IBM Tivoli Decision Support for z/OS
Version 1.8.2

Language Guide and Reference

SH19-6817-13

IBM Tivoli Decision Support for z/OS
Version 1.8.2

Language Guide and Reference

SH19-6817-13

Before using this information and the product it supports, read the information in “Notices” on page D-1.

Fourteenth Edition (March 2015)

This edition applies to version 1, release 8, modification level 2 of Tivoli Decision Support for z/OS (program number 5698-B06) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Part 2. Log collector language reference

Part 3. Report definition language guide

Part 4. Report definition language reference

Part 5. Appendixes

Figures

Tables

Preface

The *Language Guide and Reference* is a user's guide and reference book for the IBM® Tivoli® Decision Support for z/OS®log collector language and report definition language. It describes how to use these languages.

IBM Tivoli Decision Support for z/OS is hereafter also referred to as Tivoli Decision Support for z/OS.

The following terms are used interchangeably throughout this book:

- MVS™, OS/390®, and z/OS
- OPC and Tivoli Workload Scheduler

Note: To use the report definition language, you must install QMF™ on your system.

Who should read this book

The *Language Guide and ReferenceLanguage Guide and Reference* is for Tivoli Decision Support for z/OS administrators and performance analysts, or programmers who are responsible for maintaining system log data and reports. To understand this book, you should be familiar with Structured Query Language (SQL) and DB2®.

What this book contains

This book is a guide to understanding and customizing Tivoli Decision Support for z/OS to bring maximum benefit to your organization and to users. The book contains the following parts:

- Part I, "Guide to the log collector language" introduces the Tivoli Decision Support for z/OS log collector and describes how to use its log collector language to define and manage logs, records, tables, and updates.
- Part II, "Reference to the log collector language" presents the reference information for each element, function, and statement of the Tivoli Decision Support for z/OS log collector language. Each reference contains a syntax diagram, a description, and a simple example of usage. A section is included on how to read the syntax diagrams.
- Part III, "Guide to the report definition language" describes how to use the Tivoli Decision Support for z/OS report definition language to define reports and report groups to Tivoli Decision Support for z/OS.
- Part IV, "Reference to the report definition language" presents the reference information for the report definition language statements.
- Part V, "Appendixes" provides information about using the log and record procedures, using the JCL to submit batch jobs, and how to obtain support for IBM software products.

A glossary and index follows the appendixes.

Publications

This section lists publications in the Tivoli Decision Support for z/OS library and any other related documents. It also describes how to access Tivoli publications online, how to order Tivoli publications, and how to submit comments on Tivoli publications.

Tivoli Decision Support for z/OS library

The following documents are available in the Tivoli Decision Support for z/OS library:

- , SH19-6816, SH19-6816

  Provides information about initializing the Tivoli Decision Support for z/OS database and customizing and administering Tivoli Decision Support for z/OS.

- , SH19-4019.

  Provides information for administrators and users about collecting and reporting performance data generated by AS/400 systems.

- , SH19-6820.

  Provides information for administrators and users about collecting and reporting performance data generated by Customer Information and Control System (CICS®).

- , SH19-4018.

  Provides information for administrators and users about collecting and reporting performance data generated by operating systems and applications running on a workstation.

- , SH19-6842.

  Provides information for users who display existing reports, for users who create and modify reports, and for administrators who control reporting dialog default functions and capabilities.

- , SH19-6825.

  Provides information for administrators and users about collecting and reporting performance data generated by Information Management System (IMS™).

- , SH19-6817.

  Provides information for administrators, performance analysts, and programmers who are responsible for maintaining system log data and reports.

- , SH19-6902.

  Provides information to help operators and system programmers understand, interpret, and respond to Tivoli Decision Support for z/OS messages and codes.

- , SH19-6901.

  Provides information for network analysts or programmers who are responsible for setting up the network reporting environment.

- , SH19-6822.

  Provides reference information for network analysts or programmers who use the Network Performance feature.

- , SH19-6821.

  Provides information for network analysts or programmers who use the Network Performance feature reports.

- , SH19-4495.

Provides information for users who want to use Tivoli Decision Support for z/OS to collect and report performance data generated by Resource Accounting for z/OS.

- , SH19-6818.

  Provides information for performance analysts and system programmers who are responsible for meeting the service-level objectives established in your organization.
- , SH19-6819.

  Provides information for administrators and users with a variety of backgrounds who want to use Tivoli Decision Support for z/OS to analyze z/OS, z/VM®, zLinux, and their subsystems, performance data.
- , SH19-4494.

  Provides information for administrators and users with a variety of backgrounds who want to use Tivoli Decision Support for z/OS to analyze z/OS, z/VM, zLinux, and their subsystems, performance data.
- , SC23-7966.

  Provides information about the functions and features of the Usage and Accounting Collector.
- *IBM Online Library z/OS Software Products Collection Kit*, SK3T-4270

  CD containing all z/OS documentation.

Accessing terminology online

The IBM Terminology Web site consolidates the terminology from IBM product libraries in one convenient location. You can access the Terminology Web site at the following Web address:

http://www.ibm.com/ibm/terminology

Accessing publications online

IBM posts publications for this and all other Tivoli products, as they become available and whenever they are updated, to the Tivoli software information center Web site. Access the Tivoli software information center by first going to the Tivoli software library at the following Web address:

http://www.ibm.com/software/tivoli/library/

Scroll down and click the **Product manuals** link. In the Tivoli Technical Product Documents Alphabetical Listing window, click the Tivoli Decision Support for z/OS link to access the product library at the Tivoli software information center.

Note: If you print PDF documents on other than letter-sized paper, set the option in the **File ″ Print** window that allows Adobe Reader to print letter-sized pages on your local paper.

Accessibility

Accessibility features help users with a physical disability, such as restricted mobility or limited vision, to use software products successfully. With this product, you can use assistive technologies to hear and navigate the interface. You can also use the keyboard instead of the mouse to operate all features of the graphical user interface.

For additional information, see the Accessibility Appendix in the *Administration Guide and Reference*.

Tivoli technical training

For Tivoli technical training information, refer to the following IBM Tivoli Education Web site:

http://www.ibm.com/software/tivoli/education/

Support information

If you have a problem with your IBM software, you want to resolve it quickly. IBM provides the following ways for you to obtain the support you need:

- Searching knowledge bases: You can search across a large collection of known problems and workarounds, Technotes, and other information.
- Obtaining fixes: You can locate the latest fixes that are already available for your product.
- Contacting IBM Software Support: If you still cannot solve your problem, and you need to work with someone from IBM, you can use a variety of ways to contact IBM Software Support.

For more information about these three ways of resolving problems, see Appendix C, “Support information,” on page C-1.

Conventions used in this book

This guide uses several conventions for special terms and actions, operating system-dependent commands and paths, and margin graphics.

The following terms are used interchangeably throughout this book:
- MVS, OS/390, and z/OS.
- VM and z/VM.

Except for editorial changes, updates to this edition are marked with a vertical bar to the left of the change.

Typeface conventions

This guide uses the following typeface conventions:

Bold

- Lowercase commands and mixed case commands that are otherwise difficult to distinguish from surrounding text
- Interface controls (check boxes, push buttons, radio buttons, spin buttons, fields, folders, icons, list boxes, items inside list boxes, multicolumn lists, containers, menu choices, menu names, tabs, property sheets), labels (such as **Tip**, and **Operating system considerations**)
- Column headings in a table
- Keywords and parameters in text

Italic

- Citations (titles of books, diskettes, and CDs)
- Words defined in text
- Emphasis of words (words as words)

- Letters as letters
- New terms in text (except in a definition list)
- Variables and values you must provide

`Monospace`

- Examples and code examples
- File names, programming keywords, and other elements that are difficult to distinguish from surrounding text
- Message text and prompts addressed to the user
- Text that the user must type
- Values for arguments or command options

Programming Interfaces Information

This book is intended to help users use the languages provided by Tivoli Decision Support for z/OS.

This book also documents Product-sensitive Programming Interfaces and Associated Guidance Information provided by Tivoli Decision Support for z/OS.

Product-sensitive programming interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of Tivoli Decision Support for z/OS. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive programming interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interfaces and Associated Guidance Information is identified where it occurs, by an introductory statement to a chapter or section.

Changes in this edition

This edition is an update of the previous edition of the same book. The changes relate to 1.8.2 GA APAR documentation.

Chapter 9. Values and expressions

Lookup expressions syntax diagram amended to include the optional `ORDER BY` parameter.

- "Lookup expressions" on page 9-14

Chapter 11. Log collector language statements

ON TIMESTAMP OVERLAP added to COLLECT statement syntax diagram:

- "Syntax" on page 11-10

New GENERATE statements:

- "GENERATE INDEX" on page 11-39.
- "GENERATE PARTITIONING" on page 11-39.
- "GENERATE TABLESPACE" on page 11-40.

Except for editorial changes, updates to this edition are marked with a vertical bar [|] to the left of the change.

Changes in this edition

Part 1. Log collector language guide

Chapter 1. Introduction to the log collector

Tivoli Decision Support for z/OS is a reporting system that collects performance data logged by computer systems, summarizes the data, and presents it in a variety of forms for use in systems management. Tivoli Decision Support for z/OS consists of a base product and several optional features.

The central part of Tivoli Decision Support for z/OS is a program called the *log collector* that reads performance data, organizes that data, and stores it in a DB2 database. You control the log collector with instructions written in the *log collector language*. Each instruction is a *statement* in the language.

This topic provides an overview of the log collector and its language.

Collecting log data

About this task

Performance data about your system is obtained from sequential data sets such as those written by system management facilities (SMF) under MVS or by Information Management System (IMS). These data sets are called *log data sets* or *logs*.

The main function of the log collector is to read data from the logs and store it in DB2 tables, called *data tables*. This process is called collecting log data. The log collector can perform extensive processing on the data before storing it, such as:

- Grouping data by hour, day, or month.
- Computing sums, maximum or minimum values, averages, or percentiles.
- Calculating resource availability.

The purpose of this processing is to transform large amounts of data into useful information. The volume of data stored in the database is usually much smaller than the volume of data read from the log.

To collect log data, you use the **DEFINE LOG** and **DEFINE RECORD** statements to describe the log to be processed and, in particular, the layout of records in the log. In addition, you use the **DEFINE UPDATE** statement to specify the processing to be performed on the data from the log and how to update data tables with the result.

When the log collector executes the **DEFINE LOG**, **DEFINE RECORD**, and **DEFINE UPDATE** statements, it stores the information contained in these statements. The log, records, and update process then become defined to the log collector. The log collector stores the definitions of the log, records, and update process, not the statements themselves.

Having defined the log, records, and update process, you can collect data from the log using the **COLLECT** statement. When the log collector executes this statement, it retrieves the stored definitions and performs the data collection specified by those definitions.

You can use the stored definitions any number of times to collect data from any number of log data sets, as long as the definitions properly describe the data sets and the required processing. Typically, you define the log, records, and update process only once, when installing Tivoli Decision Support for z/OS. You then use the stored definitions to periodically collect performance data generated by your installation.

Listing log data

About this task

You might need to examine the contents of a log data set without updating the data tables. If you have defined the log and its records to the log collector, you can use the **LIST RECORD** statement. When the log collector executes this statement, it uses the stored log and record definitions to interpret the log data, then formats the data as specified by the **LIST RECORD** statement. You can specify the result to be a printable file or a file in the integration exchange format (IXF).

To count the number of records for each record type contained in the log, use the **LOGSTAT** statement.

Maintaining data tables

About this task

Not all performance data is kept indefinitely; it is discarded when no longer useful. You might, for example, discard daily statistics when they are one month old and monthly statistics when they are one year old.

You can use the log collector to discard old data. The basic principle is the same as for collecting data. You store the definition of the job to be done, and then repeatedly use the stored definition to perform that job.

Using the **DEFINE PURGE** statement, you specify a *purge condition* for a data table. The condition identifies the data to be discarded, depending upon the current date and time. When the log collector executes this statement, it stores the purge condition, which becomes *defined* to the log collector. Having defined the purge condition for one or more tables, you can discard old data using the **PURGE** statement. When the log collector executes this statement, it retrieves the stored purge conditions and purges the tables based on those conditions.

Occasionally, data entered into a data table is incorrect or the data table is damaged. To repair the data, you can use SQL statements executed from Query Management Facility (QMF) or the log collector. You can also use the RECALCULATE statement to alter data stored by the log collector.

Maintaining definitions

About this task

The main principle of using the log collector is that you must define the log, records, update process, and purge conditions only once. Having defined them, you can use the stored definitions repeatedly for production runs.

Sometimes, however, you might need to change the stored definitions. For example, you might install a new version of a product that generates slightly different records in its log, or you might decide to collect more information.

The log collector language includes several **ALTER** statements for modifying stored definitions. You can also delete an entire stored definition using the **DROP** statement, and then store a modified definition using one of the **DEFINE** statements.

You can document the stored definitions by adding comments to them using the **COMMENT ON** statement. When you display the definitions online using the administration dialog, you see the comments you stored.

Ready-made definitions

Tivoli Decision Support for z/OS provides definitions for most of the standard IBM logs. These definitions are provided in the form of **DEFINE LOG** and **DEFINE RECORD** statements. You can use these definitions as they are, or modify them for your needs. You can also use them as a pattern for creating your own definitions.

The Tivoli Decision Support for z/OS features also provide many table and update definitions that you can use.

Summary of log collector statements

The log collector language consists of these statements:

- Definition statements
  - DEFINE LOG
  - DEFINE PURGE
  - DEFINE RECORD
  - DEFINE RECORDPROC
  - DEFINE UPDATE
  - ALTER LOG
  - ALTER RECORD
  - ALTER RECORDPROC
  - ALTER UPDATE
  - SET
  - DROP
  - COMMENT ON
- Log processing statements
  - COLLECT
  - LIST RECORD
  - LOGSTAT
- Table maintenance statements
  - PURGE
  - RECALCULATE
- Other statements
  - SQL

Summary of log collector statements

Chapter 2. How to use the log collector language

This topic uses a simple example to describe how to use the log collector language.

In this scenario you want to determine how many read and write errors are produced per hour by three applications; APPL1, APPL2, and APPL3. These applications write information about read and write errors to a log data set called RWSTAT.EXAMPLE. The applications update this data set hourly.

You want to collect data from RWSTAT.EXAMPLE, process it, and store the result in a DB2 data table.

Table 2-1 shows the structure of the records in RWSTAT.EXAMPLE.

Table 2-1. Structure of records containing data about read and write errors

Field Name	**Offset**	**Length**	**Data format**	**Description**
A_NAME	0	10	Character string	Contains the name of the application writing to this data set (APPL1, APPL2, or APPL3)
DATE	10	4	Packed decimal in the idd:break>format 0*cyydddF*	Contains the date, where: **c** Century **yy** Year within the century **ddd** Day within the year
TIME	14	6	Character string in the idd:break>format *hhmmss*	Contains the time, where: **hh** Hour **mm** Minute **ss** Second
R_ERR	20	4	Binary	The number of read errors
W_ERR	24	4	Binary	The number of write errors

Table 2-2 shows the data (in hexadecimal) contained in RWSTAT.EXAMPLE.

Table 2-2. Contents of RWSTAT.EXAMPLE (in hexadecimal)

A_NAME	**DATE**	**TIME**	**R_ERR**	**W_ERR**
X'C1D7D7D3F1'	X'0093001F'	X'F0F1F0F0F0F1'	X'00000003'	X'00000005'
X'C1D7D7D3F2'	X'0093001F'	X'F0F1F0F0F0F2'	X'00000001'	X'00000003'
X'C1D7D7D3F3'	X'0093001F'	X'F0F1F0F0F0F3'	X'00000002'	X'00000000'
X'C1D7D7D3F1'	X'0093001F'	X'F0F2F0F0F0F1'	X'00000000'	X'00000000'
X'C1D7D7D3F2'	X'0093001F'	X'F0F2F0F0F0F2'	X'00000002'	X'00000001'
X'C1D7D7D3F3'	X'0093001F'	X'F0F2F0F0F0F3'	X'00000005'	X'00000003'
X'C1D7D7D3F1'	X'0093001F'	X'F0F3F0F0F0F1'	X'00000004'	X'00000006'
X'C1D7D7D3F2'	X'0093001F'	X'F0F3F0F0F0F2'	X'00000001'	X'00000003'
X'C1D7D7D3F3'	X'0093001F'	X'F0F3F0F0F0F3'	X'00000002'	X'00000002'
X'C1D7D7D3F1'	X'0093001F'	X'F0F4F0F0F0F1'	X'00000002'	X'00000006'
X'C1D7D7D3F2'	X'0093001F'	X'F0F4F0F0F0F2'	X'00000000'	X'00000000'
X'C1D7D7D3F3'	X'0093001F'	X'F0F4F0F0F0F3'	X'00000004'	X'00000005'

Table 2-2. Contents of RWSTAT.EXAMPLE (in hexadecimal) (continued)

A_NAME	DATE	TIME	R_ERR	W_ERR
X'C1D7D7D3F1'	X'0093001F'	X'F0F5F0F0F0F1'	X'00000001'	X'00000006'
X'C1D7D7D3F2'	X'0093001F'	X'F0F5F0F0F0F2'	X'00000004'	X'00000007'
X'C1D7D7D3F3'	X'0093001F'	X'F0F5F0F0F0F3'	X'00000002'	X'00000004'
X'C1D7D7D3F1'	X'0093001F'	X'F0F6F0F0F0F1'	X'00000001'	X'00000001'
X'C1D7D7D3F2'	X'0093001F'	X'F0F6F0F0F0F2'	X'00000004'	X'00000000'
X'C1D7D7D3F3'	X'0093001F'	X'F0F6F0F0F0F3'	X'00000003'	X'00000005'

Table 2-3 shows the results you want to obtain when you collect data from RWSTAT.EXAMPLE.

Table 2-3. Contents of data table after data collection

T_DATE	T_HOUR	RD_ERR	WR_ERR	TOT_ERR
1993-01-01	1	6	8	14
1993-01-01	2	7	4	11
1993-01-01	3	7	11	18
1993-01-01	4	6	11	17
1993-01-01	5	7	17	24
1993-01-01	6	8	6	14

The data table contains these fields:

T_DATE
Date the read and write errors occurred.

T_HOUR
Hour within the date the read and write errors occurred.

RD_ERR
Total number of read errors generated per hour.

WR_ERR
Total number of write errors generated per hour.

TOT_ERR
Combined total of read and write errors generated per hour.

To collect log data and produce the data table in Table 2-3, you must define to the log collector:

- The location and structure of the source data
- How to process that data
- How to store the results in a data table

You write these definitions in the log collector language, and then use the log collector to store the definitions.

You must also create the data table using SQL.

Defining a log

Figure 2-1 shows the DEFINE LOG statement used to define RWSTAT.EXAMPLE to the log collector.

```
-- Define RWSTAT log type to Tivoli Decision Support for z/OS
DEFINE LOG RWSTAT;
COMMENT ON LOG RWSTAT IS 'Log definition for RWSTAT'
```

Figure 2-1. DEFINE LOG statement

In Figure 2-1, you identified the log by specifying a name for the log (RWSTAT). Using the COMMENT ON statement, you also specified a description for the log, which appears when you list the log online using the administration dialog. For more information about the administration dialog, refer to the *Administration Guide and Reference*.

Defining a record

To define records to Tivoli Decision Support for z/OS, use the DEFINE RECORD statement in Figure 2-2.

```
-- Define R_REC record type to Tivoli Decision Support for z/OS
DEFINE RECORD R_REC IN LOG RWSTAT
  FIELDS
   (A_NAME   OFFSET 0  LENGTH 10 CHAR,
    DATE     OFFSET 10 LENGTH 4  DATE(0CYYDDDF),
    TIME     OFFSET 14 LENGTH 6  TIME(HHMMSS),
    R_ERR    OFFSET 20 LENGTH 4  BINARY,
    W_ERR    OFFSET 24 LENGTH 4  BINARY);
COMMENT ON RECORD R_REC IS 'Definition of R_REC record in RWSTAT';
```

Figure 2-2. DEFINE RECORD statement

In Figure 2-2, you identified the record (using the name R_REC) and specified that it occurs in the RWSTAT log. Then, you identified each of the fields in the record. For each field, you specified the name, where it occurs in the record, the length, and the format. Consider this field description:

```
R_ERR    OFFSET 20 LENGTH 4  BINARY,
```

The field called R_ERR begins at byte 20 of the R_REC record. It is 4 bytes long and contains data in a binary format.

You define the log and record to the log collector by executing these statements. See “Performing log collector statements” on page 2-6 for more information about executing log collector statements.

Creating a data table

To store data from R_REC records into a data table, you must create the table using SQL. You can issue SQL statements from QMF. For more information about using QMF to create a DB2 data table, refer to *Query Management Facility: Learner's Guide*.

You can also use the log collector language statement SQL to issue SQL commands from the same data set that contains other log collector statements.

Note: This section assumes that you know how to use SQL to create DB2 tables. If you are not an experienced using SQL, refer to *DB2 SQL Reference* for more information.

Figure 2-3 shows how to use the SQL statement to create the data table shown in Table 2-3 on page 2-2.

```
-- Submit SQL statements to create data table
SQL CREATE TABLE DRL.RWSTAT
  (T_DATE    DATE,
   T_HOUR    SMALLINT,
   RD_ERR    INTEGER,
   WR_ERR    INTEGER,
   TOT_ERR   INTEGER);
```

Figure 2-3. Creating a DB2 data table

In Figure 2-3, you identify the table (called DRL.RWSTAT) and specify each of the columns in the table.

When you use the log collector to store definitions (as discussed in “Performing log collector statements” on page 2-6), the SQL statements are executed.

Defining an update

Figure 2-4 shows how to use the DEFINE UPDATE statement to store data from the log into the data table.

```
-- Define update to store R_REC data in DRL.RWSTAT
DEFINE UPDATE TOT_ERRS
  FROM R_REC
  TO DRL.RWSTAT
  GROUP BY
   (T_DATE = DATE,
    T_HOUR = HOUR(TIME))
  SET
   (RD_ERR  = SUM(R_ERR),
    WR_ERR  = SUM(W_ERR),
    TOT_ERR = SUM(R_ERR + W_ERR));
```

Figure 2-4. DEFINE UPDATE statement

In Figure 2-4, you specified a name for this update process (TOT_ERRS) and specified how the source data is processed using the GROUP BY and SET clauses.

Understanding the GROUP BY clause

In each row of the data table, you want to collect data from records written during the same hour on the same day. The day is identified by the DATE field of each record. The TIME field identifies the exact time when the record was written. You can obtain the hour part of this time using the HOUR function specifying HOUR(TIME). First, you sort all records into groups with the same value of DATE and HOUR(TIME) using the GROUP BY clause:

```
GROUP BY
 (T_DATE = DATE,
  T_HOUR = HOUR(TIME))
```

The expressions to the right of the equal signs identify the values used for grouping. To visualize the grouping process, imagine that the log collector maintains a number of buckets labeled with different dates and hours.

When the log collector reads each record, it computes DATE and HOUR(TIME) for the record and then, drops the record into the appropriate bucket. After all records are processed, each bucket contains the group of records that were written during the same hour on the same day.

For each group of records, the log collector creates one row in the data table. It stores the values of DATE and TIME(HOUR) for each group in the columns T_DATE and T_HOUR, respectively. These are columns identified on the left of the equal sign.

Figure 2-5 on page 2-6 shows how source records are grouped based on the GROUP BY values specified in Figure 2-4 on page 2-4.

