **insert**, as well as control flow statements, such as **if** and **while**, and the status statements, **message** and **return**.

When you create and use database procedures, there are several considerations to remember:

- Within a database procedure, all object references are resolved when a procedure is created. This means that if a procedure references a public table when it is created, the procedure always uses that table, even if executed by a user having a private table with an identical name.

- All referenced objects must exist at the time the procedure is created and when it is executed. Between the time of creation and the time of execution, you can modify, reorder, or drop and recreate objects such as tables and columns without affecting the procedure definition. If an object is redefined in a way that invalidates the procedure definition, then the definition must be dropped and recreated. An example of this is a column whose data type is changed from numeric to string.

- The procedure's query execution plan is created when the procedure is created. If the procedure is modified in a way that invalidates the plan, then the plan is recreated at the next invocation of the procedure.

The following is an example of a database procedure. This example, "move_emp," accepts as input an employee id number. The employee matching that id is moved from the "employee" table and added to the "emptrans" table. Both tables are inaccessible to users except through the procedure. When the procedure is invoked, the executing application passes a single integer parameter.

```
CREATE PROCEDURE move_emp (id INTEGER NOT NULL) AS
BEGIN
      INSERT INTO emptrans
            SELECT *
                  FROM employee
                  WHERE id = :id;
      DELETE FROM employee
            WHERE id = :id;
END;
```

Permissions on Procedures

A procedure is owned by the person who creates it. If the creator is the DBA, then the procedure is public and available to any user having the DBA's permission. A procedure created by any other user is private to that user. If the DBA and a user have identically named procedures, the user has access only to the private procedure.

Procedures provide the DBA with greater control over database access. The DBA can grant a user permission to execute a procedure even if the user has no direct access to the underlying tables. In this way, the DBA controls exactly what operations a user can perform on a database.

The DBA uses the following statement to grant permissions to users:

grant execute
on procedure *procedure_name* to *user_list*

Error Handling

Unless the procedure programmer provides explicit error handling mechanisms, either within the procedure itself or within the calling application, the default action when an error occurs is to continue to the next statement.

Database procedures make use of the control flow statements, **if** and **while**, and two built-in variables, "iirowcount" and "iierrornumber", to process errors. An application that invokes a database procedure must use the SQLCA to process errors occurring inside the database procedure. "iirowcount" and "iierrornumber" are only available within the database procedure. (Refer to the *ODT-DATA Embedded SQL User's Guide* for information about using the SQLCA.)

"iirowcount" is an integer that indicates the number of rows affected by the last executed SQL statement. If the statement was not a statement that affects rows or if an error occurred, then "iirowcount" is set to -1. If the statement was a row-affecting statement, but no rows were affected, then the value of "iirowcount" is set to 0. The initial value of "iirowcount" is 0.

"iierrornumber" is an integer that holds the error number associated with an error occurring during the execution of a statement. If no error occurs, the value of "iierrornumber" is set to 0. The error number is a positive number, equivalent to errorno in 4GL. The initial value of this variable is 0.

The execution of each statement sets the value of "iierrornumber" either to zero (no errors) or an error number. To check the execution status of any particular statement, "iierrornumber" must be examined immediately after the statement's execution.

Errors occurring in **if**, **while**, **message**, and **return** statements do not set *"iierrornumber"*. However, any errors that occur during the evaluation of the condition of an **if** or **while** statement terminate the procedure and return control to the calling application.

Message Handling

Database procedures, like embedded forms applications, use the **message** statement to display text on the screen while executing. It is possible to provide alternative instructions for message processing using the **whenever** statement within the Embedded SQL. Refer to the *ODT-DATA Embedded SQL User's Guide* for information about using the **whenever** statement and processing procedure messages.

Creating and Executing a Procedure

A database procedure can be created with Interactive SQL or within Embedded SQL. The syntax for the statement is:

[create] procedure *proc_name*
    [(*param_name* [=] *param_type* { , *param_name* [=] *param_type*})]
    =|**as**
  [*declare_section*]
begin
  statement_list
end;

where *proc_name* is the name of the procedure to be created and *param_name* is the name of the procedure parameter.

Procedure parameters are treated as local variables in the procedure body, although they have an initial value assigned when the procedure is invoked. You can also assign values to procedure parameters within the body of the procedure. (Local variables are discussed later.)

param_type is the procedure parameter's type. All types may have the NULL or DEFAULT clauses. For example, the following procedure fragment accepts three parameters, a non-null integer, a varying length string, and a date:

```
CREATE PROCEDURE eval_emp (id INTEGER NOT NULL)
     comment VARCHAR(100),
     meeting DATE NOT NULL) AS ...
```

The *declare_section* declares a list of local variables that can be referenced within the procedure. The syntax for this statement is:

declare
  var_name { ,*var_name*} [=] *var_type*;
  {*var_name* { ,*var_name*} [=] *var_type*;}

where *var_name* is the name of the local variable.

var_type is the type of the variable. Variable names must be unique within the procedure. If a variable is nullable, it is initialized to NULL. If a variable is not nullable, it is initialized to the default value.

You can substitute local variables and procedure parameters for any constant value in statements in the procedure body. A preceding colon (:) is only necessary if the referenced name could be misinterpreted as an SQL column name. For example, if a procedure parameter and a referenced column (in a procedure statement) have the same name, the referenced column name must be preceded by a semicolon. The following example illustrates this rule.

In this example, the procedure retrieves the name of an employee who matches an employee id. Both the employee id column and the procedure parameter are named "id." The colon in the where clause distinguishes the column from the parameter.

```
CREATE PROCEDURE name_of_emp (id INTEGER NOT NULL) AS
DECLARE
      name CHAR(50);
BEGIN
      SELECT fname + ' ' + lname
            INTO :name
            FROM employee
            WHERE id = :id;
      MESSAGE :name;
END;
```

The *statement_list* may include local variable assignments and any of the following statements:

insert
delete
update
commit
rollback
select
if
while
return
message

You cannot issue any data definition statements, such as **create table**, from inside a database procedure.

Refer to the statement summary in Chapter 2, "SQL Commands," for detailed information about the syntax of the **create procedure** statement.

Executing a Procedure

Procedures are invoked from within an Embedded SQL application. You cannot invoke a procedure interactively or from inside another procedure. The statement that invokes a procedure is **execute procedure**. Refer to the *ODT-DATA Embedded SQL User's Guide* for information about executing a database procedure.

Dropping a Procedure

Dropping a procedure removes the procedure's definition from the database. You must be the owner of a procedure to **drop** a procedure. Procedures may be dropped using Interactive SQL or within an Embedded SQL application. You cannot drop a procedure from inside another procedure.

The syntax of the statement is:

drop procedure *proc_name*

where *proc_name* is the name of the procedure you want to drop.

The statement takes effect immediately. Executions of the procedure in progress, invoked by other users, continue until they are completed. However, no additional references to the procedure are allowed.

Multi-Filesystem Databases

To accommodate large databases within a finite computer system, ODT-DATA enables users to locate the user tables of a single database on more than one filesystem.

Merely by establishing names for discrete areas of a given disk, an ODT-DATA system administrator can preserve the usefulness of an ODT-DATA database, even when it becomes extremely large.

ODT-DATA Locationnames and Areas

Locationnames are labels that denote subpaths to ODT-DATA directories. These labels are independent of the operating system. In the UNIX operating system, an area would be defined as a directory or subdirectory (for example, **/usr/cormac/new** or **../mydb/other**).

Each locationname maps to exactly one area; however, many different locationnames can map to the same area. Locationnames follow the ODT-DATA naming convention: they must begin with a letter and then must be alphanumeric. Underscores are allowed after the first letter, and the maximum length is 24 characters. The area designation can be up to 255 characters and must follow the syntax of the host operating system in directory names.

Locationnames and areas are specified with the **accessdb** command, described in Appendix B of *Administering ODT-DATA*.

Locationnames may be used in the **createdb** and **finddbs** utilities, as well as in the **create table, create index** and **modify** commands. If a locationname is not specified in a utility or SQL command, then the appropriate default is used. C language programs using SQL can be written in a manner independent of the operating system because all references to devices can be locationnames. Each installation has a set of default locationnames. These are **ii_database, ii_journal** and **ii_checkpoint**. These locationnames map to the environment variables II_DATABASE, II_JOURNAL and II_CHECKPOINT, respectively.

Assigning Database Tables to Single Areas

ODT-DATA, as mentioned previously, assigns a table or index in a database to a default area unless it is otherwise specified on the **create table** or **create index** statement. However, if disk space on the default filesystem that stores the database becomes too filled, the table can be relocated to another filesystem in the computer system.

The process of relocating a database's user tables to a different device requires three steps:

- Make sure there is a valid ODT-DATA directory for databases there.
- The ODT-DATA system administrator extends the database to the additional area(s) by assigning the requisite locationname(s). The ODT-DATA system administrator uses the **accessdb** command, described in *Administering ODT-DATA*, to accomplish the database extension.
- The ODT-DATA user relocates the user table to a new area using an SQL command. To move a table from one area to another, the **modify to reorganize** command, described in Chapter 2, has the syntax:

  modify *tablename* **to reorganize with**
  location = (*locationname*)

  The **modify** command, of course, requires that the user be the table's owner.

Multi-Location Tables

Tables and indexes may also be physically partitioned across multiple areas. A table may be assigned to multiple areas when it is created (using the **create table** or **create index** statement) by way of the **with location** = (*location-list*) clause. For example:

```
create table large (wide varchar (2000))
with location = (area1, area2, area3);
```

The specified areas must already exist. (See **accessdb** description in *Administering ODT-DATA*.)

Alternatively, a table may be spread over several areas using the **modify to reorganize** statement:

```
modify large to reorganize with location = (area1,
area2, area3);
```

A table, or part of a table, may be relocated to a corresponding area or set of areas by using the **modify to relocate** statement:

```
modify large to relocate
      with  oldlocation = (area1, area2, area3),
            newlocation = (area4, area5, area6);
```

or

```
modify small to relocate
      with  oldlocation = (area1),
            newlocation = (area2);
```

The difference between **modify to relocate** and **modify to reorganize** is that with the relocate option, the relocation is strictly physical, with the data from each area in the old location list being moved "as is" to the corresponding area in the newlocation list. For example:

```
modify medium to relocate with
      oldlocation = (area1, area2),
      newlocation = (area3, area4);
```

The data for table **large** in area1 is moved to area3 and the data in area2 is moved to area4. The number of areas in the oldlocation list must be equal to the number of areas in the newlocation list.

A portion of a table may be relocated by specifying only specific areas in the location lists; for example:

```
modify large to relocate with
      oldlocation = (area3),
      newlocation = (area5);
```

This only relocates the table's data that resides in area3, leaving area1 and area2 unchanged.

The **Modify to relocate** statement with only one area in the location lists is analogous to the **relocate** statement. The **relocate** statement continues to be supported, but does not extend to include multiple locations support, and eventually is undocumented.

With the **reorganize** option, the table is not just moved, but also reorganized. That is, a table that is spread across three areas can be reorganized to be spread across only two areas, or five areas. For the **reorganize** form of **modify**, it is not necessary (or allowed) to specify the old locations. The entire table is reorganized. The only parameter in the **with** clause that is accepted is the *location* = (*location* [,*location* ...]) clause.

The algorithm for spreading a table or index across multiple areas is very simple (that is, efficient) from an internal standpoint, but may be a bit confusing from an external point of view.

In this example, if a table is spread over three areas:

```
create table large (wide varchar(2000),
      with location = (area1, area2, area3);
```

As rows are added to the table, they are added to each area in 16 page (approximately 32 Kbytes) chunks. When the first 16 blocks are filled area1, the following 16 pages of data are put in area2. Then area3 starts to fill up 16 pages. Then it goes back to area1.

If it is not possible to allocate 16 full pages on an area when it is that area's turn to be filled, the table is out of space, even if there is plenty of room in the table's other areas.

Chapter 2
SQL Commands

SQL (Structured Query Language) consists of explicit, keyword commands that perform a range of functions for data definition, data manipulation, and database administration. This chapter presents each of these commands. Each section is devoted to a single command; within each section, subsections present the command name, syntax, description, and examples.

When consulting the material in this chapter and others in the *ODT-DATA SQL Reference Manual*, please remember that this manual is intended to provide the definitive description of ODT-DATA's functions. Chapter 3, "ODT-DATA Terminal Monitor," of this manual describes how to use these functions interactively by using the ODT-DATA SQL Terminal Monitor. For guidance in how to use ODT-DATA's functions within a host language program, please consult the *ODT-DATA Embedded SQL User's Guide* and the *ODT-DATA Embedded SQL Companion Guide for C*.

commit

Commits the current transaction.

Syntax

commit [work]

Description

This statement commits the current transaction. Once committed, the transaction cannot be aborted, and all changes it made become visible to all users by using the **select** statement. Once executed, the current transaction is terminated; a new one is automatically started just before the next SQL command. Any open cursors are closed.

The optional word **work** has no effect. It is included for compatibility with other versions of SQL.

The statement analogous to **commit** in pre-6.0 versions of INGRES is **end transaction.**

copy

Copies data into/from a table from/into a file.

Syntax

copy table *tablename* (*columnname* = *format* [**with null** (*value*)]
{, *columnname* = *format* [**with null**(*value*)]}) **into** | **from** '*filename*'
[*with-clause*]

A *with-clause* consists of the word **with** followed by a comma-separated list of any number of the following items:

on_error = terminate | continue
error_count = *n*
rollback = enabled | disabled
log = '*filename*'

Description

The **copy** command moves data between ODT-DATA tables and standard files. **Table** is a keyword and must be typed as shown. *Tablename* is the name of an existing table. In general, *columnname* identifies a column in the table. *Format* indicates the storage format for the column's values in the file.

The **with null** *value* clause allows you to specify the value NULLs have in the target table. If you specify **with null** but do not specify a value, you get an ODT-DATA binary data value. It has non-printable characters as part of the data representation because every data value has a trailing byte specifying whether the value is NULL. Therefore, you must specify the value in a **with null** clause when using the **c0**, **vchar(0)**, **char(0)**, and **varchar(0)** data types.

If you do specify *value* in the **with null** clause, NULL values are represented by the value specified and there is no byte to represent the NULL. So be sure the file you are copying does not contain the value specified by *value* as a legitimate non-NULL value (that is, do not set *value* to "35" for a column of people's ages; use "-1" instead).

To write a file, use the **into** *filename* form of the **copy** command. To copy data from a file to an ODT-DATA table, use the **from** *filename* form of the command. *Filename* must be enclosed in single quotation marks. *Filename* is assumed to be in the front-end process's current directory unless the full pathname is specified.

The **with on_error** clause lets you specify that **copy** should not be terminated due to an error processing a row. If **continue** is set, the front end does not terminate the copy if it encounters errors converting between row and file format. The **copy** continues to be terminated by errors reading or writing the copy file, back-end errors, or other errors that signify a problem with copy processing in general rather than a problem confined to a single row. If **terminate** is set, **copy** terminates at the first conversion error. **Terminate** is the default.

If an error is encountered while **on_error** is set to **continue**, a warning message corresponding to the type of error is printed and that row is skipped. When the **copy** is finished, the following message is displayed:

```
COPY: Warning: Copy completed with %d warnings. %d
rows successfully copied.
```

The **error_count** = *n* clause instructs **copy** to terminate after *n* errors instead of just one. This clause is meaningful only if **on_error = terminate** is set. It is an error to specify an **error_count** if **on_error = continue** is specified. The default **error_count** is 1.

The **with rollback** clause lets you specify whether rows appended to the database during a copy should be backed out if the **copy** is terminated due to an error. This option is meaningful only with **copy from**, because rows are never backed out of the copy file if **copy into** is terminated.

If **rollback = enabled** is specified, all rows added to the database during a **copy** statement are backed out if the **copy** is terminated abnormally. This is the default setting.

The **rollback = disabled** option does *not* mean that a transaction cannot be rolled back. Database engine internal errors that may indicate data corruption still causes back out, and rows are still not committed until the transaction is complete. This option means only that rows are automatically backed out if an error occurs.

There are two error messages that indicate that **copy** has been interrupted abnormally due to an error or interrupt. If you are running **copy from** and either **rollback = enabled** is set or the termination is due to a database engine error, you will get the error message:

```
COPY: Copy has been aborted
```

Any other abnormal termination produces the error message:

```
COPY: Copy terminated abnormally, %d rows successfully
copied.
```

The **with log** *'filename'* clause lets you send rows that **copy** cannot process to the file specified. In a **copy from**, rows written to the log file are exactly the same as they were in the copy file. In a **copy into**, they are in the format of the rows in the database.

This option is especially useful in a **copy from** statement when the **on_error = continue** option is set. In this case, the **copy** continues to completion even though there may be rows in the copy file that cannot be processed. Warnings are given for each row that cannot be appended to the database, and those rows are written to the log file. You can then edit the log file and fix up the rows to load them into the database.

If an error occurs opening the log file, the **copy** halts. The log file is opened prior to the start of data transfer, so this happens immediately.

If an error occurs writing to the log file, a warning is given and the **copy** continues.

If the specified log file already exists, it is overwritten with the new values or truncated if the new **copy** statement produces no bad rows.

On a **copy from** a file to a table, the table can have an index, but performance is much slower than for the same table without an index. You must have update permission on the table, and the table must be updatable (that is, it cannot be an index or system table). You cannot use the **copy** command to add data to a view. If you **copy** to add rows to a table that has integrity constraints, the integrity constraints are ignored.

To execute a **copy into** a file, either you must be the owner of the table, or the table must have retrieve permission for all users (or all permissions for all users).

The formats allowed in this mode are the following:

integer1, smallint, integer	Values are stored as integers of 1-, 2- or 4-byte length in the file.
float4, float	Values are stored as floating-point numbers (either single or double precision) in the file.
c1,...,2000 **char(1),...,char(2000)**	Values are stored as fixed-length strings of type c or **char**.
vchar(1),...,vchar(2000) **varchar(1),...,varchar(2000)**	Values are stored as fixed-length strings of type **vchar** or **varchar**.
c0, char(0)	Variable length character string (any data type).

vchar(0), varchar(0)	Variable length **vchar** string (any data type).
d0,d1,...,d255	Dummy column of variable (d0) or fixed (d1,...,d255) length (that is, skip it).
date	Values stored as internal ODT-DATA date value.
money	Values stored as internal ODT-DATA money value.

Corresponding columns in the table and their entries in the file need not be of the same type or length. For example, most applications read and write numeric data from files stored in character format and, therefore, primarily use the **c0** or **vchar(0)** type format. The **copy** command converts as necessary. When converting anything except character to character, copy mode checks for overflow. When converting from character to character, the **copy** command pads character strings with blanks or truncates strings on the right, as necessary.

The column names should be ordered according to how they are to appear in the file. Columns are matched according to name. Thus the order of the columns in the table and the file need not be the same.

The **copy** command provides for variable length strings and dummy columns. The action taken depends on whether it is a **copy into** or a **copy from** command. Delimiters for variable length strings and dummy columns can be selected from the following list:

nl	newline character
tab	tab character
sp	space
nul or **null**	NULL character
comma	comma
colon	colon
dash	dash
lparen	left parenthesis
rparen	right parenthesis
x	any single character *x* (excluding digits, that is, **0, 1, 2, 3, 4, 5, 6, 7, 8,,** or **9**)

In the file, the special meaning of any delimiter can be suspended by preceding the delimiter with a backslash (\), unless the field format is **vchar(0)***delim*.

Copying from a File into a Table

When copying data from a file into a table, columns in the ODT-DATA table that are not assigned values from the file are assigned the default NULL values of zero for numeric columns and blanks for character columns. When copying data in this direction, the following special meanings apply:

c0*delim*, **vchar(0)***delim or* **char(0)***delim*

Values in the file are variable length character strings terminated by the *delim* delimiter. If *delim* is not specified, the first comma, tab, or newline encountered terminates the string. The delimiter is not copied. For example:

pnum=c0	String ending in comma, tab or nl.
pnum=vchar(0)nl	String ending in nl.
pnum=c0nl	String ending in nl.
pnum=c0sp	String ending in a space.
pnum='c0Z'	String ending in the "Z" character.
pnum=vchar(0)'Z'	String ending in the "Z" character.
pnum='c0%'	String ending in the "%" character. A string in the file can contain the delimiter by preceding it with a backslash character (\), but only if the format is **c0***delim*. For example: name = c0 When using this, the string "Blow\, Joe" is accepted into the column as "Blow, Joe."

d0*delim*

Values in the file are variable-length character strings delimited by *delim*. Each string is read and discarded. The delimiter rules are identical for **c0** and **d0**. The column name is ignored.

d1,d2,...,d255	Values in the file are fixed-length character strings. Each string is read and discarded. The column name is ignored.
vchar(1),..., vchar(2000)	Values in the file are fixed-length **vchar** strings. The field must be padded with NULL characters to the given length.
varchar(0)	Values in the file are variable-length **varchar** strings preceded by a 2-byte length specifier.

When copying from a fixed format file, be sure to take into account the newline characters at the end of each line because there is no requirement that the rows you read from the file correspond to the records in the file. For example, if you have a table called "employee" containing the columns "name," "age," and "department," and a text file containing employee data in a fixed format, such as

```
Jones,J.^^^^^32^^^Anytown,USA^^^toy
Smith,P.^^^^^41^^^New York,NY^^^admin
```

where "^" is a blank space.

A valid **copy** statement would be something like:

```
copy table employee (name=c12, age=c3, xxx=d17,
     department=c0nl) from ... ;
```

Note that the dummy column name "xxx," which is not in the table, is an arbitrary name for the skipped field from the file. The name itself has no particular meaning. The last field in the fixed field file, in this case "department," is most conveniently specified as **c0nl**. This instructs ODT-DATA to read the remainder of the line into the "department" column of the table.

Note that the format indicators in the **copy from** command should describe how values are represented in the file. This is not necessarily the same format as the corresponding table column. For example, the file record might contain a numeric field holding a string of ASCII characters, such as "1927.63," which would be converted on input and stored in the ODT-DATA table in a column of type float. In this case, the **copy** command should describe the field as a **c** format item, not **float**.

Finally, note that copying from a file into an empty, non-journaled table without indexes runs significantly faster than copying into a table that contains one or more rows, is journaled, or has indexes. The copy is fastest when the table is in the **heap** storage structure.

Copying Data to a File

When the direction is **into**, **copy** transfers data into the file from the table. If the file already exists, it is overwritten, if allowed by the UNIX environment.

When copying in this direction, the following special meanings apply:

c0, char(0)

The column is converted to a fixed-length character string and written into the file. For character columns, the length is the same as the column length. For numeric columns, the standard ODT-DATA conversions take place as specified by the **-i**, **-f** and **-c** flags. (See the **sql** command in Chapter 4, "ODT-DATA Operating System Commands.")

varchar(0)

The column is converted to **varchar** and written as a variable length string preceded by a 2-byte length specifier.

c0*delim*, **char(0)***delim*

The column is converted according to the rules for **c0** above. The one-character delimiter is inserted immediately after the column.

NOTE: For numeric columns, **c0sp** and **char(0)sp** are not meaningful and results in input errors, because the data are converted by right justification and blankfill.

vchar(0)*delim*

The column is converted to a **vchar** string and written into the file. The one-character delimiter is inserted immediately after the column. For **c** and **char** columns, the length is the same as the column length. For **vchar** and **varchar** columns, the length varies according to the number of characters in each text value. For numeric columns, the standard ODT-DATA conversions take place as specified by the **-i** and **-f** flags. (See the **sql** command in Chapter 4, "ODT-DATA Operating System Commands."

NOTE: For numeric columns, **vchar(0)sp** is not meaningful and results in input errors, because the data are converted by right justification and blankfill.

vchar(1),...,vchar(2000)	The column is converted to a **vchar** string and written into the file, according to the rules for **vchar(0)** *delim*. No delimiter is used. If necessary, the column is padded with NULL characters to the given length.
d1,d2,...,d255	The column name is taken to be the name of the delimiter. It is written into the file once for d1, twice for d2, and so forth.
d0	This format is ignored on a **copy into** command.
d0*delim*	The *delim* is written into the file. The column name is ignored.
varchar(0)	The variable-length-only specifier and data are written to the file.

Note that arbitrary delimiters can be specified independently of columns on a **copy into** command. If you want to specify a newline character at the end of a line, include "nl=d1" at the end of the list of columns. This alerts ODT-DATA to add one (d1) newline (nl) character. Do not confuse "l" (lowercase L) and "1" (number one).