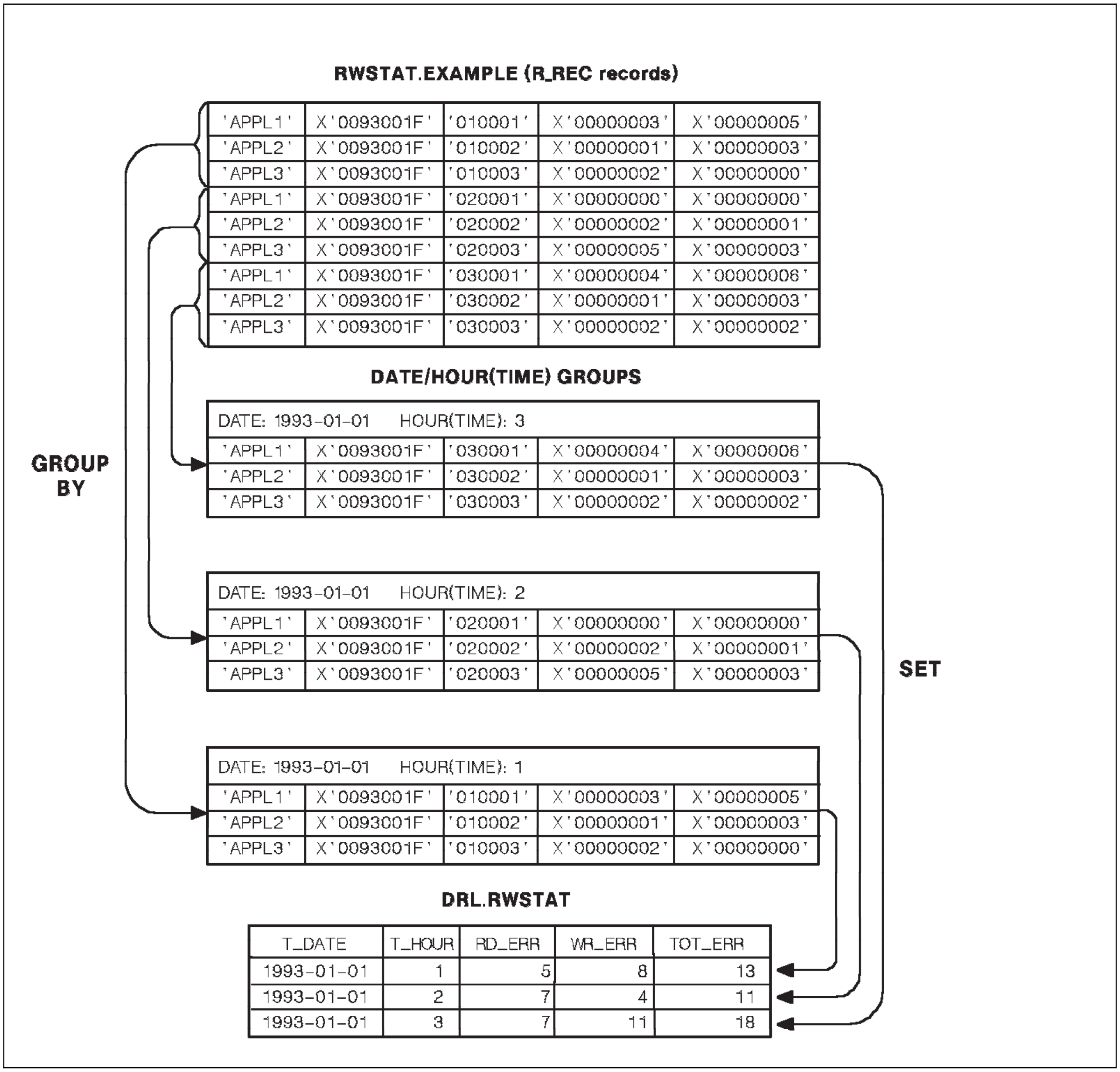
Understanding the SET clause

In each row of the data table, you want to store the sum of certain values from all records in the group represented by that row. You identified the sums to be stored in Figure 2-4 on page 2-4 using the SET clause:

```
SET
 (RD_ERR  = SUM(R_ERR),
  WR_ERR  = SUM(W_ERR),
  TOT_ERR = SUM(R_ERR + W_ERR));
```

Each expression to the right of the equal sign specifies what to compute. The name to the left of the equal sign identifies the column in the data table where the result is stored. For example, `TOT_ERR = SUM(R_ERR + W_ERR)` is an instruction to compute R_ERR + W_ERR for each record in the group, add up the resulting numbers, and store the sum in the column TOT_ERR of the row.

Figure 2-5 on page 2-6 shows how expressions you specified using the SET clause are applied to the groups of records and how the results are stored in the data table.



RWSTAT.EXAMPLE (R_REC records)

'APPL1'	X'0093001F'	'010001'	X'00000003'	X'00000005'
'APPL2'	X'0093001F'	'010002'	X'00000001'	X'00000003'
'APPL3'	X'0093001F'	'010003'	X'00000002'	X'00000000'
'APPL1'	X'0093001F'	'020001'	X'00000000'	X'00000000'
'APPL2'	X'0093001F'	'020002'	X'00000002'	X'00000001'
'APPL3'	X'0093001F'	'020003'	X'00000005'	X'00000003'
'APPL1'	X'0093001F'	'030001'	X'00000004'	X'00000006'
'APPL2'	X'0093001F'	'030002'	X'00000001'	X'00000003'
'APPL3'	X'0093001F'	'030003'	X'00000002'	X'00000002'

DATE/HOUR(TIME) GROUPS

DATE: 1993-01-01 HOUR(TIME): 3

'APPL1'	X'0093001F'	'030001'	X'00000004'	X'00000006'
'APPL2'	X'0093001F'	'030002'	X'00000001'	X'00000003'
'APPL3'	X'0093001F'	'030003'	X'00000002'	X'00000002'

DATE: 1993-01-01 HOUR(TIME): 2

'APPL1'	X'0093001F'	'020001'	X'00000000'	X'00000000'
'APPL2'	X'0093001F'	'020002'	X'00000002'	X'00000001'
'APPL3'	X'0093001F'	'020003'	X'00000005'	X'00000003'

DATE: 1993-01-01 HOUR(TIME): 1

'APPL1'	X'0093001F'	'010001'	X'00000003'	X'00000005'
'APPL2'	X'0093001F'	'010002'	X'00000001'	X'00000003'
'APPL3'	X'0093001F'	'010003'	X'00000002'	X'00000000'

DRL.RWSTAT

T_DATE	T_HOUR	RD_ERR	WR_ERR	TOT_ERR
1993-01-01	1	5	8	13
1993-01-01	2	7	4	11
1993-01-01	3	7	11	18

Figure 2-5. Example of GROUP BY and SET processing

Performing log collector statements

After writing statements to define the log, records within the log, and the update process, you can execute these statements to store the definitions. Then, you can use the definitions to collect log data.

For example, assume that you have typed all of the statements in a single data set called STATSDEF, which is a member of DRL.LOCAL.DEFS. Figure 2-6 on page 2-7 shows the contents of STATSDEF.

```
-- Define RWSTAT log type to Tivoli Decision Support for z/OS
DEFINE LOG RWSTAT;
COMMENT ON LOG RWSTAT IS 'Log definition for RWSTAT'

-- Define R_REC record type to Tivoli Decision Support for z/OS
DEFINE RECORD R_REC IN LOG RWSTAT
  FIELDS
   (A_NAME   OFFSET 0  LENGTH 10 CHAR,
    DATE     OFFSET 10 LENGTH 4  DATE(0CYYDDDF),
    TIME     OFFSET 14 LENGTH 6  TIME(HHMMSS),
    R_ERR    OFFSET 20 LENGTH 4  BINARY,
    W_ERR    OFFSET 24 LENGTH 4  BINARY);
COMMENT ON RECORD R_REC IS 'Definition of R_REC record in RWSTAT';

-- Submit SQL statements to create data table
SQL CREATE TABLE DRL.RWSTAT
  (T_DATE    DATE,
   T_HOUR    SMALLINT,
   RD_ERR    INTEGER,
   WR_ERR    INTEGER,
   TOT_ERR   INTEGER);

-- Define update to store R_REC data in DRL.RWSTAT
DEFINE UPDATE TOT_ERRS
  FROM R_REC
  TO DRL.RWSTAT
  GROUP BY
   (T_DATE = DATE,
    T_HOUR = HOUR(TIME))
  SET
   (RD_ERR  = SUM(R_ERR),
    WR_ERR  = SUM(W_ERR),
    TOT_ERR = SUM(R_ERR + W_ERR));
```

Figure 2-6. Contents of STATSDEF data set

To store the log, record, and update definitions, and to execute the SQL statement to create the data table, submit the JCL shown in Figure 2-7.

```
//jobname  JOB parameters
//LC       EXEC PGM=DRLPLC,PARM=('SYSPREFIX=DRLSYS SYSTEM=DSN'),
//         REGION=1M
//STEPLIB  DD DISP=SHR,DSN=DRL180.SDRLLOAD
//DRLIN    DD DISP=SHR,DSN=DRL.LOCAL.DEFS(STATSDEF)
//DRLOUT   DD SYSOUT=*
//DRLDUMP  DD SYSOUT=*
```

Figure 2-7. JCL for storing log and record definitions

To submit the job shown in Figure 2-7, you must add the appropriate high-level qualifier to the DD statements. You might also need to modify these parameters:

SYSTEM=DSN
Specifies DSN as the DB2 subsystem. If your DB2 system has a different name, specify it instead.

SYSPREFIX=DRLSYS
Specifies DRLSYS as the prefix of the log collector system tables. If the log collector system tables have a different prefix, specify it instead. If you do not know the prefix of the system tables, consult your system administrator.

//STEPLIB DD DISP=SHR,DSN=DRL180.SDRLLOAD

Specifies DRL180.SDRLLOAD as the name of the Tivoli Decision Support for z/OS load library. If the load library has a different name, specify it instead. If you do not know the name of the load library, consult your system administrator.

You might also need to add a DD statement for the DB2 load library to the STEPLIB statement. For example:

```
//        DD DISP=SHR,DSN=DB2.V810.SDSNLOAD
```

To execute the log collector statements in Figure 2-6 on page 2-7, run the job in Figure 2-7 on page 2-7 (modified as needed). If the statements are correct, the log collector stores the definitions and creates the data table. If the log collector finds errors in a statement, it will not execute that statement.

The DRLOUT data set will contain messages confirming the completion of each statement. It will also contain messages for statements that were not executed, explaining why they were not executed.

Verifying record definitions

After storing log and record definitions, use the LIST RECORD statement to verify that the record definition is correct. When you use the LIST RECORD statement, you do not collect data. Instead, the record definition is applied to the log data set.

Figure 2-8 shows the LIST RECORD statement for listing R_REC records.

```
LIST
  RECORD R_REC
  FIELDS DATE,
         TIME,
         R_ERR,
         W_ERR;
```

Figure 2-8. LIST RECORD statement

To use this LIST RECORD statement, submit the JCL shown in Figure 2-9.

```
//jobname  JOB parameters
//LC       EXEC PGM=DRLPLC,PARM=('SYSPREFIX=DRLSYS SYSTEM=DSN'),
//         REGION=1M
//STEPLIB  DD DISP=SHR,DSN=DRL180.SDRLLOAD
//DRLLOG   DD DISP=SHR,DSN=RWSTAT.EXAMPLE
//DRLIN    DD *
   LIST RECORD R_REC
        FIELDS DATE,
               TIME,
               R_ERR,
               W_ERR;
//DRLOUT   DD SYSOUT=*
//DRLLST1  DD SYSOUT=*
//DRLDUMP  DD SYSOUT=*
```

Figure 2-9. JCL for listing records

When you submit the JCL, change any system-dependent parameters (see "Performing log collector statements" on page 2-6).

At the completion of the job, DRLOUT contains messages from the log collector as a result of the LIST RECORD statement execution. Figure 2-10 shows an example of the messages that appear in DRLOUT.

```
DRL0300I List started at 1993-02-25-00.14.21.
DRL0302I Processing RWSTAT.EXAMPLE on TSOL01.
DRL0380I 18 records read from the input log.
DRL0003I
DRL0315I Records read from the log or built by log procedure:
DRL0317I Record name        │     Number
DRL0318I -------------------│----------
DRL0319I R_REC              │         18
DRL0318I -------------------│----------
DRL0321I Total              │         18
DRL0381I 20 records written to the DRLLST1 file.
DRL0301I List ended at 1993-02-25-00.14.21.
```

Figure 2-10. Messages resulting from LIST RECORD statement execution

The file DRLLST1 contains a list of the record fields and the data contained in those fields. Figure 2-11 shows an example of DRLLST1 contents.

```
DATE        TIME            R_ERR        W_ERR
----------  --------  -----------  -----------
1993-01-01  01.00.01            3            5
1993-01-01  01.00.02            1            3
1993-01-01  01.00.03            2            0
1993-01-01  02.00.01            0            0
1993-01-01  02.00.02            2            1
1993-01-01  02.00.03            5            3
1993-01-01  03.00.01            4            6
1993-01-01  03.00.02            1            3
1993-01-01  03.00.03            2            2
1993-01-01  04.00.01            2            6
1993-01-01  04.00.02            0            0
1993-01-01  04.00.03            4            5
1993-01-01  05.00.01            1            6
1993-01-01  05.00.02            4            7
1993-01-01  05.00.03            2            4
1993-01-01  06.00.01            1            1
1993-01-01  06.00.02            4            0
1993-01-01  06.00.03            3            5
```

Figure 2-11. Records listed by the LIST RECORD statement

Collecting log data

After successfully storing definitions, you can collect log data using those definitions.

Collecting log data in batch

To collect log data in batch, use the JCL in Figure 2-12 on page 2-10.

```
//jobname  JOB parameters
//LC       EXEC PGM=DRLPLC,PARM=('SYSPREFIX=DRLSYS SYSTEM=DSN'),
//         REGION=1M
//STEPLIB  DD DISP=SHR,DSN=DRL180.SDRLLOAD
//DRLIN    DD *
           COLLECT RWSTAT;
//DRLLOG   DD DISP=SHR,DSN=RWSTAT.EXAMPLE
//DRLOUT   DD SYSOUT=*
//DRLDUMP  DD SYSOUT=*
```

Figure 2-12. JCL used to collect log data

In Figure 2-12, you specify that you want to collect RWSTAT logs. When you execute this JCL, the log collector processes all update definitions that you have stored for this log. Here, the log collector processes the TOT_ERRS update, reading data from R_REC records and storing it in DRL.RWSTAT. The data set specified by the DRLOUT statement contains error messages that occur during processing.

At the completion of the job, DRLOUT contains messages from the log collector as a result of the COLLECT statement execution. Figure 2-13 shows an example of the messages that appear in DRLOUT.

```
               COLLECT RWSTAT
DRL0300I Collect started at 1993-03-31-22.47.34
DRL0302I Processing RWSTAT.EXAMPLE on TSOL02
DRL0310I A database update started after 18 records due to end of log
DRL0313I The collect buffer was filled 0 times. Consider increasing collect buffer size
DRL0003I
DRL0315I Records read from the log or built by log procedure:
DRL0317I Record name        |    Number
DRL0318I -------------------|----------
DRL0319I R_REC              |        18
DRL0318I -------------------|----------
DRL0321I Total              |        18
DRL0003I
DRL0323I                               -------Buffer------ ------Database-----
DRL0324I Table name                  |   Inserts   Updates   Inserts   Updates
DRL0325I ----------------------------|----------------------------------------
DRL0326I DRL     .RWSTAT             |         6        12         6         0
DRL0325I ----------------------------|----------------------------------------
DRL0327I Total                       |         6        12         6         0
DRL0003I
DRL0301I Collect ended at 1993-03-31-22.47.37
```

Figure 2-13. Messages resulting from COLLECT statement execution

DRL.RWSTAT contains the data shown in Table 2-4.

Table 2-4. Contents of DRL.RWSTAT after data collection

T_DATE	T_HOUR	RD_ERR	WR_ERR	TOT_ERR
2000-01-01	1	6	8	14
2000-01-01	2	7	4	11
2000-01-01	3	7	11	18
2000-01-01	4	6	11	18
2000-01-01	5	7	17	24
2000-01-01	6	8	6	14

Collecting log data online

You can perform the data collection process online using the administration dialog. From the administration dialog, you can:

- Execute log collector statements
- Verify record definitions
- Collect log data using the COLLECT statement

For more information about using the administration dialog, refer to the *Administration Guide*.

Chapter 3. Defining logs and records

Chapter 2, “How to use the log collector language,” on page 2-1, described how to write definitions and collect log data from a simple log data set. However, log data sets are typically much more complex. Many log data sets contain more than one record type, each with a different record structure.

This chapter describes more about writing record definitions. It also explains how to define more complex record structures and how to modify record definitions after they are stored.

Learning more about writing record definitions

About this task

Figure 3-1 shows the record definition used to define R_REC records.

```
-- Define R_REC record type to Tivoli Decision Support for z/OS
DEFINE RECORD R_REC IN LOG RWSTAT
  FIELDS
    (A_NAME   OFFSET 0  LENGTH 10 CHAR,
     DATE     OFFSET 10 LENGTH 4  DATE(0CYYDDDF),
     TIME     OFFSET 14 LENGTH 6  TIME(HHMMSS),
     R_ERR    OFFSET 20 LENGTH 4  BINARY,
     W_ERR    OFFSET 24 LENGTH 4  BINARY);
COMMENT ON RECORD R_REC IS 'Definition of R_REC record in RWSTAT;
```

Figure 3-1. Record definition for R_REC record type

For each of the fields, a field name, offset, length, and field format were specified. When you define information about the fields, you can also use these specifications:

Field name

You must provide a field name if you plan to collect data from that field. Otherwise, you can use an asterisk (*) for the field name.

For example, you could specify this record definition:

```
DEFINE RECORD R_REC IN LOG RWSTAT
  FIELDS
   (*        OFFSET  0 LENGTH 10 CHAR,
    DATE     OFFSET 10 LENGTH  4 DATE(0CYYDDDF),
    TIME     OFFSET 14 LENGTH  6 TIME(HHMMSS),
    R_ERR    OFFSET 20 LENGTH  4 BINARY,
    W_ERR    OFFSET 24 LENGTH  4 BINARY);
```

This definition specifies that a field begins at offset 0 and is 10 bytes long. When you use an asterisk for the field name, however, you cannot refer to this field using any other log collectorlog collector statement (such as DEFINE UPDATE or LIST RECORD).

Field offset

You can explicitly identify each offset as shown in Figure 3-1, or you can leave the offset blank. For example, you could specify this record definition:

```
DEFINE RECORD R_REC IN LOG RWSTAT
  FIELDS
   (A_NAME   OFFSET  0 LENGTH 10 CHAR,
    DATE     OFFSET 10 LENGTH  4 DATE(0CYYDDDF),
    TIME               LENGTH  6 TIME(HHMMSS),
    R_ERR              LENGTH  4 BINARY,
    W_ERR    OFFSET 24 LENGTH  4 BINARY);
```

The DATE field begins at offset 10 and is 4 bytes long. An omitted offset for the TIME field means that it immediately follows the DATE field and begins at offset 14 (the offset of the DATE field plus the length of the DATE field). Because the R_ERR field immediately follows the TIME field, it begins at offset 20.

Field length

You can explicitly specify the length of a field or use the default length, which is determined by the field format.

For example, the default length for a binary field is 4 bytes. So, you could specify the R_ERR field without a length:

```
DEFINE RECORD R_REC IN LOG RWSTAT
  FIELDS
   (A_NAME   OFFSET  0 LENGTH 10 CHAR,
    DATE     OFFSET 10 LENGTH  4 DATE(0CYYDDDF),
    TIME               LENGTH  6 TIME(HHMMSS),
    R_ERR                        BINARY,
    W_ERR    OFFSET 24 LENGTH  4 BINARY);
```

Field format

If you do not specify a format for a field, the default format is hexadecimal. Table 11-1 on page 11-23 shows a complete list of field formats.

You can also specify the length of some fields when you specify the format. For example, instead of using the field length when you specify a character field format, you can use CHAR(4).

Note: You can specify the field length and field format together for the CHAR and BIT field formats. This notation is not allowed on any other field formats.

When you define a record using the DEFINE RECORD statement, you must specify only the fields from which you plan to collect data. For example, Figure 3-2 shows another way to define fields in the R_REC record:

```
DEFINE RECORD R_REC IN LOG RWSTAT
  FIELDS
   (DATE     OFFSET 10         DATE(0CYYDDDF),
    TIME                       TIME(HHMMSS),
    R_ERR                      BINARY,
    W_ERR                      BINARY);
```

Figure 3-2. Defining the R_REC using defaults

In Figure 3-2, you do not specify the A_NAME field because it does not provide information that is meaningful to the total number of read and write errors per hour. You also use defaults for the field offset and field length (the offset for the DATE field is necessary because the field does not begin at offset 0).

Although using the defaults shown in Figure 3-2 on page 3-2 makes typing definitions quicker, you should be careful. Assume that you defined the fields in R_REC with no explicit offsets. This means that A_NAME begins at offset 0 and DATE begins at offset 10. If you later edit the definition and delete A_NAME, DATE would be listed as the first field in the record and begin at offset 0. The offset of all other fields would be reduced by 10 (the length of A_NAME). Collecting data with this definition would produce invalid results.

Another problem could occur if you specified an incorrect length for a field. Because offsets are calculated using lengths, the error would result in an incorrect offset for all remaining fields.

Defining sections within a record

About this task

Many log data set records, such as SMF records, contain *sections*. A section is a series of adjacent bytes that contain data located within a record. In records containing sections, information about the section (such as the offset within the record where the section occurs and the length of the section) can be stored within the record itself. This information can be fixed or can vary, depending on the data in the record (and it can differ for each record).

A record can also have *repeated sections*, which are sections that occur more than once in a record, and *nested sections*, which are sections within sections. For more information about repeated sections, see "Using repeated sections within records" on page 5-2. For more information about nested sections, see "Using nested sections within records" on page 5-6.

Defining a record containing a section

Assume that you want to collect data about the subsystems running under MVS in your organization. The data is contained in a section called SUB_1 of the SUB_REC record. Two fields (SUB_OFF and SUB_LEN) in the record provide information about the location and length of the subsystem section. The SUB_1 section begins at the offset specified in the SUB_OFF field. Its length is specified in the SUB_LEN field.

Table 3-1 shows the structure of the SUB_REC record.

Table 3-1. Structure of a record containing a section

Field name	Offset	Length	Data format	Description
REC_LEN	0	2	Binary	Length of the record
REC_TYPE	2	2	Character	Type of record
REC_SID	4	4	Character	System identifier
REC_DATE	8	4	Packed Decimal	Date record was written
REC_TIME	12	6	Character	Time record was written
SUB_OFF	18	2	Binary	Offset of subsystem section
SUB_LEN	20	4	Binary	Length of subsystem section
• • (Other fields within the record) •				
SUB_1 section				

Table 3-1. Structure of a record containing a section (continued)

Field name	Offset	Length	Data format	Description
SUB1_TYPE	0	2	Character	Subsystem identifier
SUB1_PNM	2	8	Character	Program name
SUB1_VER	10	2	Character	Version number of program
SUB1_REL	12	2	Character	Release level of program

Figure 3-3 shows the format of the SUB_REC record.

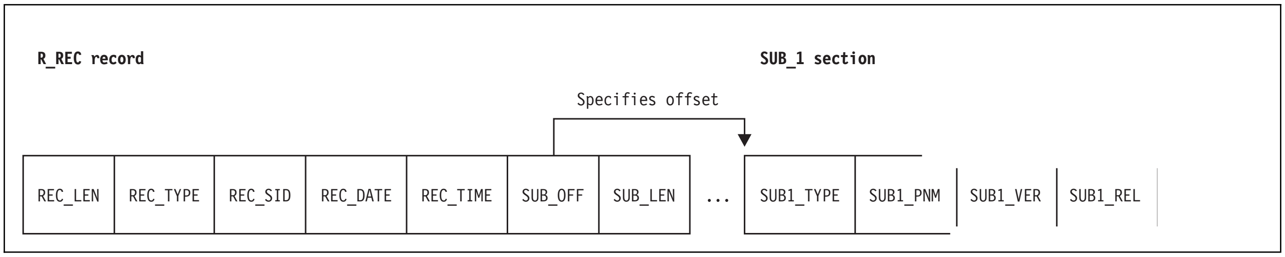


Figure 3-3. Structure of the SUB_REC record and SUB_1 section

To write a record definition for the SUB_REC record, use the DEFINE RECORD statement shown in Figure 3-4.

```
-- Define SUB_REC record
DEFINE RECORD SUB_REC IN LOG SUB_LOG
  FIELDS
   (REC_LEN    OFFSET  0  LENGTH 2 BINARY,
    REC_TYPE   OFFSET  2  LENGTH 2 CHAR,
    REC_SID    OFFSET  4  LENGTH 4 CHAR,
    REC_DATE   OFFSET  8  LENGTH 4 DATE(0CYYDDDF),
    REC_TIME   OFFSET 12  LENGTH 6 TIME(HHMMSS),
    SUB_OFF    OFFSET 18  LENGTH 2 BINARY,
    SUB_LEN    OFFSET 20  LENGTH 4 BINARY)

  -- Define section SUB_1
  SECTION SUB_1
   OFFSET  SUB_OFF
   LENGTH  SUB_LEN
   FIELDS
    (SUB1_TYPE  OFFSET 0   LENGTH 2 CHAR,
     SUB1_PNM   OFFSET 2   LENGTH 8 CHAR,
     SUB1_VER   OFFSET 10  LENGTH 2 CHAR,
     SUB1_REL   OFFSET 12  LENGTH 2 CHAR)

  -- End of definition for section SUB_1
-- End of definition for record SUB_REC
;
COMMENT ON RECORD SUB_REC IS 'Definition for record with a section';
```

Figure 3-4. Defining a record with a section

In Figure 3-4, you identify the record (called SUB_REC) and specify that it occurs in a log called SUB_LOG.

Next, you identify each of the fields in the record. Remember that you need only define the fields that you want to reference later. However, **you must** identify the SUB_OFF and SUB_LEN fields because they are used in the SECTION clause.

The SECTION clause specifies that section SUB_1 occurs in the SUB_REC record and identifies the fields that occur in the section:

```
SECTION SUB_1
 OFFSET  SUB_OFF
 LENGTH  SUB_LEN
 FIELDS
  (SUB1_TYPE  OFFSET 0  LENGTH 2 CHAR,
   SUB1_PNM   OFFSET 2  LENGTH 8 CHAR,
   SUB1_VER   OFFSET 10 LENGTH 2 CHAR,
   SUB1_REL   OFFSET 12 LENGTH 2 CHAR)
```

Using the OFFSET clause, you specify that the value contained in field SUB_OFF is the offset where SUB_1 begins. The length of SUB_1 is the value contained in SUB_LEN.

You identify fields in a section the same way you identified fields in a record. The offsets of a field in a section begin from the start of the section. So, the first field, SUB1_TYPE, begins at offset 0.

You can document sections within your definitions by adding comments:

```
-- Define SUB_REC record
   -- Define section SUB_1
   -- End of definition for section SUB_1
-- End of definition for record SUB_REC
```

Typically, record definitions are complex and contain definitions for many different sections. Adding comments throughout record definitions make them easier to read.

Defining multiple record types

So far, you have assumed that all records within a log data set are the same. However, log data sets typically contain many different kinds of records (called *record types*).

For example, assume that you have a log data set (called RWINFO.LOG) that contains data about read and write errors. Some applications (APPL1, APPL2, and APPL3) write records of type A to RWINFO.LOG.

Table 3-2 shows the structure of type A records. Notice that records of type A always have a field called REC_TYPE that contains the value A.

Table 3-2. Structure of Type A records in RWINFO.LOG log data set

Field name	**Offset**	**Length**	**Data format**	**Description**
REC_TYPE	0	2	Character	Contains the record type (here, it is A)
A_NAME	2	10	Character	Contains the name of the application writing to this data set
DATE	12	4	Packed decimal in format 0*cyydddf*	Contains the date, where: **0c** Century **yy** Year within the century **ddd** Day within the year **f** Any character

Table 3-2. Structure of Type A records in RWINFO.LOG log data set (continued)

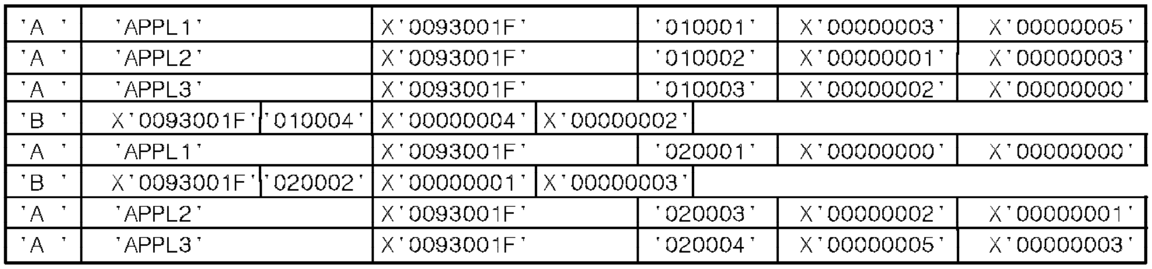
Field name	Offset	Length	Data format	Description
TIME	16	6	Character string in the format *hhmmss*	Contains the time where: **hh** Hour **mm** Minute **ss** Second
R_ERR	22	4	Binary	The number of read errors
W_ERR	26	4	Binary	The number of write errors

Other applications (APPL4, APPL5, and APPL6) write records of type B to RWINFO.LOG. Table 3-3 shows the structure of type B records. Notice that records of type B always have a field called REC_TYPE that contains the value B.

Table 3-3. Structure of Type B records in RWINFO.LOG log data set

Field name	Offset	Length	Data format	Description
REC_TYPE	0	2	Character	Contains the record type (here, it is B)
DATE	2	4	Packed decimal in the format 0*cyydddF*	Contains the date where: **0c** Century **yy** Year within the century **ddd** Day within the year **f** Any character
TIME	6	6	Character string in the format *hhmmss*	Contains the time where: **hh** Hour **mm** Minute **ss** Second
R1_ERR	12	4	Binary	The number of read errors
W1_ERR	16	4	Binary	The number of write errors

Figure 3-5 shows the contents of the records in RWINFO.LOG.



'A '	'APPL1'	X'0093001F'	'010001'	X'00000003'	X'00000005'
'A '	'APPL2'	X'0093001F'	'010002'	X'00000001'	X'00000003'
'A '	'APPL3'	X'0093001F'	'010003'	X'00000002'	X'00000000'
'B '	X'0093001F' '010004'	X'00000004'	X'00000002'		
'A '	'APPL1'	X'0093001F'	'020001'	X'00000000'	X'00000000'
'B '	X'0093001F' '020002'	X'00000001'	X'00000003'		
'A '	'APPL2'	X'0093001F'	'020003'	X'00000002'	X'00000001'
'A '	'APPL3'	X'0093001F'	'020004'	X'00000005'	X'00000003'

Figure 3-5. Contents of RWINFO.LOG data set.

Tivoli Decision Support for z/OS processes records according to the following standard:

- If a record is fixed length, the first 2 bytes identify the record type.
- If a record is variable length, the second 2 bytes identify the record length.

Tivoli Decision Support for z/OS can process customized records that do not contain these fields.

Note: If customized records differ only in the first 2 or 4 bytes and are otherwise identical, Tivoli Decision Support for z/OS assumes that the log has already been processed.

Defining the records

About this task

Figure 3-6 shows how to define both type A and type B records.

```
-- Create the log and record definitions
DEFINE LOG RWINFO;
COMMENT ON LOG RWINFO IS 'Definition of log with multiple records';

-- Create record definition for Type A records
DEFINE RECORD TYPA_REC IN LOG RWINFO
  IDENTIFIED BY REC_TYPE='A'
  FIELDS
   (REC_TYPE OFFSET  0 LENGTH  2 CHAR,
    A_NAME   OFFSET  2 LENGTH 10 CHAR,
    DATE     OFFSET 12 LENGTH  4 DATE(0CYYDDDF),
    TIME     OFFSET 16 LENGTH  6 TIME(HHMMSS),
    R_ERR    OFFSET 22 LENGTH  4 BINARY,
    W_ERR    OFFSET 26 LENGTH  4 BINARY);
COMMENT ON RECORD TYPA_REC IS 'Definition for type A records';

-- Create record definition for Type B records
DEFINE RECORD TYPB_REC IN LOG RWINFO
  IDENTIFIED BY REC_TYPE='B'
  FIELDS
   (REC_TYPE OFFSET  0 LENGTH  2 CHAR,
    DATE     OFFSET  2 LENGTH  4 DATE(0CYYDDDF),
    TIME     OFFSET  6 LENGTH  6 TIME(HHMMSS),
    R1_ERR   OFFSET 12 LENGTH  4 BINARY,
    W1_ERR   OFFSET 16 LENGTH  4 BINARY);
COMMENT ON RECORD TYPB_REC IS 'Definition for type B records';
```

Figure 3-6. Defining multiple records

In Figure 3-6, you create a separate definition for each record type. You distinguish between different record types using the IDENTIFIED BY clause. Whenever REC_TYPE='A', the record definition for type A records apply. Whenever REC_TYPE='B', the record definition for type B records apply.

“Storing data from multiple sources in a single data table” on page 4-1 describes how to use these definitions to collect data and update a data table.

Changing log and record definitions

About this task

After you have stored log and record definitions you can change them:

- Using the DROP statement to delete the existing definition and then using the DEFINE LOG or the DEFINE RECORD statement to write a new log or record definition
- Using the ALTER LOG or the ALTER RECORD statement to change a log or record definition

Using the DROP statement to delete a record definition

About this task

You can use the DROP statement to delete a stored record definition. For example, assume that you wanted to delete the stored definition for SUB_REC records. To delete the definition, use this statement:

```
DROP RECORD SUB_REC;
```

You can also use the DROP statement in combination with the DEFINE RECORD statement to make modifications to a stored definition. Assume that you want to add N_FIELD to the SUB_1 section beginning at offset 0. Because you have explicitly defined the offsets for each field, you must redefine each offset. One way to redefine them would be to use the statements in Figure 3-7.

```
DROP RECORD SUB_REC;

-- Define SUB_REC record
DEFINE RECORD SUB_REC IN LOG SUB_LOG
  IDENTIFIED BY REC_TYPE='5'
  FIELDS
   (REC_LEN    OFFSET  0  LENGTH 2 BINARY,
    REC_TYPE   OFFSET  2  LENGTH 2 CHAR,
    REC_SID    OFFSET  4  LENGTH 4 CHAR,
    REC_DATE   OFFSET  8  LENGTH 4 DATE(0CYYDDDF),
    REC_TIME   OFFSET 12  LENGTH 6 TIME(HHMMSS),
    SUB_OFF    OFFSET 18  LENGTH 2 BINARY,
    SUB_LEN    OFFSET 20  LENGTH 4 BINARY)

  -- Define section SUB_1 record
  SECTION SUB_1
   OFFSET  SUB_OFF
   LENGTH  SUB_LEN
   FIELDS
    (N_FIELD    OFFSET  0  LENGTH 2 BINARY,
     SUB1_TYPE  OFFSET  2  LENGTH 2 CHAR,
     SUB1_PNM   OFFSET  4  LENGTH 8 CHAR,
     SUB1_VER   OFFSET 12  LENGTH 2 CHAR,
     SUB1_REL   OFFSET 14  LENGTH 2 CHAR)

  -- End of definition for section SUB_1

-- End of definition for record SUB_REC
;
COMMENT ON RECORD SUB_REC IS 'Definition for record with a section';
```

Figure 3-7. Using the DROP statement to redefine a record

The statement `DROP RECORD SUB_REC` deletes the stored definition of SUB_REC. The statement `DEFINE RECORD SUB_REC` stores a new definition.

Using the ALTER RECORD statement

About this task

You can use the ALTER RECORD statement to change a stored record definition. However, you typically want to make only quick changes using the ALTER RECORD statement, because you cannot see the original DEFINE RECORD statement when you use the ALTER RECORD statement. In addition, if you typed the original DEFINE RECORD statement into a data set and then used the ALTER RECORD statement to change it, the data set would no longer contain the latest record definition.

For example, assume you had this record definition:

```
-- Define SUB_REC record
DEFINE RECORD SUB_REC IN LOG SUB_LOG
  IDENTIFIED BY REC_TYPE='5'
  FIELDS
   (REC_LEN    OFFSET  0  LENGTH 2 BINARY,
    REC_TYPE   OFFSET  2  LENGTH 2 CHAR,
    REC_SID    OFFSET  4  LENGTH 4 CHAR,
    REC_DATE   OFFSET  8  LENGTH 4 DATE(0CYYDDDF),
    REC_TIME   OFFSET 12  LENGTH 6 TIME(HHMMSS),
    SUB_OFF    OFFSET 18  LENGTH 2 BINARY,
    SUB_LEN    OFFSET 20  LENGTH 4 BINARY)

  -- Define section SUB_1 record
  SECTION SUB_1
   OFFSET  SUB_OFF
   LENGTH  SUB_LEN
   FIELDS
    (SUB1_TYPE  OFFSET  0  LENGTH 2 CHAR,
     SUB1_PNM   OFFSET  2  LENGTH 8 CHAR,
     SUB1_VER   OFFSET 10  LENGTH 2 CHAR,
     SUB1_REL   OFFSET 12  LENGTH 2 CHAR)

  -- End of definition for section SUB_1
-- End of definition for record SUB_REC
;
COMMENT ON RECORD SUB_REC IS 'Definition for record containing section';
```

Figure 3-8. Sample record definition

You can modify fields and change record procedures using the ALTER RECORD statement. For example, assume that you wanted to add a field called N_FIELD to the end of section SUB_1. You could add the field using the ALTER RECORD statement in Figure 3-9.

```
ALTER RECORD SUB_REC
  ADD FIELDS(N_FIELD OFFSET 14 LENGTH 2 BINARY) IN SECTION SUB_1;
```

Figure 3-9. Changing a record definition

Executing this ALTER RECORD statement adds N_FIELD to the section SUB_1, starting at offset 14.

Chapter 4. Updating, storing, and managing data in tables

Although it can be useful to store data from a single record type in a log data set into a data table, you can also perform more complex tasks with data tables. For example, you can store data from multiple record types into a data table. Then, you can take the data from that data table, summarize it, and store the result in another data table.

This chapter describes how to update a data table from multiple record types and how to store data from one data table into another data table. It also explains how to use log collector language statements to manage data within tables.

Storing data from multiple sources in a single data table

About this task

"Defining multiple record types" on page 3-5 described how to define two record types (type A and type B) that occur in RWINFO.LOG. These log and record definitions were used:

```
-- Create the log and record definitions
DEFINE LOG RWINFO;
COMMENT ON LOG RWINFO IS 'Definition of log containing multiple records';

-- Create record definition for Type A records
DEFINE RECORD TYPA_REC IN LOG RWINFO
  IDENTIFIED BY REC_TYPE='A'
  FIELDS
   (REC_TYPE OFFSET  0 LENGTH  2 CHAR,
    A_NAME   OFFSET  2 LENGTH 10 CHAR,
    DATE     OFFSET 12 LENGTH 44 DATE(0CYYDDDF),
    TIME     OFFSET 16 LENGTH  6 TIME(HHMMSS),
    R_ERR    OFFSET 22 LENGTH  4 BINARY,
    W_ERR    OFFSET 26 LENGTH  4 BINARY);
COMMENT ON RECORD TYPA_REC IS 'Definition for type A records';

-- Create record definition for Type B records
DEFINE RECORD TYPB_REC IN LOG RWINFO
  IDENTIFIED BY REC_TYPE='B'
  FIELDS
   (REC_TYPE OFFSET  0 LENGTH  2 CHAR,
    DATE     OFFSET  2 LENGTH  4 DATE(0CYYDDDF),
    TIME     OFFSET  6 LENGTH  6 TIME(HHMMSS),
    R1_ERR   OFFSET 12 LENGTH  4 BINARY,
    W1_ERR   OFFSET 16 LENGTH  4 BINARY);
COMMENT ON RECORD TYPB_REC IS 'Definition for type B records';
```

Figure 4-1. Definitions used in RWINFO.LOG

You want to collect data from both records and store the result in a single data table. To do so, you must create the table and define the update process for storing data in the data table.

Creating the data table

About this task

The DRL.STATS_H data table is used to store the collected data. The table contains these columns (the data is derived from both record types):

- D_DATE is the date the records are written.
- D_HOUR is the hour in the date the records are written.
- RD_ERR is the total read errors (from both record types) generated per hour.
- WR_ERR is the total write errors (from both record types) generated per hour.
- TOT_ERR is the total number of read and write errors generated per hour.

Figure 4-2 shows the SQL log collector language statement you use to create this table.

```
-- Create a data table to store the collected data
SQL CREATE TABLE DRL.STATS_H
  (D_DATE    DATE,
   D_HOUR    SMALLINT,
   RD_ERR    INTEGER,
   WR_ERR    INTEGER,
   TOT_ERR   INTEGER);
```

Figure 4-2. Creating the DRL.STATS_H data table

Writing the update definition

About this task

To store data from both record types (A and B) into a single data table, use two DEFINE UPDATE statements. Each DEFINE UPDATE statement is similar to the one used in "Defining an update" on page 2-4:

```
-- Define update to store R_REC data in DRL.RWSTAT
DEFINE UPDATE TOT_ERRS
  FROM R_REC
  TO DRL.RWSTAT
  GROUP BY
   (T_DATE = DATE,
    T_HOUR = HOUR(TIME))
  SET
   (RD_ERR  = SUM(R_ERR),
    WR_ERR  = SUM(W_ERR),
    TOT_ERR = SUM(R_ERR + W_ERR));
```

When you store data from more than one record type into a single data table, follow these rules:

- All update definitions must use the same data table columns to store grouping values. That is, the column names listed for the GROUP BY clause must be the same for both record types.
- For example, to store data from two records in the same column, you must accumulate that data in the same way. For example, if data from two records is to be stored in column COL_A, enter:

  ```
  COL_A = SUM(REC1_FIELD)
  ```

  for the first record type. You must also use the SUM function for the field from the second record type that is to be stored in column COL_A:

  ```
  COL_A = SUM(REC2_FIELD)
  ```

Figure 4-3 shows the DEFINE UPDATE statements used to collect data from both records and to store the result in a single data table.

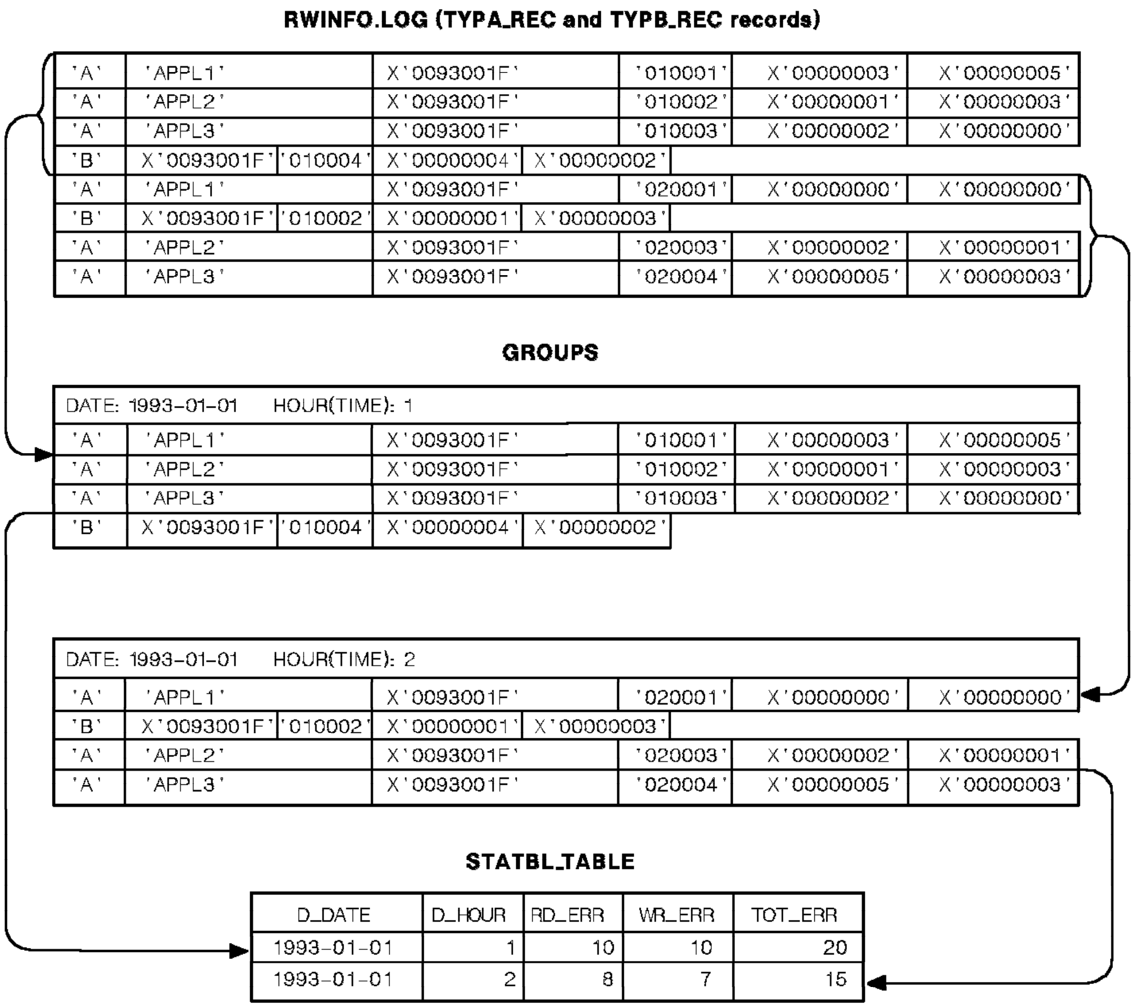
```
-- Create update definition for TYPA_REC records
DEFINE UPDATE ALL_ERRS
  FROM TYPA_REC
  TO DRL.STATS_H
  GROUP BY
   (D_DATE = DATE,
    D_HOUR = HOUR(TIME))
  SET
   (RD_ERR  = SUM(R_ERR),
    WR_ERR  = SUM(W_ERR),
    TOT_ERR = SUM(R_ERR + W_ERR));

-- Create update definition for TYPB_REC records
DEFINE UPDATE ALL1_ERRS
  FROM TYPB_REC
  TO DRL.STATS_H
  GROUP BY
   (D_DATE = DATE,
    D_HOUR = HOUR(TIME))
  SET
   (RD_ERR  = SUM(R1_ERR),
    WR_ERR  = SUM(W1_ERR),
    TOT_ERR = SUM(R1_ERR + W1_ERR));
```

Figure 4-3. Creating multiple update definitions for a single data table

Based on these update definitions, data from both record types will be grouped together by date and hour. The functions specified by the SET clause will be applied to these groups and the result for each date and hour group will be stored as a single row in DRL.STATS_H.

Figure 4-4 on page 4-4 shows the process used to store data from two different records in a single data table.

Storing data from multiple sources in a single data table



RWINFO.LOG (TYPA_REC and TYPB_REC records)

'A'	'APPL1'	X'0093001F'	'010001'	X'00000003'	X'00000005'
'A'	'APPL2'	X'0093001F'	'010002'	X'00000001'	X'00000003'
'A'	'APPL3'	X'0093001F'	'010003'	X'00000002'	X'00000000'
'B'	X'0093001F'	'010004'	X'00000004'	X'00000002'	
'A'	'APPL1'	X'0093001F'	'020001'	X'00000000'	X'00000000'
'B'	X'0093001F'	'010002'	X'00000001'	X'00000003'	
'A'	'APPL2'	X'0093001F'	'020003'	X'00000002'	X'00000001'
'A'	'APPL3'	X'0093001F'	'020004'	X'00000005'	X'00000003'

GROUPS

DATE: 1993-01-01 HOUR(TIME): 1

'A'	'APPL1'	X'0093001F'	'010001'	X'00000003'	X'00000005'
'A'	'APPL2'	X'0093001F'	'010002'	X'00000001'	X'00000003'
'A'	'APPL3'	X'0093001F'	'010003'	X'00000002'	X'00000000'
'B'	X'0093001F'	'010004'	X'00000004'	X'00000002'	

DATE: 1993-01-01 HOUR(TIME): 2

'A'	'APPL1'	X'0093001F'	'020001'	X'00000000'	X'00000000'
'B'	X'0093001F'	'010002'	X'00000001'	X'00000003'	
'A'	'APPL2'	X'0093001F'	'020003'	X'00000002'	X'00000001'
'A'	'APPL3'	X'0093001F'	'020004'	X'00000005'	X'00000003'

STATBL_TABLE

D_DATE	D_HOUR	RD_ERR	WR_ERR	TOT_ERR
1993-01-01	1	10	10	20
1993-01-01	2	8	7	15

Figure 4-4. Processing two update definitions.

After you store the update definitions and collect log data, DRL.STATS_H will contain the data in Table 4-1.

Note: The data used to produce the third row of the data table is not shown in the log data set.

Table 4-1. Contents of DRL.STATS_H data table after collecting log data

D_DATE	D_HOUR	RD_ERR	WR_ERR	TOT_ERR
1993-01-01	1	10	10	20
1993-01-01	2	8	7	15
1993-01-01	3	7	11	18

Storing data in multiple data tables

About this task

When you collect data, you can:
- Store data about hourly activities in one data table
- Summarize the hourly activities and store the result in another data table

- Store a weekly summary in yet another data table

Storing data in one table, summarizing it, and storing the result in another table is called a *cascaded update*.

Defining a cascaded update

Assume that, when you collect data and store it in DRL.STATS_H (on an hourly basis), you also want to summarize the data on a daily basis and store the result in another data table, called DRL.STATS_D.

To write a cascaded update definition, you must create a data table to store the summary data. Then, you can write the update definition to store data from the first data table into the summary data table.