If no columns appear in the **copy** command (that is, **copy table** *tablename* () **into**|**from** *filename*), then the **copy** command automatically performs a bulk copy of all columns, using the order and formats of the columns in the table. This is provided as a convenient shorthand notation for copying and restoring entire tables.

In the "Examples" section later in this chapter, the first two examples illustrate different ways of representing numeric data in a file. In the first example, several fields are represented in 2-byte integer format, and "sal" is represented as a 4-byte floating-point item. These items would not be readable as characters with the text editor. The **copy** command loads them into ODT-DATA table columns, which may or may not have the same format as the file data.

The second example copies some of the same data out of the "employee" table into a file. This time, all items are written as character data. This means, for instance, that "sal" would be converted from its format in the ODT-DATA table (say, **float4** or **float**) to ASCII characters in the result file.

Performance of Copying from a File into a Table

Copying from a file into a non-journaled **heap** table without secondary indexes runs significantly faster than copying into a **btree, isam,** or **hash** table, or one that is journaled or has secondary indexes.

For example, consider the following two queries. The first runs more slowly because the table's **btree** index must be dynamically maintained as data are copied from the external file into the table:

```
CREATE TABLE employee (name vchar(12), age integer2,
      department vchar(8) ) ;
MODIFY employee TO btree ON name ;
COPY TABLE employee (name=c12, age=c3, xxx=d17,
      department=c0nl) FROM ... ;
```

The following query, on the other hand, runs more quickly because the "employee" table is a heap while data are copied and ODT-DATA, therefore, does not need to maintain an index structure for the table during the copy operation. After the **copy** statement is complete, the table is modified to **btree:**

```
CREATE TABLE employee (name vchar(12), age integer2,
      department vchar(8) ) ;
COPY TABLE employee (name=c12, age=c3, xxx=d17,
      department=c0nl) FROM ... ;
MODIFY employee TO btree ON name ;
```

Depending on the initial size of the database table and the amount of data to be copied from the external file, it may be faster to modify the database table to **heap** before copying data into it. For example, if "departments" is an existing table that is **btree** on column "department," it may be faster to copy with the first of the following two scripts:

```
MODIFY departments TO heap ;
COPY TABLE departments (department=c8, ... ) FROM ... ;
MODIFY departments TO btree ON department ;  /* restore
      original structure */
```

This second copy script, below, may run more slowly because it requires ODT-DATA to maintain the index structure on the "departments" table during the copy operation:

```
COPY TABLE departments (department=c8, ... ) FROM ... ;
```

As a general rule, if the external file contains more rows than the database table, then you may get better performance by modifying **to heap** before doing the copy and then modifying to the correct structure when the **copy** is complete. Note that if the database table is empty, it is nearly always better to modify to **heap** before doing the copy.

Effect of Table Structure on Copy from Performance

Isam Avoid copying large amounts of data into a database table that has an **isam** structure (modify table to **heap** first, as described earlier). It is particularly dangerous, from a performance standpoint, to copy into an empty **isam** table.

The index structure of an **isam** table is fixed (as opposed to **btree**, whose index structure grows dynamically as data are added to the table) and therefore does not grow as you add data; the net result is overflow chains that can significantly degrade performance.

Hash You can get good performance copying data into a **hash** structure, If the **hash** structure has been preallocated with enough space for the new data, and if many rows do not **hash** to the same page and produce overflow. The **minpages** parameter is used to preallocate space in **hash** tables.

Copying into a **heap** structure gives the best performance. Use of the **btree** structure is not as fast as **heap** because the index structure must be maintained, but should be faster than **isam** because of the lack of long overflow chains. The **hash** structure is will be fast if the table has enough empty space to hold the new data.

Examples

Copy data into the "employee" table.

```
copy table employee (eno=integer2, ename=c10,
age=integer2,
      job=integer2, sal=float4, dept=integer2, xxx=d1)
      from '/usr/mydir/files/myfile.in';
```

Copy employee names, numbers, and salaries into a file, inserting commas and newline characters so that the file can be printed or edited.

```
copy table employee (ename=c0, comma=d1, eno=c0, comma=d1,
      sal=c0, nl=d1)
      into '/usr/mydir/files/mfile.out';
```

or

```
copy table employee (ename=c0comma, eno=c0comma, sal= c0nl)
      into '/usr/mydir/files/mfile.out';
```

Bulk copy the "employee" table into a file.

```
copy table employee () into
'/usr/mydir/files/ourfile.dat';
```

Bulk copy the "employee" table from a file.

```
copy table employee ()
      from '/usr/mydir/another.fil';
```

create index

Creates an index on an existing base table.

Syntax

create [**unique**] **index** *indexname* **on** *tablename*
(*columnname* [**asc** | **desc**] {,*columnname* [**asc** | **desc**]})
[*with-clause*]

A *with-clause* consists of the word **with** followed by a comma-separated list of any of the following items:

structure = **cbtree** |**btree** | **cisam** | **isam** | **chash** | **hash**
key = (*columnlist*)
fillfactor = *n*
minpages = *n*
maxpages = *n*
leaffill= *n*
nonleaffill= *n*
location = (*locationname* ...)

Description

The **create index** command creates an index on an existing base table. The index key is constructed of columns from the specified table in the order given. A maximum of 32 *columnnames* may be specified per index, but you can build any number of indexes for a table. Only the owner of a table is allowed to create indexes on that table.

asc and **desc** specify ascending and descending sort sequence, respectively. **asc** is the default. **btree** and **isam** indexes are always kept in ascending order, even if the **desc** option is specified.

If **key**=(*column list*) is specified, the columns in *column list* must be an ordered subset of the columns specified in the index definition. In addition, they must be the leading columns in the index definition. For example, an index defined on columns *a*, *b*, *c*, and *d* may be keyed on *a*, or *ab*, or *abc*, or *abcd*. (The default is *abcd* if the **key** clause is omitted.)

The **fillfactor** specifies the percentage (from 1 to 100) of each primary data page that should be filled with rows, under ideal conditions. The **fillfactor** may be used with **isam**, **cisam**, **hash**, **chash**, **btree**, and **cbtree**. When creating a table with storage structure **btree** or **cbtree**, **nonleaffill** determines the percentage of each index page to fill. Care should be taken when specifying large fillfactors because a non-uniform distribution of key values could later result in overflow pages and thus degrade access performance for the table.

minpages specifies the minimum number of primary pages a **hash** or **chash** table must have. **maxpages** specifies the maximum number of primary pages a **hash** or **chash** table may have. **minpages** and **maxpages** must be at least one. If both **minpages** and **maxpages** are specified in a **create index** command, **minpages** cannot exceed **maxpages**.

Default values for **fillfactor**, **minpages**, and **maxpages** are as follows:

	fillfactor	minpages	maxpages
hash	50	10	no limit
chash	75	1	no limit
cisam	100	NA	NA
btree	80	NA	NA
cbtree	100	NA	NA

The **leaffill** parameter of the **create index** command applies only to tables stored in **btree** and **cbtree** structures. The **leaffill** parameter specifies percentages to fill each index page for a **btree** or **cbtree** table.

The **leaffill** specifies a percentage *n*, where *n* ranges from 1 to 100, and its percentage specifies how much each index page should be filled at the time the table is modified to **btree** or **cbtree**. This parameter contrasts with the **fillfactor** parameter, which specifies the percentage occupancy of data pages (not index pages) when a table is converted to **btree** or **cbtree**.

The **leaffill** parameter allows you to control locking contention in **btree** and **cbtree** index pages. By retaining a percentage of open space on these index pages, more concurrent users can access the **btree** without contention while their queries descend the index tree. Note, however, that you must strike a balance between preserving space in index pages and creating a greater number of index pages; more levels of index pages require more I/O to locate a data row.

Default value for **leaffill** is 60 (percent). This default applies to both **btree** and **cbtree** indexes.

The *Locationname* refers to the area(s) on which the new index is created. The *locationname(s)* must be defined on the system, and the database must have been extended to the corresponding area(s). If no *locationname* is specified, the default area for the database is assumed. If multiple *locationnames* are specified, the index is physically partitioned across the areas. (See Chapter 1 for more information about ODT-DATA locationnames and areas and multi-location tables.)

To maintain the integrity of the index, users are not permitted to update indexes directly. However, whenever a table is changed, its indexes are automatically updated by the system. Indexes may be modified to increase even further the access efficiency of the table. When an index is first created, it is automatically modified to an **isam** storage structure on all its columns. If this structure is undesirable, you may override the default structure with the **-n** flag (see the **sql** command in Chapter 4, "ODT-DATA Operating System Commands"), by entering a **modify** command directly, or by specifying the modify parameters in the **with** clause of the **create index** command.

Once created, an index improves query processing "silently." That is, if you retrieve data from a table based on an indexed column, you need not indicate to ODT-DATA that it should consult the index; ODT-DATA automatically uses indexes to accelerate query processing once the indexes are created.

If a **modify** or **drop** command is used on a table, all indexes on that table are destroyed. Note also that the **modify** and **drop** commands can be executed directly on an index.

You are not allowed to create indexes on system tables. No more than 32 columns may appear in the index key.

Examples

Create an index called "x" for the columns "ename" and "age" on table "employee":

```
create index x on employee (ename, age);
```

Create an index called "ename" and have it located on the area referred to by the locationname "remote":

```
create index ename on employee (ename, age) with
location = (remote);
```

create integrity

Defines integrity constraints on a base table.

Syntax

create integrity on *tablename* [*corr_name*] **is** *search_condition*

Description

The **create integrity** command creates an integrity constraint for the specified base table. After the constraint is defined, all updates to the table must satisfy the specified search condition. The search condition must be true for every existing row in the table when the **create integrity** statement is issued; if it is not true, a diagnostic is issued, and the integrity constraint is rejected.

In the current implementation, integrity constraints that are violated are not specifically flagged. Updates that violate any integrity constraints are simply not performed.

The search condition must not involve any tables (or their correlation names) other than the one specified in the **on** clause. The search condition must also not contain a subselect.

The **create integrity** statement may be issued only by the table owner.

Examples

Make sure that all employee salaries are not less than 6000.

```
create integrity on employee is salary = 6000;
```

create procedure

Creates a named database procedure definition.

Syntax

[create] procedure *proc_name*
    [(*param_name* [=] *param_type* { , *param_name* [=] *param_type*})]
    =las
  [*declare_section*]
begin
  statement { ; *statement*}[;]
end;

Description

The **create procedure** statement creates a named database procedure definition that is managed as a named object by ODT-DATA as part of the database.

The *proc_name* is the name of the procedure. The name must be a legal ODT-DATA SQL name (see Chapter 1, "SQL Syntax").

The *param_name* is the formal name of the procedure parameter.

The *param_type* is the procedure parameter's type. The *Param_type* can be any of the ODT-DATA types (see Chapter 1, "SQL Syntax"). All types may have the NULL or DEFAULT clauses.

The *declare_section* declares a list of local variables that you can reference in the procedure body. The syntax for this section is:

declare
  var_name { , *var_name*} [=] *var_type*;
  {*var_name* { , *var_name*} [=] *var_type*;}

Refer to the summary of the **declare** command in this manual for full information about this syntax.

The *Statements* may include local variable assignments and any of the following:

commit	**return**
delete	**rollback**
if	**select**
insert	**update**
message	**while**

A procedure cannot contain any data definition statements, such as **create table**, nor may it **create**, **drop**, or **execute** another procedure. Additionally, unlike the Embedded SQL versions of some of these statements, you cannot use the **repeated** clause in a statement in the procedure body. (Using the procedure itself provides the same performance benefits as the **repeated** clause.)

select statements inside a procedure must assign their results to local variables. Also, they can return only a single row of data. If more rows are returned, no error is issued, but only the first row retrieved is in the result variables.

Both procedure parameters and local variables can be used in place of any constant value in statements in the procedure body. Procedure parameters are treated as local variables inside the procedure body, although they have an initial value assigned when the procedure is invoked. Preceding colons (:) are only necessary if the referenced name could be interpreted to refer to more than one object.

Local variable assignments use the "=" or ":=" operator.

All statements, except a statement preceding an **end**, **endif**, or **endwhile**, must be terminated by a semicolon.

You can replace the keywords **begin** and **end** with braces {}, but the terminating semicolon must follow the closing brace if another statement is entered interactively after the **create procedure** statement and before committing the transactions.

Examples

This database procedure, "mark_emp," accepts as input an employee id number and a label string. The employee matching that id is labeled and an indication is returned.

```
CREATE PROCEDURE mark_emp
      (id INTEGER NOT NULL, label VARCHAR(100)) AS
BEGIN
      UPDATE employee
         SET comment = :label
         WHERE id = :id;
      IF iirowcount =1 THEN
         MESSAGE 'Employee was marked'       ;
         COMMIT;
         RETURN 1;
      ELSE
         MESSAGE 'Employee was not marked - record error'   ;
         ROLLBACK;
         RETURN 0;
      ENDIF;
END;
```

In this next example, the database procedure "add_n_rows" accepts as input a label, a base number, and a number of rows. It inserts the specified number of rows in to the table "blocks," starting from the base number. If an error occurs, then the procedure terminates and the current row number is returned.

```
CREATE PROCEDURE add_n_rows
      (base INTEGER, n INTEGER, label VARCHAR(100)) AS
DECLARE
      limit INTEGER;
      err INTEGER;
BEGIN
      limit = base + n;
      err = 0;
      WHILE (base < limit) AND (err = 0) DO
            insert into blocks VALUES (:label, :base);
            IF iierornumber > 0 THEN
                  err = 1;
            ELSE
                  base = base + 1;
            ENDIF;
      ENDWHILE;
      RETURN :base;
END;
```

create table

Creates a new base table.

Syntax

create table *tablename*
(*columnname format* {, *columnname format*})
[*with-clause*]

create table *tablename*
[(*columnname* {, *columnname*})]
as *subselect*
[with-clause]

A *with-clause* consists of the word **with** followed by a comma-separated list of any number of the following items:

location = (*locationname* ...)
[no]journaling
[no]duplicates

For the syntax of *subselect*, see the "select" section later in this chapter.

Description

The **create table** command creates a new base table. The table is owned by the user issuing the **create table** command and, by default, has no expiration date. However, the expiration date can be changed by using the **save** statement. At this time there is no means by which expired tables can be destroyed; they are not destroyed automatically.

The created table has a name as specified by the indicated *tablename*, with columns named as specified by the indicated *columnnames*. If no *columnnames* are specified, then an **as** clause must be specified, and the table inherits *columnnames* in the obvious way from the **select** clause of the subselect in that **as** clause. Even if an **as** clause is specified, *columnnames* **must** also be specified if two or more columns of the table would otherwise have the same name. If there is an expression in the **select** clause of the subselect, columnnames are assigned randomly.

The *formats* specify the data type of a column as well as how unspecified values are to be handled. The *formats* must be omitted if an **as** clause is specified (in which case the *formats* are inherited in the obvious way from the expressions in the **select** clause of the subselect). When an **as** clause is *not* specified, *formats* must be included.

The *formats* have the syntax:

datatype [**not null** [**with** | **not default**] | **with null**]

The **with|not null** clause determines what happens during an **insert** to a field for which no value is specified. There are three possible settings for this clause:

with null
not null with default
not null not default

The **with null** clause means that a field into which no data has been inserted is marked as having no value. The **not null with default** clause means the default value ("0" or blank) is entered. The **not null not default** clause means an error condition is created when the insert is attempted.

If no **with|not null** clause is specified, **with null** is assumed. If **not null** alone is specified, **not null not default** is assumed.

Pre-6.0 versions of INGRES entered a field value of "0" or blank if no value was specified. This corresponds to **not null with default**.

A table can have a maximum of 127 columns and can be a maximum of 2008 bytes wide. Note that a **varchar** or **vchar** column requires two more bytes than the value specified in the format; for example, **varchar**(20) stores values up to 20 characters long, but requires 22 bytes in storage. A table cannot be defined to have a name beginning with "ii".

If an **as** clause is specified, the table is populated with the set of rows resulting from execution of the specified subselect; otherwise the table is created empty. If **as** is specified, the new table is created with the storage structure defined by the most recent **set result_structure** command within the session (see the **modify** command); the default is compressed heap. If **as** is not specified, the new table is created as heap.

The *locationname* refers to the area(s) (see Chapter 1, "SQL Syntax") on which the new table is created. The *locationname(s)* must be defined on the system, and the database must have been extended to the corresponding area(s). If no *locationname* is specified, the default area for the database is assumed. If multiple *locationnames* are specified, the table is physically partitioned across the areas, as described in the section of Chapter 1 titled "Multiple-Location Tables." (Please see Appendix B, "Authorizing User Access to ODT-DATA and Databases," in *Administering ODT-DATA*.)

If **with journaling** is set, journaling occurs for the table only if journaling is enabled for the database as a whole by using the **ckpdb** command (see Chapter 4, "ODT-DATA Operating System Commands"). Enabling journaling causes ODT-DATA to keep a record of all changes to the table (**inserts, updates,** and **deletes**) in the journal for the containing database, and thus allows the ODT-DATA recovery system to reconstruct the table after a disk crash. It also allows an audit trail to be built for the table, and thus can be useful for monitoring updates or for maintaining change histories.

It is not necessary to enable journaling to recover from operating system or ODT-DATA crashes; this kind of recovery is handled by normal query processing.

The **duplicates/noduplicates** option does not affect a table created as a heap; this type of storage structure allows duplicate rows regardless of the setting of this option. The **duplicates/ noduplicates** setting affects only those tables created as, or later modified to be, structures other than heap.

Additionally, this setting can be overridden by specifying a unique key for a table by using the **modify** command. See the **modify** command for more information about table structures with unique keys.

Examples

Create the "employee" table with columns "eno," "ename," "age," "job," "salary," and "dept," with journaling enabled.

```
create table     employee
     (eno        smallint,
     ename       varchar(20) not null with default,
     age         integer1,
     job         smallint,
     salary      float4,
     dept        smallint)
     with        journaling;
```

Create a table with some other data types.

```
create table      debts
      (acct       varchar(20) not null not default,
      owes        money,
      due         date not null with default);
```

Create a table listing employee numbers for employees who make more than the average salary.

```
create table      highincome
      as select   eno
      from        employee
      where       salary  all
                  (select     avg (salary)
                  from  employee);
```

Create a table that spans two locations.

```
create table      emp as
      select eno from employee
            which location = (location1, location2);
```

create view

Defines a virtual table.

Syntax

create view *view_name* [(*columnname* {, *columnname*})] **as** *subselect*
[**with check option**]

The syntax of the *subselect* is described in the **select** command summary later in this chapter.

Description

The syntax of the **create view** statement is very similar to that of the **as** form of **create table**. However, data are not retrieved. Instead, the definition is stored. When the *view_name* is later used in an SQL statement, the statement operates on the tables that are used to define the view, called the base tables.

All selects on views are fully supported. Simply use a *view_name* in place of a *tablename* in any SQL retrieval.

Only a limited set of updates on views is supported because of anomalies that can occur. Generally, no updates are supported on views that have more than one base table. No updates are allowed on columns that are in the qualification of the view definition, or on any column whose source is not a simple column name (that is, set functions or computations). Updates not allowed on view columns may, of course, be performed on columns in the base tables.

In general, updates are supported only if it can be guaranteed (without looking at the actual data) that the result of updating the view is identical to updating the corresponding base table.

Although a person who defines a view need not own all tables upon which a view is based, use of the view is restricted to those who have all necessary permissions to the base tables. Permissions may be granted by the DBA using the **grant** *priv* command.

When a table used in the definition of a view is dropped, the view is also dropped.

The **with check option** clause causes each insert and update done using the view to be checked to make sure that a row inserted through the view actually appears in the view, and that an update of a row in the view does not cause the row to disappear from the view.

For example, consider the following two commands:

```
create view       v
     as select    *
     from         t
     where        c  10

update      v
set         c = 5
```

Once "c" is set to the value "5" when "t" is updated, the updated rows are no longer in the view. If the view had been created with the **with check option**, the update would not be allowed.

By default, the **with check option** is not set. All views created in pre-6.0 releases of Ingres had the **check option** attribute automatically set.

Example

Define a view of employee data including names, salaries, and managers' names.

```
create view       empdpt (ename, sal, dname)
     as select    employee.name, employee.salary,
                        dept.name
     from         employee, dept
     where        employee.mgr = dept.mgr;
```

declare

Declares a list of local variables for use in a database procedure.

Syntax

declare
var_name { , *var_name*} [=] *var_type*;
{*var_name* { , *var_name*} [=] *var_type*;}

Description

This statement is used only in a database procedure definition, to **declare** a list of local variables for use in the procedure. The statement is optional and, if used, is placed before the **begin** clause.

The *var_name* is the name of the local variable. Variable names must be unique within the procedure body. Also, a variable and a procedure parameter may not have the same name.

The *var_type* is the type of the variable. A local variable type may be any of the ODT-DATA SQL data types. Nullable variables are initialized to NULL; non-nullable variables are initialized to the default value. For example, a non-nullable floating-point variable is initialized to 0.0. Any non-nullable variables declared without an explicit default value are initialized to the ODT-DATA default value.

Example

This procedure fragment demonstrates some declarations and uses of local variables. Note that some of these statements cause an error.

```
CREATE PROCEDURE variables (vmny MONEY NOT NULL) AS
DECLARE
        vi4     INTEGER NOT NULL;
        vf8     FLOAT;
        vc11    CHAR(11) NOT NULL;
        vdt     DATE;
BEGIN
        vi4 = 1234;
        vf8 = NULL;
        vc11 = '26-jun-1957';
```

```
        SELECT  DATE(:vc11)  INTO  :vdt;
        vc11  =  vmny;          -data  type  conversion  error
        vmny  =  vf8;           -null  to  non-null  conversion  error
        RETURN  :vi4;
END;
```

delete

Deletes rows from a table.

Syntax

delete from *tablename* [*corr_name*]
[**where** *search_condition*]

Description

The **delete** command removes rows that satisfy *search_condition* from the specified table. If the **where** clause is omitted, the command deletes all rows in the table. The result is a valid but empty table.

Note that **delete** does not automatically recover the space in a table left by the deleted rows. However, if you add new rows later, the empty space may be re-used. If you delete many rows from a table, you may want to run the **modify** command to recover the lost space. You can specify any storage structure and still recover the empty space. In particular, if you want to delete all rows from a table, you can use the special **modify** *tablename* **to truncated** to delete all rows and recover the space at the same time. (See the **modify** command in this chapter for more information.)

To delete rows from a table, you must either be its owner or have **select** and **delete** permission on the table.

Example

Remove all employees who make over $35,000.

```
delete from employee where salary > 35000;
```

drop

Destroys one or more tables, indexes, or views.

Syntax

drop *tablename*|*indexname*|*viewname* {, *tablename*|*indexname*|*viewname*}

Alternate forms:

drop table *tablename* {,*tablename*}
drop index *indexname* {,*indexname*}
drop view *viewname* {,*viewname*}

Description

The specified tables, indexes, and views are removed from the database. Only the owner of the view or table is allowed to drop it. Likewise, only the owner of an indexed table is allowed to drop an index.

If a table is dropped, any indexes and views defined on that table are automatically dropped too.

If the **drop table**|**view**|**index** form is used, the object name is checked to be sure it is the "right" type. So **drop table** *viewname* is not permitted. Similarly, **drop table** *tablename*, *viewname* drops the table and not the view.

If the generic **drop** form is used, the objectnames may be any mixture of the three types.

Example

Drop the "employee" and "dept" tables.

```
drop employee, dept;
```

drop integrity

Destroys one or more integrity constraints.

Syntax

drop integrity on *tablename integer* {, *integer*}

The keyword **all** can appear in place of the list of integers.

Description

The specified integrity constraints are removed from the database. The constraints are specified by integers whose values can be obtained by using the **help integrity** command. Alternatively, the keyword **all** can be specified, meaning all integrity constraints currently defined for the table in question.

Only the owner of the table to which a given constraint applies is allowed to drop that constraint.

Example

Drop integrity constraints 0, 4, and 5 on "job."

```
drop integrity on job 0, 4, 5;
```

drop permit

Destroys one or more permissions.

Syntax

For tables and views:

drop permit on *tablename integer* {, *integer*}

For procedures:

drop permit on procedure *proc_name*
integer | **all**

Description

The specified permissions are removed from the database. The permissions are specified by integers whose values can be obtained by using the **help permit** command. Alternatively, the keyword **all** can be specified, meaning all permissions currently defined for the table or procedure in question.

Only the owner of the table to which a given permission applies is allowed to drop that permission.

Examples

Drop all permissions on "job."

```
DROP PERMIT ON job ALL;
```

Drop the second permission on procedure "AddEmp."

```
DROP PERMIT ON PROCEDURE AddEmp 2;
```

drop procedure

Removes a procedure definition from the database.

Syntax

drop procedure *proc_name*

Description

This statement removes a database procedure definition from the database. When executed, it takes effect immediately. Executions in progress, invoked by other users, are allowed to continue until they are completed. A procedure can only be dropped by its owner.

proc_name is the name of the procedure to be removed.

Example

This statement removes the procedure named "salupdt."

```
DROP PROCEDURE salupdt
```

grant

Grants privileges on a table, view, or procedure.

Syntax

grant all [**privileges**] **on** [**table**] *tablename* {, *tablename*} **to public**

grant all [**privileges**] **on** [**table**] *tablename* {, *tablename*} **to** *username* {, *username*}

grant *priv* {, *priv*} **on** [**table**] *tablename* {, *tablename*} **to public**

grant *priv* {, *priv*} **on** [**table**] *tablename* {, *tablename*} **to** *username* {, *username*}

grant *priv* **on procedure** *proc_name* { , *proc_name*} **to public** | *username* {, *username*}

Description

priv represents one of the following privileges:

- **select**
- **insert**
- **delete**
- **update** (*columnname* {, *columnname*})
- **execute**

This command grants one or more of these privileges to any set of users on the tables, views, or procedure specified. The **Select**, **insert**, **update**, and **delete** privileges can only be granted on tables or views. The **Execute** privilege can only be granted on procedures.

The **Grant** statement must be issued by the DBA of the current database, who must own the procedure or all the tables and views specified. If a non-DBA issues a **grant** statement, an error is returned. If the DBA issues a **grant** statement that includes tables or views that the DBA does not own, processing continues on all the tables or views that the DBA does own.

If the DBA issues a **grant** statement to allow a user to use a view or procedure, then the user can do so without permission(s) on the underlying tables or views.

The optional words **privileges** and **table** have no effect. They are included for compatibility with other versions of SQL.

help

Gets information about SQL, or about tables in the database.

Syntax

help [[**all**]|[*tablename*{, *tablename*}]]
help view *viewname* {, *viewname*}
help permit|integrity *tablename* {, *tablename*}
help help
help sql
help *sql_statement*

Description

The **help** function can be used to display information about ODT-DATA features, definitions of views, protections or permissions, or information about the contents of the database and specific tables in the database. In addition, **help** can be used inside the ODT-DATA SQL Terminal Monitor to obtain information regarding SQL, including such features as the syntax of SQL statements and the available datatypes. The legal forms are as follows:

help	Lists all user (that is, not system) tables that exist in the current database.
help all	Gives information about the makeup of all user (that is, not system) tables in the database.
help *tablename* {, *tablename*}	Gives information about the specified table(s).
help view *viewname*}	Prints view definition of specified view(s).
help permit *tablename* {, *tablename*}	Prints permissions on specified table(s).
help integrity *tablename* {, *tablename*}	Prints current integrity constraints on specified table(s).
help help	Prints a list of SQL features for which help is available.
help sql	Prints information of a general nature pertaining to SQL.
help *sql_statement*	Prints information on the specified *sql_statement*.

The **permit** and **integrity** forms of the **help** command print out unique integer identifiers for each constraint. The **drop permit** and **drop integrity** commands use these identifiers to remove individual constraints. (See the sections in this chapter on **drop integrity** and **drop permit**.)

Examples

Retrieve a list of all tables in the database.

```
help;
```

Retrieve help about the "employee" table.

```
help employee;
```

Retrieve help about the "employee" and "dept" tables.

```
help employee, dept;
```

Retrieve the definition of the "highpay" view.

```
help view highpay;
```

List all permits issued on the "job" and "employee" tables.

```
help permit job, employee;
```

List all integrity constraints issued on the "dept" and "employee" tables.

```
help integrity dept, employee;
```

List information on the **select** statement.

```
help select;
```

if-then-else

Chooses between alternative paths of execution inside a database procedure.

Syntax

if *boolean_expr*
then *statement*; {*statement*;} {**elseif** *boolean_expr* **then**
statement; {*statement*;}} [**else**
statement; {*statement*;}] **endif**

Description

In SQL, this statement can only be issued from within the body of a database procedure.

A boolean expression (*boolean_expr*) must always evaluate to "true" or "false." As discussed in Chapter 1, "SQL Syntax," a boolean expression can include comparison operators ("=", ">" and so on) and the logical operators **and**, **or**, and **not**. Boolean expressions involving NULL values frequently evaluate to "unknown", which will behave exactly as if it evaluated to "false."

The simplest variant of the **if** statement performs an action only if the boolean expression evaluates to true. The syntax for this variant is:

```
if boolean_expr then
      statement; {statement;}
endif
```

If the boolean expression evaluates to true, the list of statements is executed. If the expression evaluates to false (or "unknown"), the statement list is not executed, and control passes directly to the statement following the **endif** terminator.

The second variant of the **if** statement includes the **else** construct. Its simplest form is:

```
if boolean_expr then
      statement; {statement;}
else
      statement; {statement;}
endif
```

In this variant, if the boolean expression is true, the statements immediately following the keyword **then** are executed. If the expression is false (or "unknown"), the statements following the keyword **else** are executed. In either case, after the appropriate statement list is executed, control passes to the statement immediately following **endif**.

The third **if** variant involves the **elseif** construct. The **elseif** construct allows the running application to test a series of conditions in a prescribed order. The statement list corresponding to the first true condition found is executed; all other statement lists connected to conditions are skipped. The **elseif** construct can be used with or without an **else** construct, which must follow all the **elseif** constructs. If an **else** construct is included, one statement list is guaranteed to be executed, because the statement list connected to the **else** is executed if all the specified conditions evaluate to false.

The simplest form of this variant is:

```
if boolean_expr then
      statement; {statement;}
elseif boolean_expr then
      statement; {statement;}
endif
```

If the first boolean expression evaluates to true, the statements immediately following the first **then** keyword are executed. In such a case, the value of the second boolean expression is irrelevant. If the first boolean expression proves false, however, the next boolean expression is tested. If the second expression is true, the statements under it are executed. If both boolean expressions test false, neither statement list is executed.

A more complex example of the **elseif** construct is:

```
if boolean_expr then
      statement; {statement;}
elseif boolean_expr then
      statement; {statement;}
elseif boolean_expr then
      statement; {statement;}
else
      statement; {statement;}
endif
```

In this case, the first statement list is executed if the first boolean expression evaluates to true. The second statement list is executed if the first boolean expression is false and the second true. The third statement list is executed only if the first and second boolean expressions are false and the third evaluates to true. Finally, if none of the boolean expressions is true, then the fourth statement list is executed. After any of the statement lists is executed, control passes to the statement following the **endif.**

Two or more **if** statements can be nested. In such cases, each **if** statement must be closed with its own **endif.**

If an error occurs during the evaluation of an **if** statement condition, the database procedure terminates and control returns to the calling application. This is true even if the statement is nested.

Example

This **if** statement performs a **delete** or an **insert** and checks to make sure the statement succeeded.

```
IF (id > 0) AND (id <= maxid) THEN
      DELETE FROM emp WHERE id = :id;
      IF iierrornumber > 0 THEN
            MESSAGE 'Error deleting specified row';
            RETURN 1;
      ELSEIF iirowcount = 0 THEN
            MESSAGE 'Specified row does not exist';
            RETURN 2;
      ENDIF;
ELSEIF (id < maxid) THEN
      INSERT INTO emp VALUES (:name, :id, :status);
      IF iierrornumber > 0 THEN
            MESSAGE 'Error inserting specified row';
            RETURN 3;
      ENDIF;
ELSE
      MESSAGE 'Invalid row specification';
      RETURN 4;
ENDIF;
```

insert

Inserts rows into a table.

Syntax

insert into *tablename* [(*column* {, *column* })]
[**values** (*expr*{, *expr*})] | [*subselect*]

Either the **values** clause or the *subselect* must appear. See the **select** command description for the subselect syntax.

Description

The **insert** statement inserts new rows into the specified table. In the **values** form, a single row is inserted; in the subselect form, all rows that result from evaluating the subselect are inserted.

The *i*th expression in the **values** list, or the *i*th expression in the **select** clause of the subselect, corresponds to the *i*th column in the list of column names. Omitting the list of column names is allowed when a subselect is used and the column names in the subselect match column names in the table, or if the **values** list corresponds exactly to the columns in the table. That is, the **values** list must have a value for each column and that value must be the appropriate data type. Additionally, the values must be listed in an order corresponding to the order of the columns in the table.

What happens in columns not specified in the column list depends on the format used when the table was created with **create table**. If the column was set **with null**, the NULL value is assigned. If the column is set **not null with default**, the appropriate default value ("0" or blank) is assigned. Otherwise, an error code is returned and the insert is not executed.

Expressions used in the **values** clause can only be constants (including the NULL constant), scalar functions on constants, or arithmetic operations on constants.

An **insert** statement may be issued only by the owner of the table or by a user with **insert** permission on the table.

Inserted data must be coercible into the data type of the target column, that is, either both must be numeric types or both must be character types.

Some common errors to watch for are:

- Use of a numeric expression to set the value of a string column or *vice versa*
- Failure to specify a value for a column that is not nullable and not defaultable
- An attempt to insert the NULL constant into a non-nullable column

Examples

1. Simply add a row to an existing table.

```
insert into emp (name, sal, bdate)
     values ('Jones, Bill', 10000, 1944);
```

2. Insert into the "job" table all rows from the "newjob" table where the job title is not "Janitor."

```
insert into job (jid, jtitle, lowsal, highsal)
select      job_no, title, lowsal, highsal
from        newjob
where       title != 'Janitor';
```

3. Add a row to an existing table, using the default columns.

```
insert into emp
     values ('Jones, bill', 10000, 1944)
```

message

Returns a message number, message text, or both to the executing application.

Syntax

message *message_text* | *message_number* | *message_number message_text*

Description

This statement can only be issued from inside the body of a database procedure.

The *Message_text* can be a string literal, or a non-null local character variable or parameter. The *Message_number* can be an integer or a non-null local integer variable or parameter. Neither can be expressions. Both the *message_text* and the *message_number* are supplied by the database procedure programmer.

When a message is returned to an application, the default behavior is to display the message on the screen. (In an interactive forms application, the message is displayed in a window at the bottom of the screen.) An application may override the default behavior by using the Embedded SQL **whenever** statement. Consult the *ODT-DATA Embedded SQL User's Guide* for more details about the **whenever** statement and processing procedure messages.

Examples

This fragment returns trace text to the application.

```
MESSAGE 'Inserting new row';
INSERT INTO tab VALUES (:val);
MESSAGE 'About to commit change';
COMMIT;
MESSAGE 'Deleting newly inserted row';
DELETE FROM tab WHERE tabval = :val;
MESSAGE 'Returning with pending change';
RETURN;
```

This example returns a message number to the application. The application can then extract the international message text out of a message file.

```
IF iierrornumber > 0 THEN
     MESSAGE 58001;
ELSEIF iirowcount != 1 THEN
     MESSAGE 58002;
ENDIF;
```

modify

Converts the storage structure of a table or index.

Syntax

modify *tablename*|*indexname* **to** *storage_structure* | *verb*[**unique**]
[**on** *columnname* [**asc**|**desc**] {,*columnname* [**asc**|**desc**]}]
[*with-clause*]

A *with-clause* consists of the word **with** followed by a comma-separated list of any number of the following items:

fillfactor=*n*
minpages=*n*
maxpages=*n*
leaffill=*n*
nonleaffill=*n*
newlocation=*(loc1[, loc2[, loc3 ...]]),*
oldlocation=*(loc1[, loc2[, loc3 ...]]),*
location=*(loc1[, loc2[, loc3 ...]]),*

Description

The **modify** command changes *tablename* or *indexname* to the specified storage structure, reorganizes a btree index, or moves a table to two or more different locations. Only the owner of a table can modify that table. This command is used to accelerate performance of queries that access the table, particularly when the table is large or frequently referenced. The *Storage_structure* can be any of the following:

isam	indexed sequential access method structure, duplicate rows allowed unless the **with noduplicates** clause is specified when the table is created.
cisam	compressed **isam**, duplicate rows allowed unless the **with noduplicates** clause is specified when the table is created.
hash	random **hash** storage structure, duplicate rows allowed unless the **with noduplicates** clause is specified when the table is created.

chash compressed **hash**, duplicate rows allowed unless the **with noduplicates** clause is specified when the table is created.

heap unkeyed and unstructured, duplicated rows allowed, even if the **with noduplicates** clause is specified when the table is created.

cheap compressed **heap**.

heapsort heap with rows sorted and duplicate rows allowed unless the **with noduplicates** clause is specified when the table is created (sort order not retained if rows are added or replaced).

cheapsort compressed heapsort.

btree dynamic tree-structured organization with duplicate rows allowed unless the **with noduplicates** clause is specified when the table is created.

cbtree compressed **btree**, duplicate rows allowed unless the **with noduplicates** clause is specified when the table is created.

The *Verb* can be any of the following:

merge special form **of modify** for **btree** and **cbtree** storage structures; modifies the tree structure only merging adjacent pages whenever possible and deleting empty pages.

relocate move a table or portion of a table from the location(s) listed in the oldlocation list to the location specified in the newlocation list.

reorganize spread the contents of the table over the location(s) in the location list.

truncated special form to delete all rows quickly and release all file space back to the operating system; structure automatically converted to heap.

The current compression algorithm suppresses trailing blanks in columns of the **c** data type.

The keyword **unique** may be used with the following storage structures:

isam	**cisam**
hash	**chash**
btree	**cbtree**

This keyword has the effect of requiring each key value in the table to be unique. (A key value is the concatenation of all key columns in a row.) If you try to use the **unique** keyword for a table containing non-unique keys, ODT-DATA returns an error message and does not change the storage structure.

Keys may be defined on nullable columns, even unique keys. For determining "uniqueness" on a key, a column or a whole row, NULL values are considered equal to other NULL values. Therefore, if you define keys on nullable columns, use **btree**, because the duplicate NULL values create overflow chains in **isam** and **hash** tables, making them very inefficient.

For determining the ordering of values in a column, NULL values are considered "greater than" any non-NULL value.

If the **on** phrase is omitted when modifying to **isam, cisam, hash, chash, btree** or **cbtree**, the table automatically keyed on the first column. When modifying to heap or cheap, the **on** phrase is meaningless and must be omitted. When modifying to heapsort or cheapsort, the **on** phrase is optional.

When a table is sorted (**isam, cisam, heapsort, cheapsort, btree**, and **cbtree**) the primary sort keys are those specified in the **on** phrase (if any). The first key (columnname) after the **on** phrase is the most significant sort key, and each successive columnname specified is the next most significant sort key. Any columns not specified in the **on** phrase is used as least significant sort keys in column number sequence.

When a table is modified to heapsort or cheapsort, the sortorder can be specified to be **asc** or **desc**, meaning ascending or descending, respectively. The default is **asc.**

The **fillfactor** specifies the percentage (from 1 to 100) of each primary data page that should be filled with rows, under ideal conditions. The **fillfactor** may be used with **isam, cisam, hash, chash, btree**, and **cbtree**. When modifying to **btree** or **cbtree**, **nonleaffill** determines the percentage of each index page to fill. Care should be taken when specifying large fillfactors because a non-uniform distribution of key values could later result in overflow pages and thus degrade access performance for the table.

The **minpages** specifies the minimum number of primary pages a **hash** or **chash** table must have. The **maxpages** specifies the maximum number of primary pages a **hash** or **chash** table may have. The **minpages** and **maxpages** must be at least one. If both **minpages** and **maxpages** are specified in a **modify** command, **minpages** cannot exceed **maxpages**.

Default values for **fillfactor**, **minpages**, and **maxpages** are as follows:

	Fillfactor	**Minpages**	**Maxpages**
hash	50	10	no limit
chash	75	1	no limit
cisam	100	NA	NA
btree	80	NA	NA
cbtree	100	NA	NA

The **leaffill** parameters of the **modify** command applies only to tables stored in **btree** and **cbtree** structures. The **leaffill** parameter specifies the percentage to fill each index page for a **btree** or **cbtree** table.

The **leaffill** specifies a percentage *n*, where *n* ranges from 1 to 100, and its percentage specifies how much each index page should be filled at the time the table is modified to **btree** or **cbtree**. This parameter contrasts with the **fillfactor** parameter, which specifies the percentage occupancy of data pages (not index pages) when a table is converted to **btree** or **cbtree**.

The **leaffill** parameter allows you to control locking contention in **btree** and **cbtree** index pages. By retaining a percentage of open space on these index pages, more concurrent users can access the **btree** without contention while their queries descend the index tree. Note, however, that you must strike a balance between preserving space in index pages and creating a greater number of index pages; more levels of index pages require more I/O to locate a data row.

Default value for **leaffill** is 60 (percent). This default applies to both **btree** and **cbtree** indexes.

ODT-DATA storage structures use existing data to build the index (for **isam** and **cisam**), the **hash** function (for **hash** and **chash**), or for sorting (**heapsort** and **cheapsort**). Therefore, it is pointless to modify a table to any of these six structures before adding data to the tables. You are thus strongly encouraged to add all data to a table as a heap before modifying a table to these structures. Then, after the table contains its data, run **modify** to optimize storage for retrievals. If you add, delete, or change the data in the table significantly, affecting, say, 20% of the data, run **modify** again to re-optimize storage. If the table is dynamically used as part of an ongoing application, periodically re-optimize it with the **modify** command. If the table is merely a static repository for data, this maintenance procedure is not needed.

When data is added to a table stored as a **btree** or **cbtree**, the **btree** index automatically expands, so there should be no need to remodify a growing **btree** index. However, a **btree** index does not shrink when rows are deleted from the **btree** table. A special form of modify can be used to shrink a **btree** index after you have deleted a significant number of rows from the **btree** table.

modify *tablename* **to merge**

Because this form of **modify** only affects the index, it usually runs a good deal faster than a normal **modify** command. This form of **modify** does not require any temporary disk space to execute.

When **modify** is run on a table, any indexes created for the table are destroyed and must be recreated (except for **modify** to merge). For more information on indexes, please refer to the **create index** command description.

For information on the **reorganize** and **relocate** clauses, see "Multi-Location Tables" in Chapter 1.

Examples

Modify the "employee" table to an indexed sequential storage structure with "eno" as the keyed column.

```
modify employee to isam on eno;
```

If "eno" is the first column of the "employee" table, the same result can be achieved by:

```
modify employee to isam;
```

Perform the same **modify** command but request a 60% occupancy on all primary pages.

```
modify employee to isam on eno with fillfactor = 60;
```

Modify the "job" table to compressed **hash** storage structure with "jid" and "salary" as keyed columns.

```
modify job to chash on jid, salary;
```

Perform the same **modify** command but also request 75% occupancy on all primary pages, a minimum of seven primary pages and a maximum of 43 primary pages.

```
modify job to chash on jid, salary with fillfactor =
     75, minpages = 7, maxpages = 43;
```

Perform the same **modify** command again but only request a minimum of 16 primary pages.

```
modify job to chash on jid, salary with
     minpages = 16;
```

Modify the "dept" table to a heap storage structure.

```
modify dept to heap;
```

Modify the "dept" table to a heap again, but have rows sorted on the "dno" column and have any duplicate rows removed.

```
modify dept to heapsort on dno;
```

Modify the "employee" table in ascending order by "ename," descending order by "age" and have any duplicate rows removed.

```
modify employee to heapsort on ename, age desc;
```

Modify the "btree on "ename," so that data pages are 50% full and index pages are initially 40% full.

```
modify employee to btree on ename
     with fillfactor = 50, leaffill = 40;
```

return

Terminates a currently executing database procedure and returns control to the calling application and, optionally, retrieves a value.

Syntax

return [*return_status*]

Description

In a database procedure, the **return** statement terminates the procedure and returns control to the application. The calling application resumes execution at the statement following **execute procedure**.

The **return** statement can return a value to the application that executed the procedure by using the *return_status*. The *return_status* must -null integer constant, variable, or parameter whose data type is compatible with the data type of the field to which its value is assigned upon the return. If the *return_status* is not specified or if a **return** statement is not executed, then 0 is returned to the calling application.

The **into** clause of the **execute procedure** statement allows the calling application to retrieve the *return_status* once the procedure has finished executing. Consult the *ODT-DATA Embedded SQL User's Guide* for information about the **execute procedure** statement.

Example

This fragment of a database procedure returns a passed parameter to the calling application.

```
CREATE PROCEDURE CHECK (okval INTEGER, failval
INTEGER) AS
BEGIN
      ...
```

```
IF (iierrornumber = 0) THEN
            COMMIT;
            RETURN :okval;
      ELSE
            ROLLBACK;
            RETURN :failval;
      ENDIF;
END;
```

rollback

Rolls back the current transaction.

Syntax

rollback [**work**] [**to** *savepointname*]

Description

This statement aborts part or all of the current transaction. If the **to** *savepointname* clause is given, the transaction is undone to the point of the savepoint declaration. Otherwise, the entire transaction is erased.

The optional word **work** has no effect. It is included for compatibility with other versions of SQL.

The statement analogous to **rollback** in pre-6.0 versions of INGRES is **abort**.

save

Saves a base table until a specified date.

Syntax

save *tablename* [**until** *month day year*]

Description

Use **save** to preserve tables until the given expiration date. Only the owner of a table can save that table. User tables, when created, default to "no expiration date."

The *month* can be an integer from 1 through 12, or the name of the month, either abbreviated or spelled out. The *day* is simply the day of the month, and *year* is the fully specified year (that is, 1982 or 1999).

If the optional **until** clause is omitted, the expiration date is set to "no expiration date."

Tables are not automatically destroyed after their expiration date. At this time there is no means by which expired tables can be destroyed.

System tables have no expiration date.

Example

Save the "employee" table until the end of February, 1989.

```
save employee until feb 28 1989;
```

savepoint

Declares a savepoint marker within a transaction.

Syntax

savepoint *savepoint_name*

Description

The **savepoint** command declares a named savepoint marker within a transaction, allowing subsequent rollbacks back to the declared savepoint at any time before the transaction is terminated. The *savepoint_name* can be any character string conforming to rules for ODT-DATA names, except that the first character need not be alphabetic. This allows the naming of savepoints with integers. Note that the *savepoint_name* is not entered with quotation marks.

There is no limit to the number of savepoints that you can declare within a transaction. You can also use the same *savepoint_name* more than once; however, only the most recently declared use of a *savepoint_name* is active within the transaction. That is, if you abort the transaction to a savepoint whose name is used more than once, the transaction is backed out to the most recent use of the *savepoint_name*.

All savepoints of a transaction are rendered inactive when the transaction is terminated (either committed with **commit** or rolled back with **rollback** or a system intervention upon deadlock). Please refer to Chapter 1, "SQL Syntax," for more information on deadlock and to the command descriptions in this chapter for information on **commit** and **rollback**, respectively.

Example

Declare savepoints among other SQL statements, then do partial rollbacks of the transaction.

```
insert into emp (name, sal, bdate)
              values ('Jones,Bill', 10000, 1945);
savepoint setone;              /* sets first savepoint marker
insert into emp (name, sal, bdate)
              values ('Smith,Stan', 20000, 1948);
savepoint 2; \g                /* set second savepoint marker
```

```
insert into emp (name, sal, bdate)
       values ('Engel,Harry', 18000, 1954);
rollback to 2; \g          /* undoes third append; first and second remain */
rollback to setone; \g /* undoes second append; first remains */
commit; \g                 /* only the first append is committed */
```

select

Retrieves values from one or more tables.

Syntax

subselect
{**union** [**all**] (*subselect*)}
[**order by** *order_column* [**asc** | **desc**] {, *order_column* [**asc** | **desc**]}]

where *subselect* has the syntax:

select [**all**|**distinct**] *expression* [**as** *result_column*] {, *expression* [**as** *result_column*]}
from *table* [*corr_name*] {, *table* [*corr_name*]}
[**where** *search_condition*]
[**group by** *column* {, *column*}]
[**having** *search_condition*]

Description

The result of a **select** statement is the union of the results of all subselects in that statement, ordered in accordance with the specifications of the optional **order by** clause.

Duplicate rows are always eliminated if **union** is specified. But if you say **union all**, duplicates are not removed. If you say **union all** once, you must say it for all **union**s within one statement. If **order by** is not specified, the rows of the result table appear in unpredictable order.

Note that all subselects in a **select** statement with **union** must have the same number of columns in their result tables. Additionally, columns of numeric type cannot be matched with columns of character type.