Creating the summary data table

About this task

Figure 4-5 shows how to create DRL.STATS_D.

```
-- Creating a summary table
SQL CREATE TABLE DRL.STATS_D
    (D_DATE  DATE,
     RD_ERR  INTEGER,
     WR_ERR  INTEGER,
     TOT_ERR INTEGER );
```

Figure 4-5. Creating a summary data table

DRL.STATS_D contains these columns:
- D_DATE is the date the record was written.
- RD_ERR is the total read errors per day.
- WR_ERR is the total write errors per day.
- TOT_ERR is the total errors per day.

Defining an update for the summary table

About this task

Next, define the update process to store data from DRL.STATS_H into DRL.STATS_D. Figure 4-6 shows the DEFINE UPDATE statement you use.

```
DEFINE UPDATE DAY_STATS
  FROM DRL.STATS_H
  TO DRL.STATS_D
  GROUP BY
   (D_DATE = D_DATE)
  SET
   (RD_ERR = SUM(RD_ERR),
    WR_ERR = SUM(WR_ERR),
    TOT_ERR = SUM(TOT_ERR));
```

Figure 4-6. Updating a data table using information from another data table

Notice that in Figure 4-6, the location of the source data (specified by the FROM clause) is the data table DRL.STATS_H.

You specify this GROUP BY clause:

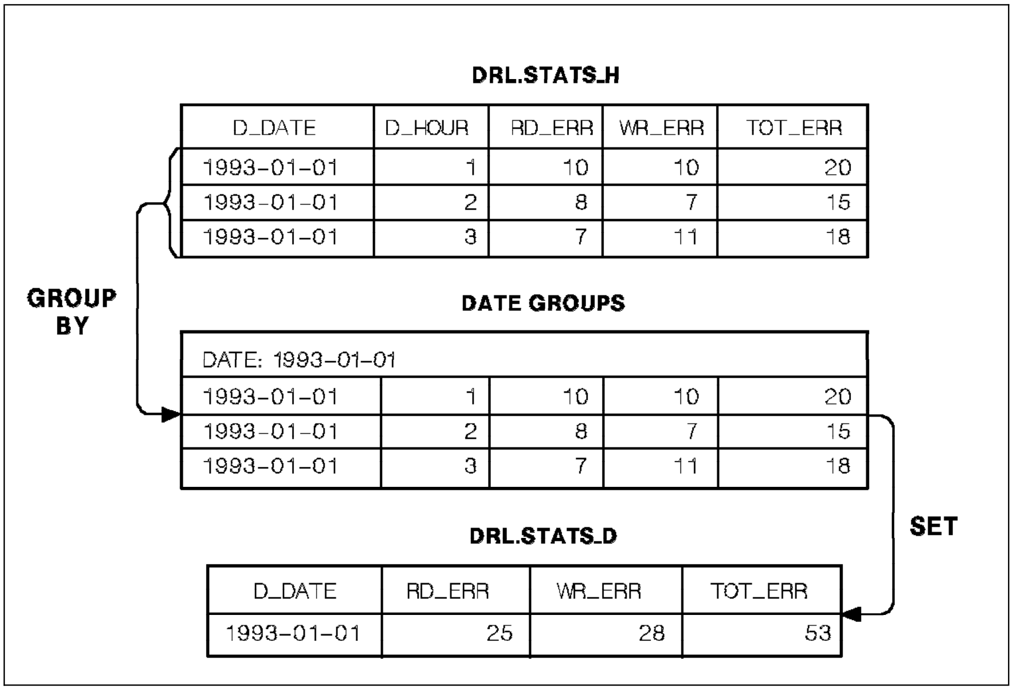
```
GROUP BY
 (D_DATE = D_DATE)
```

Based on this GROUP BY clause, the rows in DRL.STATS_H are grouped by the D_DATE column (see “Understanding the GROUP BY clause” on page 2-4). Each group is stored as a single row in DRL.STATS_D.

Use the SET clause to determine how the data in each group of rows from DRL.STATS_H is processed:

```
SET
 (RD_ERR = SUM(RD_ERR),
  WR_ERR = SUM(WR_ERR),
  TOT_ERR = SUM(TOT_ERR));
```

Figure 4-7 shows how data is stored in DRL.STATS_H, summarized by D_DATE, and stored in DRL.STATS_D. When you perform this cascaded update, the DRL.STATS_D table is updated using only the data entered into the DRL.STATS_H table in the same data collection process. For example, if you start with an empty DRL.STATS_D table and with DRL.STATS_H containing some data, then, after data collection, DRL.STATS_D contains the summary of only the data that you just collected. If you want to include the earlier contents of DRL.STATS_H in the summary, use the RECALCULATE statement as discussed in “Managing data within tables” on page 4-7.



DRL.STATS_H

D_DATE	D_HOUR	RD_ERR	WR_ERR	TOT_ERR
1993-01-01	1	10	10	20
1993-01-01	2	8	7	15
1993-01-01	3	7	11	18

DATE GROUPS

DATE: 1993-01-01				
1993-01-01	1	10	10	20
1993-01-01	2	8	7	15
1993-01-01	3	7	11	18

DRL.STATS_D

D_DATE	RD_ERR	WR_ERR	TOT_ERR
1993-01-01	25	28	53

Figure 4-7. Cascaded update process

When you collect log data, based on the update definitions in Figure 4-3 on page 4-3 and Figure 4-6 on page 4-5, DRLOUT will contain messages like those in Figure 4-8 on page 4-7.

```
  COLLECT RWINFO ;
DRL0300I Collect started at 1993-04-02-02.11.32.
DRL0302I Processing RWINFO.LOG on TSOL02.
DRL0310I A database update started after 12 records due to end of log.
DRL0313I The collect buffer was filled 0 times.
DRL0003I Consider increasing the collect buffer size.
DRL0315I Records read from the log or built by log procedure:
DRL0317I Record name        |    Number
DRL0318I -------------------|----------
DRL0319I TYPA_REC           |         9
DRL0319I TYPB_REC           |         3
DRL0318I -------------------|----------
DRL0321I Total              |        12
DRL0003I
DRL0323I                               -------Buffer------ ------Database-----
DRL0324I Table name                  |   Inserts   Updates   Inserts   Updates
DRL0325I ----------------------------|----------------------------------------
DRL0326I DRL     .STATS_D            |         1         2         1         0
DRL0326I DRL     .STATS_H            |         3         9         3         0
DRL0325I ----------------------------|----------------------------------------
DRL0327I Total                       |         4        11         4         0
DRL0003I
DRL0301I Collect ended at 1993-04-02-02.11.36.
```

Figure 4-8. Messages resulting from data collection for cascaded update

Table 4-2 shows the data stored in DRL.STATS_D after you collect data.

Table 4-2. Contents of DRL.STATS_D after collecting log data

D_DATE	RD_ERR	WR_ERR	TOT_ERR
1993-01-01	25	28	53

Managing data within tables

About this task

You can use log collector statements to manage data contained in data tables. You can also modify a data table and reflect modifications in other tables if they derive data from the first table. For example, you can change data in DRL.STATS_H and reflect the changes in DRL.STATS_D.

You can use log collector statements to:

- Delete data from tables. Use the PURGE statement to delete data from tables. The PURGE statement performs the deletion based on the criteria you specify with the DEFINE PURGE statement.
- Modify data within tables. You can use the RECALCULATE statement to correct invalid data. You can also insert and delete rows using the RECALCULATE statement.

Deleting data

About this task

You can delete data from data tables using the PURGE statement. Use the PURGE statement when you want to regularly delete certain data from the table, such as when you want to delete old data.

To use the PURGE statement, you must first specify the purge conditions using the DEFINE PURGE statement. For example, the table DRL.STATS_H stores the read and write errors by date. Assume that you want to delete all data that is more than 14 days old.

To specify the purge condition, use the DEFINE PURGE statement in Figure 4-9.

```
DEFINE PURGE FROM DRL.STATS_H
    WHERE D_DATE < CURRENT DATE - 14 DAYS;
```

Figure 4-9. Using the DEFINE PURGE statement

The DEFINE PURGE statement indicates the data table for which the purge conditions are effective. The WHERE clause on the DEFINE PURGE statement must be a valid SQL search condition. However, the log collector must be able to recognize its tokens.

To store a purge condition for DRL.STATS_H, use the DEFINE PURGE statement in Figure 4-9. You can then purge data using the PURGE statement in Figure 4-10.

```
PURGE
  EXCLUDE DRL.STATS_D;
```

Figure 4-10. Using the PURGE statement

Executing PURGE will purge data from all tables that have a purge condition.

In Figure 4-10, you specify PURGE, but you also use the EXCLUDE clause to exclude DRL.STATS_D from the purge. To purge data from DRL.STATS_H only, you could also use the PURGE statement in Figure 4-11.

```
PURGE
  INCLUDE DRL.STATS_H;
```

Figure 4-11. Using the PURGE statement

When you specify a table using the INCLUDE clause, only that table is processed for the purge.

Changing data within tables

You can use the RECALCULATE statement to change data that is stored in the data tables, and then update more tables based on the changed information.

Correcting data

Assume that you are verifying the accuracy of the data in DRL.STATS_H, which contains the following values:

Table 4-3. Contents of DRL.STATS_H before the RECALCULATE statement is executed

D_DATE	D_HOUR	RD_ERR	WR_ERR	TOT_ERR
1993-01-01	1	10	10	20
1993-01-01	2	8	7	15
1993-01-01	3	7	11	18

You determine that the number of read errors produced in hour 3 is incorrect. The correct number of read errors should be 4 instead of 7. So, the TOT_ERR column is also incorrect. The total errors produced in hour 3 were 15 instead of 18.

Use the RECALCULATE statement in Figure 4-12 to correct DRL.STATS_H.

```
RECALCULATE DRL.STATS_D
  UPDATE DRL.STATS_H
    SET
     (RD_ERR = 4,
      TOT_ERR = 15)
    WHERE
     (D_HOUR = 3
      AND D_DATE = '1993-01-01');
```

Figure 4-12. Using the RECALCULATE statement

In Figure 4-12, you specify that you want to change the fields RD_ERR and TOT_ERR in the row where D_DATE is 1993-01-01 and D_HOUR is 3. When you execute the RECALCULATE statement, DRL.STATS_H contains the data in Table 4-4.

Table 4-4. Contents of DRL.STATS_H after the RECALCULATE statement is executed

D_DATE	D_HOUR	RD_ERR	WR_ERR	TOT_ERR
1993-01-01	1	10	10	20
1003-01-01	2	8	7	15
1993-01-01	3	4	11	15

In the row where D_DATE is 1993-01-01 and D_HOUR is 3, RD_ERR now contains a value of 4 and TOT_ERR has a value of 15.

In Figure 4-12, you specified `RECALCULATE DRL.STATS_D`. As a result, DRL.STATS_D reflects the changed data.

Deleting and adding rows

About this task

You can delete rows from tables or add more rows to tables using the RECALCULATE statement. For example, assume that you want to delete the number of read errors and write errors for the row that contains an D_HOUR column value of 3 and a D_DATE of 2000-01-01. To do so, use the RECALCULATE statement in Figure 4-13.

```
RECALCULATE
  DELETE FROM DRL.STATS_H
  WHERE
   (D_HOUR = 3
    AND D_DATE = 2001-01-01);
```

Figure 4-13. Deleting a row from a data table

The DRL.STATS_D table is also changed. However, this change never results in rows being deleted from the DRL.STATS_D table. Even if you delete all rows for the date 1993-01-01 from the DRL.STATS_H table, the DRL.STATS_D table still contains a row for that date, with all error counts 0.

You can also insert more rows into a table. For example, assume that you want to add the row you just deleted back into DRL.STATS_H.

Note: When you add a row into a table, you must specify the column names and the data to go into the columns. If you leave out column names, columns (in the order they appear in the table) must be assigned a value.

To insert a row into DRL.STATS_H, use the RECALCULATE statement in Figure 4-14.

```
RECALCULATE
  INSERT INTO DRL.STATS_H
   (D_DATE, D_HOUR, RD_ERR, WR_ERR, TOT_ERR)
    VALUES ('1993-01-01',3,5,7,12);
```

Figure 4-14. Inserting a row into a data table

Chapter 5. Defining update definitions

This chapter describes how to use update definitions to specify more complex processing. You can write update definitions to read data from repeated sections, compute averages and percentiles, and determine resource availability.

You want to store the information in a data table called DRL.CPUTAB that contains these fields:

- JOB_DAY is the day portion of date.
- JOB_CNT is the number of jobs.
- AVE_CPU is the average CPU time per job.

Figure 5-1 shows the update definition used to calculate the average CPU time used per job:

```
DEFINE UPDATE CPU_CAL
  FROM CPU_INFO
  TO DRL.CPUTAB
  GROUP BY
   (JOB_DAY = DAY(DATE))
  SET
   (AVE_CPU = AVG(JOB_CPU, JOB_CNT),
    JOB_CNT = COUNT(JOB_CPU));
```

Figure 5-1. Calculating averages

In Figure 5-1, the records are grouped by the day portion of the DATE field. The average is determined by these lines:

```
SET
 (AVE_CPU = AVG(JOB_CPU, JOB_CNT)
  JOB_CNT = COUNT(JOB_CPU))
```

The AVG function calculates the average value of JOB_CPU for all records in a group. To use the function, you have to specify a column of the target table that will contain the number of JOB_CPU values in the group. Here, you used the column JOB_CNT, with the value specified as COUNT(JOB_CPU).

Table 5-1 shows the results stored in DRL.CPUTAB after data collection:

Table 5-1. Contents of DRL.CPUTAB after data collection

JOB_DAY	JOB_CNT	AVE_CPU
1	5	1.86
2	4	1.33
3	3	1.50
4	5	2.00

Notice that the value of the column specified as the second argument of the AVG function (like JOB_CNT here) must be defined by either the COUNT function or the SUM function. If you specify a column with value defined by COUNT, as in this example, the AVG function computes the ordinary average. If you specify a column with value defined by SUM, the AVG function computes a weighted average. The values specified as the argument of SUM are then used as the weights.

Using repeated sections within records

About this task

Log records often contain sections and, in many such records, sections are repeated. The number of times a section appears in the record can be fixed, or it can be specified by data within the record.

Assume that you want to collect statistics about data sets processed in your subsystem. For each data set processed, the subsystem writes a section in a REP_REC record. The section, called SUBIO, contains the data set name, the number of blocks processed, and the size of the blocks. The number of SUBIO sections in the record is specified by a field called SIO_OCC.

Table 5-2 shows the structure of REP_REC records:

Table 5-2. Structure of a record containing a repeated section

Field name	**Offset**	**Length**	**Data format**	**Description**
REC_TYPE	0	4	Binary	Record type
REC_DATE	4	4	Packed decimal in the format *0cyydddf*	Date the record was written
REC_TIME	8	6	Character string in the format *hhmmss*	Time the record was written
TOT_DSNS	14	4	Binary	Total number of data sets read
SIO_OFF	18	4	Binary	Offset of the first SUBIO section
SIO_LEN	22	4	Binary	Length of each SUBIO section
SIO_OCC	26	4	Binary	Number of SUBIO sections in the record
• • (Other fields within the record) •				
SUBIO section (multiple occurrences of this section exist within a REP_REC record with the number of sections specified by SIO_OCC)				
SIO_DDN	0	8	Character	Data set identifier
SIO_BLK	8	4	Binary	Number of blocks processed in the data set
SIO_BSZ	12	4	Binary	Size of the blocks

Figure 5-2 shows the location of the SUBIO sections within REP_REC.

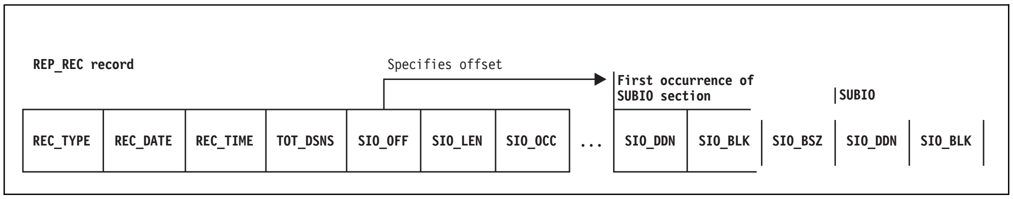


Figure 5-2. A record containing a repeated section

Defining a record with a repeated section

About this task

Figure 5-3 shows how to define a record that contains a repeated section.

```
DEFINE RECORD REP_REC IN LOG SUB_LOG
  IDENTIFIED BY REC_TYPE=5
  FIELDS
   (REC_TYPE   OFFSET 0   LENGTH 4 BINARY,
    REC_DATE   OFFSET 4   LENGTH 4 DATE(CYYMMDDF),
    REC_TIME   OFFSET 8   LENGTH 6 TIME(HHMMSS),
    TOT_DSNS   OFFSET 14  LENGTH 4 BINARY,
    SIO_OFF    OFFSET 18  LENGTH 4 BINARY,
    SIO_LEN    OFFSET 22  LENGTH 4 BINARY,
    SIO_OCC    OFFSET 26  LENGTH 4 BINARY)

  -- Section definition for SUBIO section (repeated)
  SECTION SUBIO
   OFFSET  SIO_OFF
   LENGTH  SIO_LEN
   NUMBER  SIO_OCC
   REPEATED
   FIELDS
    (SIO_DDN OFFSET  0 LENGTH 8  CHAR,
     SIO_BLK OFFSET  8 LENGTH 4 BINARY,
     SIO_BSZ OFFSET 12 LENGTH 4 BINARY)

  -- End of definition for repeated section SUBIO
;
COMMENT ON RECORD REP_REC IS 'A record with a repeated section';
```

Figure 5-3. Defining a record with a repeated section

Specifying a repeated section is similar to specifying a section that is not repeated. To identify the section, use the SECTION clause. The value contained in the SIO_OFF field specifies the offset of the first occurrence of this section. The value in SIO_LEN specifies the length of each occurrence. To specify the fields in the section, use the FIELDS clause in the same way you specify the fields in the record itself.

To identify the section as repeated, use the REPEATED keyword.

Defining updates for records with repeated sections

About this task

You control the processing of repeated sections with the SECTION clause of the DEFINE UPDATE statement. If you do not code the SECTION clause, the log collectorlog collector ignores the repeated section. The records are processed in the usual way, but your definition can only access the stem of the record that consists of all fields outside the repeated sections.

If you code the SECTION clause, the log collectorlog collector generates an internal record for each occurrence of the repeated section. The record contains all data from that occurrence and all data from the record stem. The GROUP BY and SET clauses are applied to these internal records, not to the original records from the log.

Table 5-3 shows an example of REP_REC records:

Table 5-3. Examples of records with repeated section

Record stem							SUBIO section (1)			SUBIO section (2)		
REC_ TYPE	REC_ DATE	REC_ TIME	TOT_ >DSNS	SIO_ OFF	SIO_ LEN	SIO_ OCC	SIO_ DDN	SIO_ BLK	SIO_ SIZE	SIO_ >DDN	SIO_ BLK	SIO_ SIZE
05	0990620F	065311	2	80	16	2	A_DSN	25	4096	B_DSN	75	4096
05	0990620F	092400	1	80	16	1	C_DSN	62	4096			
05	0990621F	010000	2	80	16	2	B_DSN	27	4096	A_DSN	53	4096
05	0990622F	151358	2	80	16	2	A_DSN	92	4096	E_DSN	29	4096

You could not collect the total number of processed data sets by just adding a DSNS column to the table in Table 5-7 on page 5-6 and adding a line `DSNS=SUM(TOT_DSNS)` to the SET clause in Figure 5-5 on page 5-5. The column thus specified would contain the numbers 5, 4, and 4 instead of the correct numbers 3, 2, and 2 obtained in Table 5-5 on page 5-5. This is because data from the stem is repeated in several internal records. To collect both, the number of data sets and the number of processed blocks, you need two update definitions, one with and one without the SECTION clause.

Accessing data from the record stem

About this task

Assume that you want to collect the total number of data sets processed per day. The field TOT_DSNS contains the number of data sets processed. This field is in the record stem, not in the repeated section.

Figure 5-4 shows the DEFINE UPDATE statement used to access data in the record stem.

```
DEFINE UPDATE T_DSNS
  FROM REP_REC
  TO DRL.TOTAL
  GROUP BY
   (DATE=REC_DATE)
  SET
   (DSNS=SUM(TOT_DSNS));
```

Figure 5-4. DEFINE UPDATE statement to access data in the record stem

In Figure 5-4, you specified the processing exactly as for records without repeated sections. When the log collector uses this update definition, it ignores the repeated section. The records of Table 5-3 are processed as if they looked like this:

Table 5-4. The accessible fields when SECTION is not specified

REC_TYPE	REC_DATE	REC_TIME	TOT_DSNS	SIO_OFF	SIO_LEN	SIO_OCC
05	0990620F	065311	2	80	16	2
05	0990620F	092400	1	80	16	1
05	0990621F	010000	2	80	16	2
05	0990622F	151358	2	80	16	2

Notice that the log collector cannot process data from the repeated sections.

In Figure 5-4 on page 5-4, you specified that the records are to be grouped by REC_DATE (specified with the GROUP clause) and that the total data sets processed per day are to be computed (specified with the SET clause).

Table 5-5 shows the results stored in DRL.TOTAL after data collection:.

Table 5-5. Contents of DRL.TOTAL after data collection

DATE	DSNS
1999-06-20	3
1999-06-21	2
1999-06-22	2

Accessing data from repeated sections

About this task

Assume that you want to determine the total number of blocks read each day. This data is stored in the SUBIO sections of each REP_REC record.

Figure 5-5 shows the DEFINE UPDATE statement used to access the data stored in repeated sections of a record.

```
DEFINE UPDATE T_BLKS
  FROM REP_REC SECTION SUBIO
  TO DRL.BLOCK
  GROUP BY
   (DATE=REC_DATE)
  SET
   (BLKS=SUM(SIO_BLK));
```

Figure 5-5. DEFINE UPDATE statement to access data in a repeated section

The SECTION SUBIO clause after the record name in Figure 5-5 states that you want to collect data from the SUBIO sections. When the log collector uses this update definition, it generates one internal record for each occurrence of the SUBIO section. For the REP_REC records of Table 5-3 on page 5-4, these internal records are:

Table 5-6. Internal records generated as a result of specifying SECTION SUBIO

REC_ TYPE	REC_ DATE	REC_ TIME	TOT_ DSNS	SIO_ OFF	SIO_ LEN	SIO_ OCC	SIO_ DDN	SIO_ BLK	SIO_ SIZE
05	0990620F	065311	2	80	16	2	A_DSN	25	4096
05	0990620F	065311	2	80	16	2	B_DSN	75	4096
05	0990620F	092400	1	80	16	1	C_DSN	62	4096
05	0990621F	010000	2	80	16	2	B_DSN	27	4096
05	0990621F	010000	2	80	16	2	A_DSN	53	4096
05	0990622F	151358	2	80	16	2	A_DSN	92	4096
05	0990622F	151358	2	80	16	2	E_DSN	29	4096

Each record contains all data from one occurrence of SUBIO and all data from the record stem. Your GROUP BY and SET clauses are applied to these records. In Figure 5-5, you specified that the records are to be grouped by REC_DATE (specified with the GROUP clause) and that the total blocks processed per day are to be computed (specified with the SET clause).

Table 5-7 on page 5-6 shows the results stored in DRL.BLOCK after data collection.

Table 5-7. Contents of DRL.BLOCK after data collection

DATE	BLKS
1999-06-20	162
1999-06-21	80
1999-06-22	121

Using nested sections within records

About this task

Records often contain nested sections within sections. These sections are called subsections or *nested sections*. Like sections, nested sections can be repeated or non-repeated.