Each *order_column* in the **order by** clause must consist of either a result column name or an integer constant in the range 1 - *n*, where *n* is the number of columns in the result table of each of the subselects. The *order_column* designations are taken only from the first subselect in a set of "unioned" subselects. The optional keywords **asc** and **desc** specify ascending and descending sort sequence, respectively. If neither is specified for a particular column, **asc** is assumed by default.

The keyword **distinct**, used in a subselect, indicates that duplicate rows are to be eliminated. If **distinct** is not specified, the subselect defaults to **all**, in which case duplicate rows are not eliminated.

The *expressions* in the **select** clause of the subselect can be any expressions constructed in accordance with the rules set forth in Chapter 1, "SQL Syntax." They may also take one of the following forms:

correlation_name.*	meaning all the columns of the table denoted by *correlation_name*.
table.*	meaning all the columns of *table*.

Note that * is considered 'wild card.'

Additionally, you can specify **select * from** *tablenames*, which returns all the columns from all the tables named in the **from** clause.

A *result_column* may be assigned to any *expression* that denotes a single column in the result table (that is, where *expression* does not use the "*" syntax). The result column then appears in the result table as the column heading for the expression. The ability to assign a result column name to an expression is of particular benefit when the expression is not simply a column from a database table. If the expression is such a column, the column heading in the result table is by default the name of that column. However, when the expression is, for example, a scalar or set function or involves a computation, ODT-DATA returns blanks for the column heading. To override this default, assign the expression an appropriate result column. The result column, whether default or explicit, may also be used in the **order by** clause.

The **from** clause is used to specify the base tables from which rows are to be selected. An optional correlation name (*corr_name*) may be chosen for each table specified (see Chapter 1, "SQL Syntax" for information about correlation names). If the **from** clause includes more than one table, and a column name in the **select** list appears in more than one of the tables in the **from** clause, column names in the **select** statement must be qualified explicitly by a table name or a correlation name. This eliminates ambiguity as to which table a column belongs.

The **from** clause may be omitted if the statement consists only of a **select** clause of a constant expression. (See the examples following this description.)

The **where** clause qualifies the selection of rows; only those rows that satisfy the *search_condition* are selected.

The *columns* in the **group by** clause of the subselect are names of columns from the table(s) identified in the **from** clause. The groups may be qualified by a **having** clause.

From a conceptual standpoint, the subselect is evaluated in the following manner:

- First, the Cartesian product of all tables identified in the **from** clause is formed. From that product, rows not satisfying the search condition specified in the **where** clause are eliminated.
- Next, the remaining rows are grouped in accordance with the specifications of the **group by** clause.
- Groups not satisfying the search condition in the **having** clause are then eliminated.
- Finally, the expressions specified in the **select** clause are evaluated.
- If the keyword **distinct** has been specified, any duplicate rows are eliminated from the result table.

If the subselect includes a **group by** clause, each expression in the **select** clause must be *single-valued per group*. That is, the only data items permitted in such an expression are the following:

- the grouping columns.
- set function references. As usual, however, such terms can be combined by using arithmetic operations, can be the arguments to scalar functions, and so on.

If the subselect includes a **having** clause, each expression in that clause must also be single-valued per group. If the **group by** clause is omitted in a subselect with a **having** clause, the entire table is considered to be a single group.

NOTE: When **select** is used to display varying length character columns, two features should be noted. First, the **select** statement pads unused space with blanks. Second, nonprinting characters and control characters are displayed as blanks. **select** assumes that each varying length character column requires a width of *n* characters on the screen, where *n* is the width specified when the column was created.

Only the table's owner or a user with **select** permission on the table may issue a **select** statement.

Examples

Find all employees who make more than their managers.

```
select      e.ename
from        employee e, dept, employee m
where       e.dept = dept.dno
and         dept.mgr = m.eno
and         e.salary > m.salary;
```

Retrieve all columns for those employees who make more than the average salary.

```
select      *
from        employee
where       salary >
            (select avg (salary)
            from employee);
```

Retrieve employee information sorted, with duplicate rows removed.

```
select      distinct e.ename, d.dname
from        employee e, dept d
where       e.dept = d.dno
order       by dname desc, ename;
```

Select lab samples from production and archive tables that were analyzed by lab #12.

```
select      *
from        samples s
where       s.lab = 12
union
select      *
from        archive_samples a
where       a.lab = 12 ;
```

Select the current user name.

```
select username();
```

Select a data conversion operation.

```
select dow(date('today') + date('3 days'));
```

set

Sets an ODT-DATA session option.

Syntax

set autocommit on |off
set journaling | nojournaling [on *tablename*]
set result_structure
heap|cheap|heapsort|cheapsort|hash|chash|isam|cisam|btree|cbtree

Set lockmode parameters for your ODT-DATA session to the desired values. Tables accessed after executing this command are governed by these locking behavior characteristics.

Description

The **set** command specifies an ODT-DATA run-time option for a single ODT-DATA session. The selected run-time option remains in effect until the end of the ODT-DATA session, by using either the ODT-DATA Terminal Monitor or a database invocation within an Embedded SQL program. Alternatively, another **set** command can change the value of a current run-time option established by a previous **set** command.

See *Administering ODT-DATA* for information about changing environment variables.

The SET AUTOCOMMIT Option

The **set autocommit on** command causes an implicit commit to occur after every successfully executed SQL query. The **set autocommit off**, the default case, means an explicit command is required to commit a transaction.

The JOURNALING/NOJOURNALING Option

The **set journaling** command causes all tables created within a session to be logged in with the ODT-DATA journaling system. Note, however, that journaling does not take effect until journaling is enabled for the entire database with the **ckpdb** command. (Please refer to Chapter 4, "ODT-DATA Operating System Commands," for information about **ckpdb**.) With the **journaling** option set, the explicit **with journaling** clause is not necessary in the **create table**

command. Additionally, tables created using the **as** clause of the **create table** command are also logged. If the **set nojournaling** command, which is the default, is set, tables are created without logging to the journal, unless the explicit **with journaling** clause appears in the **create table** command. The **set journaling** command, when used with an optional table name, causes journaling to begin at the next checkpoint for the named table.

The RESULT_STRUCTURE Option

The **set result_structure** command sets the default storage structure for tables created with the **as** clause of the **create table** command. If the value of **heap** or **cheap** is selected as the default, tables are created exactly as through the **select** command, which may result in duplicate rows. However, performance of the **create table as** command is best with the **heap** or **cheap** option specified. You can optionally set the default structure of tables created by **create table as** to any of the structures described in the **modify** command, that is, **heap**, **cheap**, **heapsort**, **cheapsort**, **hash**, **chash**, **btree**, **cbtree**, **isam** or **cisam**. For example, this first sequence of statements does the same thing as the second sequence:

```
set result_structure hash;
create temp as select id ... ;

create temp as select id ... ;
modify temp to hash;
```

Either sequence results in the "temp" table being stored in a **hash** structure, hashed on the first column, "id" in this case. For **hash**, **chash**, **isam**, and **cisam**, the newly created table is automatically indexed on the first column.

If you do not execute a **set result_structure** command, the default storage structure for a table created by the **create table as** command is **cheap**. If **distinct** is specified, the **cheapsort** structure is used if the default storage structure is **cheap**, **chash**, or **cisam**; if the default storage structure is **heap**, **hash** or **isam**, **heap** or **btree**, then **heapsort** is used.

The SET LOCKMODE Option

You can also use the **set** command to determine how the ODT-DATA locking system operatse when ODT-DATA accesses data in a table. The **set lockmode** option allows you to establish a number of different types and levels of locks.

You should know that ODT-DATA provides a default strategy for locking in query processing. If you have no interest in overriding this default, you need not make use of the **set lockmode** option. The **set lockmode** option is provided to allow you to optimize performance or enforce stricter validation and/or concurrency controls.

set lockmode acknowledges three basic types of locking: (1) locking provided by default, that is, by the ODT-DATA system; (2) locking instituted for an ODT-DATA session; and (3) locking specified on an *ad hoc* basis. Therefore, **set lockmode** allows you to switch among any of these three types of locking at any time in your ODT-DATA session, except where specifically disallowed, such as within a multi-statement transaction (see Appendix B, "The ODT-DATA System Catalogs," on the **begin transaction** command).

set lockmode provides four different parameters to govern the nature of locking in an ODT-DATA session:

- **level:** This refers to the level of granularity desired when the table is accessed. You can specify any of the following locking levels:

page	Specifies locking at the level of the data page (subject to escalation criteria; see "maxlocks" later in this list).
table	Specifies table-level locking in the database.
session	Specifies the current default for your ODT-DATA session.
system	Specifies that ODT-DATA starts with page-level locking, unless it estimates that more than *maxlocks* pages are referenced, in which case table level locking is used.

- **readlock:** This refers to locking in situations where table access is for reading of data only (as opposed to updates of data). You can specify any of the following readlock modes:

nolock	Specifies no locking when reading data.
shared	Specifies the default mode of locking when reading data.
exclusive	Specifies exclusive locking when reading data (useful in "select-for-update" processing within a multi-statement transaction).
session	Specifies the current readlock default for your ODT-DATA session.
system	Specifies the general readlock default for the ODT-DATA system.

- **maxlocks:** This refers to an escalation factor, or number of locks on data pages, at which locking escalates from page level to table level. The number of locks available to you is dependent upon your system configuration. You can specify the following maxlocks escalation factors:

n	Specifies a specific (integer) number of page locks to allow before escalating to table-level locking. *n* now defaults to 10. *n* must be greater than 0.
session	Specifies the current maxlocks default for your ODT-DATA session.
system	Specifies the general maxlocks default for the ODT-DATA system.

  NOTE: If you specify page-level locking, and the number of locks granted during a query exceeds the system-wide lock limit, or if the operating system's locking resources are depleted, locking escalates to table level. This escalation occurs automatically and is independent of the user.

- **timeout:** This refers to a time limit, expressed in seconds, for which a lock request should remain pending. If ODT-DATA cannot grant the lock request within the specified time, then the query that requested the lock aborts. You can specify the following timeout characteristics:

n	Specifies a specific (integer) number of seconds to wait for a lock (setting *n* to 0 requires ODT-DATA to wait indefinitely for the lock).
session	Specifies the current timeout default for your ODT-DATA session (which is also the ODT-DATA default).
system	Specifies the general timeout default for the ODT-DATA system.

Against the backdrop of these **set lockmode** parameters and options are, of course, the ODT-DATA system defaults for each of the parameters:

level	dynamically determined by ODT-DATA
readlock	shared
maxlocks	10
timeout	0 (no timeout)

If you select the **system** option for any of the **set lockmode** parameters, the values above are automatically supplied. When you begin your ODT-DATA session, the ODT-DATA system defaults are in effect. If you override them with other values using the **set lockmode** command, you can revert back to the system defaults easily.

Similarly, if you set session parameters (that is, locking behavior for all user tables accessed by queries in your ODT-DATA session), you can further set parameters for individual tables on an as needed basis. After setting the as needed locking behavior, you can return it to either the session defaults or the ODT-DATA system defaults.

Examples

Within an ODT-DATA session, create three tables with journal logging enabled and one without.

```
set journaling;
create table withlog1 ( ... );
create table withlog2 ( ... );
set nojournaling;
create table withlog3 ( ... ) with journaling;
create nolog1 ( ... );
```

Create a few tables with different structures.

```
create table a as ...; /* heap */
set result_structure 'hash';
create table b as select id ...;    /* hash on 'id' */
set result_structure 'heap';
create table d as select id ...;    /* heap again */
```

Set lockmode parameters for your ODT-DATA session to the desired values. Tables accessed after executing this command are governed by these locking behavior characteristics.

```
set lockmode session where level = page, readlock = nolock,
      maxlocks = 50, timeout = 10;
```

Set the lockmode parameters explicitly for a particular table.

```
set lockmode on employee
      where level = table, readlock = exclusive,
      maxlocks = session, timeout = 0;
```

Reset your ODT-DATA session default locking characteristics to the ODT-DATA system defaults.

```
set lockmode session where level = system, readlock = system,
      maxlocks = system, timeout = system;
```

update

Updates values of columns in a table.

Syntax

update *tablename* [*corr_name*]
set *columnname* = *expression* {, *columnname* = *expression*}
[**where** *search_condition*]

Description

The **update** statement replaces the values of the specified columns by the values of the specified expressions for all rows of the table that satisfy the *search_condition*.

The expressions in the set clause may only use constants or columns from the table specified by *tablename*.

Only the owner of the table or a user with **update** permission on the table is allowed to **update** a table. If a given row update would violate an integrity constraint on the table, that row remains unchanged. Any data used to update a table must come from that same table.

Numeric columns may be updated by values of any numeric type. Update values are converted to the type of the result columns. Character string columns may be updated by values of any character string type. Nullable columns may be set to the NULL value by using the NULL constant.

NOTE: Use a numeric expression to set the value of a numeric column and use a string expression to set the value of a string column. Mixing them does not work.

Examples

Give all employees who work for Smith a 10% raise.

```
update      emp
set   salary = 1.1 * salary
where dept in
      (select     dno
      from        dept
      where       mgr in
            (select     eno
            from        emp
            where       ename = '*Smith'));
```

Set all salaried people who work for Smith to null.

```
update      emp
set   salary = null
where dept in
      (select     dno
      from        dept
      where       mgr in
            (select     eno
            from        emp
            where       ename = '*Smith'));
```

while - endloop

Repeats a series of statements while a specified condition is true.

Syntax

[*label*:] **while** *boolean_expr* **do**
statement; {*statement*;}
endwhile

Description

This statement may only be issued within the body of a database procedure.

A boolean expression (*boolean_expr*) must always evaluate to "true" or "false." A boolean expression can include comparison operators ("=", ">," and so on) and the logical operators **and**, **or**, and **not**.

The statement list may include any series of legal database procedure statements, including another **while** statement.

As long as the condition represented by the boolean expression remains true, the series of statements between **do** and **endwhile** is executed. The condition is tested only at the start of each loop; if values change inside the body of the loop so as to make the condition false, execution still continues through the current iteration of the statement list, unless an **endloop** statement is encountered.

The **endloop** statement may be used to break out of a **while** loop. When **endloop** is encountered, the loop is immediately closed, and execution continues with the first statement following **endwhile**. For example:

```
while condition_1 do
    statement_list_1
    if condition_2 then
        endloop;
    endif;
    statement_list_2
endwhile;
```

In this case, if *condition_2* is true, *statement_list_2* is not executed in that pass through the loop, and the entire loop is closed. Execution resumes at the statement following the **endwhile** statement.

A **while** statement may also be labeled to allow **endloop** to break out of a nested series of **while** statements to a specified level. The label precedes **while** and is specified by a unique alphanumeric identifier followed by a colon, as in:

```
A:  while
```

The label must be a legal ODT-DATA SQL name (see Chapter 1, "SQL Syntax"). The **endloop** statement uses the label to indicate which level of nesting to break out of. The following is one example of the use of labels in nested **while** statements:

```
label_1: while condition_1 do
             statement_list_1
label_2:     while condition_2 do
                 statement_list_2
                 if condition_3 then
                     endloop label_1;
                 elseif condition_4 then
                     endloop label_2;
                 endif;
                 statement_list_3
             endwhile;
             statement_list_4
         endwhile;
```

In this example, there are two possible breaks out of the inner loop. If *condition_3* is true, both loops are closed, and control resumes at the statement following the outer loop. If *condition_3* is false but *condition_4* is true, the inner loop is exited and control resumes at *statement_list_4*.

If no label is specified after **endloop**, only the innermost loop currently active is closed.

If an error occurs during the evaluation of a **while** statement, the database procedure terminates and control returns to the calling application.

Example

This database procedure "delete_n_rows" accepts as input a base number and a number of rows. The specified rows are deleted from the table "tab," starting from the base number. If an error occurs, then the loop terminates.

```
CREATE PROCEDURE delete_n_rows
      (base INTEGER, n INTEGER) AS
DECLARE
      limit INTEGER;
      err INTEGER;
BEGIN
      limit = base + n;
      err = 0;
      WHILE (base < limit) DO
            DELETE FROM tab WHERE val = :base;
            IF iierrornumber > 0 THEN
                  err = 1;
                  ENDLOOP;
                  base = base + 1;
            ENDWHILE;
            RETURN :err;
END;
```

Chapter 3
ODT-DATA Terminal Monitor

The ODT-DATA Terminal Monitor is the primary user interface to SQL. The Terminal Monitor allows you to enter a query and execute it. After executing the query, you can either enter a new query or edit the existing query if minor changes are to be made. The Terminal Monitor also allows you to read or write files containing queries or execute operating system level commands from within.

The Terminal Monitor is invoked by typing the system-level command **sql** at your terminal. (See the **sql** command description in Chapter 4, "ODT-DATA Operating System Commands," for details). At its simplest, you can then type a SQL query, type \g (for go) to run the query, and see the results of the query at your terminal. Simply by typing additional queries, followed by \g, any of the capabilities of SQL can be invoked. To exit the Terminal Monitor, type \q (for quit).

Messages and Prompts

The Terminal Monitor gives a variety of messages to keep the user informed of the status of the monitor and the query buffer.

As the user logs in, a message is printed. This typically tells the version number and the login time. It is followed by the dayfile, which gives information pertinent to users.

When the Terminal Monitor is empty and ready to accept input, the message **go** is printed. The message **continue** means there is something in the query buffer. After a \go command, the query buffer is cleared if another query is typed in, unless a command that affects the query buffer is typed first. Commands that retain the query buffer contents are:

\append or **\a**
\edit or **\e**
\print or **\p**
\bell
\nobell

For example, type:

```
help parts
\go
print parts
```

This results in the query buffer containing:

```
print parts
```

Now type:

```
help parts
\go
\print
print parts
```

This results in the query buffer containing:

```
help parts
print parts
```

An asterisk is printed at the beginning of each line as the prompt character.

Commands

A number of commands may be entered by the user to manipulate either the contents of the query buffer or the user's environment. They are all preceded by a backslash (\), and all are executed immediately (rather than at execution time, like queries).

Some commands may take a filename. In such commands, the filename is designated by a string defined by the first significant character after the end of the command until the end of the line. These commands may have no other commands on the line with them. Commands that do not take a filename may be stacked on a single line. For example:

```
\date\go\date
```

This returns the time both before and after execution of the current query buffer.

\r or **\reset**	Erase the entire query (reset the query buffer). The former contents of the buffer are lost and cannot be retrieved.
\p or **\print**	Print the current query. The contents of the buffer are printed on the user's terminal.
\e or **\ed** or **\edit** or **\editor** [*filename*]	Enter the operating system's text editor (designated by the ODT-DATA startup file). Use the appropriate editor command to return to the ODT-DATA monitor. If no filename is given, the current contents of the query buffer are sent to the editor, and upon return, the query buffer is replaced with the edited query. If a filename is given, the query buffer is written to that file. On exit from the editor, the file contains the edited query, but the query buffer remains unchanged.
\g or **\go**	Process the current query. The contents of the buffer are transmitted to ODT-DATA and run.
\a or **\append**	Append to the query buffer. Typing **\append** after completion of a query overrides the auto-clear feature and guarantees that the query buffer is not reset until executed again.
\time or **\date**	Print out the current time and date.
\s or **\sh** or **\shell**	Escape to the UNIX System shell (command line interpreter). Typing **Ctrl D** causes you to exit the shell and return to the ODT-DATA Terminal Monitor.
\q or **\quit**	Exit ODT-DATA terminal monitor.
\cd or **\chdir** *dir_name*	Change the working directory of the monitor to the named directory.
\i or **\include** or **\read** *filename*	Read the named file into the query buffer. Backslash characters in the file are processed as they are read.
\w or **\write** *filename*	Write the contents of the query buffer to the named file.

\script [*filename*]	Write/stop writing the subsequent SQL statements and their results to the specified file. If no filename is supplied with the \script command, output is logged to a file called **script.ing** in the current directory. The \script command toggles between logging and not logging your ODT-DATA session to a file. If you supply a *filename* on the \script command that terminates logging to a file, the *filename* is ignored. You can use this command to save result tables from SQL statements for output. The \script command in no way impedes the terminal output of your session.
\bell and \nobell	Tell the Terminal Monitor to include (\bell) or not to include (\nobell) a bell (that is, Ctrl G) with the **continue** or **go** prompt. The default is \nobell.
any other character	Ignore any possible special meaning of character following \. This allows the backslash itself to be inserted as a literal character. (See also Chapter 1, "SQL Syntax," on character strings).

Flags

Certain flags may be included on the **sql** command line. These flags affect the operation of the Terminal Monitor. Among the most useful of these flags are:

-a	Disable the autoclear function. This means that the query buffer is never automatically cleared; it is as though the \append command were inserted after every \go. Note that this flag requires that the user must explicitly clear the query buffer using \reset after every query.
-d	Turn off printing the dayfile.
-s	Turn off printing of all messages (except errors) from the monitor, including the login and logout messages, the dayfile and prompts. It is used for executing "canned queries," that is, queries redirected from files.

For a complete list of flags available with the **sql** command, consult Chapter 4, "ODT-DATA Operating System Commands," of this manual.

Diagnostics and Messages

`go`	You may begin a new query.
`continue`	The previous query is finished and you are back in the Terminal Monitor.
`Executing ...`	The query is being processed by ODT-DATA.
`>>editor`	You have entered the text editor.
`Non-printing character` ***nnn*** `converted to blank:`	ODT-DATA maps non-printing ASCII characters into blanks; this message indicates that one such conversion has been made.

Chapter 4
ODT-DATA Operating System Commands

A number of ODT-DATA commands are entered at the level of the computer's operating system. These "utility" commands control the overall database organization, its creation, backup, maintenance and the like. Unlike the SQL commands, these do not affect the data in the database but rather the database as a whole.

Parameter name conventions in the syntax of these commands are:

dbname The ODT-DATA database name, which must be nine characters or less.

flags The set of flags used to select special options to the command. Flags are one letter names, preceded by a sign and optionally followed by a parameter value. If an option provides a "+" or "-" choice before the name, the "+"sign means to turn the option on and the "-" sign means to turn it off. If only a "-" is shown before the name, specification of the "-" turns the option on. For example, a flag is described as:

+ - x

or

[=x|-x]

This means that option *x* has two settings (which are explained in the option description). A **+x** means to turn on the option, and **-x** means to turn it off. However, if a flag is described as **-x**, specifying **-x** invokes the option.

tablename The name of a table in the database.

username The login user name for a valid ODT-DATA user.

accessdb

Authorizes access to database.

Syntax

accessdb

Description

The **accessdb** command is a forms-based interface to list and modify the databases you may access, the locationnames known to the system, and the extensions allowed for databases. You can also use **accessdb** to add users to the ODT-DATA system. See *Administering ODT-DATA* for a complete description of this utility.

auditdb

Audits a database.

Syntax

auditdb [**-b***dd-mmm-yyyy:hh:mm:ss*] [**-e***dd-mmm-yyyy:hh:mm:ss*] [**-f**] [**-i***username*] [-s] [**-t***tablename*] [**-u***username*] {*dbname*}

Description

The **auditdb** command allows the user to print selected portions of the journal for a database or to create an ODT-DATA readable audit trail of the changes made to a particular table. The **auditdb** command operates on all journal entries that have been moved to the journal files. The flags are interpreted as follows:

-b Print journal entries for ODT-DATA transactions committed after the time following the **-b** flag.

-e Print journal entries for ODT-DATA transactions committed before the time following the **-e** flag.

-f Create a file in "bulk copy" (that is, binary) format containing rows appended to, deleted from, or copied into the table specified in the t command. This file may be copied into an ODT-DATA database table that has been created in the following manner:

create auditrel (date = date, user = c24, oper = c8, tranid1 = i4, tranid2 = i4, tbl_idbase = i4, tbl_id_index = i4 { columns of *tablename* })

The first eight columns of "auditrel" contain control information that allow you to identify the user and transaction that performed the operation, the operation itself and the tuple identifier of the row modified. The rest of the columns are identical to the columns in the table being audited. Note that the column information restricts this function to tables that have less than 120 columns and less than 1948 bytes per row.

The **-f** flag creates a filenamed audit.trl in your current directory. This file may be copied into the table created above with the following command:

copy auditrel () from "/usr/dir/audit.trl"

-i Print journal entries for actions taken by the specified user only.

-s Invoke "super user" status for system-wide access to any database.

-t Print the journal entries for the table specified in the **-t** flag.

-u Print the journal, with specified options, for databases owned by the indicated user.

Only the database administrator, who created the database, or the ODT-DATA system administrator (if the **-s** flag is specified) may run the **auditdb** command on a database.

Note that **auditdb** does not necessarily give you a complete list of all transactions since the last checkpoint. There are two reasons for this:

- Because **auditdb** does not exclusively lock the database, other users may complete a transaction while **auditdb** is running.
- In some cases, a completed transaction might not yet have been moved to the journal files.

If you need an absolutely accurate list of transactions since the last checkpoint, make sure all users exit the database before you run **auditdb**.

Some possible diagnostic messages you may receive and their causes are:

`You are not a valid ODT-DATA user`	The current username is not entered in the ODT-DATA users file.
`You may not use the -s flag`	You have tried to use the -s flag, but you do not have ODT-DATA system administrator privileges.
`You are not the dba for` *dbname*	You have tried to audit a database for which you are not the database administrator.
`Cannot enter` *dbname*	The database does not exist.

Examples

Audit the "empdata" database.

```
auditdb empdata
```

Audit "empdata," creating an ODT-DATA-readable audit trail for the "employee" table; then copy this into ODT-DATA.

```
auditdb -temployee -f empdata
sql empdata
 create empaudit(date = date,
     user =          c24, oper = c8, tranid1 = i4, tranid2 = i4,
     tbl_base =      i4, tbl_index = i4, eno = i2,
     ename =         c10, age = i1, job = i2, salary = money, dept = i2)
* copy empaudit () from "/usr/directory/audit.trl"
\g
```

catalogdb

Lists databases that you own.

Syntax

catalogdb [**-u***username*]

Description

catalogdb is a forms-based interface to list your databases, the databases that you may access, the location names known to the system, and the extensions made to your databases. You may also use **catalogdb** to view your user capabilities. See the **accessdb** command for information on how to modify these attributes.

The optional flag for **catalogdb** and its purpose is:

-u Allows the system administrator to use **catalogdb** as the user specified by *username*.

catalogdb, as with other ODT-DATA forms-based products, requires that you specify the type of terminal you are using. For information on defining your terminal, please refer to *Using ODT-DATA Through Forms and Menus*.

As with other ODT-DATA forms-based products, command menus accompany forms on the terminal screen. When you invoke the **catalogdb** command, the main menu appears, offering the following items:

```
Catalog  Database  User  Help  Quit :
```

Note that this menu, along with others, contains an entry for help. After you select one of these operations, the screen clears, and a new form appears. Each menu item evokes a different form.

The `User` item displays a summary of information about your username. The form lists the permissions accorded to your account, the database that you own and the private databases to which you have access. Notice that the two lists (that is, the databases you own and those you may access) appear in a table field. Table fields contain more than one entry and display only a limited number of entries at one time. You can scroll among the entries of a table field using

the techniques described in *Using ODT-DATA Through Forms and Menus*. You can only browse through the forms and cannot change the data. This is true with all forms in **catalogdb**. To change any values displayed in the fields, you must run the **accessdb** program, described in *Administering ODT-DATA* for the current ODT-DATA release. You must be the ODT-DATA system administrator to run this program.

To leave the form invoked by the **User** option, select **e** or **End** from the menu, and you are returned to the main menu for **catalogdb**. You may then select another main menu item.

You can use the **Help** menu command for a full description of the main menu items. In summary, these options do the following:

Command	Function
Catalog	Submenu of additional operations (see below)
Database	Detailed information on one database
Users	Summary information about you
Help	Help message
Quit	Exit the **catalogdb** program

Each command evokes a form for you to browse. Each form includes its own command menu, including a **Help** command, which provides a help message, and an **End** command, which returns you to the main menu.

The **Catalog** operation calls a submenu offering the following items:

```
Databases  DbExtensions  LocationNames  Help  End
```

In summary, these options perform the following functions:

Command	Function
Databases	Table of all your databases
DbExtensions	Table of all your database extensions

Command	Function
LocationNames	Table of all locationname or area mappings on the system
Help	Help message
End	Return to **catalogdb** menu

Note that the **Database** command prompts you for the name of one of the databases that you own. Simply enter the full name of the database to invoke the form with information about that database.

Examples

Browse through data on your own account and database(s).

```
catalogdb
```

As system administrator, browse the data for another user.

```
catalogdb -uPeter
```

ckpdb

Checkpoints a database.

Syntax

ckpdb [**-d**] [**+j**|**-j**] [**-m***device*] [**-u***username*] [**-s**] [**+w**|**-w**] {*dbname*}

Description

The **ckpdb** command creates a new checkpoint for the named database(s) and marks all journal entries up to this checkpoint as expired. Because there is a new checkpoint, previous journal entries are no longer needed. Command line flags have the following interpretations:

-d	Destroy the most recently expired checkpoint and journal files.
+j|**-j**	Enable/disable journaling for a database. When this flag is not specified, the current journaling status of the database is maintained.
-m	Place the new checkpoint onto the specified tape device rather than on disk.
-s	Invoke "super user" (system administrator) status for system-wide access to any database. You must be the system administrator.
-u	Execute the **ckpdb** command on specified or all databases owned by the indicated user.
+w|**-w**	Wait/do not wait for the database to be "free." Note that this flag can be used only in interactive sessions and not in batch mode. The default is **-w**.

Only the database administrator who created the database, or the ODT-DATA system administrator (if the **-s** flag is specified), may run the **ckpdb** command on a database. If neither **+j** nor **-j** is specified, then the current status of journaling for the database as a whole is maintained.

If you wish, you can write the checkpoint to a specified tape device instead of to disk. Note that you can write only one checkpoint per tape.

If no databases are specified, all databases for which you are the database administrator are affected. All databases can have new checkpoints created if the ODT-DATA system administrator uses the **-s** flag.

The **ckpdb** command locks the database because errors can occur if the database is active while the **ckpdb** command is running. If a database is busy, the **ckpdb** command reports this and proceeds to the next database, if any. If the **-w** flag is specified, the **ckpdb** command does not wait, regardless of standard input. The **+w** flag always causes the **ckpdb** command to wait.

Examples

Checkpoint "empdata" and initiate journaling on "empdata".

```
ckpdb +j empdata
```

Checkpoint all databases for which you are database administrator, retaining only the newest checkpoints.

```
ckpdb -d
```

Checkpoint empdata to tape.

```
ckpdb -m/dev/rmt0 empdata
```

compform

Compiles a form.

Syntax

compform [**-u***username*] *dbname form txtfile*

Description

The **compform** command compiles a form that is already stored in a database and places the compiled form in a text file. The command is entered at the operating system.

The flags and parameter names for **compform** have the following meanings:

-u*username*	if specified, compiles the form owned by the stated user. You must be the database administrator for the database or an ODT-DATA super user to use this flag.
dbname	is the name of the database.
form	is the name of the form. You may compile only one form at a time.
txtfile	is the name of the text file in which the compiled form is placed.

Example

To compile the form "employees" which is stored in the "emp" database, and place it in the file **empform.c**, use the command line:

```
compform emp employees empform.c
```

Compiling a Compiled Form

Before you can link the compiled form to your application, you must translate the compiled form into object code.

If the text file is in C language format and the file is "form.c," the following command translates the form into object code:

```
cc form.c
```

If the symbol for the C language compiler at your installation is not "cc," substitute the appropriate compiler symbol in the place of "cc."

The **compform** command automatically generates the correct header file **include** statement for a compiled form in C language format so you do not have to worry about header files when calling the C language compiler to generate object code for a compiled form.

copydb

Creates command files to copy out a database and restores it.

Syntax

copydb [-u*username*] [-c] [-d*pathname*] *dbname* {*tablename*}

Description

The **copydb** command creates two ODT-DATA command files in the current directory:

- **copy.out**, which contains SQL instructions to copy all tables owned by the user into files in the named directory; and
- **copy.in**, which contains SQL instructions to copy the files into tables, create indexes, and perform modifications.

The **copydb** command does not copy the database but creates SQL commands that do the copying. Run **sql** using the commands in the **copy.in** and **copy.out** files to copy the database (see the examples). The name of a file created by **copy.out** consists of the name of the table, truncated to eight characters if necessary, followed by an extension made up of the first three letters of the owner's login name. If filenames collide, a unique digit replaces the last character of the table name segment. The directory must not be the same as the database's actual directory, **$II_DATABASE/ingres/data/default/dbname**, because the files have the same names as the table files.

The optional flags have the following purposes:

-u Run **copydb** with the user identification specified by *username*. This flag may only be used by the database administrator or an ODT-DATA super user. The fact that the **copydb** command creates the copy files does not necessarily mean that the user can access the specified table. If table names are specified, only those tables are included in the copy files.

-d Store the **copy.in** and **copy.out** files in the directory specified by *directory-specification* instead of the default current directory. The specification may be either a full or relative pathname.

-c Cause the copy commands in the generated command files to use a portable format. That is, all data are copied in and out as ASCII characters. This is useful for transporting databases between computer systems whose internal representations of non-ASCII data differ.

NOTE: The **copydb** command automatically converts data stored in this format back to the appropriate ODT-DATA type for the corresponding table column.

Because databases recreated with the **copy.in** file are new, be sure to run **sysmod** after recreating the databases to reinstitute the optimizing effects of storage structures.

It is important that the database is recreated with **copy.in** before doing any work (for example, creating tables, forms, applications, reports, and so on.) in the new database.

Note that system catalogs may not be copied using **copydb**. Use **unloaddb** to copy a complete database, including system catalogs.

Examples

Copy "mydb" to tape.

```
cd /usr/mydir/backup
            /*  Or whatever directory you wish */
copydb mydb /usr/mydir/backup
sql mydb  copy.out
tar c .
rm *
```

Copy tape to "mydb."

```
cd /usr/mydir/backup    /* Again, your choice */
tar xrpf /dev/rmt0
sql mydb  copy.in
sysmod mydb
```

copyform

Copies a form created with the ODT-DATA/Visual-Forms-Editor (VIFRED) from one database to another.

Syntax

copyform [**-s**] [**-u***username*] *dbname filename form* {*form*}

copyform -i [**-s**] [**-u***username*] [**-r**] *dbname filename*

Description

copyform is a utility of the Visual-Forms-Editor (VIFRED) for copying a form from one database to another. Additionally, it can be used to change ownership of a form by copying out a form owned by a particular user into a text file and copying it back into the database under the ownership of another user, effectively changing its owner. Using **copyform** is a two-step process. First, you must copy one or more forms from a database to a text file, using the first variant of the command, as presented above. Next, by using the **copyform** command with the **-i** flag, as shown in the second syntax statement above, you can copy the forms from the text file into a database. See *Using ODT-DATA Through Forms and Menus* for a complete description and examples of this utility.

The flags and parameter names for copying forms from a database into a text file (the first variant shown in the syntax section) have the following meanings:

-s	Suppresse status messages.
-u*username*	Copy forms owned by the stated user. This flag can be used only by the database administrator for the database or the ODT-DATA system administrator.
dbname	The name of the database containing the forms.
filename	The name of a text file in which to write the forms.
form	The name of the form(s). Any number of forms may be specified on the command line.

The flags and parameter names for copying forms from a text file into a database (the second variant shown in the syntax section) have the following meanings:

-i	A required parameter, telling **copyform** that this is an "input" operation.
-s	Suppress status messages.
-u*username*	Add the forms into the database owned by *username*. This flag can be used only by the database administrator for the database or the ODT-DATA system administrator.
-r	Suppress the verification prompt for overwriting existing forms. If a form exists in the database under the same name and owner, it is overwritten by the form from the file. If this flag is not specified, the user is prompted for verification.
dbname	The name of the database to which the forms are being copied.
filename	The name of the text file previously created by **copyform**, which contains the forms to be copied into the database.

copyrep

Copies a report specification from a database to a text file.

Syntax

copyrep [**-s**] [**-u***username*] [**-f**] [**-c***numactions*] *dbname filename report* {*report*}

Description

copyrep is a utility of ODT-DATA REPORTS that can be used, in conjunction with the **sreport** command, to copy a report from one database to another. Additionally, the two commands can be used to change ownership of a report by copying out a report owned by a particular user into a text file and copying the report back into the database under the ownership of another user.

The reports to be copied may have been created by either Report-By-Forms (RBF) or the Report-Writer. Copying a report into a new database is a two-step process. First, you must copy one or more reports from a database to a text file, using the **copyrep** command. Next, by using the **sreport** command, you can copy the reports from the text file into a database. See the *ODT-DATA Report-Writer Reference Manual* for a complete description and examples of these utilities.

The flags and parameter names for the **copyrep** command have the following meanings:

-s	Suppress status messages.
-u*username*	Copy reports owned by the stated user. This flag can be used only by the database administrator for the database or the ODT-DATA system administrator.
-f	Write the reports out in the same format as is done with the **FileReport** option in the Catalogs frame of RBF. For reports created with RBF, this strips out many of the commands.

-cnumactions	If specified, this sets the number of Report-Writer action commands to be processed within one buffer to "numactions." This can be useful to minimize real memory usage on systems where this is a concern. Default value is 32,000, which is large enough to cover all known cases. If the value is set too large, only the actual number of commands is used in computing the value.
dbname	The name of the database containing the reports.
filename	The name of a text file in which to write the report definitions.
report	The name of one or more reports that are to be written to the text file.

createdb

Creates a database.

Syntax

createdb [**-u***username*] [**-p**] *dbname* [**-c***locationname*] [**-d***locationname*] [**-j***locationname*]

Description

The **createdb** command creates a new ODT-DATA database. The person who executes this command becomes the database administrator (DBA) for the database. The database administrator has special powers not granted to ordinary users.

The name of the database to be created (**dbname**) must be unique among all ODT-DATA users. It must begin with an alphabetic character, and it may have a maximum of 12 characters.

The optional flags and their purposes are:

-u Allow the system administrator to create a database as the user specified by *username*.

-p Restrict access to the database to only the database administrator and other users specifically named in the **accessdb** command. (By default, the database is created with access permitted to all ODT-DATA users, although access to any tables in the database must be explicitly granted.) The **accessdb** command, used by the ODT-DATA system administrator, allows additional users access to a private database. For more information about the **accessdb** command, please refer to *Administering ODT-DATA* for the current ODT-DATA release.

-c Store the checkpoint files at the location specified by *locationname*. The default location is ii_checkpoint.

-d Store the database files at the location specified by *locationname*. The default location is ii_database.

-j Store the journaling files at the location specified by *locationname*. The default location is ii_journal.

Note that before you can specify any of the *locationnames* mentioned above, the location-names must be created by the ODT-DATA system administrator using **accessdb**. The procedures for creating *locationnames* are described in *Administering ODT-DATA*. If you do not specify one of the flags, the files are placed on the area corresponding to the default *locationname* for the relevant aspect of the database (that is, checkpoint, database, and journal). Note that databases and their associated journaling files should not reside on the same device.

If **createdb** fails for any reason, the partially created database should be destroyed using **destroydb.**

There are two ways to use the **-c**, **-d**, and **-j** flags to place database components in directories other than the default. This capability is particularly designed to enable you to locate various database (as well as checkpoints and journals) on different filesystems in your UNIX System installation, and thus on different disks.

One alternative is to name a directory after the flag by the end of its pathname. For example:

```
createdb -daltdir newdb
```

This command creates the "newdb" database in the **$II_DATABASE/ingres/data/altdir** instead of the **$II_DATABASE/ingres/data/default** location. Because *altdir* could be mounted as a filesystem on a UNIX System, this technique provides the capability of placing different databases on different disks. Please note that you must create such an alternate directory in 700 mode (read, write and execute permission for ODT-DATA and owned by ODT-DATA before using the directory name in a **createdb** command. The same is true for checkpoints and journals.

```
createdb -caltdir newdb
```

This command creates a database and locates its checkpoints in **$II_CHECKPOINT/ingres/ckp/altdir** instead of the **$II_CHECKPOINT/ingres/ckp/default** directory.

The second way to use the **-c**, **-d**, and **-j** flags is to supply a prefix of the directory pathname, beginning with a "/" character. Consider the command:

```
createdb -d/other newdb
```

In this case, a new database is created in a directory named **/other/ingres/data/default** as opposed to the default location. The directories at all these levels must already exist prior to executing the particular **createdb** command.

The first part of the pathname, in the previous case **/other**, can be whatever you choose, including additional directory levels. Thus, **/aa/other** would also work. (But note the limit on the number of characters, specified below.) The lower level directories, starting with "ingres," must have the same names as shown in this example.

The ownership and permissions for the sample directories should be:

```
/other/ingres                  -rwxr-xr-x
/other/ingres/data             -rwxr-xr-x
/other/ingres/data/default     -rwxrwxrwx
```

Note, however, that whichever alternative you use, the part of the directory name supplied after the **-c**, **-d**, or **-j** flags may be no more than 12 characters.

Examples

Create a private database on the default device(s).

```
createdb -p mydb
```

Create public databases under different user names.

```
createdb -ueric ericsdb
```

Create a database with files for the database, checkpoints, and journal on different devices.

```
createdb bigdb -ddb_ingres -cnewdev_ingres
-jotherdev_ingres
```

Files

$II_SYSTEM/ingres/files/dbtmplt/*
$II_SYSTEM/ingres/data/default/dbname/* [This is the default.]
$II_SYSTEM/ingres/ckp/default
$II_SYSTEM/ingres/jnl/default

destroydb

Destroys an existing database.

Syntax

destroydb [**-s**] [**-p**] [**-u***username*] *dbname*

Description

The **destroydb** command removes all references to an existing database. The directory of the database and all files in that directory are removed.

To execute this command, you must be either the database administrator for *dbname* or the ODT-DATA system administrator and have the **-s** flag specified.

The optional flags have the following meanings:

-s Indicates that you are the ODT-DATA system administrator.

-p Requires ODT-DATA to ask if you are sure that you want to destroy the database.

-u Allows the system administrator to use **destroydb** as the user specified by *username*.

Examples

```
destroydb empdata
destroydb -s empdata
destroydb -uBrad video
```

Files

$II_DATABASE/ingres/data/default/dbname/*

esqlc

Invokes the Embedded SQL/C preprocessor.

Syntax

esqlc {*flags*} {*filename*}

Description

The **esqlc** command invokes the Embedded SQL/C preprocessor. See the *ODT-DATA Embedded SQL Companion Guide for C* for a complete description of this command. The flags and parameter names have the following meanings:

-l	Write preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a filenamed *filename*.**lis**, where *filename* is the name of the input file.
-lo	Like **-l**, but the generated C code also appears in the listing file.
-f[*filename*]	Write preprocessor output to the named file. If no *filename* is specified, the output is sent to standard output, one screen at a time.
-s	Read input from standard input and generate C code to standard output. This is useful for testing statements you are not familiar with. If the **-l** option is specified with this flag, the listing file is called **stdin.lis**. To terminate the interactive session, type **Ctrl D**.
-o. *ext*	Include files preprocessed by Embedded SQL are output to a file with the given extension, *ext*. The default is **.c**.
-o	If no extension is given, output files are not generated for include files.
-{**#** | **p**}	Generate # line directives to the C compiler (by default, they are in comments). This flag can prove helpful when debugging the error messages from the C compiler.

-d	Add debugging information to the run-time database error messages generated by Embedded SQL. The source filename, line number, and the erroneous statement itself are printed along with the error message.
-?	Show what command-line options are available for **esqlc**.
filename	The input file containing the Embedded SQL program.

finddbs

Recovers databases when the database database is corrupted, or when an entry in a database is missing.

Syntax

finddbs [**-a**|**-r**] [**-p**]

Description

The **finddbs** command is used to recover ODT-DATA when the database database (iidbdb) has been corrupted. Only the ODT-DATA system administrator can use it. See *Administering ODT-DATA* for a complete description of this utility. The flags have the following meanings:

-a Run **finddbs** in analyze mode (the default), informing you of possible errors in the database table.

-r Run **finddbs** in replace mode, rebuilding the dbdb database table by scanning a list of directories for databases.

-p Cause all databases rebuilt in replace mode to be made private, except for the iidbdb. By default, replace mode makes all databases globally accessible.

ingmenu

Invokes ODT-DATA/MENU.

Syntax

ingmenu [**-e**] *dbname* [**-u***username*]

Description

The **ingmenu** command invokes ODT-DATA/MENU, a forms-based interface for accessing the capabilities and subsystems of ODT-DATA. See *Using ODT-DATA Through Forms and Menus* for a complete description of ODT-DATA/MENU. The flags have the following meanings:

-e	Invoke ODT-DATA/MENU in "empty" mode. This flag is passed to the **QBF**, **RBF**, **TABLES**, or **VIFRED** capabilities of ODT-DATA/MENU. In essence, it causes any catalog of applications, join definitions, tables, reports or other objects to be initially displayed empty, so that specific names of such objects may be entered by the user.
-u*username*	Invoke ODT-DATA/MENU as the user with the login name *username*. This flag may only be used by the database administrator for the database or by the ODT-DATA system administrator.

isql

Initiates Interactive SQL system.

Syntax

isql *dbname*

Description

The **isql** evokes the Interactive SQL system, as described in *Using ODT-DATA Through Forms and Menus*.

Example

Invoke Interactive SQL on the "employee" database.

```
isql employee
```

optimizedb

Generates statistics for use by the query optimizer.

Syntax

optimizedb [**-zf***filename*] [**-zv**] [**-zh**] [**-zk**] [**-zx**] [**-zu#**] [**-zr#**] [*ODT-DATA flags*]
dbname [{**-r***tablename* {**-a***columnname*}}]

Description

The **optimizedb** command retrieves values from the specified tables and columns. These values are used to generate statistics, which are stored in the "iistats" and "iihistograms" system catalogs. These statistics are used by the query optimizer to select an efficient query processing strategy. Such statistics should be generated for all columns that may appear in the qualification of a query statement. Statistics for columns named in the target list of a query or a query's sort list are not used. After running **optimizedb**, it is prudent to run **sysmod** to restructure the "iistats" and "iihistograms" catalogs. This is especially true the first time **optimizedb** is run on a database.

More complete and accurate statistics in the "iistats" and "iihistograms" system catalogs generally result in more efficient query execution strategies, which further results in faster system performance. The process of generating such complete and accurate statistics may require some time, but a tradeoff between accurate statistics and the time to generate them can be achieved by specifying the **-zx** flag described later. Another compromise relies on how often you regenerate the statistics. The statistics need only infrequent regeneration, usually when a significant change has occurred in the distribution of a column's values.

There need be no statistics for any columns whatsoever, and any statistics may be incorrect. The only effect is on the speed of query processing, not whether the query executes or not.

The statistics generated by the **optimizedb** command for any column consist of two basic elements: (1) the number of unique values in a column, and (2) a histogram with a variable number of variable-width cells. The accuracy of the histograms can be controlled by the **-zu#** and **-zr#** flags described later. Increasing the number of cells in the histograms increases the amount of space required for the "iihistograms" table and thus increases somewhat the amount of space and time used by the query optimizer. However, the increased accuracy of the statistics generally results in more efficient query execution strategies.

NOTE: While **optimizedb** is running, the database is not locked. Only the current table being optimized has a read lock.

The **optimizedb** command-line flags have the following functions:

-zf*filename*	If this is the first command-line flag, the following argument is taken as a filename containing all further flags on the current invocation of **optimizedb**. This file must contain only one flag per line. (Please refer to the following examples.) **NOTE:** If this flag is used, the **optimizedb** command line can contain only three items: the keyword **optimizedb**, the **-zf** flag, and the *filename*, which contains any other flags, database names, or other permissible arguments.
-zv	Verbose. Print information about each column as it is being processed.
-zh	Histograms. Print the histogram that was generated for each column. This flag also implies the **-zv** flag.
-zk	In addition to any columns specified for the table, statistics for columns that are keys on the table or are indexed are also generated.
-zx	Do minimal statistics only. Determine only the minimum and maximum values for each column. Because minimum and maximum values for columns from the same table can be determined by a single scan through the table, this flag provides a quick way to generate a minimal set of statistics.
-zr#	The histogram can contain no more than this number of cells (but see the **-zu#** flag next). Larger numbers require more processing time by the query optimizer but, because they are more accurate, generally result in more efficient query execution strategies. The default for this flag is 15.
-zu#	If there are not too many unique values for a column, then it is worth allowing more cells to produce an exact histogram. This number, therefore, specifies the number of unique values that the histogram is automatically extended to accommodate. The default for this flag is 50.

[*ODT-DATA flags*]	ODT-DATA flags on the **optimizedb** command line are automatically passed to ODT-DATA. Consult the **sql** command summary for a description of the ODT-DATA flags.
-r*tablename*	If no table name is specified, then all columns for all tables in the database are processed. Otherwise only columns for the specified tables are processed.
-a*columnname*	If the **-r***tablename* flag is specified, then (and only then) can individual columns be specified for the generation of statistics. When table(s) and column(s) are specified, then statistics processing occurs only for the specified columns (but see the **-zk** flag earlier in this table).