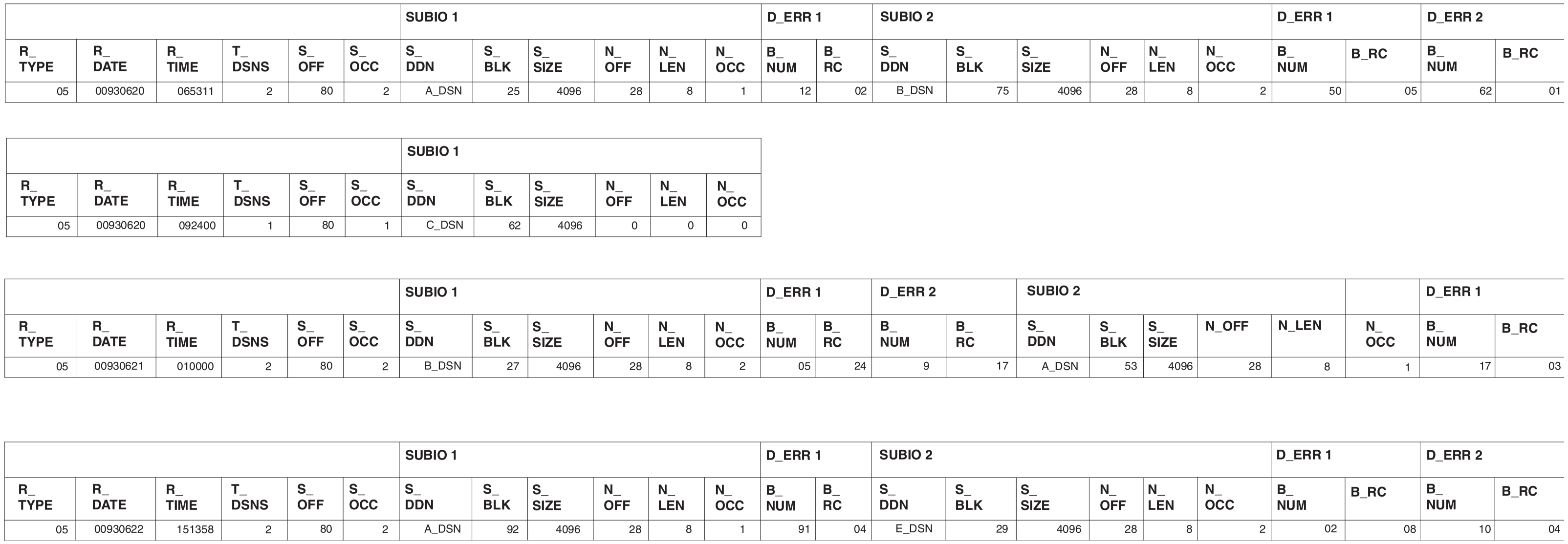
Assume that you have a record similar to REP_REC. This record, called DSERR_REC, contains nested sections within each SUBIO section that describe errors encountered while processing data sets.

Table 5-8 shows the structure of DSERR_REC records:

Table 5-8. Structure of a record containing nested sections

Field name	Offset	Length	Data format	Description
R_TYPE	0	4	Binary	Record type
R_DATE	4	4	Packed decimal in the format 0*cyydddf*	Date the record was written
R_TIME	8	6	Character string in the format *hhmmss*	Time the record was written
T_DSNS	14	4	Binary	Total number of data sets read
S_OFF	18	4	Binary	Offset of the first SUBIO section
S_OCC	22	4	Binary	Number of SUBIO sections that occur in the record
• • (Other fields within the record) •				
SUBIO section (multiple occurrences of this section exist within a DSERR_REC record with the number of sections specified by SIO_OCC)				
S_DDN	0	8	Character	Data set identifier
S_BLK	8	4	Binary	Number of blocks processed in the data set
S_BSZ	12	4	Binary	Size of the blocks
N_OFF	16	4	Binary	Offset of the first D_ERR nested section
N_LEN	20	4	Binary	Length of each D_ERR nested section
N_OCC	24	4	Binary	Number of occurrences of the D_ERR nested section
D_ERR nested section (multiple occurrences of this section exist within a SUBIO section with the number of sections specified by N_OCC)				
B_NUM	0	4	Binary	Number of blocks with error
B_RC	4	4	Binary	Highest return code of the error

The following figure, "Example of records containing nested sections" shows the contents of DSERR_REC records. Note that each occurrence of the SUBIO section has a different length, depending on the number of occurrences of D_ERR section contained in it.



						SUBIO 1						D_ERR 1		SUBIO 2						D_ERR 1		D_ERR 2	
R_ TYPE	R_ DATE	R_ TIME	T_ DSNS	S_ OFF	S_ OCC	S_ DDN	S_ BLK	S_ SIZE	N_ OFF	N_ LEN	N_ OCC	B_ NUM	B_ RC	S_ DDN	S_ BLK	S_ SIZE	N_ OFF	N_ LEN	N_ OCC	B_ NUM	B_RC	B_ NUM	B_RC
05	00930620	065311	2	80	2	A_DSN	25	4096	28	8	1	12	02	B_DSN	75	4096	28	8	2	50	05	62	01

						SUBIO 1					
R_ TYPE	R_ DATE	R_ TIME	T_ DSNS	S_ OFF	S_ OCC	S_ DDN	S_ BLK	S_ SIZE	N_ OFF	N_ LEN	N_ OCC
05	00930620	092400	1	80	1	C_DSN	62	4096	0	0	0

						SUBIO 1						D_ERR 1		D_ERR 2		SUBIO 2						D_ERR 1	
R_ TYPE	R_ DATE	R_ TIME	T_ DSNS	S_ OFF	S_ OCC	S_ DDN	S_ BLK	S_ SIZE	N_ OFF	N_ LEN	N_ OCC	B_ NUM	B_ RC	B_ NUM	B_ RC	S_ DDN	S_ BLK	S_ SIZE	N_OFF	N_LEN	N_ OCC	B_ NUM	B_RC
05	00930621	010000	2	80	2	B_DSN	27	4096	28	8	2	05	24	9	17	A_DSN	53	4096	28	8	1	17	03

						SUBIO 1						D_ERR 1		SUBIO 2						D_ERR 1		D_ERR 2	
R_ TYPE	R_ DATE	R_ TIME	T_ DSNS	S_ OFF	S_ OCC	S_ DDN	S_ BLK	S_ SIZE	N_ OFF	N_ LEN	N_ OCC	B_ NUM	B_ RC	S_ DDN	S_ BLK	S_ SIZE	N_ OFF	N_ LEN	N_ OCC	B_ NUM	B_RC	B_ NUM	B_RC
05	00930622	151358	2	80	2	A_DSN	92	4096	28	8	1	91	04	E_DSN	29	4096	28	8	2	02	08	10	04

Figure 5-6. Example of records containing nested records

Notice that each record contains all data from one occurrence of D_ERR section, all data from the containing occurrence of the SUBIO section, and all data from the record stem. Your GROUP BY and SET clauses are applied to these internal records. Table 5-9 shows the results stored in DRL.PROERR after data collection.

Table 5-9. Contents of DRL.PROERR after data collection

P_DATE	TPRO_ERR
1999-06-20	124
1999-06-21	31
1999-06-22	103

Defining a record with nested sections

About this task

Figure 5-6 on page 5-8 shows how to define the DSERR_REC record shown in Figure 5-7:

```
DEFINE RECORD DSERR_REC IN LOG N_LOG
  IDENTIFIED BY R_TYPE = 05
  FIELDS
   (R_TYPE   OFFSET  0  LENGTH 4 BINARY,
    R_DATE   OFFSET  4  LENGTH 4 DATE(CYYMMDDF),
    R_TIME   OFFSET  8  LENGTH 6 TIME(HHMMSS),
    T_DSNS   OFFSET 14  LENGTH 4 BINARY,
    S_OFF    OFFSET 18  LENGTH 4 BINARY,
    S_OCC    OFFSET 26  LENGTH 4 BINARY)

  -- Section definition for SUBIO section (repeated)
  SECTION SUBIO
    OFFSET  S_OFF
    LENGTH  N_OFF + N_OCC*N_LEN
    NUMBER  S_OCC
    REPEATED
    FIELDS
     (S_DDN OFFSET  0 LENGTH  8 CHAR,
      S_BLK OFFSET  8 LENGTH  4 BINARY,
      S_BSZ OFFSET 12 LENGTH  4 BINARY,
      N_OFF OFFSET 16 LENGTH  4 BINARY,
      N_LEN OFFSET 20 LENGTH  4 BINARY,
      N_OCC OFFSET 24 LENGTH  4 BINARY)

  -- End of definition for repeated section SUBIO

    -- Definition for D_ERR section (nested section)
    SECTION D_ERR
      IN SECTION SUBIO
      OFFSET  N_OFF
      LENGTH  N_LEN
      NUMBER  N_OCC
      REPEATED
      FIELDS
       (B_NUM OFFSET  0 LENGTH 4 BINARY,
        B_RC  OFFSET  4 LENGTH 4 BINARY)

    -- End of definition for nested section D_ERR
;
COMMENT ON RECORD DSERR_REC IS 'Record containing nested section';
```

Figure 5-7. Defining a record with nested sections

You specify a nested section like you specify a section. But notice the IN clause:

```
SECTION D_ERR
  IN SECTION SUBIO
```

The IN clause identifies D_ERR as a nested section, occurring in the SUBIO section. Notice also that the length of each occurrence of SUBIO must be computed from the number of occurrences of D_ERR within it:

```
LENGTH  N_OFF + N_OCC*N_LEN
```

Accessing data in nested sections

About this task

Assume you want to determine the number of blocks that had processing errors. Figure 5-8 shows the DEFINE UPDATE statement you use to gather the data.

```
DEFINE UPDATE E_DSNS
   FROM DSERR_REC SECTION D_ERR
   TO DRL.PROERR
   GROUP BY
    (P_DATE=R_DATE)
   SET
    (TPRO_ERR=SUM(B_NUM));
```

Figure 5-8. DEFINE UPDATE statement to access nested sections in a record

The clause SECTION D_ERR after the record name in Figure 5-8 indicates that you want to collect data from the D_ERR section of DSERR_REC record. When the log collector uses this update definition, it generates one internal record for each occurrence of the D_ERR section. For the DSERR_REC records shown in the previous table “Example of records containing nested sections,” the internal records are:

Table 5-10. Internal records generated for nested repeated section

R_ TYPE	R_ DATE	R_ TIME	T_ DSNS	S_ OFF	S_ OCC	S_ DDN	S_ BLK	S_ SIZE	N_ OFF	N_ LEN	N_ OCC	B_ NUM	B_ RC
05	0990620F	065311	2	80	2	A_DSN	25	4096	28	8	1	12	02
05	0990620F	065311	2	80	2	B_DSN	75	4096	28	8	2	50	05
05	0990620F	065311	2	80	2	B_DSN	75	4096	28	8	2	62	01
05	0990621F	010000	2	80	2	B_DSN	27	4096	28	8	2	05	24
05	0990621F	010000	2	80	2	B_DSN	27	4096	28	8	2	9	17
05	0990621F	010000	2	80	2	A_DSN	53	4096	28	8	1	17	03
05	0990622F	151358	2	80	2	A_DSN	92	4096	28	8	1	91	04
05	0990622F	151358	2	80	2	E_DSN	29	4096	28	8	2	02	08
05	0990622F	151358	2	80	2	E_DSN	29	4096	28	8	2	10	04

Understanding how to access data from records with sections

When you specify processing of a repeated section using the SECTION clause of the DEFINE UPDATE statement, the log collector generates an internal record for each occurrence of that repeated section. Your GROUP BY and SET clauses are then applied to these internal records, rather than records from the log.

Each internal record contains fields from one occurrence of the specified repeated section. It also contains fields from all sections containing that occurrence, from the record stem, and from certain non-repeated subsections.

Figure 5-9 shows a record of type REC with different kinds of sections.

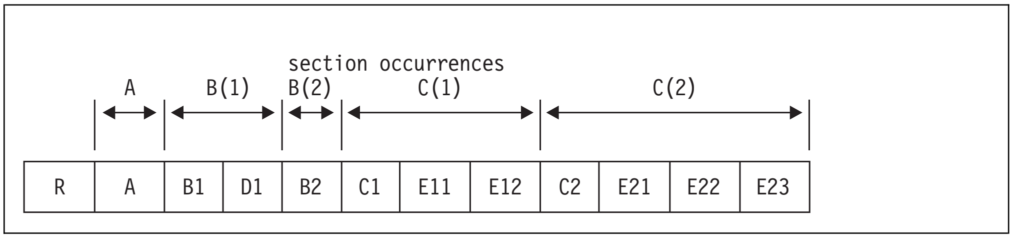


Figure 5-9. Example of a record with different kinds of sections

The record contains these sections:

A A non-repeated section.

B A repeated section that contains a nested non-repeated section (D).

C A repeated section that contains a nested repeated section (E).

D A non-repeated section contained in section B. Section D is present only in the first occurrence of section B.

E A nested repeated section contained in C. Two occurrences of section E are contained in the first occurrence of section C, and three occurrences of section E are contained in the second occurrence of section C.

The different parts of the record in Figure 5-9 are:

R Fields in the record.

A Fields in section A.

B1 Fields in the first occurrence of section B.

B2 Fields in the second occurrence of section B.

C1 Fields in the first occurrence of section C.

C2 Fields in the second occurrence of section C.

D1 Fields in subsection D in the first occurrence of B.

E11 Fields in the first occurrence of E in the first occurrence of C.

.
.
.

E23 Fields in the third occurrence of E in the second occurrence of C.

Figure 5-10 on page 5-12 shows the internal records generated by the log collector, depending on the SECTION clause.

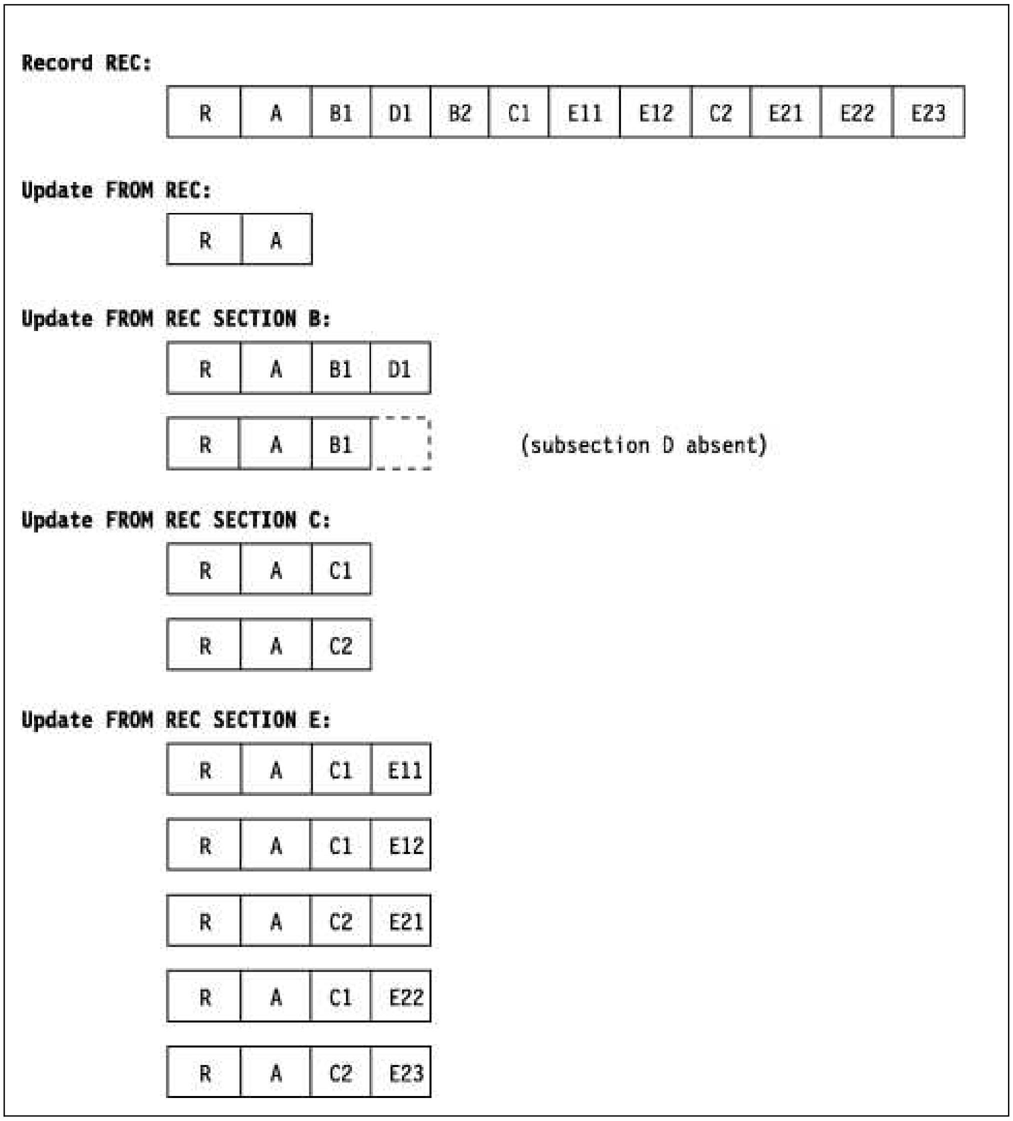


Figure 5-10. Data available for collection, depending on SECTION clause

To find out which fields are included in the generated records, you can use the following method. Represent sections in the record by a tree structure as in Figure 5-11 on page 5-13. The top node represents the record, and the remaining nodes represent the repeated sections. Each node includes its non-repeated subsections. A line such as from C to E shows that E is a subsection of C.

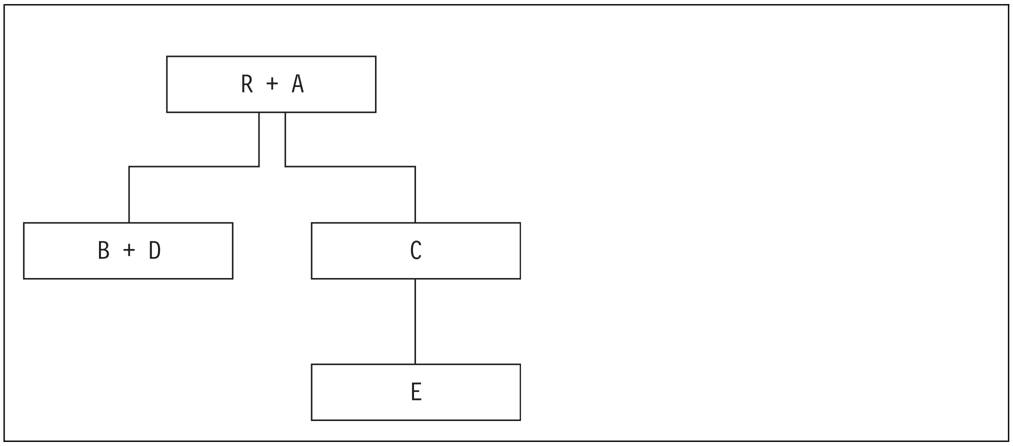


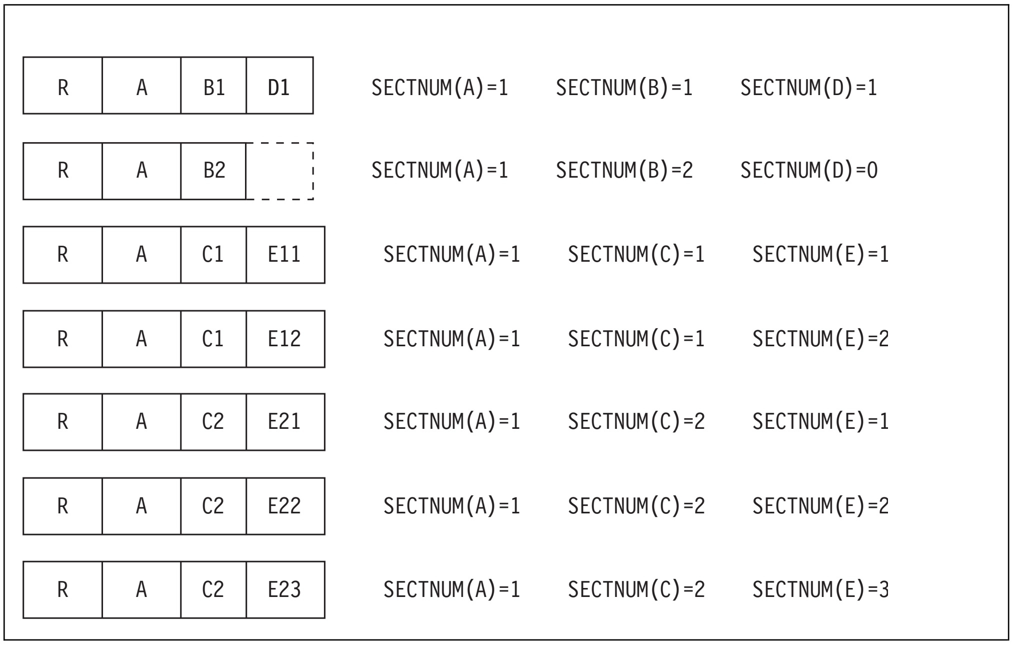
Figure 5-11. Tree structure of a record with repeated sections

Your SECTION clause specifies one of the repeated sections. This repeated section is represented by one of the nodes of the tree. The internal records generated for this section contain all fields from the sections represented by that node and by all nodes on the path leading upwards.

Obtaining a section occurrence number

When processing any of the internal records generated from a repeated section, you can use the SECTNUM function to identify the occurrence of each section included in the record. The function returns the occurrence number of the specified section within its containing section. If you apply the SECTNUM function to a non-repeated section, the result is always either 1 (if the section is present) or 0 (if the section is absent).

As an example, Figure 5-12 shows the result of SECTNUM for different records from Figure 5-10 on page 5-12:



R	A	B1	D1	SECTNUM(A)=1	SECTNUM(B)=1	SECTNUM(D)=1
R	A	B2		SECTNUM(A)=1	SECTNUM(B)=2	SECTNUM(D)=0
R	A	C1	E11	SECTNUM(A)=1	SECTNUM(C)=1	SECTNUM(E)=1
R	A	C1	E12	SECTNUM(A)=1	SECTNUM(C)=1	SECTNUM(E)=2
R	A	C2	E21	SECTNUM(A)=1	SECTNUM(C)=2	SECTNUM(E)=1
R	A	C2	E22	SECTNUM(A)=1	SECTNUM(C)=2	SECTNUM(E)=2
R	A	C2	E23	SECTNUM(A)=1	SECTNUM(C)=2	SECTNUM(E)=3

Figure 5-12. Result of SECTNUM for different internal records

Accessing specific sections in a record

When processing the internal records generated for repeated sections, you can access the fields of the original record using the FIELD function. For example, you can obtain the contents of the field E_FIELD in the occurrence E23 of section E by writing:

```
FIELD(E_FIELD,2,3)
```

Each of the two indexes (2 and 3) refers to one level of nested repeated sections containing the field. The field E_FIELD is contained in two levels of repeated sections, C and E. The index 2 identifies the second occurrence of C, and the index 3 identifies the third occurrence of E (within that second occurrence of C). To specify a field D_FIELD in D1, write:

```
FIELD(D_FIELD,1)
```

Section D is contained in only one level of repeated sections (B). The index 1 identifies the first occurrence of B.

You can also specify an asterisk (*) as an index in the FIELD function. An asterisk specifies the occurrence contained in the currently processed internal record. For example:

```
FIELD(E_FIELD,*,3)
```

means the field E_FIELD in E23 when the log collector processes the record generated for C2. When the log collector processes the record for C1, it means the field in E13, and yields a null value as the result, because E13 does not exist.

Note: The indexes need not be constants as in the examples; they may be any expressions referring to the fields of the record.

The log collector actually generates the internal records for repeated sections by selecting portions of the original record, and not by constructing new records. So, generating these records does not result in a performance penalty.

However, accessing fields with the FIELD function has a definite price in performance, because the log collector recalculates section lengths and offsets each time it executes the FIELD function.

Determining averages

You can use the DEFINE UPDATE statement to determine averages. Assume that you want to determine the average amount of CPU time used per job. The records that contain this information are of type CPU_INFO and contain the data shown in Table 5-11:

Table 5-11. Contents of CPU_INFO records

DATE	JOB_ID	JOB_CPU
00006231F	4546	2.5
00006231F	5367	1.7
00006231F	5893	1.9
00006231F	6192	1.3
00006231F	7338	1.9
00006232F	1600	.8

Table 5-11. Contents of CPU_INFO records (continued)

DATE	JOB_ID	JOB_CPU
00006232F	1775	2.1
00006232F	1990	1.4
00006232F	2222	1.0
00006233F	1752	1.1
00006233F	3193	1.9
00006233F	4000	1.5
00006234F	1655	1.7
00006234F	1883	2.3
00006234F	2122	2.0
00006234F	2775	2.3
00006234F	5721	1.7

Determining percentiles

You can use the DEFINE UPDATE statement to determine percentiles. You might need to know, for example, whether 95% of the transactions for a particular application have a response time of less than 1 second.

Assume that you have these records in a log data set:

DATE	TIME	APPL_ID	TRANS_NO	RES_TIME
0990305F	090901	APP_A	1332	.33
0990305F	111022	APP_A	2110	.95
0990305F	131500	APP_A	2413	.24
0990305F	150020	APP_A	4010	.99
0990305F	164315	APP_A	5121	.75
0990305F	185307	APP_A	6567	.53
0990305F	190000	APP_A	6800	.46
0990305F	211908	APP_A	7548	.39
0990305F	221500	APP_A	8812	.57
0990305F	230000	APP_A	9325	.37
0990305F	231912	APP_A	9794	.39

You want to determine which response time represents the 95th percentile for your transactions. You want to store the resulting data in a data table called DRL.RTIME that has these columns:

- T_DATE is the date of the transactions.
- T_APPL is the application name.
- NUM_RESP is the total number of responses per date.
- RESP_95 is the 95th percentile of response per date.