Some possible diagnostic messages you may receive and their causes are:

`More than 1000 arguments`	There are too many lines in the argument file, specified with the **-zf** flag.
`Bad unique cells value`	The value specified in the **-zu#** flag was not a number, was less than 1 or greater than 249.
`Bad regcells value`	The value specified in the -zr# flag was not a number, was less than 2 or greater than 499.

Examples

Generate full statistics for all columns in all tables in the "empdata" database.

```
optimizedb empdata
```

Generate statistics for key or indexed columns in the "employee" and "dept" tables, and additionally generate statistics for the "dno" column in the "dept" table.

```
optimizedb -zk empdata -remployee -rdept -adno
```

Do the same as the second example, but from a file.

```
optimizedb -zf flagfile
```

The "flagfile" contains:

```
-zk
empdata
-remployee
-rdept
-adno
```

Generate statistics for all key or indexed columns in "employee," "dept" and "salhist." Also process the "eno" column in "employee," whether "eno" is a key or an indexed column or not. Generate statistics with only minimum and maximum values from the columns. Print status information as each column is processed.

```
optimizedb -zk -zv -zx empdata -remployee -aeno       -rdept
-rsalhist
```

Allow up to 100 unique values from each column in the "employee" table before merging adjacent values into the same histogram cell.

```
optimizedb -zu100 empdata -remployee
```

printform

Places an image of the form and a description of the form and its fields into a text file.

Syntax

printform [-u*username*] *dbname form txtfile*

Description

The **printform** command places an image of the form and a description of the form and its fields into a text file.

The parameter names for **printform** have the following meanings:

-u*username*	if specified, prints the form owned by the stated user. You must be the database administrator or an ODT-DATA super user to use this flag.
dbname	is the name of the database.
form	is the name of the form. You may print only one form at a time.
txtfile	is the name of the text file to which **printform** prints the form.

Example

To print the form "employees," which is stored in the "emp" database, into the file **emp.prf**, use the following command line:

```
printform emp employees emp.prf
```

qbf

Invokes ODT-DATA/Query-By-Forms (QBF).

Syntax

qbf *dbname* [**-s**] [**-m***mode*] [**-u***username*] [[**-f**|**-j**|**-t**|**-l**] [*querytarget*]

Description

The **qbf** command invokes Query-By-Forms, a forms-based interface for manipulating data in a database. See *Using ODT-DATA Through Forms and Menus* for a complete description of this system. The flags and parameter names have the following meanings:

dbname	The name of the database.
-s	Put QBF into silent mode, eliminating verbose messages.
-m*mode*	Bypass the Join Definition phase of QBF, putting you directly into the *mode* function for Query Execution where *mode* is **retrieve**, **append**, **update**, or **all**. If you use the **-m***mode* flag, you must also specify a *querytarget* on the command line.
-u*username*	Invoke QBF as if you were the user with the login name *username*. This flag can be used only by the database administrator for the database or the ODT-DATA system administrator.
querytarget	Either a QBFName, JoinDef, or Table you want to access in your query. Specifying *querytarget* brings you directly into Query Execution phase. If you specify it without also specifying **-m***mode*, you have the option of switching to the Join Definition phase. You can specify the type of *querytarget* to QBF by using the **-f**, **-j**, or **-t** flag, as described below. An alternative is to use the **-l** flag described below and let QBF figure out the *querytarget*'s type. If no flag is specified for *querytarget*, QBF assumes that the type is Table and generates an error if it cannot find a table with that name.

-f Indicate that *querytarget* is a QBFName. This invokes QBF with a VIFRED (Visual-Forms-Editor) Form.

-j Indicate that the *querytarget* is a JoinDef.

-t Indicate that *querytarget* is a Table. This flag further indicates that a table field format is used to query the Table.

-l Cause QBF to look for a QBFName first, then a JoinDef, and finally a Table until it finds the *querytarget* specified.

query

Invokes Query Execution phase of ODT-DATA/Query-By-Forms (QBF).

Syntax

query *dbname* [**-m***mode*] [**-u***username*] [**-fl-jl-t**] *querytarget*

Description

The **query** command invokes the Query Execution phase only of Query-By-Forms, a forms-based interface for manipulating data in a database. Through the Query Execution phase you can append, retrieve, or modify data. See *Using ODT-DATA Through Forms and Menus* for a complete description of Query Execution. The flags and parameter names have the same meaning as for the **qbf** command, except that here *querytarget* is required. Unless otherwise specified, **query** uses the same order for looking up *querytarget* as the flag **-l** in the **qbf** command - QBFName, JoinDef, Table.

rbf

Invokes ODT-DATA/Report-By-Forms (RBF).

Syntax

rbf [**-s**] [**-u***username*] [**-r**]|[**-m**[*mode*]] [**-l***mxline*] [**-c***numactions*] [**-e**] *dbname* [*reportname*|*tablename*]

Description

The **rbf** command invokes Report-By-Forms, a forms-based interface for specifying reports. See *Using ODT-DATA Through Forms and Menus* for a complete description of this system. The flags and parameter names have the following meanings:

-s	Request that status messages, including prompts, be suppressed.
-u*username*	Request that RBF pretend you are the user with login name *username*. This can only be used by the database administrator for a database, or by the ODT-DATA system administrator.
-r	Tell RBF that a report and *not* a table is specified on the command line. This returns an error if the named report is not found. Without the **-r** flag, RBF first looks for the named report, and if the report is not found, and a table with the same name exists, a default report for that table is set up.
-m[*mode*]	Tell RBF that a table and not a report is specified. This instructs RBF to format a default report for the specified table and not to check for a report of the given name first. If the optional *mode* value of **wrap**, **column**, or **block** is specified after the **-m** flag, that style of report is used rather than the default.

-l*mxline*	The line length to use when generating default reports. By default, default reports use a line length appropriate to the type of terminal on which they are run (either 80 or 132 characters). This default can be changed by using the **-l** flag.
-c*numactions*	Set the number of Report-Writer action commands to be processed within one buffer to the *numactions* value. This can be used to minimize real memory usage on systems where this is a concern. The default value of 32,000 is large enough to cover all known cases. If the value is set too high, only the actual number of commands is used in computing the value.
-e	Cause the RBF Reports Catalog frame to appear without data in its table field. This flag is designed to accelerate the process of selecting a report definition for editing, for the benefit of users who are quite familiar with the contents of a database's reports catalog. To use this flag with a particular report definition, move the cursor to the **Name** column, enter the desired report name and select the appropriate operation.
dbname	The name of the ODT-DATA database containing the report data.
reportname	The name of a report as specified in a previous RBF session.
tablename	The name of a table or view in your database for which you want a default report formatted.

report

Runs a default report or a report created with the **sreport** or **rbf** command.

Syntax

report [**-c***numactions*] [**-f***filename*] [**-s**] [**-u***username*] [**-r**]|[**-m**[*mode*]] [**-l***mxline*] [**-q***mxquer*] [**-w***mxwrap*] [**+t**|**-t**] [**+b**|**-b**] [**-h**] [**-5**] [**-v***pagelength*] *dbname reportname*|*tablename* ['({*parameter=value*})']

Description

The **report** command writes a report set up by the **sreport** or **rbf** commands, or sets up a default report for a table in the database. See *Using ODT-DATA Through Forms and Menus* and the *ODT-DATA Report-Writer Reference Manual* for a complete description of this command. The flags and parameter names have the following meanings:

-c*numactions*	If specified, this sets the number of Report-Writer action commands to be processed within one buffer to **numactions**. This can be useful to minimize real memory usage on systems where this is a concern. Default value is 32,000, which is large enough to cover all known cases. If the value is set too large, only the actual number of commands is used in computing the value.
-f*filename*	Direct the formatted report to *filename* for subsequent output. If this option is not specified, the report is written to the standard output file (normally your terminal), or, in the case of a report specified by the Report-Writer, to the file designated by the **.output** command in the report specification file.
-s	Request that status messages, including prompts, be suppressed.
-u*username*	Request that the Report-Writer pretend you are the user with login name *username*. This can only be used by the database administrator for a database, or by the ODT-DATA system administrator.

-r Tell the Report-Writer that a report is being specified, rather than a table. This returns an error if no report with the given name is found. By default, the Report-Writer first looks for a report of the given name, and if one is not found, and a table of the given name does exist, a default report for that table is set up.

-m[*mode*] Tell the Report-Writer that a table has been specified, rather than a report. This instructs the Report-Writer to format a default report for the specified table and not to check for a report of the given name first. If the optional *mode* value of **wrap**, **column** ,or **block** is specified after the **-m** flag, that style of default report is used rather than the default.

-l*mxline* Set the maximum output line size to *mxline* characters. By default, the maximum output line size is 132 characters if output is to a file; otherwise, the default maximum line size is the width of the terminal. This option is needed only if reports are written that contain unusually long lines.

-q*mxquer* Set the maximum length of the query after all substitutions for run-time parameters have been made to *mxquer* characters. By default, the maximum query size is 1000 characters. This option is needed only for particularly long queries.

-w*mxwrap* Set *mxwrap* as the maximum number of lines to wrap with one of the column "C" formats, or the maximum number of lines that can be used within any block. By default, the maximum value is 100 lines. This means that a column written with a format such as "c0.20" (which writes a character string in a column 20 characters wide) contains a maximum of 100 lines. This maximum is provided as a protection against misspecified columns and is rarely needed.

-t|+t if turned on (**+t**), this flag causes aggregates and breaks to occur over rounded values for any floating-point column whose format has been specified in a **.format** command as numeric **F** or template. Each value in the column is rounded to the precision given by its format. Additionally, breaks for date columns using a date template occur over the actual values appearing for the dates. **+t** is the default. If this flag is turned off (**-t**), aggregates and breaks use the underlying values, not the rounded values.

-b|+b If turned on (**+b**), this flag forces formfeeds at the end of each page. If turned off (**-b**), this flag suppresses formfeeds for the end of each page. The flag overrides any **.formfeed** or **.noformfeed** command occurring in the report specification file.

-h If specified, a report that retrieves no rows is provided a null set of data. All header and footer sections are executed. The detail section is suppressed.

-a If specified, the report is made compatible with version 5.0 of ODT-DATA. (The default is that the -a flag is not specified.) To ensure compatibility, the following assumptions are made:

- I+t option is the default.
- Only floating-point arithmetic is used. Integer columns are converted to floating-point before use in computation.
- The month part of the *current_date* function is displayed in capitals if no format is specified. Normally, the system displays the month names in lowercase letters. For example, what is now displayed as "01-feb-1985" would, with the flag set, be displayed as "01-FEB-1985".

-v*pagelength*	If specified, this flag sets *pagelength* as the number of lines for each page of output. *pagelength* must be a positive integer. This flag overrides any **.pagelength** command in the report specification file. The default is 61 lines per page if the report is written to a file, and 23 lines per page if written to a terminal.
dbname	The name of the ODT-DATA database containing the report data.
reportname	The name of a report as specified in a report specification that is set up with RBF or **sreport**.
tablename	The name of a table or view in your database for which you want a default report formatted.
parameter	The name of a parameter used in the report. Parameter/value combinations on the command line may be separated by blanks, tabs, or commas.
value	The value that is replaced for every occurrence of the corresponding *parameter* name in the report specifications. *value* should be surrounded by quotes (which are removed when it is processed) if you want to pass through a string or date value.

rolldb

Recovers the database from the last checkpoint and the current journal.

Syntax

rolldb [+c|-c] [+j|-j] [-m*device:*] [-s] [-u*username*]
[-v] [+w|-w] {*dbname*}

Description

The **rolldb** command recovers the named database(s) from the last checkpoint and the current journal. The recommended procedure is to recover the last checkpoint, then recover from the journal (see the following examples).

The command line flags have the following interpretations:

+c|-c	Recover/do not recover the database from the last checkpoint. The flag defaults to **+c**.
+j|-j	Recover/do not recover the database from the journal. The flag defaults to **+j**.
-m	Recover the checkpoint from the specified tape device rather than from disk.
-s	Invoke "super user" (system administrator) status for system-wide access to any database. You must be the system administrator.
-u*username*	Pretend you are the user with login name *username*. This may only be used by the database administrator for the database or by the ODT-DATA system administrator.
-v	Recover the database from the journal in verbose mode, which provides diagnostic information detailing all operations executed during the recovery process.
+w|-w	Wait/do not wait for the database to be "free." The flag defaults to **-w**.

If you have written to tape the checkpoint from which you want to restore the journal, you can use the **-m** flag to read in the checkpoint from a tape device.

Only the database administrator, who created the database, or the ODT-DATA system administrator (if the **-s** flag is specified) may run the **rolldb** command on a database.

If no databases are specified, all databases for which you are the database administrator are affected. All databases can be purged if the ODT-DATA system administrator uses the -s flag.

The **rolldb** command locks the database because errors can occur if the database is active while the **rolldb** command is running. If a database is busy, the **rolldb** command reports this and proceeds to the next database, if any. If the **-w** flag is specified, the **rolldb** command does not wait, regardless of standard input. The **+w** flag always causes the **rolldb** command to wait.

Some possible diagnostic messages you may receive and their causes are:

`You are not a valid ODT-DATA user`	The current login name is not entered in the ODT-DATA users file.
`You may not use the -s flag`	You have tried to use the -s flag, but you do not have ODT-DATA system administrator privileges.
`You are not the dba for` *dbname*	You have tried to recover a database for which you are not the database administrator.
`Cannot enter` *dbname*	The database does not exist.

Examples

Recover the "empdata" database from the last checkpoint and journal. This assumes that both the journal and the checkpoint are currently online. If not, they should be placed online before executing these commands.

```
rolldb -v empdata
```

Recover all databases for which you are database administrator.

```
rolldb -v
```

Recover "empdata" from tape, and then apply the journals.

```
rolldb +c +j -m/dev/rmt0 empdata
```

sql

Invokes the ODT-DATA relational database management system.

Syntax

sql [*flags*] [<*altin*] [*altout*] *dbname*

Description

This command invokes ODT-DATA. *dbname* is the name of an existing database. The optional flags have the following meanings:

+U|-U	Enable/disable user updating of the system catalog tables and secondary indexes. You must have the "update system tables" privilege obtained through **accessdb**. This option is provided for system debugging and is strongly discouraged for normal use. The default is **" Note that this flag causes an exclusive lock of the database during the session for which it is specified."**
-u*username*	Pretend you are the user with login name *username* (found in the users file). If *name* is of the form *:xx*, then *xx* is the two-character user code of a user. This may only be used by the database administrator for the database or by the ODT-DATA system administrator.
-c*N*	Set the minimum field width for printing character columns to *N*. The default is 6.
-t*N*	Set the minimum field width for printing text columns to *N*. The default is 6.
-i*kN*	Set integer output column width to *N*. *k* may be 1, 2, or 4 for i1's, i2's, or i4's, respectively. The default for N is 6 for i1 and i2 fields, and 13 for i4 fields.

-f*kxM.N* Set floating-point output column width to *M* characters (total), including *N* decimal places, and (if warranted) *e+-xx* and the decimal indicator character itself. *k* may be 4 or 8 to apply to f4's or f8's, respectively. *x* may be **E, F, G,** or **N** (upper- or lower-case) to specify an output format. For a number to be displayed in **E** (that is, exponential) format, either **E** must be specified in the flag or the number must be too large for the format indicated in the flag. **E** is exponential form, **F** is floating-point form, and **G** and **N** are identical to **F** unless the number is too large to fit in that field, when it is output in **E** format. **G** format guarantees decimal-point alignment; **N** does not. The default display format for both f4 and f8 is **n10.3**, unless your computer supports the IEEE standard for floating-point numbers, in which case the display format for f4 and f8 is **n11.3.**

-v*X* Set the column separator for retrievals to the terminal and print commands to be *X*. The default is vertical bar (|).

-n*M* Set modify mode on the index command to *M*. *M* can take as values any of the storage structures described in the **modify** command, which are **heap, cheap, heapsort, cheapsort, isam, cisam, btree, cbtree, hash** and **chash. heap,** and **btree.** The default is **isam.**

+a|-a Set/clear the autoclear option in the Terminal Monitor. It defaults to +a.

-l Lock the database for your exclusive use. When you specify this flag, no one else can open the database while you are in it. If you attempt to use this flag on a database that is already opened, the system informs you that the database is temporarily unavailable.

+d|-d Print/do not print the dayfile. It defaults to **+d.**

+s|-s Print/do not print any of the monitor messages, including prompts. This flag is normally set to **+s.** If cleared, it also clears the **-d** flag. It defaults to **+s.**

+w|-w

Wait/do not wait for the database. If the +w flag is present, ODT-DATA waits, provided that certain processes are running (**sql -l**, **sql -U**, **verifydb**, **rollforwarddb** and/or **sysmod**) on the given database. Upon completion of those processes, ODT-DATA proceeds. When the -w flag is present, a message is returned and execution is stopped if the database is not available. When the +w|-w flag is omitted and the database is unavailable, then the error message is returned if ODT-DATA is running in foreground (more precisely, if the standard input is from a terminal). Otherwise the wait option is invoked.

NOTE: This flag can be used only in interactive sessions and not in batch mode. The flag defaults to **-w**.

-x*k*

Set arithmetic handling mode. *k* may be **f** or **w**. **f** indicates that all arithmetic exceptions (floating overflow and underflow, integer overflow, and divide by zero) should be treated as fatal errors. In this mode, the detection of an arithmetic exception terminates query processing. **w** indicates that warning messages should be generated for arithmetic exceptions. In this mode, the query is run to completion, and a summary of exceptions detected is generated. The default condition is to ignore exceptions.

<*altin*

Use an alternate file to input Terminal Monitor commands to ODT-DATA. The file *altin* should contain all the terminal monitor commands needed to run an ODT-DATA session. This can be used to run "canned" ODT-DATA procedures, such as processing the output of the **copydb** command.

altout

Use an alternate file for all output from the terminal monitor. This option can capture the output of a terminal session for later reference.

NOTE: You do not see any output from ODT-DATA if you use this option.

Some possible diagnostic messages you may receive and their causes are:

`Too many options to ODT-DATA`	You have stated too many flags as ODT-DATA options.
`Bad flag format`	You have stated a flag in an unintelligible format, or an entirely bad flag.
`Too many parameters`	You have given a database name and "something else," which ODT-DATA cannot decipher.
`No database name specified`	
`Improper database name`	The database name is not legal.
`You may not access database` *name*	According to the users file, you do not have permission to open this database.
`You are not authorized to use the` *flag* `flag`	The flag specified requires some special authorization, which you do not have.
`Database` *name* `does not exist`	
`You are not a valid ODT-DATA user`	You are not entered into the user's file; you may not use ODT-DATA at all.
`Database temporarily unavailable`	Someone else is currently performing some operation on the database; you cannot start ODT-DATA now.
`Error starting up ODT-DATA: 19: Request for lock failed`	The database you tried to access is currently exclusively reserved for another user.

Examples

Open the "empdata" database.

```
sql empdata
```

Open "empdata," suppressing the dayfile message.

```
sql -d empdata
```

Open "empdata," suppressing the dayfile message and the Terminal Monitor prompts and messages; read into the workspace the contents of the **batchfile** file.

```
sql -s empdata batchfile
```

Open "empdata," display f4 columns in G format with two decimal places and i1 columns with three spaces.

```
sql -f4g12.2 -i13 empdata
```

Files

$II_SYSTEM/ingres/files/users
$II_DATABASE/ingres/data/default/dbname/*

sreport

Sets up report specifications created with the Report-Writer in a database.

Syntax

sreport [**-s**] [**-u***username*] *dbname filename*

Description

The **sreport** command writes a report definition specified with the Report-Writer report definition language into the database. This command may also be used in conjunction with the **copyrep** command to copy a report from one database to another, or to change a report's ownership. See the *ODT-DATA Report-Writer Reference Manual* for a complete description of **sreport.** The flags and parameter names have the following meanings:

-s	Suppress status messages.
-u*username*	Request that **sreport** act as if you are a user with login name *username* when creating a report. This may only be used by the database administrator for the database or by the ODT-DATA system administrator.
dbname	The name of the ODT-DATA database that is to contain the report.
filename	The name of a text file containing report-formatting commands for one or more reports.

statdump

Prints statistics contained in the "iistats" and "iihistograms" system catalogs.

Syntax

statdump [**-zq**] [**-zdl**] [*ODT-DATA flags*] *dbname* [{**-r***tablename* {**-a***columnname*}}]

Description

The **statdump** command allows you to inspect the "iistats" and "iihistograms" tables. These system tables contain statistical information about columns used by the query optimizer as it selects an efficient query processing strategy. The data in these tables are usually loaded by the **optimizedb** command.

The command-line flags have the following meanings:

-zq	Quiet mode. Print out only the information contained in the "iistats" table and not the histogram information contained in the "iihistograms" table.
-zdl	Delete statistics from the system catalogs. When this flag is included, the statistics for the specified tables and columns (if any are specified) are deleted rather than displayed.
[*ODT-DATA flags*]	Pass any of these flags to ODT-DATA. For more information about the ODT-DATA flags, refer to the **sql** command description in this chapter.
-r*tablename*	Produce statistics for all columns in the table specified. If no table is specified, then statistics for all columns in all tables are produced.
-a*columnname*	Produce statistics for the specified column(s) only. Note that to specify individual columns, you must first specify a table name with the **-r** flag, as the syntax summary indicates.

Examples

Print the statistical information for all columns in the "employee" table in the "empdata" database.

```
statdump empdata -remployee
```

For all columns in all tables of the "empdata" database, print out only the information in the "iistats" system table.

```
statdump -zq empdata
```

Delete statistics for all columns in the "employee" table.

```
statdump -zdl empdata -remployee
```

NOTE: If a specified table or column cannot be found, then a warning message is printed and processing continues.

sysmod

Modifies system tables to predetermined storage structures.

Syntax

sysmod [-s] [+|-w] *dbname* [*tablename* { , *tablename*}]

Description

The **sysmod** command modifies a database's system tables to the most appropriate storage structure, usually hash, for accelerating query processing. You can run **sysmod** on the whole database or on specified tables. The user must be either the database administrator for the specified database or the ODT-DATA system administrator, in which case the -s flag must be specified.

The flags have the following meanings:

- **-s** Allow the ODT-DATA system administrator to use **sysmod** on another user's database.

- **-w** Cause ODT-DATA to wait or not wait until the database is free before executing **sysmod**. This can only be used in interactive sessions, not in batch mode.

  sysmod locks the database while it modifies the system tables, to prevent errors. If the database is in use, **sysmod** reports that the database is not free, and **sysmod** does not execute. If standard input is not a terminal, **sysmod** waits for the database to be free. If the **-w** flag is stated, **sysmod** does not wait, regardless of standard input. The **+w** flag causes **sysmod** to wait until the database is no longer in use, regardless of standard input.

sysmod should be run on a database periodically to maintain peak performance. This is particularly true whenever many tables and secondary indexes are created and/or destroyed, in which case **sysmod** should be run even more often.

Examples

Optimize the system tables in "empdata."

```
sysmod empdata
```

Optimize the "iirelation" and "iiindexes" system tables in "empdata," but only if the database is not currently busy.

```
sysmod -w empdata iirelation iiindexes
```

unloaddb

Creates command files for complete unloading and reloading of a database.

Syntax

unloaddb [-u*username*] [-c] [-d*pathname*] [-l*sql*] *dbname*

Description

The **unloaddb** command creates a set of command files that can be run by the database administrator for a database to unload all tables in the database. The **unloaddb** utility works in the same way as the **copydb** command except that it also unloads all views, integrity constraints, permissions, forms, graphs, and report definitions in the database. Also, unlike the **copydb** command, **unloaddb** unloads all user-defined tables, views, and so on, in the database of which you are database administrator, not merely those items that you own. This utility can be used when a database must be totally rebuilt or for checkpointing the database.

The **unloaddb** utility creates two command files in the current directory that can then be executed by the database administrator:

- **unload.ing** contains commands to read sequentially through the database, copying every user table into its own file in the named directory.
- **reload.ing** contains commands to reload the database with the information contained in the files created by the **unload.ing** command file.

Note that the **unloaddb** command does not actually do the unloading or reloading of the database. The command files created by **unloaddb** must be executed by the database administrator to accomplish these tasks. The directory specified in the **unloaddb** command must not be the actual database directory **$II_DATABASE/ingres/data/default/dbname** because the files created by **unloaddb** may have the same names as the tables in the database.

The optional flags and their purposes are:

-u Allows you to run **unloaddb** as the user specified by *username*.

-c Causes the commands in the generated command files to use a portable format. That is, all data are copied in and out as ASCII characters. This is useful for transporting databases between computer systems whose internal representations of non-ASCII data differ.

-d Stores the **unload.ing** and **reload.ing** in the location specified by *directory-specification* instead of the default current directory. The specification may be either a full or relative directory specification.

Because databases recreated with the **reload.ing** file from **unloaddb** are new databases, you should be sure to run the **sysmod** command after recreating the database to re-optimize performance.

It is important that the database be recreated with reloading before doing any work (for example, creating tables, forms, reports, and so on.) in the new database.

Example

Unload and reload the "empdata" database.

```
cd /mydir/backup unloaddb empdata unload.ing destroydb
empdata createdb empdata reload.ing sysmod empdata
```

The **unloaddb** command uses a version of the **copydb** utility to generate the copy commands in the **unload.ing** and **reload.ing** files. Thus all limitations of the **copydb** command apply to the **unloaddb** command.

vifred

Invokes the ODT-DATA/Visual-Forms-Editor (VIFRED).

Syntax

vifred *dbname* [*objectname* [**-f**|**-t**|**-j**]] [**-e**] [**-u***username*]

Description

The **vifred** command invokes the Visual-Forms-Editor, a forms-based interface for editing the appearance of a form. See *Using ODT-DATA Through Forms and Menus* for a complete description of this system. The flag and parameter names have the following meanings:

dbname	The name of an ODT-DATA database.
objectname	If specified, is the name of a form, table, or joindef.
-f	If specified, indicates that *objectname* is the name of an existing form, already created with VIFRED. By default, *objectname is* a form name; therefore, this flag may be omitted.
-t	If specified, indicates that *objectname* is a database table name.
-j	If specified, indicates that *objectname* is the name of a joindef.
-e	If specified, starts VIFRED with an empty table field in the Catalog frame. The user can then access the desired form directly by entering its name in the table field.
-u*username*	Request that the Visual-Forms-Editor act as if you are a user with login name *username*. This may be used only by the database administrator for the database or by the ODT-DATA system administrator.

Appendix A
Keywords

ODT-DATA SQL

The following identifiers are keywords in ODT-DATA SQL:

abort	**count**	**endwhile**	**is**	**privileges**	**then**
all	**create**	**execute**	**like**	**procedure**	**to**
and	**current**	**exists**	**max**	**public**	**union**
any	**cursor**	**for**	**message**	**relocate**	**unique**
as	**declare**	**from**	**min**	**return**	**until**
asc	**delete**	**grant**	**modify**	**revoke**	**update**
alter	**desc**	**group**	**not**	**rollback**	**user**
at	**describe**	**having**	**null**	**save**	**using**
avg	**distinct**	**if**	**of**	**savepoint**	**values**
between	**do**	**immediate**	**on**	**select**	**where**
by	**drop**	**in**	**open**	**set**	**while**
check	**else**	**index**	**or**	**some**	**with**
close	**elseif**	**insert**	**order**	**sql**	**work**
commit	**endif**	**integrity**	**permit**	**sum**	
copy	**endloop**	**into**	**prepare**	**table**	

ODT-DATA Embedded SQL

The following list contains the keywords specific to ODT-DATA Embedded SQL. Note that all the SQL keywords listed above are also reserved in Embedded SQL.

activate	**deleterow**	**getform**	**loadtable**	**scroll**
addform	**disconnect**	**getoper**	**menuitem**	**scrolldown**
breakdisplay	**display**	**getrow**	**message**	**scrollup**
call	**down**	**go**	**next**	**sleep**
clear	**enddata**	**goto**	**notrim**	**stop**
clearrow	**enddisplay**	**help**	**open**	**submenu**
close	**endforms**	**helpfile**	**out**	**tabledata**
column	**endloop**	**identified**	**print**	**unloadtable**
command	**endselect**	**include**	**prompt**	**up**
connect	**fetch**	**indicator**	**putform**	**validate**
continue	**field**	**initialize**	**putrow**	**validrow**
current	**finalize**	**inittable**	**redisplay**	**whenever**
cursor	**formdata**	**inquire_frs**	**repeated**	
declare	**forminit**	**inquire_ingres**	**resume**	
descriptor	**forms**	**insertrow**	**screen**	

Double Reserved Words

The following words are reserved when they appear together on the same line with only spaces separating them.

add node
begin declare
begin transaction
create link
create permanent
create temporary
drop link
drop permanent
drop temporary
end transaction
remove node

ANSI SQL

The following list comprises the proposed ANSI standard keywords that are not currently reserved in ODT-DATA SQL or ODT-DATA Embedded SQL. You may wish to treat these as reserved words to ensure compatibility with other implementations of SQL.

authorization	**double**	**module**	**public**
char	**float**	**numeric**	**real**
character	**fortran**	**option**	**schema**
cobol	**found**	**pascal**	**smallint**
constraints	**int**	**pli**	**sqlcode**
dec	**integer**	**precision**	**sqlerror**
decimal	**language**	**procedure**	

Host Language Keywords

You cannot use host language keywords, including language-defined data types, as objects in Embedded SQL statements.

Appendix B
The ODT-DATA System Catalogs

This appendix describes the Standard Catalog Interface views, the Extended System Catalogs, and lists the DBMS System Catalogs.

System catalogs are tables, just like user tables in a database. Each system catalog has a distinct set of columns (attributes), each of which has a distinct database management function. These catalogs can be used in programs to access (but not update) information about the system. Each row in a system catalog reflects some aspect of the database.

The Standard Catalog Interface is a group of views defined on the system catalogs. These views are the supported catalogs, and users who need to query the system catalogs should use them.

To reduce coding time, the definitions for the columns in the catalogs let the programmer know (1) that all values are left-justified in a column unless otherwise noted, (2) that all columns are uppercase unless otherwise noted, and (3) what are valid values for the column.

Columns are assumed to be non-nullable, except where explicitly noted.

Many columns that are **char(25)** names are valid ODT-DATA names. ODT-DATA names are described in Chapter 1, "SQL Syntax."

Allowable values for those columns described as ODT-DATA usernames are determined by operating systems in general, but should be drawn from the list of values in the **iidbconstants** catalog, which contains the current username and current dbaname.

All **char(24)** fields described as ODT-DATA standard dates have the following format:

yyyy_mm_dd hh:mm:ss GMT

where:

yyyy	is the year (for example, 1987)
mm	is the month (for example, 11)
dd	is the day of the month (for example, 21)
hh	is the military hour (for example, 14)
mm	is the minute (for example, 43)
ss	is the second (for example, 32)
GMT	is for Greenwich Mean Time

The underscores and colons are required between the parts of the date.

Standard Catalog Interface

All database users can read the Standard Catalog Interface views, but they may only be updated by a privileged ODT-DATA user who specifies the +U flag when the database is accessed.

The iidbcapabilities Catalog

The **iidbcapabilities** view contains information about the capabilities the DBMS provides.

The following table describes the columns in the **iidbcapabilities** catalog:

Column Name	Data Type	Description
cap_capability	**char(24)**	Contains one of the values listed in the following table. If the cap_capability has a value, it is activated by the value in the cap_value column.
cap_value	**char(24)**	Most capabilities are binary and ise set to the string "Yes" or "No"; either the DBMS supports them or it does not. Some, however, have values. For those, this field contains the value of the capability.
cap_description	**varchar(240)**	Contains a description of the capability this row represents.

The cap_capability column in the **iidbcapabilities** catalog contains one or more of the following values:

Capability	Value
DB_NAME_CASE	The type of case sensitivity the database has with respect to database objects. It takes on the value of "LOWER," "UPPER," or "MIXED." If not present, this capability defaults to "LOWER." Database objects may be specified in programs and queries in either mixed, lower, or uppercase if the value is "LOWER" or "UPPER." If the value is "MIXED," be careful to preserve the case specified by the user for database objects. Database objects are stored in the system catalogs, as specified by DB_NAME_CASE. For database and user names, the names are stored in uppercase if the value of DB_NAME_CASE is "UPPER" or "MIXED." If is it "LOWER," then they are stored in lowercase in the system catalogs. Note that this applies only to database objects (tables, columns, and users). Front-end objects, such as forms and reports, are always lowercase.
UNIQUE_KEY_REQ	Set to "Yes" if the database service requires that all tables have a unique key. Set to "No" or not present if the database service allows tables without unique keys.
INGRES	Set to "Yes" if the DBMS supports, in ALL respects, 100% of ODT-DATA. Otherwise "No," Defaults to "Yes."
INGRES/SQL_LEVEL	Version of INGRES/SQL support provided by the DBMS. Examples: 00600 - DBMS supports INGRES/SQL Version 6.0 00601 - DBMS supports INGRES/SQL Version 6.1 00000 - DBMS does not support INGRES/SQL Default is 00600.

Capability	Value
OPEN/SQL_LEVEL	Version of OPEN/SQL support provided by the DBMS. Examples: 00600 - DBMS supports OPEN/SQL Version 6.0 Default is 00600.
SAVEPOINTS	"Yes" if savepoints behave exactly as in ODT-DATA, else "No." Default is "Yes."
DBMS_TYPE	What type of DBMS the application is communicating with. Valid values are the same as those accepted by the *with DBMS* = clause used in queries. Some are "INGRES," "RMS." The default value is "INGRES."

The iidbconstants Catalog

The **iidbconstants** view contains a list of values that must be known by the ODT-DATA frontends.

The following table describes the columns in the **iidbconstants** catalog:

Column Name	Data Type	Description
user_name	**char(24)**	The name of the current user.
dbaname	**char(24)**	The name of the db's owner.

The iitables Catalog

The **iitables** catalog contains an entry for each queryable object in the database. In ODT-DATA, these objects are tables, views, and indexes. The **iitables** catalog contains basic system-independent logical information. User programs can query this catalog to find out what tables, views, and indexes exist in a database.

In ODT-DATA, this view is keyed on table_name and table_owner, so the best way to query this view is with a query such as:

```
select      *
from        iitables
where       (table_name = (anyname))
and         (table_owner = (myname) or table_owner =
            (dbaname))
```

Column Name	Data Type	Description
table_name	**char(32)**	The object's name. This is an ODT-DATA name.
table_owner	**char(32)**	The object's owner, expressed as an ODT-DATA user-name. Generally the creator of the object.
create_date	**char(25)**	The object's creation date, expressed as an ODT-DATA standard date. This is blank if unknown.
alter_date	**char(25)**	The last time this table was altered, expressed as an ODT-DATA standard date. For ODT-DATA tables, this date is the same as the create date. This date is updated whenever the logical structure of the table changes, either through changes to the columns in the table or changes in the primary key itself. Physical changes to the table, such as changes to data, indexes, or physical keys, do not change this date. This is blank if unknown.

Column Name	Data Type	Description
table_type	**char(8)**	This describes the type of the query object. The possible values are: "T" if the object is a table "V" if the object is a view "I" if the object is an index Further information about tables can be found in system catalogs **iiphysical_tables** and about views in **iiviews.**
table_subtype	**char(8)**	This describes the type of table or view that this is. Possible values are: "No" - (native) for standard ODT-DATA databases "I" - (imported tables) for gateways " " - for unknown
table_version	**char(8)**	This is the version of the object, which allows the frontends to determine where additional information about this particular object is stored. This reflects the database type, as well as the version of an object within a given database. For ODT-DATA tables, the value for this field is "ING6.0".
system_use	**char(1)**	Specifies whether the object is a system object or a user object. The system_use field is used by the front ends to screen lists of tables in catalog displays. Values are " " (if unknown). The distinction between "S" and "U" is used in utilities to know which tables need reloading. If the value is unknown, the utilities use the naming convention of "ii" for tables to distinguish between system and user catalogs. Also, any table beginning with **ii_** is assumed to be a frontend object, rather than a DBMS system object. The standard system catalogs themselves must be included in the **iitables** catalog and are considered system tables.

The following columns in **iitables** have values only if the table_type is "T" or "I."

Gateways that do not supply this information must set these columns to the default values: -1 for numeric data types and a blank for character data types.

Column Name	Data Type	Description
table_stats	**char(8)**	"Yes" if this object has entries in the **iistats** table, or "No" if this object does not have entries. Whether this is blank or not is not a determinant of "Yes" or "No." If the field is blank, then a probe of the **iistats** table should be done to determine if they exist. This column is used only for optimization of ODT-DATA databases.
table_indexes	**char(8)**	"Yes" if this object has entries in the **iiindexes** table that refer to this as a base table, or "No" if this object does not have entries. Whether this is blank or not is not a determinant of "Yes" or "No." If the field is blank, then a probe of the **iiindexes** table on the **base_table** column should be done to determine if they exist. This field is used only for optimization of ODT-DATA databases.
is_readonly	**char(8)**	"No" if updates are physically allowed, or "Yes" if no updates are allowed. This is blank if it is unknown. This is used for tables that are defined to a gateway only for retrieval, such as tables in hierarchical database systems. If this field is set to "Yes" then no updates works, independent of what permissions might be set. If it is set to "No," updates may be allowed, depending on whether the permissions allow it or not.
num_rows	**integer**	The estimated number of rows in the table. Set to -1 if it is unknown.

Column Name	Data Type	Description
storage_structure	**char(16)**	The storage structure for the table. It is one of the following: "HEAP" if the table is a heap "HASH" if the table is a hash structure "ISAM" if the table is an isam structure "BTREE" if the table is a btree, Blank if the table structure is unknown.
is_compressed	**char(8)**	Set to "Yes" if the table is stored in compressed format, or "No" if the table is uncompressed. This is blank if this is unknown.
duplicate_rows	**char(8)**	"D" if the table, as created, allows duplicate rows or "U" if it does not. The table storage structure (unique vs. non-unique keys, and so on) can override this setting. This column is blank if this information is unknown.
unique_rule	**char(8)**	The value may be either "U," "D," or a blank. If the value is "U" and the object is an ODT-DATA object, then it indicates that the object has a unique storage structure key(s). Refer to the key_sequence column of the **iicolumns** catalog for the key(s). If the value is "U" and the object is not an ODT-DATA object, then it indicates that the object has a unique key, described in either **iicolumns** or **iialt_columns**. If the value is "D," it indicates that duplicate physical storage structure keys are allowed. (A unique alternate key may exist in **iialt_columns** and any storage structure keys may be listed in **iicolumns**.) If this value is blank, uniqueness is unknown or does not apply.
number_pages	**integer**	The estimated number of physical pages in the table. Set to -1 if unknown.
overflow_pages	**integer**	The estimated number of overflow pages in the table. Set to -1 if unknown.

The following columns are used by the ODT-DATA DBMS. If a gateway does not supply this information, they should set these column to the default values: -1 for numeric columns and a blank for character columns.

Column Name	Data Type	Description
expire_date	**integer**	Expiration date of table. This is an ODT-DATA _bintime date.
table_integrities	**char(1)**	"Yes" if this object has ODT-DATA style integrities. If the value is blank, a probe of the **iiintegrities** table determines if integrities exist or not.
table_permits	**char(1)**	"Yes" if this object has ODT-DATA style permissions. A value of blank is not determinant on entries in the **iipermits** table.
all_to_all	**char(1)**	"Yes" if this object has an ODT-DATA permit all to all, or "No" if not.
ret_to_all	**char(1)**	"Yes" if this object has an ODT-DATA permit retrieve to all, or "No" if not.
row_width	**integer**	The size, in bytes, of the uncompressed binary value for a row of this query object.
is_journaled	**char(1)**	"Yes" if ODT-DATA journaling is enabled on this object, or "No" if it is not. Set to "C" if journaling is enabled at the next checkpoint. This is blank if ODT-DATA journaling does not apply.
view_base	**char(1)**	"Yes" if this is a base for a view definition, "No" if it is not, or blank if this is unknown.
modify_date	**char(25)**	The date on which the last physical modification to the storage structure of the table occurred. This is an ODT-DATA standard date. This is blank if unknown or inapplicable.

Column Name	Data Type	Description
table_ifillpct	**smallint**	This is the fill factor for the index pages used on the last **modify** command in the nonleaffill clause, expressed as a percentage from 0 to 100. This is used for ODT-DATA **btree** structures to rerun the last **modify** command.
table_dfillpct	**smallint**	This is the fill factor for the data pages used on the last **modify** command in the fillfactor clause, expressed as a percentage from 0 to 100. This is used for ODT-DATA **btree**, **hash**, and **isam** structures to rerun the last **modify** command.
table_lfillpct	**smallint**	This is the fill factor for the leaf pages used on the last **modify** command in the leaffill clause, expressed as a percentage from 0 to 100. This is used for ODT-DATA **btree** structures to rerun the last **modify** command.
table_minpages	**integer**	This is the minpages parameter from the last execution of the **modify** command. This is used for ODT-DATA hash structures only.
table_maxpages	**integer**	This is the maxpages parameter from the last execution of the **modify** command. This is used for ODT-DATA hash structures only.
location_name	**char(24)**	The first location of the table. If there are additional locations for a table, they are shown in the **iimulti_locations** table and multi_locations are set to "Yes."
table_reltid	**integer**	The first part of the internal relation id. This is used to derive the filename for the table.
table_reltidx	**integer**	The second part of the internal relation id. This is used to derive the filename for the table.

Column Name	Data Type	Description
multi_locations	**char(1)**	Indicates if the database is located in more than one area. "Yes" if it is in multiple locations, "No" if not.
table_relstamp	**integer**	High part of last create or modify timestamp for the table.
table_relstamp2	**integer**	Low part of last create or modify timestamp for the table.

The iicolumns Catalog

For each object in the **iitables** catalog, there are one or more entries in the **iicolumns** catalog. Each row in **iicolumns** contains the logical information on a column of the query object. This view is used by the frontends and user programs to perform dictionary operations and dynamic queries.

Column Name	Data Type	Description
table_name	**char(32)**	The name of the table. This is an ODT-DATA name.
table_owner	**char(32)**	The owner of the table. This is an ODT-DATA user-name.
column_name	**char(32)**	The column's name. This is an ODT-DATA name.
column_datatype	**char(32)**	The column's data type name. This is one of the type names: **integer, smallint, int, float, real, double precision, char, character, varchar, c, text, date,** and **money.**

Column Name	Data Type	Description
column_length	**integer**	The length of the column as specified by the user. If a data type contains two length specifiers, this column uses the first length. For the data types which are specified without length (**money** and **date**), this is set to zero. **NOTE:** This length is not the actual length of the column's internal storage.
column_scale	**integer**	This is the second number in a two-part user-length specification, that is, for typename(len1, len2) it is len2.
column_nulls	**char(8)**	Tells whether the column can contain null values. It is "No" if the column cannot contain null values. It is "Yes" if the column can contain null values.
column_defaults	**char(8)**	Tells whether the column is given a default value. It is "No" if the column is **not** given a default value on insert. It is "Yes" if the column is given a default value on insert.
column_sequence	**integer**	The number of this column in the corresponding table's **create** statement, numbered from 1.
key_sequence	**integer**	The order of this column in the primary key, numbered from 1. For an ODT-DATA table, this indicates the column's order in the primary storage structure key. If 0, then this column is not part of the primary key. This is unique if the unique_rule column for the table's corresponding entry in **iitables** is set to "U."
sort_direction	**char(8)**	Set to "A" for ascending or "D" for descending when key_sequence is greater than 0. Otherwise, this value is a blank.

Column Name	Data Type	Description
column_ingdatatype	**smallint**	This value indicates the ODT-DATA data type of the column. If the value is positive then the column is not nullable; if the value is negative, then the column is nullable. The data types and their corresponding values are: integer-30/30 float-31/31 c-32/32 text-37/37 date-3/3 money-5/5 char-20/20 varchar-21/21

The iiviews Catalog

The **iiviews** catalog contains one or more entries for each view in the database. (Views are represented in **iitables** by table type = "V.") Because the text_segment column is limited to 240 characters per row, a single view may require more than one entry to represent all its text. There are as many entries in this table as needed to represent all the text of a view.

The text may be broken in mid-word across the sequenced rows. The text column is pure text. Also, the text may or may not contain newline characters.

Column Name	Data Type	Description
table_name	**char(24)**	The view name. This is an ODT-DATA name.
table_owner	**char(24)**	The view's owner. This is an ODT-DATA username.
view_dml	**char(8)**	The language the view was created in. "S" (for SQL).
check_option	**char(8)**	Set to "Yes" if the check option was specified in the **create view** statement and "No" if not. This is blank if unknown.
text_sequence	**integer**	The sequence number for the text field, numbered from 1.
text_segment	**varchar(240)**	The text of the view definition.

The iiindexes Catalog

Each table with a **table_type** of "I" in the **iitables** table has an entry in **iiindexes**. In ODT-DATA, all indexes also have an entry in **iiphysical_tables**.

Column Name	Data Type	Description
index_name	**char(24)**	The index name. This is an ODT-DATA name.
index_owner	**char(24)**	The index owner. This is an ODT-DATA username.
create_date	**char(24)**	Creation date of index. This is an ODT-DATA standard date.
base_name	**char(24)**	The base table name. This is an ODT-DATA name.
base_owner	**char(24)**	The base table owner. This is an ODT-DATA name.
storage_structure	**char(16)**	The storage structure for the index. It is one of the following: "HEAP" if the table is a heap "HASH" if the table is a hash structure "ISAM" if the table is an isam structure "BTREE" if the table is a btree Blank if the table structure is unknown.
is_compressed	**char(8)**	Set to "Yes" if the table is stored in compressed format, or "No" if the table is uncompressed. This is blank if this is unknown.
unique_rule	**char(8)**	"U" if the index is unique, or "D" if duplicate key values are allowed or blank if unknown.

The iiindex_columns Catalog

For indexes, any ODT-DATA columns that are defined as part of the index has an entry in **iiindex_columns**.

Column Name	Data Type	Description
index_name	**char(24)**	The index containing *column_name*. This is an ODT-DATA name.
index_owner	**char(24)**	The index owner. This is an ODT-DATA username.
column_name	**char(24)**	The name of the column. This is an ODT-DATA name.
key_sequence	**integer**	Sequence of column within the key, numbered from 1.
sort_direction	**char(8)**	Set to "A" for ascending or "D" for descending.

The iialt_columns Catalog

For each alternate key, any columns that are defined as part of the key have an entry in **iialt_columns**.

Column Name	Data Type	Description
table_name	**char(24)**	The table that column_name belongs to.
table_owner	**char(24)**	The table owner.
key_id	**integer**	The number of the alternate key for this table.
column_name	**char(24)**	The name of the column.
key_sequence	**smallint**	Sequence of column within the key, numbered from 1.

The iistats Catalog

If a column has statistics, then it has a row in this table.

Column Name	Data Type	Description
table_name	**char(24)**	The name of the table. This is an ODT-DATA name.
table_owner	**char(24)**	The owner of the table. This is an ODT-DATA username.
column_name	**char(24)**	The column name to which the statistics apply. This is an ODT-DATA name.
create_date	**char(24)**	Date statistics were gathered, as an ODT-DATA standard date.
num_unique	**float8**	The number of unique values in the column.
rept_factor	**float8**	The repetition factor, or the inverse of the number of unique values (number of rows/number of unique values).
has_unique	**char(8)**	This is "Yes" if the column has unique values. "No" otherwise.
pct_nulls	**float8**	The percentage (fraction of 1.0) of the table that contains NULL for the column.
num_cells	**integer**	The number of cells in the histogram.

The iihistograms Catalog

The **iihistograms** table contains histogram information used by the optimizer.

Column Name	Data Type	Description
table_name	**char(24)**	The table for the histogram. This is an ODT-DATA name.
table_owner	**char(24)**	The table owner. This is an ODT-DATA username.
column_name	**char(24)**	The name of the column. This is an ODT-DATA name.
text_sequence	**integer**	The sequence number for the histogram, numbered from 1. There may be several rows in this table, used to order the "optdata" data when histogram is read into contiguous memory.
text_segment	**char(228)**	The histogram data, created by **optimizedb**. This is encoded.

The iipermits Catalog

The **iipermits** catalog contains one or more entries for each permit defined. Because the text of the permit definition may contain more than 240 characters, **iipermits** may contain more than one entry for a single permit. The text may or may not contain newlines and may be broken mid-word across rows.

This table is keyed on **object_name** and **object_owner**.

Column Name	Data Type	Description
object_name	**char(24)**	The table or procedure name. This is an ODT-DATA name.
object_owner	**char(24)**	The owner of the table or procedure. This is an ODT-DATA username.