Figure 5-13 on page 5-16 shows the update definition used to determine the 95th percentile.

```
DEFINE UPDATE DET_PERT
  FROM RESP_DATA
  TO DRL.RTIME
  GROUP BY
   (T_DATE  = DATE,
    T_APPL  = APPL_ID)
SET
   (NUM_RESP = COUNT(RES_TIME),
    RESP_95 = PERCENTILE(RES_TIME,NUM_RESP,95));
```

Figure 5-13. Calculating the 95th percentile

The 95th percentile is determined using these lines:

```
SET
 (NUM_RESP = COUNT(RES_TIME)
  RESP_95 = PERCENTILE(RES_TIME,NUM_RESP,95))
```

The PERCENTILE function returns the response time that is the 95th percentile of all response times within the group. To use the PERCENTILE function, you must specify a column of the target table that will contain the number of RES_TIME values in the group. Here, you used the column NUM_RESP, with the value specified as COUNT(RES_TIME).

Because you are grouping records by date and application, the percentile function uses all response times recorded for a specific application on a given date.

Table 5-12 shows the results stored in DRL.RTIME after data collection.

Table 5-12. Contents of DRL.RTIME

T_DATE	T_APPL	NUM_RESP	RESP_95
1999-03-05	APP_A	11	.97

Note: When using the PERCENTILE function, you should process all input values at the same time. Processing input values during different data collections gives average percentiles of each collect, rather than one overall percentile.

Distributing measurements

Many times, you want to determine statistics over specific periods, such as clock hours. You know the statistics over an interval, which does not coincide with any of these periods, and may contain several periods. You can use the DISTRIBUTE clause of the DEFINE UPDATE statement to distribute the interval statistics evenly over the periods.

For example, assume that you know the amount of CPU time used per job. Table 5-13 shows CPU_IN records that contain this data. Notice that the jobs take several hours and do not start or end at a full hour.

Table 5-13. CPU_IN records containing data to be distributed

APPL	STA_DTE	STA_TME	END_DTE	END_TME	CPU_TME
APP_A	0990705F	163000	0990705F	193000	36.0
APP_A	0990705F	204500	0990705F	234500	18.0

You want to determine the amount of CPU time used each hour (assuming the CPU time is used at an even rate throughout the job). You want to store this data in a table called DRL.DIST that contains these columns:

- DATE is the date of the period.
- HOUR is the hour of the period.
- CPU_TIME is the CPU time used during the period.
- USAGE is the total number of seconds that different jobs were running during the period.

Figure 5-14 shows how to distribute the CPU time contained in CPU_IN records:

```
DEFINE UPDATE DIST_CPU
  FROM CPU_IN
  TO DRL.DIST
  DISTRIBUTE CPU_TIME
    BY 3600
    START TIMESTAMP(STA_DTE,STA_TME)
    END TIMESTAME(END_DTE,END_TME)
    TIMESTAMP CUR_TME
    INTERVAL CUR_DUR
  GROUP BY
   (DATE = DATE(CUR_TIME),
    HOUR = HOUR(CUR_TIME))
  SET
   (CPU_TIME = SUM(CPU_TME),
    USAGE = SUM(CUR_DUR));
```

Figure 5-14. Creating an update definition for measurement distribution

The clause BY 3600 in Figure 5-14 specifies the periods as consecutive one-hour (3600-second) periods starting at midnight. To specify the start and end of the interval that you want to distribute, you use the START and END clauses. In Figure 5-14, you specify the start of the interval using the TIMESTAMP function, which produces a timestamp from the starting date (STA_DTE) and starting time (STA_TIME) fields. You specify the end of the interval using the TIMESTAMP function to produce a timestamp from the ending date (END_DTE) and ending time (END_TME) fields.

When the log collector executes the DISTRIBUTE clause, it first splits each interval at the period boundaries. Then, it generates one internal record for each part resulting from the split.

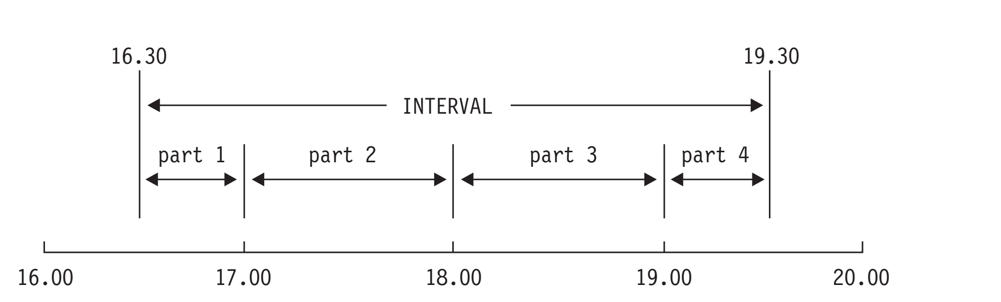


Figure 5-15. Splitting the interval at one-hour boundaries

Figure 5-15 on page 5-17 illustrates the process of splitting the interval from the first record of Table 5-13 on page 5-16. When the log collector processes the records shown in Table 5-13 on page 5-16, it generates these internal records:

```
STA_DTE   STA_TME  END_DTE   END_TME  CPU_TIME    CUR_TIME         CUR_DUR

 100187F  163000    100187F  193000     6.0     2000-07-05-16.30.00  1800
 100187F  163000    100187F  193000    12.0     2000-07-05-17.00.00  3600  First input record
 100187F  163000    100187F  193000    12.0     2000-07-05-18.00.00  3600  produces these records
 100187F  163000    100187F  193000     6.0     2000-07-05-19.00.00  1800


 100187F  204500    100187F  234500     1.5     2000-07-05-20.45.00   900
 100187F  204500    100187F  234500     6.0     2000-07-05-21.00.00  3600  Second input record
 100187F  204500    100187F  234500     6.0     2000-07-05-22.00.00  3600  produces these records
 100187F  204500    100187F  234500     4.5     2000-07-05-23.00.00  2700
```

Each internal record contains the same fields as the original record. The values of fields you have listed after the keyword DISTRIBUTE (CPU_TIME in this case) are distributed proportionally to the length of the part represented by the row. The contents of the remaining fields are copied unchanged from the original record.

Each record also contains two more fields: CUR_TIME and CUR_DUR. They contain, respectively, the start and length of the part represented by the row. You specify the names of these columns using the TIMESTAMP and INTERVAL clauses.

Your GROUP BY and SET clauses are applied to the internal records and the result is used to update the data table.

Table 5-14 shows the results stored in DRL.DIST after data collection.

Table 5-14. Contents of DRL.DIST after data collection

DATE	HOUR	CPU_TIME	USAGE
1999-07-05	16	6.0	1800
1999-07-05	17	12.0	3600
1999-07-05	18	12.0	3600
1999-07-05	19	6.0	1800
1999-07-05	20	1.5	900
1999-07-05	21	6.0	3600
1999-07-05	22	6.0	3600
1999-07-05	23	4.5	2700

Determining resource availability

An important aspect of system performance management is the ability to determine the availability of a particular resource at any given time and the ability to compare that availability with the scheduled availability of the resource.

Typically, information about the availability of a resource comes from several different sources. For example, you can determine that a particular resource is available, or *up*, if that resource is being used by a job running in your system, if the resource is using CPU time, or if it is producing messages from transactions. If applications are trying to use a particular resource and generating error messages showing that they could not use it, the resource is unavailable or *down*.

Using the MERGE clause of the DEFINE UPDATE statement, you can put all this information together to obtain the status of the resource at different times. You may think of this process as reconstructing facts from different pieces of evidence. **This evidence is usually incomplete, or even conflicting, and the log collector must make assumptions that conform best to the collected data.**

Assume that you want to determine the availability of certain resources (such as database servers), using data stored in log files of type RES_DATA. A log of type RES_DATA contains these records:

- Records of type A, which contain data about jobs using a resource.
- Records of type B, which are written when an application attempts to access a resource and receives no answer before the timeout period.
- Records of type C, which are created whenever a resource is started by the system operator.

Records of type A have this layout:

Table 5-15. Layout of Type A records (RES_DATA_A)

Field name	Offset	Length	Data format	Description
REC_TYPE	0	1	Character	Character string A.
RESOURCE	2	8	Character	Resource name.
START_DATE	12	6	Character	Start date of job using the resource, in format *yymmdd*.
START_TIME	20	6	Character	Start time of job using the resource, in format *hhmmss*.
END_DATE	28	6	Character	End date of job using the resource, in format *yymmdd*.
END_TIME	36	6	Character	End time of job using the resource, in format *hhmmss*.

Records of type B have this layout:

Table 5-16. Layout of Type B records (RES_DATA_B)

Field name	Offset	Length	Data format	Description
REC_TYPE	0	1	Character	Character string B.
RESOURCE	2	8	Character	Resource name.
DATE	12	6	Character	Date of attempted access, in format *yymmdd*.
TIME	20	6	Character	Time of attempted access, in format *hhmmss*.
TIMEOUT	28	6	Character	Time, in seconds, that the application waited without obtaining a response.

Records of type C have this layout:

Table 5-17. Layout of Type C records (RES_DATA_C)

Field name	Offset	Length	Data format	Description
REC_TYPE	0	1	Character	Character string C.
RESOURCE	2	8	Character	Resource name.
START_DATE	12	6	Character	Date the resource was started, in format *yymmdd*.
START_TIME	20	6	Character	Time the resource was started, in format *hhmmss*.

A log file containing these records might look like this:

```
record 1:   C DBSERV1   990623  010000
record 2:   A DBSERV1   990623  040000  990623  060000
record 3:   A DBSERV1   990623  050000  990623  070000
record 4:   B DBSERV1   990623  080000    1800
record 5:   C DBSERV1   990623  100000
record 6:   A DBSERV1   990623  130000  990623  140000
record 7:   A DBSERV1   990623  141999  990623  150000
record 8:   C DBSERV1   990623  180000
record 9:   A DBSERV1   990623  181999  990623  190000
```

Figure 5-16. Log file containing RES_DATA records

All records in this log contain data about the same resource, namely DBSERV1, during the same day, namely June 23, 1999. (A log normally contains data about many resources, and may cover more than one day.) Reading the log, you can reason like this to establish the availability of DBSERV1 at different times:

Record number 1
Shows that DBSERV1 was started at 01.00. After this, it must have been up for at least some short time. Nothing is then known about DBSERV1 until 04.00.

Record number 2
Shows that a job was using DBSERV1 from 04.00 through 06.00, and record number 3 indicates that another job was using it from 05.00 through 07.00. These two records together provide evidence that DBSERV1 was up all the time from 04.00 to 07.00.

Record number 4
Shows that at 08.00, an application attempted to use DBSERV1 and did not receive any answer for the subsequent 30 minutes. This provides evidence that the resource was down from 08.00 to 08.30, but you cannot tell exactly when it went down. Record number 5 indicates that the operator restarted DBSERV1 at 10.00, but the interval from 08.30 to 10.00 is too long to conclude that DBSERV1 was down all that time.

Records number 6 and 7
Show that DBSERV1 was up from 13.00 to 14.00 and then from 14.20 to 15.00. Although you do not have any positive evidence for the 20 minute period between 14.00 and 14.20, it seems likely that DBSERV1 was up during that time; you may assume that it was up all the time from 13.00 to 15.00.

Records number 8 and 9
Show that DBSERV1 was restarted at 18.00 and then used from 18.20 through 19.00. Again, you may assume that DBSERV1 was up during the 20 minute interval for which you have no positive evidence.

Figure 5-17 on page 5-21 illustrates the availability of DBSERV1 thus obtained from the log. A double line (`==`) represents the resource being up, and crosses (`XX`) represent the resource being down. A vertical bar (`|`) represents a change of status. Blank spaces represent unknown status.

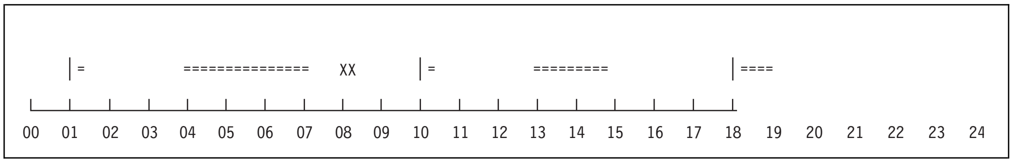


Figure 5-17. Availability of DBSERV1 between 00.00 and 24.00 on June 23, 1999

When the log collector processes a RES_DATA log, it works in essentially the same way. At the end, it stores the resulting availability data in a database table. The table containing these data might look as follows:

Table 5-18. Data table DRLAVAIL_STATUS: an example of availability data

RES_ID	TYPE	INT_START	INT_END	QUIET
DBSERV1		1999-06-23-00.00.00	1999-06-23-01.00.00	0
DBSERV1	|==	1999-06-23-01.00.00	1999-06-23-01.00.01	3600
DBSERV1		1999-06-23-01.00.01	1999-06-23-04.00.00	0
DBSERV1	===	1999-06-23-04.00.00	1999-06-23-07.00.00	1800
DBSERV1		1999-06-23-07.00.00	1999-06-23-08.00.00	0
DBSERV1	XXX	1999-06-23-08.00.00	1999-06-23-08.30.00	0
DBSERV1		1999-06-23-08.30.00	1999-06-23-10.00.00	0
DBSERV1	|==	1999-06-23-10.00.00	1999-06-23-10.00.01	3600
DBSERV1		1999-06-23-10.00.01	1999-06-23-13.00.00	0
DBSERV1	===	1999-06-23-13.00.00	1999-06-23-15.00.00	1800
DBSERV1		1999-06-23-15.00.00	1999-06-23-18.00.00	0
DBSERV1	|==	1999-06-23-18.00.00	1999-06-23-19.00.00	1800
DBSERV1		1999-06-23-19.00.00	1999-06-23-24.00.00	0

Each row of the table corresponds to one interval in the figure illustrating the availability. The columns INT_START and INT_END contain the start and end of an interval, stored as timestamps. (For readability, the timestamps are shown without the microsecond part.) The column TYPE contains a three-character code representing the interval type. This code is similar to the symbols used in the figure. The possible codes are listed in Table 5-19:

Table 5-19. Interval type codes for resource availability

Interval type	Resource status	Resource status at start	Resource status at end
===	Up	Active	Active
|==	Up	Started	Active
==|	Up	Active	Stopped
|=|	Up	Started	Stopped
XXX	Down	Inactive	Inactive
|XX	Down	Stopped	Inactive
XX|	Down	Inactive	Started
|X|	Down	Stopped	Started

The column QUIET contains additional information needed by the log collector to process the evidence that may come at a later time. It is used to bridge the gaps such as between the log records number 6 and 7, or 8 and 9. The number in the column is the maximum length, in seconds, of the gap that can be so bridged. For example, the number 1800 in the fourth row in Table 5-18 on page 5-21 means:

If, in the future, you receive evidence that DBSERV1 was up at any time between 07.00 and 07.00 plus 1800 seconds (that is, between 07.00 and 07.30), you may assume it was up all the time between 07.00 and that instant.

The number in the column QUIET can be seen as the likely length of a “quiet” period when the resource is not used, and therefore cannot provide any evidence of its status.

The table in the example contains availability data for only one resource, but normally contains data for many resources. The column RES_ID then identifies the resource.

To process a RES_DATA log in the way described, and obtain the availability data shown in Table 5-18 on page 5-21, you must provide suitable instructions to the log collector. These instructions have the form of three update definitions, one for each record type. These update definitions might look like this:

```
DEFINE UPDATE AVAIL_A
  FROM RES_DATA_A
  TO DRL.AVAIL_STATUS
  GROUP BY
   (RES_ID = RESOURCE)
  MERGE
   (TYPE      = '===',
    INT_START = TIMESTAMP(START_DATE,START_TIME),
    INT_END   = TIMESTAMP(END_DATE,END_TIME),
    QUIET     = 1800);

DEFINE UPDATE AVAIL_B
  FROM RES_DATA_B
  TO DRL.AVAIL_STATUS
  GROUP BY
   (RES_ID = RESOURCE)
  MERGE
   (TYPE      = 'XXX',
    INT_START = TIMESTAMP(DATE,TIME),
    INT_END   = TIMESTAMP(DATE,TIME) + TIMEOUT SECONDS,
    QUIET     = 0);

DEFINE UPDATE AVAIL_C
  FROM RES_DATA_C
  TO DRL.AVAIL_STATUS
  GROUP BY
   (RES_ID = RESOURCE)
  MERGE
   (TYPE      = '|==',
    INT_START = TIMESTAMP(START_DATE,START_TIME),
    INT_END   = TIMESTAMP(START_DATE,START_TIME) + 1 SECOND,
    QUIET     = 3600);
```

Figure 5-18. Using the MERGE clause

In each of the three updates, you specified grouping of the records by RESOURCE. As a result, the records are grouped so that each group consists of records containing information about the same resource. The MERGE clause is then applied to each such group of records to derive the availability information. Because all three update definitions specify the same target table, the information from all three types of records is combined together before it is stored in the table.

Notice that the MERGE clause generates several rows for each group (for example, all rows in DRL.AVAIL_STATUS are generated from the same group). In this respect, the MERGE clause is different from the SET clause, which summarizes each group in a single row.

Understanding the MERGE clause

The MERGE clause derives from each record an interval similar to those in the DRL.AVAIL_STATUS table. The intervals thus obtained are merged to obtain the final result.

The way the interval is derived from a record depends on the record type. A record of type A is evidence that the resource was up from the start of a job at START_DATE, START_TIME to the end of the job at END_DATE, END_TIME. This is represented as an "up" interval (===) between these two points of time. The MERGE clause for records of type A specifies that interval, for example:

```
MERGE
 (TYPE      = '===',
  INT_START = TIMESTAMP(START_DATE,START_TIME),
  INT_END   = TIMESTAMP(END_DATE,END_TIME),
  QUIET     = 1800);
```

The expressions to the right of the equal signs specify the interval. The first expression specifies the interval type. The second and the third expression specify the start and end of the interval. The fourth expression specifies the quiet period. (The start and end of the interval must be specified as timestamps; the TIMESTAMP function constructs the timestamp from date and time. Table 5-19 on page 5-21 lists the possible interval type codes.)

The names to the left of the equal signs in the MERGE clause are names of columns in the target table that receive the specified values. These columns must be the same in all three update definitions.

A record of type B is evidence that the resource was down for TIMEOUT seconds starting at DATE, TIME. This is represented as a "down" interval (XXX), in this way:

```
MERGE
 (TYPE      = 'XXX',
  INT_START = TIMESTAMP(DATE,TIME),
  INT_END   = TIMESTAMP(DATE,TIME) + TIMEOUT SECONDS,
  QUIET     = 0);
```

A record of type C is evidence that at START_DATE, START_TIME, the resource changed status from "down" to "up", and then was up for some time, perhaps one second, but possibly much more. This is represented as a one-second interval of type |== with a long quiet period:

```
MERGE
 (TYPE      = '|==',
  INT_START = TIMESTAMP(START_DATE,START_TIME),
  INT_END   = TIMESTAMP(START_DATE,START_TIME) + 1 SECOND,
  QUIET     = 3600);
```

Figure 5-19 shows the process of merging the intervals derived from the individual records. A single line (----) in the figure represents quiet periods.

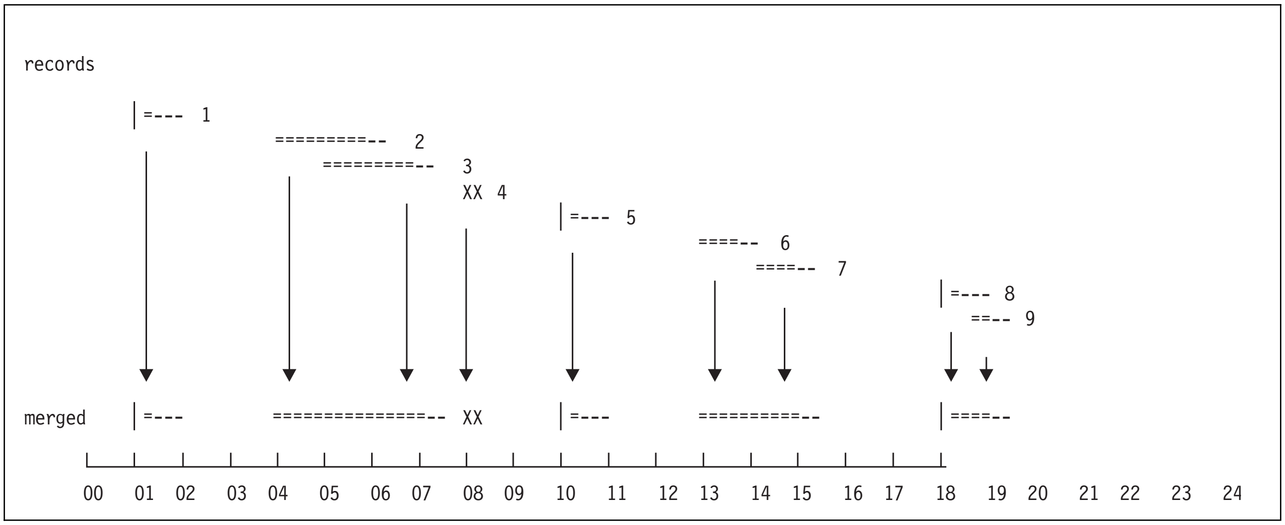


Figure 5-19. Merging of intervals derived from the records

Comparing actual availability to scheduled availability

After determining the actual availability of the resource as thoroughly as possible based on collected data, you can compare that availability with the scheduled availability for the resource.

The scheduled availability is stored in a table named DRLSYS.SCHEDULE. This table is a *control table* and should be set up by your system administrator. (For more information about control tables, refer to the *Administration Guide*.)

The DRLSYS.SCHEDULE table contains *schedules*. A schedule specifies the periods during the day when a resource is scheduled to be available. For example, the table on your system might contain the information shown in Table 5-20.

Table 5-20. Example of a schedule in DRLSYS.SCHEDULE table

SCHEDULE_NAME	DAY_TYPE	START_TIME	END_TIME
STANDARD	MON	08.00.00	11.00.00
STANDARD	MON	12.00.00	17.00.00
• • (Other rows for each day of the week) •			
STANDARD	FRI	08.00.00	17.00.00
STANDARD	SUN	02.00.00	22.00.00
STANDARD	HOLIDAY	02.00.00	22.00.00

You can read in the table that, for example, the schedule named STANDARD requires a resource to be available from 08.00.00 to 11.00.00 and from 12.00.00 to 17.00.00 on Mondays, from 08.00.00 to 17.00.00 on Tuesdays, and so on. (The codes appearing in the DAY_TYPE column are the so-called *day types*. For more information on day types, see the description of the DAYTYPE function.)

Assume that you have a table DRL.AVAIL_STATUS, containing availability data as shown in Table 5-18 on page 5-21. Assume that the schedules at your installation are defined by the above DRLSYS.SCHEDULE table. You want to compute the total number of seconds that each resource was up when it was scheduled to be up. You want to store the result in a data table called DRL.AVAIL_IN_SCHED that has these columns:

Column
Contains

DTE A date.

RESRCE
The name of a resource.

UP_TIME
The number of seconds the resource was up within the schedule on that date.

Assume further that June 23, 1999 is of type FRI (it is a Friday). This figure shows the schedule for June 23, 1999 together with the availability data for that day:

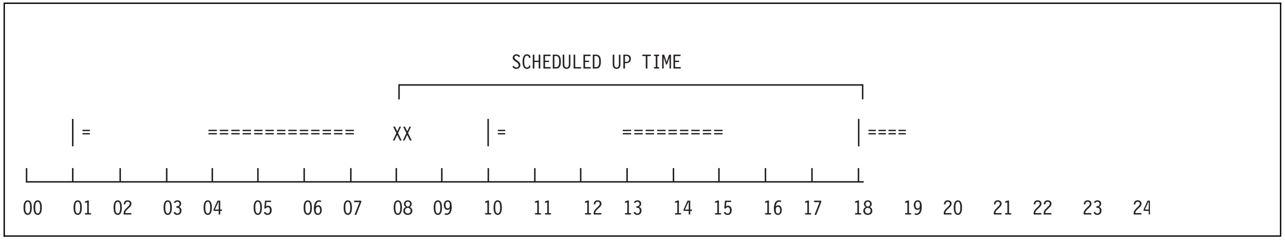


Figure 5-20. Status of the resource and the schedule for June 23, 1999

To summarize the up time within the schedule, use this update definition:

```
DEFINE UPDATE APPLY_SCHEDULE
  FROM DRL.AVAIL_STATUS
  TO   DRL.AVAIL_IN_SCHED
  APPLY SCHEDULE 'STANDARD'
    TO TYPE, INT_START, INT_END
    STATUS SCHED
  GROUP BY
   (DTE    = DATE(INT_START),
    RESRCE = RES_ID)
  SET
   (UP_TIME = SUM(CASE
                  WHEN SUBSTR(TYPE,2,1) = '='
                  AND  SCHED = '='
                  THEN INTERVAL(INT_START,INT_END)
                END));
```

Figure 5-21. Using the APPLY SCHEDULE clause

Understanding the APPLY SCHEDULE clause

When executing the update definition shown above, the log collector first creates a temporary internal table. This table is a copy of the source table, with two modifications:

- The intervals that cross the boundary between the schedule periods are split at these boundaries.
- An additional column indicates if the interval is within the schedule.