Column Name	Data Type	Description
create_date	**char(24)**	The permit's creation date. This is an ODT-DATA standard date.
permit_user	**char(24)**	The username to which this permit applies.
permit_number	**smallint**	The number of this permit.
text_sequence	**smallint**	The sequence number for the text, numbered from 1.
text-segment	**varchar(240)**	The text of the permission definition.

The iiintegrities Catalog

The **iiintegrities** catalog contains one or more entries for each integrity defined on a table. Because the text of the integrity definition may contain more than 240 characters, **iiintegrities** may contain more than one entry for a single integrity. The text may or may not contain newlines and may be broken mid-word across rows.

This table is keyed on **table_name** and **table_owner**.

Column Name	Data Type	Description
table_name	**char(24)**	The table name. This is an ODT-DATA name.
table_owner	**char(24)**	The table's owner. This is an ODT-DATA user-name.
create_date	**char(24)**	The integrity's creation date. This is an ODT-DATA standard date.
integrity_number	**smallint**	The number of this integrity.
text_sequence	**smallint**	The sequence number for the text, numbered from 1.
text_segment	**varchar(240)**	The text of the integrity definition.

The iimulti_locations Catalog

Because a table, due to size or space constraints, may be located on multiple volumes, this table contains an entry for each additional location on which a table resides. The first location for a table can be found in the **iitables** catalog.

This table is keyed on **table_name** and **table_owner.**

Column Name	Data Type	Description
table_name	**char(24)**	The table name. This is an ODT-DATA name.
table_owner	**char(24)**	The table's owner. This is an ODT-DATA username.
sequence	**integer**	The sequence of this location in the list of locations, as specified in the **modify** command. This is numbered from 1.
location_name	**char(24)**	The name of the location.

The iiprocedures Catalog

The **iiprocedures** catalog contains one or more entries for each database procedure defined on a database. Because the text of the procedure definition may contain more than 240 characters, **iiprocedures** may contain more than one entry for a single procedure. The text may or may not contain newlines and may be broken mid-word across rows.

This table is keyed on **procedure_name** and **procedure_owner.**

Column Name	Data Type	Description
procedure_name	**char(24)**	The database procedure name, as specified in the **create procedure** statement.
procedure_owner	**char(24)**	The procedure's owner. This is an ODT-DATA username.
create_date	**char(24)**	The procedure's creation date. This is an ODT-DATA standard date.
text_sequence	**smallint**	The sequence number for the text_segment.
text_segment	**varchar(240)**	The text of the procedure definition.

Extended System Catalogs

Extended system catalogs used by the ODT-DATA frontend products, such as VIFRED and RBF, to store information on frontend objects such as applications, forms, and reports. These catalogs are also known as the ODT-DATA frontend catalogs.

Object id

Every ODT-DATA frontend object (form, report, QBF JoinDef, and so on.) is identified in the extended system catalogs by a unique number, the object id. The object id is generated by ODT-DATA when the frontend object is created. For each database, ODT-DATA stores the largest object id issued to date in the table **ii_id**; this value is incremented and issued as the id for each new frontend object.

An object's name, owner, and other information is stored once only, in the **ii_objects** catalog. In all other extended system catalogs, objects are identified by their object id.

User programs that insert objects into the extended system catalogs should be careful to first generate a unique object id for each new object. New object ids are generated by incrementing the id column in the **ii_id** catalog while inside a transaction. Be sure to keep the transaction that updates **ii_id.***id* as short as possible and to recover properly from errors; see the *ODT-DATA Embedded SQL User's Guide* for information on writing a transaction that recovers from errors.

Copying the Extended System Catalogs

Extended system catalogs should only be copied into new databases, never into existing databases that contain frontend objects (forms, reports, and so on).

Copying extended system catalogs with the ODT-DATA copy statement does not create new object ids for the copied objects. If the target database already contains frontend objects, then copying extended system catalogs into that database with the **copy** statement can result in catalogs that contain different objects with the same object id (for example, both a form and a report with the same object id). That will cause serious problems in the target database's extended system catalogs. Use the appropriate copy utility (**copyform, copyrep**, and so on) to copy objects to existing databases. The copy utilities generate a new object id for each object copied into the target database.

Querying the Extended System Catalogs

These are examples of queries you can issue to get information from the extended system catalogs. Note that each query specifies the class code for the type of object being selected. A table of class codes accompanies the description of the **ii_objects** extended system catalog.

Find information on every report in the database.

```
select report=o.name, o.owner, o.short_remark, r.reptype
from ii_objects o, ii_reports r
where (o.class = 1501 or o.class = 1502 or o.class = 1511)
      /* classes 1501, 1502, 1511 = reports */
and   o.id = r.id
```

Find the name and tabbing sequence number of every simple field and table field on form "empform" (empform is owned by user "susan").

```
select  form=o.name, f.fldname, f.flseq, f.fltype
from  ii_objects o, ii_fields f
where o.class = 1601            /* class 1601 = "form" */
and   o.name  = 'empform'
and   o.owner = 'susan'
and   o.id    = f.id
and   (f.fltype = 0 or f.fltype = 1) /* simple field or  table
field */
order by flseq
```

Select object information and long remarks, when available, by performing an outer join of **ii_objects** with **ii_longremarks.**

```
select o.name, o.class, o.owner, o.short_remark,
      l.long_remark
from ii_objects o, ii_longremarks l
where o.id = l.id

union all

select o.name, o.class, o.owner, o.short_remark, "
from ii_objects o
where not exists
( select *
from ii_longremarks
where ii_longremarks.id = ii_objects.id )
order by name
```

Catalogs Shared by All Frontend (FE) Objects

This section contains a description of the extended system catalogs that are not unique to any particular frontend (FE), but they are used by them all.

The ii_id Catalog

This catalog consists of one column with a single row. The value in this catalog is the highest object id currently allocated within this database. For a newly created database, this value is initialized to 10000 and can grow as large as the largest positive **integer(4)** value.

The ii_id catalog is a heap table with 1 row.

Column Name	Data Type	Description
object_id	**integer(4)**	The highest current object id in this database.

The ii_objects Catalog

This catalog contains a row for every FE object in the database. Its columns provide basic information about each object, such as name, owner, object id, object class, and creation date. Objects in this table often have additional information represented in rows of one or more other FE catalogs. For example, form objects are also represented by rows in **ii_forms**, **ii_fields**, and **ii_trim**. In all cases, the object id is used as the key column to join information from multiple catalogs about a single object.

The **ii_objects** catalog is structured as btree on the object_class, object_owner, and object_name columns. It also has a secondary index, which is btree unique on the object_id column.

Column Name	Data Type	Description
object_id	**integer(4)**	The object identifier, unique among FE objects in the database.
object_class	**integer(4)**	The object's class. Tells what type of object this is (form, report, and so on). See the following for a table of object classes.

Column Name	Data Type	Description
object_name	**varchar(32)**	The name of the object.
object_owner	**varchar(32)**	The object owner's username.
object_env	**integer(4)**	Currently unused.
is_current	**integer(1)**	Currently unused.
short_remark	**varchar(60)**	A short descriptive remark associated with the object.
object_language	**integer(2)**	Currently unused.
create_date	**char(24)**	The time and date at which the object was initially created.
alter_date	**char(24)**	The time and date at which the object, or associated information, was most recently altered or saved.
alter_count	**integer(4)**	A count of the number of times this object has been altered or saved.
last_altered_by	**varchar(32)**	The name of the user who last altered or saved this object.

The following table describes each object class. Object class is a column in the **ii_objects** catalog.

Object Class	Description
1002	JoinDef
1501	Generic Report
1502	Report-Writer Report

Object Class	Description
1511	RBF Report
1601	Form
2021	Host Language Procedure
2190	Undefined Procedure
2201	QBFName
2220	Report Frame
2230	QBF Frame
2249	GBF Frame
2250	Undefined Frame
3501	Dependency Type: member of
3502	Dependency Type: database reference
3503	Dependency Type: call with no use of return code
3504	Dependency Type: call with use of return code

The II_longremarks Catalog

This catalog contains the "long remarks" text associated with FE objects. Only those objects that have an associated long remark are entered in this catalog. Consequently, unless all objects being selected have a long remark entered, joins between **ii_objects** and **ii_longremarks** should be outer joins. See "Querying the Extended System Catalogs" earlier in this appendix for an example of an outer **join** between the **ii_objects** and **ii_longremarks** catalogs. The current implementation restricts long remarks to a single row; the sequence column is provided for a future enhancement to allow remarks of arbitrary length.

The **ii_longremarks** catalog is structured as btree unique on the object_id column.

Column Name	Data Type	Description
object_id	**integer(4)**	Object id of the frontend object this remark belongs to. Various other information about this object (such as name, owner, and object class) is kept in the **ii_objects** catalog.
remark_sequence	**integer(2)**	A sequence number for (future) representation of multiple segments of text comprising one object's long remarks.
long_remark	**varchar(600)**	The long remarks text associated with the object.
remark_language	**integer(2)**	Currently unused.

Forms System Catalogs

This section contains a description of the extended system catalogs that are unique to the forms system.

The ii_forms Catalog

This catalog contains one row for each form in a database.

The **ii_forms** catalog is structured as btree unique on the object_id column.

Column Name	Data Type	Description
object_id	**integer(4)**	Unique identifier (object id) for identifying this form in the **ii_objects** catalog. Other information about the form (such as name, owner, and object class) is stored in the **ii_objects** catalog.
frmaxcol	**integer(2)**	The number of columns the form occupies.
frmaxlin	**integer(2)**	The number of lines the form occupies.

Column Name	Data Type	Description
frposx	**integer(2)**	The **x** coordinate for the upper left corner of the form.
frposy	**integer(2)**	The **y** coordinate for the upper left corner of the form.
frfldno	**integer(2)**	The number of updateable regular or table fields in the form.
frnsno	**integer(2)**	The number of display-only regular fields in the form.
frtrimno	**integer(2)**	The number of trim and box graphic trim strings in the form.
frversion	**integer(2)**	Version number of the form.
frscrtype	**integer(2)**	Reserved for future use.
frscrmaxx	**integer(2)**	Reserved for future use.
frscrmaxy	**integer(2)**	Reserved for future use.
frscrdpix	**integer(2)**	Reserved for future use.
frscrdpiy	**integer(2)**	Reserved for future use.
frflags	**integer(4)**	The attributes of the form, such as whether this a pop-up or normal form.
fr2flags	**integer(4)**	More attributes for the form. Currently unused.
frtotflds	**integer(4)**	The total number of records in the **ii_fields** catalog for the form.

The ii_fields Catalog

This catalog contains information on the fields in a form. For every form, there is one row in this catalog for each field, table field, and table field column. As used below, the word *field* refers to a simple field, a table field, or a column in a table field. See "Querying the Extended System Catalogs" earlier in this appendix for an example of a query that selects information about fields on a form.

The **ii_fields** catalog is structured as btree unique on the object_id and flsubseq columns.

Column Name	Data Type	Description
object_id	**integer(4)**	Unique identifier (object id) for identifying the form this field belongs to in the **ii_objects** catalog. Other information about the form (such as name, owner, and object class) is stored in the **ii_objects** catalog.
flseq	**integer(2)**	The sequence number of the field in the form (or column in a table field). This determines the tabbing order among fields and among columns in a table field. If the number is less than zero (0), it is a display-only field or column.
fldname	**varchar(32)**	The name of the field.
fldatatype	**integer(2)**	The field's data type. Possible values are listed below with nullable data types being the negative of the listed value: 3 - date 5 - money 20 - char 21 - varchar 30 - integer 31 - floating-point 32 - c 37 - text

Column Name	Data Type	Description
fllength	**integer(2)**	The internal ODT-DATA data length of the field in bytes. Note that this may not be the same as the user-defined length. This is the length used by ODT-DATA.
flprec	**integer(2)**	Reserved for future use.
flwidth	**integer(2)**	The number of characters displayed in the field on the form including wrap characters. For example, if the format for the field is *c20.10*, flwidth is 20.
flmaxlin	**integer(2)**	The number of lines occupied by the field (title and data).
flmaxcol	**integer(2)**	The number of columns occupied by the field (title and data).
flposy	**integer(2)**	The **y** coordinate of the upper left corner of the field.
flposx	**integer(2)**	The **x** coordinate of the upper left corner of the field.
fldatawidth	**integer(2)**	The width of the data entry area for the field. If field format is *c20.10*, fldatawidth is 10.
fldatalin	**integer(2)**	The **y** coordinate position of the data entry area relative to the upper left corner of the field.
fldatacol	**integer(2)**	The **x** coordinate position of the data entry area relative to the upper left corner of the field.
fltitle	**varchar(50)**	The field title.
fltitcol	**integer(2)**	The **x** coordinate position of the title relative to the upper left corner of the field.

Column Name	Data Type	Description
fltitlin	**integer(2)**	The y coordinate position of the title relative to the upper left corner of the field.
flintrp	**integer(2)**	Reserved for future use.
fldflags	**integer(4)**	The field attributes, such as boxing the field or displaying the data entry in reverse video.
fld2flags	**integer(4)**	More attributes for the field, such as is the field scrollable.
fldfont	**integer(2)**	Reserved for future use.
fldptsz	**integer(2)**	Reserved for future use.
fldefault	**varchar(50)**	The default value for the field.
flformat	**varchar(50)**	The display format for the field (for example, *c10* or *f10.2*).
flvalmsg	**varchar(100)**	The message to be displayed if the validation check fails.
flvalchk	**varchar(240)**	The validation check for the field.
fltype	**integer(2)**	Indicates if the record describes a regular field, a table field, or a column in a table field. Possible values are: 0 - simple field 1 - table field 2 - table field column
flsubseq	**integer(2)**	A unique identifying record number with respect to the set of records that describe all the fields in a form.

The ii_trim Catalog

This catalog contains the trim strings and box graphic trim for a form. There is one row for each trim string and for each box graphic trim. This table allows duplicates to support box/trim graphics that start at the same location.

The **ii_trim** catalog is structured as compressed btree unique on the object_id, trim_col and trim_lin columns.

Column Name	Data Type	Description
object_id	**integer(4)**	Unique identifier (object id) for identifying the form, this trim string belongs in the **ii_objects** catalog. Other information about the form (such as name, owner, and object class) is stored in the **ii_objects** catalog.
trim_col	**integer(2)**	The **x** coordinate for the starting position of the trim string or box graphic trim.
trim_lin	**integer(2)**	The **y** coordinate for the starting position of the trim string or box graphic trim.
trim_trim	**varchar(150)**	The actual trim string or encoding of box graphic trim size (number of rows and columns).
trim_flags	**integer(4)**	Attributes of the trim string.
trim2_flags	**integer(4)**	More attributes for the trim string. Currently unused.
trim_font	**integer(2)**	Reserved for future use.
trim_ptsz	**integer(2)**	Reserved for future use.

The ii_encoded_forms Catalog

This catalog contains encoded versions of forms. The encoding allows forms to be retrieved from the database faster.

The **ii_encoded_forms** catalog is structured as compressed btree unique on the object_id and cfseq columns.

Column Name	Data Type	Description
object_id	**integer(4)**	Unique identifier (object id) for identifying this form in the **ii_objects** catalog. Other information about this form (such as name, owner, and object class) is stored in the **ii_objects** catalog.
cfseq	**integer(2)**	Sequence number of this record for a particular encoded form. Record sequence numbering start at zero (0).
cfdatsize	**integer(4)**	Number of bytes of actual data in column cfdata.
cftotdat	**integer(4)**	Total number of bytes needed to hold an encoded form.
cfdata	**varchar(1960)**	Data area used for holding an encoded form.

QBF System Catalogs

This section contains a description of the extended system catalogs that are unique to the QBF system.

The ii_qbfnames Catalog

This catalog contains information used by QBF on the mapping between a form and a corresponding table or JoinDef.

The **ii_qbfnames** catalog is structured as compressed btree unique on the object_id column.

Column Name	Data Type	Description
object_id	**integer(4)**	Unique identifier (object id) for identifying this QBFName in the **ii_objects** catalog. Other information about this QBFName (such as its name, owner, and object class) is stored in the **ii_objects** catalog.
relname	**varchar(32)**	The name of a table or JoinDef.
frname	**varchar(32)**	The name of a form corresponding to the table or JoinDef.
qbftype	**integer(2)**	Indicates if the QBFName is mapping a form to a table (value 0) or JoinDef (value 1).

The ii_joindefs Catalog

This catalog contains additional information about join definitions (JoinDefs) used in QBF. Basic information about the JoinDef is contained in a row in the **ii_objects** catalog. Each JoinDef can have several rows in **ii_joindefs** associated with it. Each row of **ii_joindefs** contains one of four types of records, each of which contains a different kind of information about the structure of the JoinDef. The interpretation of the respective columns differs depending on the value of "qtype."

The **ii_joindefs** catalog is structured as compressed btree unique on the object_id and qtype columns.

Column Name	Data Type	Description
object_id	**integer(4)**	Unique identifier (object id) for identifying this JoinDef in the **ii_objects** catalog. Other information about the JoinDef (such as its name, owner, and object class) is stored in the **ii_objects** catalog.

Column Name	Data Type	Description
qtype	**integer(4)**	The low order byte of this column indicates the record type of this row, as follows: 0 - Indicates if a table field is used in the JoinDef; 1 - Table information; 2 - Column information; 3 - Join information. (The high order byte is used as a sequence number for multiple entries of a particular record type.) Each JoinDef has exactly one row with qtype = 0; it has one row with qtype = 1 for each table used in the JoinDef; it has one row with qtype = 2 for each field displayed in the JoinDef; it has one row with qtype = 3 for each pair of columns joined in the JoinDef.
qinfo1	**varchar(32)**	If qtype = 0, then qinfo1 indicates if the JoinDef is built with a table field format (y = yes, n = no). If qtype = 1, then qinfo1 contains the name of a table used in the JoinDef. If qtype = 2, then qinfo1 contains a correlation name (range variable) for the table used in the JoinDef that contains the column named in qinfo2. If qtype = 3, then qinfo1 contains a correlation name (range variable) for a column named in qinfo2 that is joined to the column referenced in qinfo3 and qinfo4.
qinfo2	**varchar(32)**	If qtype = 0, then qinfo2 is not used. If qtype = 1, then qinfo2 indicates whether the table named in qinfo1 is a Master (0) or Detail (1) table. If qtype = 2, then qinfo2 contains the name of the column to be used in conjunction with the correlation name in qinfo1. If qtype = 3, then qinfo2 contains the name of the column to be joined to the column referenced in qinfo3 and qinfo4.

Column Name	Data Type	Description
qinfo3	**varchar(32)**	If qtype = 0, then qinfo3 is not used. If qtype = 1, then qinfo3 contains a correlation name (range variable) for the table named in qinfo1. If qtype = 2, then qinfo3 contains the field name in the form corresponding to the column identified by qinfo2. If qtype = 3, then qinfo3 contains a correlation name (range variable) for a column named in qinfo4 that is joined to the column referenced in qinfo1 and qinfo2.
qinfo4	**varchar(32)**	If qtype = 0, then qinfo4 is not used. If qtype = 1, then qinfo4 contains the delete rules for the table named in qinfo1 (0 = no, 1 = yes). If qtype = 2, then qinfo4 contains the status codes for the column identified by qinfo1 and qinfo2. These status codes are expressed as a 3-character text string; the first character denotes update rules for values in this column (0 = no, 1 = yes); the second character denotes whether this column is part of a join (0 = no, 1 = yes); the third character denotes whether this column is a displayed column (0 = no, 1 = yes). Typically, if the column is not part of a join the third character is not used by QBF. If qtype = 3, then qinfo4 contains the name of the column I-Catalogs (QBF system);described!E to be joined to the column referenced in qinfo1 and qinfo2.

Report-Writer System Catalogs

This section contains a description of the extended system catalogs that are unique to the Report-Writer system.

The ii_reports Catalog

This catalog contains information about reports. There is one row for every report in the database. Both reports created through RBF and reports created through **sreport** contain entries in **ii_reports**. See "Querying the Extended System Catalogs" earlier in this appendix for an example of a query that selects information about reports.

The **ii_reports** table is structured as btree unique on the object_id column.

Column Name	Data Type	Description
object_id	**integer(4)**	Unique identifier (object id) for identifying this report in the **ii_objects** catalog. Other information about the report (such as name, owner, and object class) is stored in the **ii_objects** catalog.
reptype	**char(1)**	The method used to create the report; "S" if the report was created by **sreport**, and "F" if the report was created by RBF.
repacount	**integer(2)**	The number of rows in the **ii_rcommands** catalog with an rcotype of "AC." This is used for internal consistency.
repscount	**integer(2)**	The number of rows in the **ii_rcommands** catalog with an rcotype of "SO." This is used for internal consistency.
repqcount	**integer(2)**	The number of rows in the **ii_rcommands** catalog with an rcotype of "QU." This is used for internal consistency.
repbcount	**integer(2)**	The number of rows in the **ii_rcommands** catalog with an rcotype of "BR." This is used for internal consistency.

The ii_rcommands Catalog

This catalog contains the formatting, sorting, break, and query commands for each report, broken down into individual commands.

The **ii_rcommands** catalog is structured as compressed btree unique on the object_id column.

Column Name	Data Type	Description
object_id	**integer(4)**	Unique identifier (object id) for identifying the report this command belongs to in the **ii_objects** catalog. Other information about the report (such as name, owner, and object class) is stored in the **ii_objects** catalog.
rcotype	**char(2)**	Report command type. Valid values are: "TA" - Table for a **.data** command "SQ" - Piece of SQL query for the **.query** command "SO" - Sort column for a **.sort** command "AC" - Report formatting or **action** command "OU" - .output filename, if specified "BR" - **.break** command information "DE" - **.declare** statement information
rcosequence	**integer(2)**	The sequence number for this row, within the rcotype.
rcosection	**varchar(12)**	The section of the report, such as header or footer, to which the commands refer if rcotype is "AC." If rcotype is "QU" or "SQ," this refers to the part of the query described. For other values of rcotype, this field is unused.
rcoattid	**varchar(32)**	If rcotype is "AC," this indicates either the column name associated with the footer/header section or contains the value "PAGE" or "REPORT" or "DETAIL." If rcotype is "SO," this is the name of the sort column. If rcotype is "QU," the range variable names are put in this column. If rcotype is "BR," the name of the break column is put in this column. If rcotype is "DE," the name of the declared variable is put in this column.

Column Name	Data Type	Description
rcocommand	**varchar(12)**	Primarily used for the names of the formatting commands when rcotype is "AC." This is also used for "SO" rcotype to indicate that the sort column is also a break column.
rcotext	**varchar(100)**	If the rcotype is "AC," then this contains the text of the formatting command. If rcotype is type "OU," then this contains the name of the output file. If rcotype is "QU" or "SQ," then this contains query text. If rcotype is "TA," then this contains the table name. If rcotype is "SO," then this contains the sort order. If rcotype is "DE," then this contains the text of the declaration. If rcotype is "BR," then this is unused.

The DBMS System Catalogs

The section provides a list of the DBMS catalogs with a short description of each. The table names of the DBMS System Catalogs may be used as arguments to the **sysmod** command (see Chapter 4, "ODT-DATA Operating System Commands").

iirelation Describes each table in the database.

iirel_idx An index table that indexes the iirelation table by table name and owner.

iiattribute Describes the properties of each column of a table.

iiindex Describes all the indexes for a table.

iidevices Describes additional locations when a user table spans more than one ODT-DATA location.

iiintegrity Contains information about the integrities applied to tables.

iiprotect	Contains information about the protections applied to tables.
iitree	Contains the DBMS internal representation of query text for views, protections, and integrities.
iiqrytext	Contains the actual query text for views, protections, and integrities.
iidbdepends	Describes the dependencies between views or protections and their base tables.
iixdbdepends	An Index table used to find the rows that reference a dependent object in the iidbdepends catalog.
iiprocedure	Contains information about database procedures.
iihistogram	Contains database histograms that are collected by the **optimizedb** program.
iistatistics	Contains database statistics that are collected by the **optimizedb** program.
iidatabase	Describes various attributes of each database in an installation.
iidbid_idx	A secondary index built on a column in the **iidatabase** catalog.
iidbaccess	Describes which users have access to private databases.
iiextend	Defines the extended data locations of a database.
iilocations	Maps locations to physical areas and indicates what that location can be used for.
iiuser	Defines valid users and their privileges in an ODT-DATA installation.

Index

Special Characrters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

15June1990

AZ10405P000

PO# 28305

PAT 050