You requested creation of the temporary table by means of these lines:

```
APPLY SCHEDULE 'STANDARD'
  TO TYPE, INT_START, INT_END
  STATUS SCHED
```

In the first line, you identified the schedule to be used.

In the second line, you specified where to find the availability data. The three names listed after the keyword TO are the names of columns that contain, respectively, the interval type code, the interval start, and the interval end.

In the last line, you specified the name for the column added to indicate the status, within or outside the schedule. It is the name appearing after the keyword STATUS.

The temporary internal table for the availability data of Table 5-18 on page 5-21 is shown below. Notice that an interval was split at 17.00. An equal sign (=) in the STATUS column indicates an interval within the schedule, and an X indicates an interval outside the schedule.

Table 5-21. Temporary internal table created by APPLY SCHEDULE

RES_ID	TYPE	INT_START	INT_END	QUIET	SCHED
DBSERV1		1999-06-23-00.00.00	1999-06-23-01.00.00	0	X
DBSERV1	|==	1999-06-23-01.00.00	1999-06-23-01.00.01	1800	X
DBSERV1		1999-06-23-01.00.01	1999-06-23-04.00.00	0	X
DBSERV1	===	1999-06-23-04.00.00	1999-06-23-07.00.00	1800	X
DBSERV1		1999-06-23-07.00.00	1999-06-23-08.00.00	0	X
DBSERV1	XXX	1999-06-23-08.00.00	1999-06-23-08.30.00	0	=
DBSERV1		1999-06-23-08.30.00	1999-06-23-10.00.00	0	=
DBSERV1	|==	1999-06-23-10.00.00	1999-06-23-10.00.01	1800	=
DBSERV1		1999-06-23-10.00.01	1999-06-23-13.00.00	0	=
DBSERV1	===	1999-06-23-13.00.00	1999-06-23-15.00.00	1800	=
DBSERV1		1999-06-23-15.00.00	1999-06-23-17.00.00	0	=
DBSERV1		1999-06-23-17.00.00	1999-06-23-18.00.00	0	X
DBSERV1	|==	1999-06-23-18.00.00	1999-06-23-19.00.00	1200	X
DBSERV1		1999-06-23-19.00.00	1999-06-23-24.00.00	0	X

Your GROUP BY and SET clauses are applied to the internal table. You specified grouping by date and resource:

```
GROUP BY
 (DTE    = DATE(INT_START),
  RESRCE = RES_ID)
```

Your SET clause specifies how to compute the total up time within the schedule:

```
SET
 (UP_TIME = SUM(CASE
                 WHEN SUBSTR(TYPE,2,1) = '='
                 AND  SCHED = '='
                 THEN INTERVAL(INT_START,INT_END)
               END));
```

The CASE expression tests whether the middle character in TYPE is an equal sign (meaning an up interval), and whether the interval is within the schedule. If both conditions are true, the result of CASE is the number of seconds between INT_START and INT_END, computed using the INTERVAL function. Otherwise, the result of CASE is a *null value*, meaning no data. The SUM function ignores the null values; its result is thus the total up time within the schedule.

The contents of the table DRL.AVAIL_IN_SCHED after data collection are:

Table 5-22. Contents of the DRL.AVAIL_IN_SCHEDULE table after data collection

DTE	RESRCE	UP_TIME
1999-06-23	DBSERV1	7201

Changing and deleting update definitions

You can modify update definitions after you have stored them using:

- The DROP statement to delete the existing update definition and using the DEFINE UPDATE statement to create a new update definition
- The ALTER UPDATE statement to change an update definition

Using the DROP statement to delete an update definition

About this task

You can use the DROP statement to delete a stored update definition. For example, assume that you wanted to delete a stored update definition, called UPD_DEF. To delete the definition, use this statement:

```
DROP UPD_DEF;
```

You can also use the DROP statement in combination with the DEFINE UPDATE statement to change a stored definition.

Assume that R_REC records contain fields that you did not use when you defined an update using the record. These fields are:

- TOTR_ATT contains the total read attempts.
- TOTW_ATT contains the total write attempts.

You want to collect data from these fields and store it in these columns of DRL.RWSTAT:

- R_ATT is the number of read attempts.
- W_ATT is the number of write attempts.
- TOT_ATT is the total number of read and write attempts.

Figure 5-22 on page 5-28 shows how to use the DROP and DEFINE UPDATE statements to replace the stored update definition.

```
DROP UPDATE TOT_ERRS;

DEFINE UPDATE TOT_ERRS
  FROM R_REC
  TO DRL.RWSTAT
  GROUP BY
   (DATE    = DATE,
    HOUR    = HOUR(TIME))
  SET
   (R_ATT   = SUM(TOTR_ATT)
    RD_ERR  = SUM(R_ERR)
    W_ATT   = SUM(TOTW_ATT)
    WR_ERR  = SUM(W_ERR)
    TOT_ATT = SUM(TOTR_ATT + TOTW_ATT)
    TOT_ERR = SUM(R_ERR + W_ERR));

COMMENT ON UPDATE TOT_ERR IS 'Definition to update DRL.RWSTAT';
```

Figure 5-22. Modifying an update definition using the DROP statement

In Figure 5-22, you specified that the update definition TOT_ERRS should be deleted, using `DROP UPDATE TOT_ERRS`. You can then modify the definition and execute the statement again.

Using the ALTER UPDATE statement

About this task

You can also use the ALTER UPDATE statement to change the update definition TOT_ERRS. However, you should usually use the ALTER UPDATE statement when you must make quick changes. The reason is that when you use the ALTER UPDATE statement, you cannot see the remainder of the original DEFINE UPDATE statement you created.

To store the additional data in columns R_ATT, W_ATT, and TOT_ATT, use the ALTER UPDATE statements in Figure 5-23.

```
ALTER UPDATE TOT_ERRS SET R_ATT  = SUM(TOTR_ATT);
ALTER UPDATE TOT_ERRS SET W_ATT  = SUM(TOTW_ATT);
ALTER UPDATE TOT_ERRS SET TOT_ATT = SUM(TOTR_ATT + TOTW_ATT);
```

Figure 5-23. Using the ALTER UPDATE statement

When you execute these statements, the additional specifications are added to the SET clause of the TOT_ERRS update.

Chapter 6. Collecting log data

This chapter describes how to control the collection process. It also describes how to collect data from logs that have already been processed and how to use the HEADER clause of the DEFINE LOG statement to verify the completeness of a log during collection.

Controlling data collection

About this task

When you collect log data, the log collector processes all updates that are defined for records in the log being collected. For example, if you specify `COLLECT SOME_LOG`, the log collector processes all stored update definitions for that log and collects data based on those definitions.

When you collect log data, you can control much of the collection process. Using the COLLECT statement, you can control:

- Which records in a log are processed
- Which tables are updated
- The size of the collect buffer
- How often records are written to the database and a commit is made
- What happens if an overflow condition occurs during the update of a table row

Limiting the collection to certain records

About this task

Sometimes, you have many record types within a log data set and you want to collect data from only certain records. For example, assume that SOME_LOG has 5 different record types. Each record type has a field called RC_TPE that has a value from 1 to 5 (corresponding to the record type).

Figure 6-1 shows how to use the COLLECT statement to collect data from record type 3.

Note: RC_TPE must be defined using the HEADER clause of the DEFINE LOG statement. For more information about using the HEADER clause, see "Verifying log data sets during data collection" on page 6-4.

```
COLLECT SOME_LOG
WHERE (RC_TPE=3);
```

Figure 6-1. Using the WHERE clause on the COLLECT statement

During data collection, only the update definitions and record description that are stored for record type 3 of SOME_LOG are processed.

Including and excluding data tables

About this task

You can control which data tables are updated during data collection processing using the INCLUDE and EXCLUDE clauses on the COLLECT statement.

In Chapter 4, “Updating, storing, and managing data in tables,” on page 4-1, data was collected, stored in DRL.STATS_H, summarized, and then stored in DRL.STATS_D. However, if you want to collect log data and update DRL.STATS_H, but you do not want to update DRL.STATS_D, use the INCLUDE clause in Figure 6-2.

```
COLLECT RWINFO.LOG
  INCLUDE DRL.STATS_H;
```

Figure 6-2. Using the INCLUDE clauses on the COLLECT statement

When you use the INCLUDE clause, you specify that data will be collected for that table only. No other tables (such as summary tables) are affected by the data collection.

You can also exclude certain tables. For example, if you want to collect data, but you did not want to update DRL.STATS_D, use the COLLECT statement in Figure 6-3.

```
COLLECT RWINFO.LOG
  EXCLUDE DRL.STATS_D;
```

Figure 6-3. Using the EXCLUDE clauses on the COLLECT statement

In Figure 6-3, you specified that all tables associated with RWINFO.LOG should be updated except DRL.STATS_D.

Including or excluding groups of tables

You can use the percent sign (%) to include or exclude groups of tables. For example, assume you stored a number of update definitions for RWINFO.LOG. These definitions update a variety of tables, all beginning with DRL.STATS. But, you do not want to update DRL.STATS_D.

To update all DRL.STATS tables except DRL.STATS_D, use the COLLECT statement in Figure 6-4.

```
COLLECT RWINFO.LOG
  INCLUDE LIKE 'DRL.STAT%'
  EXCLUDE DRL.STATS_D
```

Figure 6-4. Using the percent sign (%)

Controlling when a COMMIT is made

About this task

You can use the COMMIT AFTER clause to determine when a COMMIT is made. A COMMIT makes permanent the changes that occur in the DB2 data tables. You can specify that the log collector write its internal buffer to the database and a COMMIT be made:

- When the buffer is full
- At the end of log data set processing
- After a certain number of records

For example, Figure 6-5 shows how to use the COMMIT AFTER clause to update the database after every 5000 records.

```
COLLECT SOME_LOG
   COMMIT AFTER 5000 RECORDS;
```

Figure 6-5. Using the COMMIT AFTER clause of the COLLECT statement

In Figure 6-5, you specify that the log collector should update the database after every 5000 records.

When you use the COMMIT AFTER clause, ensure that the buffer is large enough to hold the records. You determine the size of the buffer by specifying the BUFFER clause.

Controlling buffer size

About this task

With the BUFFER clause, you can determine the number of bytes used by the log collector's internal buffer. Figure 6-6 shows how to use the BUFFER clause to specify a buffer size of 500 000 bytes.

```
COLLECT SOME_LOG
   BUFFER SIZE 500K;
```

Figure 6-6. Using the BUFFER clause of the COLLECT statement

The minimum size of the buffer is 10 KB, the default is 10 MB. The maximum size of the buffer is limited to the available virtual storage.

Note: The log collector sometimes requires more buffer space than you specify. It will abend if it cannot obtain the extra space.

Handling table row overflows

About this task

You can determine what the log collector collector does if an overflow condition occurs during the update of a table row. An overflow is a situation when the result of an accumulation function exceeds the capacity of the column that should receive it.

You can specify that the log collector either continue processing, re-initializing the value that had the overflow, or stop processing.

Collecting data more than once

About this task

Sometimes, you might need to collect data again after you have already updated data tables. The log collector keeps track of the processed logs to prevent collecting the same data twice. If you try to collect data on the same log more than once, the log collector issues a message saying that the log has already been processed. However, if you must collect data from a log again anyway, use the REPROCESS keyword on the COLLECT statement shown in Figure 6-7:

```
COLLECT SOME_LOG
  REPROCESS;
```

Figure 6-7. Using the REPROCESS keyword

As a result, the log collector collects data from the log again.

Note: When you use the REPROCESS keyword, data that is already stored in data tables is not replaced. Instead, new data is added to the existing data. So, if you attempt to use REPROCESS and you want to start over (store data in an empty data table), you must first create a PURGE definition as discussed in “Deleting data” on page 4-7. Then, you can reprocess a log and store the data in the data table.

Collecting data from partially processed logs

About this task

Occasionally, a system failure or other condition may cause the collection process to halt prematurely. If the log has not been completely processed, you can restart the data collection by issuing the COLLECT statement again. The log collector will begin processing data again, starting after the last completely processed record.

Verifying log data sets during data collection

You can use the DEFINE LOG statement to provide more than the name of the log you are defining. You can also use the DEFINE LOG statement to verify that the log is complete and to ensure that the log collector does not process the log more than once.

Assume that you want to define a log (called SUB_LOG). Although the log contains many different record types, these fields are common to all records in SUB_LOG:

Table 6-1. Fields that are common to all records in SUB_LOG

Field name	Offset	Length	Data format	Description
R_LEN	0	2	Binary	Length of the record
R_TYPE	2	2	Character	Record type
R_TIME	4	6	Hexadecimal	Time the record was written
R_DATE	10	4	Hexadecimal	Date the record was written
S_ID	14	4	Character	System identifier
P_ID	18	8	Character	Program identifier

The log always contains a record of type 2 (R_TYPE = 2) as the first record in the log and a record of type 3 (R_TYPE = 3) as the last record.

Figure 6-8 shows how to use the DEFINE LOG statement to define this log.

```
DEFINE LOG SUB_LOG
  HEADER
   (R_LEN    OFFSET 0  LENGTH 2 BINARY,
    R_TYPE   OFFSET 2  LENGTH 1 BINARY,
    R_TIME   OFFSET 3  LENGTH 6 TIME(HHMMSS),
    R_DATE   OFFSET 9  LENGTH 4 DATE(0CYYDDDF),
    S_ID     OFFSET 13          CHAR(4),
    P_ID     OFFSET 17          CHAR(8))
  TIMESTAMP TIMESTAMP(R_DATE,R_TIME)
  FIRST RECORD R_TYPE=2
  LAST RECORD  R_TYPE=3;
COMMENT ON LOG SUB_LOG IS 'Log definition for SUB_LOG';
```

Figure 6-8. Using the HEADER, TIMESTAMP, FIRST RECORD, and LAST RECORD clauses of the DEFINE LOG statement

The HEADER clause identifies the fields that are common to all records in SUB_LOG. You have to identify only the fields you need. Here, you must identify R_DATE, R_TIME, and R_TYPE because you use them in the TIMESTAMP, FIRST RECORD, and LAST RECORD clauses of the DEFINE LOG statement.

Note: The header field definitions apply only to the log definition itself. If you plan to use these fields in the data collection process, you must define them again using the DEFINE RECORD statement.

The header fields are defined in the same way that the record fields were defined in “Defining a record” on page 2-3.

When you collect log data, the first record and last record conditions are checked to determine whether this is a complete and valid log data set. If not, an warning message is issued to the data set you specified on the DRLOUT JCL statement.

If you specify the TIMESTAMP clause, the log collector stores the timestamps of the first and last records. It also writes these timestamps to the DRLOUT file. Using the TIMESTAMP clause, you know the time period covered by the log data set. You can also ensure that the log collector only processes the log once (unless you specify the REPROCESS clause).

Part 2. Log collector language reference

Chapter 7. How to read the syntax diagrams

The syntax diagrams in Chapter 8, “Elements of the log collector language,” on page 8-1 through Chapter 11, “Log collector language statements,” on page 11-1 graphically illustrate the coding options available for the log collector functions and statements. These diagrams give you a quick visual method for determining whether:

- An element is a required, optional, or default element.
- A word or value is repeatable.
- An element is a constant or a variable.

The lines and arrows in a diagram symbolize the way these elements are combined to form a valid function or statement.

Note: The syntax diagrams show the sequence of tokens, not of individual characters. For more information, see “How your text is processed” on page 8-5.

Syntax is described using these conventions:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►─── symbol indicates the beginning of a statement.

  The ───► symbol indicates that the statement syntax is continued on the next line.

  The ►─── symbol indicates that a statement is continued from the previous line.

  The ───►◄ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ►─── symbol and end with the ───► symbol.
- Required items appear on the horizontal line (the main path).

```
►►──required_item──required_item──────────────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──required_item──┬───────────────┬──────────────────────────────────────►◄
                   └─optional_item─┘
```

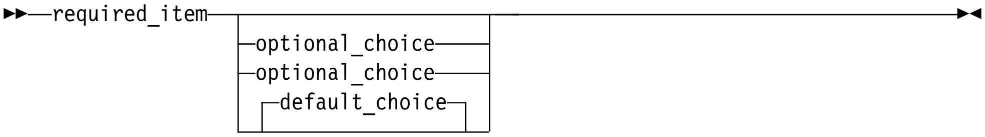
- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.

```
►►──required_item──┬─required_choice1─┬───────────────────────────────────►◄
                   └─required_choice2─┘
```

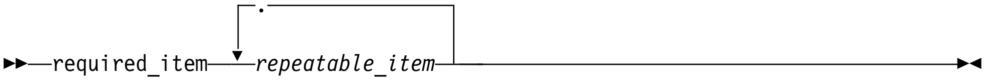
  If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──required_item──┬──────────────────┬───────────────────────────────────►◄
                   ├─optional_choice1─┤
                   └─optional_choice2─┘
```

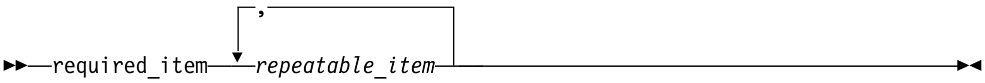
If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.

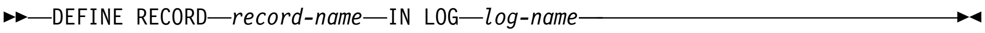


If the repeat arrow contains a comma, you must separate repeated items with a comma.



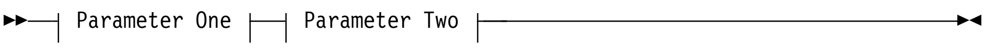
A repeat arrow above a stack indicates that you can repeat the items in a stack.

- Keywords appear in uppercase (for example, DEFINE RECORD). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *record-name*). They represent user-supplied names or values.

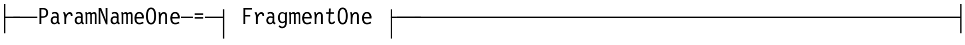


- A *fragment* contains a large element or group of elements that appear more than once in the syntax diagram. The fragment appears at the end of the diagram, before any syntax notes.

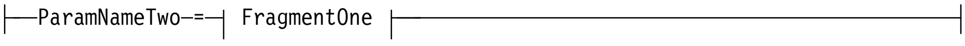
  All fragment references inside the diagram are enclosed by vertical bars. The text in a fragment reference matches the title of the fragment it references:



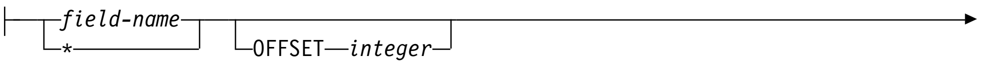
Parameter One



Parameter Two



FragmentOne:



```
►──┬──────────────────────────────────────────┬──────────────────────────────────┤
   └─LENGTH──┬─integer─┬──┬──────────────┬─┘
             └─*───────┘  └─field-format─┘
```

- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Chapter 8. Elements of the log collector language

This chapter introduces basic elements common to all statements: keywords, identifiers, constants, and delimiters. It explains how to enter text in a data set, and how to use blanks and comments to format and annotate the text.

The chapter also introduces the naming convention for DB2 tables and describes how you can parameterize your definitions using variables.

Characters

The basic symbols of the log collector language are single-byte EBCDIC characters. Within some language elements, you can also enter sequences of double-byte characters. Each such sequence must be enclosed between (single-byte) shift-out and shift-in characters. Unless otherwise stated, all characters named below are single-byte EBCDIC characters.

A letter is one of the characters A through Z and a through z, or any of the three alphabetic extenders for national languages. (The three alphabetic extenders are X'5B', X'7B' and X'7C'. Using code pages 37 and 500, they display as $, #, and @, respectively.)

A digit is any of the characters 0 through 9.

Tokens

The smallest building blocks of the language are tokens. The syntax diagrams show the sequence of tokens, rather than individual characters. The tokens are of six kinds:
- Words
- Delimited words
- String constants
- Integer constants
- Floating-point constants
- Delimiters

Words

A *word* is a letter followed by zero or more characters, each of which is a letter, a digit, or the underscore character (_).

Examples

```
A   ABC   Length   Z_121
```

Before processing, all lowercase letters in a word are translated to uppercase. This means, for example, that `length`, `Length`, and `LENgth` are all interpreted as `LENGTH`.

The words are used as language keywords and as identifiers. The keywords must be as specified in the syntax diagrams. The identifiers are names you give things you work with. See “Identifiers” on page 8-6 for more information about identifiers.

Delimited words

A delimited word is a sequence of one or more characters enclosed within quotation marks ("). The sequence may contain any characters, but quotation marks may only appear in pairs (""). Any unpaired quotation mark is interpreted as the end of the token.

Examples

```
"A 2"   "JobClass ""A"""
```

Before processing, the enclosing quotation marks are removed, and each pair of quotation marks within the token is replaced by a single quotation mark. The two examples become `A 2` and `JobClass "A"`, respectively. Notice that each contains a blank in the middle.

The delimited word is afterwards treated the same as a word. Thus, delimited words let you code words that contain characters other than uppercase letters, digits, and underscore.

A delimited word may contain sequences of double-byte characters enclosed between shift-out and shift-in characters. The quotation marks are single-byte characters; they are recognized only outside a double-byte sequence.

The delimited words are used as identifiers. See “Identifiers” on page 8-6 for more information about identifiers.

String constants

A string constant is a sequence of zero or more characters enclosed within apostrophes ('). The sequence can contain any characters, but apostrophes may only appear in pairs ('). Any unpaired apostrophe is interpreted as the end of the token.

Examples

```
'A 2'   'a:b'   'JobClass 'A''   ''
```

A string constant represents the character string obtained by removing the enclosing apostrophes, and replacing each pair of apostrophes within the string by a single apostrophe. The first three constants in the example thus represent the strings `A 2`, `a:b`, and `JobClass 'A'`, respectively. Notice that the first and the third contain a blank in the middle. The last example represents the *empty string*: a sequence of zero characters.

A string constant may contain sequences of double-byte characters, each enclosed between shift-out and shift-in characters. The apostrophes are single-byte characters and will only be recognized outside a double-byte sequence.

The maximum length of a string represented by a string constant is 254 bytes. This includes any shift characters enclosing sequences of double-byte characters.

Integer constants

An integer constant is a sequence of one or more digits.

Examples

```
62   100   32767   720176   0000000015
```

An integer constant represents a whole number in decimal notation. The number must not exceed 2 147 483 647. The maximum length of the token is 32 characters.

Floating-point constants

A floating-point constant is a sequence of one or more digits followed by a decimal point and zero or more digits, optionally followed by an E and a signed or unsigned number of at most two digits.

Examples

```
25.5   1000.   0.0   37589.33333   15E1   2.5E5   2.2E-1   5.E+22
```

A floating-point constant represents a 64-bit floating-point number of System/390® architecture. The number is represented in decimal notation, with E*nn* meaning *multiplied by 10 to power nn*. For example, `2.5E5` means 2.5×10^5, and `2.2E-1` means 2.2×10^{-1}. The specified value is rounded to the closest value that can be represented as a 64-bit floating-point number.

The number must not exceed 16^{63}-16^{49}, which is approximately 7.2E75. The smallest possible value different from 0 is 16^{-65}, which is approximately 5.4E-79.

The maximum length of the token is 32 characters.

Delimiters

A delimiter is any of these characters or character pairs:

```
(   )   ,   .   ;   :   +   -   /   *   =   <   >   <>   >=   <=   ||
```

Input lines

You enter the log collector language text in a sequential data set. The data set may have either fixed-length records (record format F or FB) or varying-length records (record format V or VB). You can use only positions 1 through 72 of each record. The log collector ignores anything beyond position 72. (In a fixed-length record, position number 1 is the first byte of the record. In a varying-length record, position number 1 is the fifth byte of the record, that is, one immediately after the record descriptor word.)

An input line is the contents of one record, starting with position 1, and ending either at position 72, or at the end of record, whatever comes first. If you use a file with 80-byte fixed-length records, this corresponds exactly to a line displayed by the ISPF editor. If you use a file with varying-length records, the actual input line may be shorter than the line shown on the screen. Notice that the line then ends at the last non-blank character, which is important if you want to code a token extending over several input lines. When the log collector processes your text, it acts as if all input lines were concatenated into one long string. The only case where a line break influences processing is when it terminates a line comment. Otherwise, line breaks are ignored.

Example

Suppose you entered these lines on the editor screen (both starting at position 1):

```
'ABC
DEF'
```

If you use fixed-length 80-byte records, the log collector interprets this as 'ABC followed by 68 blanks followed by DEF'. If you use varying-length records, the log collector interprets this as 'ABCDEF'.

Blanks

Some tokens cannot be entered immediately one after another, because together, they form another token. For example, you cannot enter the word LENGTH immediately followed by the integer constant 2, because the log collector would treat them as one word LENGTH2. To separate these tokens, enter one or more blanks between them.

Apart from separating tokens, all blanks between tokens are ignored by the log collector.

Blanks between tokens are allowed, and ignored, even if they are not needed to separate the tokens. For example, A + B is correct and equivalent to A+B. You normally use these blanks to arrange the text so that it is readable.

Blanks within tokens are significant and not ignored.

Comments

To make your text readable, you can include explanations that are ignored by the log collector. You can enter these explanations, or comments, anywhere between the tokens. Any sequence of comments and blanks between the tokens is allowed and ignored. Blanks and comments are also ignored before the first and after the last token. If you wish, you can use a comment instead of blanks to separate tokens. The comments are of two kinds:

- Line comments
- Block comments

Line comments

A line comment is any sequence of characters starting with a double minus sign (--) up to the end of the current input line.

Examples

```
-- This is a line comment.
-- Another line comment. Notice that it may contain unpaired ' and " .
```

The comment may contain sequences of double-byte characters enclosed between shift-out and shift-in characters. The line break that terminates the comment must occur within a single-byte sequence. If the line ends in a double-byte sequence, the next line will be interpreted as starting in the single-byte mode, usually resulting in an error.

Block comments

A block comment is any sequence of characters starting with slash asterisk (/*) up to the nearest following asterisk slash (*/).

Example

```
/* This is a block comment.
   Notice that it can extend over several lines.
   It can contain -- and unpaired ' or "  */
```

The comment can contain sequences of double-byte characters enclosed between shift-out and shift-in characters. The asterisk and slash that terminate the comment are single-byte characters and will only be recognized outside a double-byte sequence.

How your text is processed

Before processing your text, the log collector converts it into a sequence of tokens. Understanding how this is done may help you write the text and interpret messages about syntax errors. The process is illustrated in this example:

Example

These two lines constitute a fragment of a log collector statement:

```
/* Summarize the data */
Group By (Job="Job Name"||'01')   -- By job name with 01 appended
```

This fragment is first split into tokens, blanks, and comments:

```
blanks:
block comment:            /* Summarize the data */
blanks:
word:                     Group
blanks:
word:                     By
blanks:
delimiter:                (
word:                     Job
delimiter:                =
delimited word:           "Job Name"
delimiter:                ||
string constant:          '01'
delimiter:                )
blanks:
line comment:             -- By job name with 01 appended
```

Then, the blanks and comments are discarded, and words and delimited words are transformed as described under “Words” on page 8-1 and “Delimited words” on page 8-2. The result is the following sequence of tokens:

```
word:                     GROUP
word:                     BY
delimiter:                (
word:                     JOB
delimiter:                =
word:                     Job Name
delimiter:                ||
string constant:          '01'
delimiter:                )
```

This sequence of tokens constitutes the proper input to the log collector, and is specified by syntax diagrams. The syntax diagram specifying this particular fragment of a statement is:

```
                  ┌─,──────────────────────────┐
                  ▼                            │
►►──GROUP BY──(───column-name = expression─────┴──)──────────────────────────►◄
```

By comparing the sequence of tokens with this syntax, the log collector recognizes GROUP and BY as keywords, and JOB as an identifier (a column name). By similarly using the syntax for expression, it recognizes Job Name as an identifier. Notice that

Job Name is treated as one word, with a blank in its middle. Notice also that it would make no difference if you coded "GROUP" (with quotation marks) instead of Group.

The log collector always processes the text in one direction. When it comes to a point where it cannot interpret the next portion of the text as a token, blank, or comment, it does not return to try an alternative interpretation, but signals an invalid character, skips the character, and proceeds to identify the next token, blank, or comment. When the log collector finds that a token does not match the syntax, it does not return to try alternatives, but signals an unexpected token. In each case, after signalling the error, the log collector skips the rest of current statement, that is, all tokens up to, and including, the nearest semicolon token.

Identifiers

The words and delimited words are mainly used as names of things you work with: logs, records, fields within records, and so on. When used in this way, they are called identifiers.

You select the identifiers. By coding an identifier as a delimited word, you can use any sequence of characters as the name, including blanks and double-byte characters. In general, the only restriction on your choice of an identifier is that it must not exceed 18 bytes. The restrictions on identifiers used for specific purposes are:

- The name of a log cannot exceed 16 bytes.
- The name of a record cannot have the asterisk (*) as both the first and the last character.
- The name of a table column must consist of uppercase letters, digits, and underscore characters. It must start with a letter and must be distinct from all SQL reserved words.
- The name of a file (a ddname) and the name of a program cannot exceed 8 bytes. It must consist of uppercase letters and digits, and must start with a letter.

All lengths include any shift characters enclosing the sequences of double-byte characters.

In some contexts, an identifier may be confused with a keyword. For this reason, do not use the words CASE, CURRENT, LOOKUP, NOT, and USER as names of anything you might reference within an expression; in particular, as names of fields in a record.

Table names

The DB2 tables have names that consist of two identifiers separated by a period (.). Both identifiers must consist of uppercase letters, digits, and underscore characters, and each must start with a letter. They must be distinct from all SQL reserved words. The first identifier cannot exceed 8 bytes.

Example

```
ABC.DAYSTAT_1
```

The first identifier (ABC in this case) is called the prefix, and the table name written in this form is called a qualified table name. You can also code a single identifier as a table name. Such an unqualified table name is implicitly prefixed with the user ID of the user who runs the log collector. Thus, for example, if your user ID is ABC, then specifying DAYSTAT_1 as a table name is equivalent to specifying ABC.DAYSTAT_1.

If not stated otherwise, a table name anywhere in this manual means a qualified or unqualified table name.

Statements

The input in the log collector language is a sequence of statements. The statements must be separated by semicolons (;). The semicolons are not considered a part of the statement and are not shown in syntax diagrams.

Using variables to modify your text

Sometimes it is convenient to leave certain details open, and fill them in at the last moment. Suppose that you are writing definitions for some installation, and you do not know the prefix used for table names at that installation. You can then code everything, except for the prefix. Instead of the prefix, you can write a marker that will be replaced by the prefix at a later time. This marker, called a variable marker, has the form of an ampersand (&) immediately followed by a word. For example, you can write:

```
DEFINE UPDATE XYZ FROM &PREFIX.JOBSTAT_D TO ...
```

The variable marker is in this case &PREFIX. One way of specifying the actual prefix is by executing a log collector statement such as:

```
SET PREFIX='ABC';
```

As explained in “SET” on page 11-53, this statement defines a *variable*PREFIX with *value*ABC. From that moment on, whenever the log collector encounters the marker &PREFIX, it logically replaces the marker by the value of the variable PREFIX. Thus, your statement will be processed as if you have coded:

```
DEFINE UPDATE XYZ FROM ABC.JOBSTAT_D TO ..
```

You may code the variable marker at any place you would code a token. To properly terminate it, you might have to follow it with a blank or a comment. Any lowercase letters in the variable marker are translated to uppercase before the marker is used to identify the variable. The replacement string can consist of any number of tokens, blanks, and/or comments.

An alternative way to define a variable is to code &PREFIX=ABC as a part of the parameter string supplied to the log collector at the invocation. See Appendix B, “JCL for the log collector language and report definition language,” on page B-1 for details.

If you use variable markers, fill in the details when the log collector processes your text. If your text is, for example an update definition, this means all details are fixed before the definition is stored. Another method, described in “Obtaining the value of a variable” on page 9-4, lets you postpone this until the moment when the stored definition is actually used.

Chapter 9. Values and expressions

This chapter discusses the data items, or values, processed by the log collector. It introduces the notion of a data type and null value, and describes how values are specified by means of expressions.

Expressions are important tools for defining what to do with the data, and are used in almost all statements of the log collector language. You will in most cases use rather simple expressions, consisting of a single identifier or a constant, or perhaps two such items and an operator. However, you can use the constructions described here to specify calculations of almost any complexity. The only limitation is the size of the expression, which cannot exceed 2 000 bytes. The size of the internal representation of the expression is also limited, which sometimes might limit the external form to less than 2 000 bytes. There is also a limit on the nesting depth of certain structures (such as parentheses), but you are unlikely to reach it.

An expression that specifies a truth value (that is, one of the values true and false) is called a condition.

This chapter does not cover everything about expressions. The special case of an expression called a function call is discussed in Chapter 10, "Functions," on page 10-1.

Data types

The main task of the log collector is to process data. The smallest unit of data processed by the log collector is called a value. A value can be obtained, for example, from a field in a record, from a DB2 table, stated explicitly in your definition, or computed from other values. The values handled by the log collector are:

- Integers, such as 5, 0, or -127
- Floating-point numbers, such as 3.333 or 1.5×10^{7}
- Character strings, such as `xy%(?z` or `ABC`
- Dates, such as April 15, 2000
- Times, such as 16 hours 32 minutes 55.123456 seconds
- Timestamps, such as 16 hours 32 minutes 55.123456 seconds on April 15, 2000
- Truth values, such as true or false

These seven kinds are called data types. For convenience, the integers and floating-point numbers are together called numbers, or numeric values. The dates, times, and timestamps are together called date/time values.

Integers

An *integer* is a whole number in the range -2147483648 to +2147483647.

Floating-point numbers

A floating-point number is a number that can be represented as a normalized 64-bit floating-point number of System/390 architecture. The numbers that can be represented in this way have absolute values up to 16^{63}-16^{49}, which is approximately 7.2×10^{75}. The smallest absolute value different from 0 that can be represented is 16^{-65} (approximately 5.4×10^{-79}).

Character strings

A character string, or string, is a sequence of 0 to 254 bytes. It can contain sequences of double-byte characters. Each sequence of double-byte characters must begin with a shift-out character, and end with a shift-in character. The length of the string is the number of bytes in the sequence. The string of length 0 is called the empty string.

Dates

A date is a three-part value (day, month, year) designating a day according to the Gregorian calendar. The range of the year part is 1 to 9 999. The range of the month part is 1 to 12. The range of the day part is 1 to *x*, where *x* depends on the month.

All computations on dates are performed as if the Gregorian calendar was in effect since year 0001.

Times

A time is a four-part value (hour, minute, second, and microsecond) designating a time of day under a 24-hour clock. The range of the hour part is 0 to 24, the range of the minute and second part is 0 to 59, and the range of the microsecond part is 0 to 999 999. If the hour is 24, the remaining parts must be 0.

Timestamps

A timestamp is a seven-part value (year, month, day, hour, minute, second, and microsecond) that designates a point in time: a day and a time of that day. The year, month, and day designate the day as specified under "Dates." The hour, minute, second, and microsecond designate the time as specified under "Times."

Truth values

A truth value is one of the values true or false.

Missing and invalid data

In some situations, the value specified by your definition cannot be obtained. For example, you specify a value from a field in a record, but the record is too short to contain the field. Alternatively, the record section that should contain the field is absent, or the field is present, but contains non-valid data.

It also happens that the value has been obtained from a field, but you specified an operation on it that cannot be performed. For example, you specified that you want the result of dividing the value from a field A by the value from a field B. If this second value happens to be 0, the requested result cannot be obtained; you cannot divide by 0.

Null value

For the purpose of specifying what to do with data, it is convenient to treat an absence of data as just another value, distinct from all other values handled by the log collector. This special value is called the null value.

The null value is useful because whenever you specify that a value should be obtained in some way, you can be sure that you **always** obtain a value (sometimes null, and sometimes not). You need not treat separately the cases of data being present and data being absent.

The rules of the log collector language specify exactly when the result of each action is null. For example, the value obtained from an absent or an invalid field is null, the result of division by `0` is null, and so on. The rules also specify what happens when an operation has null as one of the arguments. For example, the result of addition or multiplication is null whenever one of the operands is null. Other operations, such as SUM or MAX, just ignore null operands. These rules correspond to how you normally understand absence of data. (For example, if you do not have the value A or B, you do not have A+B either.) If the rules do not suit your purpose, you can always replace a null value by a default. You can use the VALUE function for this purpose. (See "VALUE" on page 10-19).

Remember that the null value is **not** the same as the empty string. The former is an absence of data; the latter is a well-defined data item, namely a string of 0 bytes.

Unknown truth value

If you specify a test such as `A>10`, and you do not have `A` (that is, `A` is null), you cannot decide whether the result of the test is true or false. You simply do not have any result. This absence of a truth value has slightly different consequences than absence of other kinds of value. It is therefore not represented by a null value, but by a new value called *unknown*. You may regard it as a third truth value besides true and false.

If you do not use the logical operator NOT, you may treat *unknown* as equivalent to *false*. This is so because whenever a truth value is used, the outcome depends only on whether the value is *true* or not *true*.

If you use the operator NOT, however, you cannot treat these values as equivalent. This is so because NOT(false)=true, while NOT(unknown)=unknown.

Error handling

Some null values result from situations that are normal, such as absent field or absent section. Other null values result from errors, such as invalid data or a violation of language rules.

If the log collector obtains null value as the result of an error, you will be notified in some way that there was an error. The notification depends on the statement that detected the error. For example, the COLLECT statement issues a message that values were set to null because of error; the LIST RECORD statement represents the null values resulting from an error in a special way.

Some errors are detected at an early stage, before they can result in a null value. For example, if you by mistake write `15/0` in your definition, you will receive an error message, and your definition will not be accepted. Notice, however, that not all errors of this kind are detected early. For example, if you code `TIMESTAMP('00.00.00.000000')`, the log collector will not detect the error (time string instead of timestamp string) until it uses your definition and tries to evaluate the TIMESTAMP function.

Some simple ways of specifying a value

About this task

This section discusses two ways in which you can specify a value:
- Write the value explicitly

- Write the name of the value, or of something that holds the value

These two ways are the simplest forms of an expression. Notice that a truth value cannot be specified in any of these two ways. It can only be computed from other values.

Specifying a value explicitly

About this task

You can specify a value explicitly by writing an integer constant, a floating-point constant, or a string constant. For example:

```
720176   2.2E-1   'JobClass 'A'
```

(For more information about constants, see "Tokens" on page 8-1).

Using the integer constant or floating-point constant alone, you can only represent non-negative numbers. To represent a negative number, you can use the minus operator (-) in front of the constant. See "Arithmetic operations" on page 9-8.

You cannot explicitly write a date/time value or a truth value, because there are no date/time constants or truth constants. To write a specific date/time value, you can use a function and a date/time string, as explained in "Date/time strings" on page 9-6. To write a specific truth value, you can use a comparison involving two constants, such as 0=0. See "Comparisons" on page 9-10.

Specifying a value using an identifier

About this task

The most common way to specify a value is to write an identifier. The identifier is usually the name of something that holds a value, for example, a field in a record, or a column in a DB2 table. It may also be the name of the value itself, for example, a value defined by means of a LET clause of DEFINE UPDATE statement. For more information about identifiers, see "Tokens" on page 8-1 and "Identifiers" on page 8-6.

Whenever a syntax diagram for a statement specifies an expression, the description of the statement shows which identifiers are allowed in that expression. For example, in the DEFINE RECORD statement, the expression defining the offset of a section can contain only names of fields in certain previously defined sections. If you specify another name (for instance, the name that is not used in the definition, or the name of a section in the record, or the name of a field that cannot be referenced), you receive an error message stating that the name is unknown, or that you cannot use it in this context.

Some expressions cannot contain any identifiers, for example, the expression following the keyword PARM in the statements DEFINE LOG and DEFINE RECORDPROC. If you use an identifier there, you receive an error message stating that the name is unknown.

Obtaining the value of a variable

About this task

As explained in "SET" on page 11-53, you can define a named string, called a *variable*, by executing a log collector statement such as

```
SET SYSTEM_ID = 'LDGMVS1';
```

This statement defines the variable SYSTEM_ID having value LDGMVS1. The variable remains defined, with its value unchanged, until the end of the current log collector run, or until you change its value by executing another SET statement. An alternative way of defining a variable is to code &SYSTEM_ID=LDGMVS1 as a part of the parameter string supplied to the log collector at the invocation. See Appendix B, “JCL for the log collector language and report definition language,” on page B-1 for details.

You can obtain the value of a variable by coding a *variable reference*. A variable reference consists of a colon (:) followed by the variable name, for example, :SYSTEM_ID.

Using a variable reference has the effect of modifying your stored definition just before it is used. Suppose you have stored an update definition containing the fragment:

```
SYSID = :SYSTEM_ID
```

If you execute a COLLECT statement using this update definition, this fragment will have the same effect as if you have coded SYSID = 'LDGMVS1' (assuming the variable was defined as in the example above).

A variable reference is a special case of expression that can be used only where an expression is allowed. You can also use variable markers to modify your statements, “Using variables to modify your text” on page 8-7. This method lets you replace any part of your text, not only an expression. However, variable markers modify the text when your definition is stored; by using a variable reference, you postpone the modification until the definition is used.

Obtaining the current date and time

About this task

The log collector maintains a timestamp identifying the time when it started the execution of the current statement. You can obtain this timestamp, or the date or time part of it, by writing one of these pairs of keywords:

```
CURRENT DATE
CURRENT TIME
CURRENT TIMESTAMP
```

These values are of type date, time, and timestamp, respectively. Because these values correspond to the start of the current statement, you obtain the same value each time you use these keywords within the same statement.

Obtaining the user ID

About this task

The log collector also has access to the user ID of the user running it. You can obtain this user ID by writing:

```
USER
```

The value specified by this keyword is a character string of length 8.

Date/time strings

About this task

To write specific date/time values, you must code expressions explicitly, for example:

```
DATE('2000-06-27')
TIME('10.32.55.123456')
TIMESTAMP('2000-06-27-10.32.55.123456')
```

These expressions are specific cases of function calls, discussed in Chapter 10, “Functions,” on page 10-1. The character strings `2000-06-27`, `10.32.55.123456`, and `2000-06-27-10.32.55.123456` are examples of date/time strings. Date/time strings are character strings of a specific format.

Date string
A character string that represents a date in the form `yyyy-mm-dd` where `yyyy` is the year, `mm` is the month, and `dd` is the day.

Time string
A character string that represents a time in the form `hh.mm.ss.uuuuuu`, where `hh` is the hour, `mm` is the minute, `ss` is the second, and `uuuuuu` is the microsecond.

Timestamp string
A character string that represents a timestamp in the form `yyyy-mm-dd-hh.mm.ss.uuuuuu` where `yyyy`, `mm`, `dd`, `hh`, `mm`, `ss`, and `uuuuuu` are as above.

You will most likely use date/time strings to write specific date/time values. These strings are more than a substitute for date/time constants; all that was stated earlier also applies to date/time strings obtained as a result of your processing, not only those specified as constants.

DATE function

The DATE function converts a date string to a date. The expression such as `DATE('2000-06-27')` specifies the result of such conversion, here, the date June 27, 2000. In a similar way, the TIME function converts a time string to a time, and TIMESTAMP converts a timestamp string to a timestamp. The expression `TIME('10.32.55.123456')` specifies thus the time 10 hours 32 minutes 55.123456 seconds, and the expression `TIMESTAMP('2000-06-27-10.32.55.123456')` specifies the timestamp 10 hours 32 minutes 55.123456 seconds on June 27, 2000.

Automatic conversions

In some special cases, you can code a date/time string instead of a date/time value, and the log collector automatically performs the conversion for you. For example, if `CREATION_DATE` specifies a date, you may code `CREATION_DATE<'2000-06-25'`. As described in “Comparisons” on page 9-10, the log collector will then automatically convert the date string to a date, and compare the result with the date specified by `CREATION_DATE`. The information on such an automatic conversion is found in the description of the operation that implies it.

Labeled durations

A labeled duration specifies a number of time units. It is used for incrementing or decrementing date/time values (see “Incrementing and decrementing date/time values” on page 9-9.) It is also used to specify the rounding factor for the ROUND function (see “ROUND” on page 10-14.) Note that a labeled duration by itself does not specify any value.

The syntax of labeled duration follows.

```
►►─┤ count ├──┬─YEAR─────────┬──────────────────────────────────►◄
              ├─YEARS────────┤
              ├─MONTH────────┤
              ├─MONTHS───────┤
              ├─DAY──────────┤
              ├─DAYS─────────┤
              ├─HOUR─────────┤
              ├─HOURS────────┤
              ├─MINUTE───────┤
              ├─MINUTES──────┤
              ├─SECOND───────┤
              ├─SECONDS──────┤
              ├─MICROSECOND──┤
              └─MICROSECONDS─┘
```

count:

```
├──┬─constant─────┬─────────────────────────────────────────────┤
   ├─identifier───┤
   ├─function─────┤
   └─(expression)─┘
```

constant
Is a constant that explicitly specifies a value. See “Specifying a value explicitly” on page 9-4.

identifier
Is the name of a value or of something that holds a value. It specifies that value. See “Specifying a value using an identifier” on page 9-4.

function
Is a function call. It specifies the result of a function. See Chapter 10, “Functions,” on page 10-1.

(expression)
Specifies the value of *expression*. See “Expressions” on page 9-17.

The count must specify a number. If this number is a floating-point number, it is converted to an integer by discarding the fractional part.

Examples

```
2 YEARS
X DAYS
(N/60) MINUTES
```

Using operators

About this task

You can specify a value either by writing it explicitly, or by writing its name. Alternatively, you can specify a value as a result of modifying or combining other values. This section discusses how to specify new values by:
- Arithmetic operations on numbers
- Incrementing and decrementing of date/time values
- Concatenation of strings
- Comparisons
- Pattern matching
- Logical operations on truth values

All these operations are specified by an operator written between the operands (an infix operator), or in front of an operand (a prefix operator). A much greater variety of operations, using another format, is described in Chapter 10, "Functions," on page 10-1.

Arithmetic operations

You can apply the prefix operator plus (+) or minus (-) to any numeric value.

Examples

```
-DOWN_TIME   +40   -23.456   -1E8
```

The result is of the same type as the operand. The prefix plus does not change its operand. The prefix minus reverses the sign of its operand.

You can apply any of the infix operators plus (+), minus (-), multiply (*), and divide (/), to any pair of numeric values.

Examples

```
A+B   N_DATASETS-5   COUNT*1E-6   RESP_TIME/60
```

The result depends on the operand types:
- If both operands are integers, the result is an integer. The operation is performed using integer arithmetic. The division is performed so that the remainder has the same sign as the dividend.
- If both operands are floating-point numbers, the result is a floating-point number. The operation is performed using long floating-point operations of System/390.
- If one of the operands is an integer and the other a floating-point number, the integer is converted to a floating-point number. The operation is then performed on the result of the conversion, using floating-point arithmetic. The result is a floating-point number.

The result of dividing an integer by another integer is also an integer. Thus, for example, if RESP_TIME is an integer less than 60, the result of RESP_TIME/60 is 0. If you want the exact result, write RESP_TIME/60.0 instead. The right-hand operand is then the floating-point number 60.0; the left-hand operand, RESP_TIME, is converted to a floating-point number, and the result is a floating-point number.

For all operators (both prefix and infix), the result is null if any of the operands is null. If the result is an integer, the result must be within the range of integers. If the result is a floating-point number, the result must be within the range of floating-point numbers. The right-hand operand of a divide operator must not be 0.

Incrementing and decrementing date/time values

About this task

You can modify a date/time value by adding or subtracting a specified number of time units. The operation is expressed by means of an infix operator plus (+) or minus (-). The left-hand operand is the date/time value. The right-hand operand must be a labeled duration. (See "Labeled durations" on page 9-7.)

Examples

```
CURRENT DATE + 2 MONTHS
INTV_END - X SECONDS
JOB_START + (N/60) MINUTES
```

These expressions specify, respectively: the date two months from now, the time X seconds before the time INTV_END, and the timestamp `N/60` minutes after the timestamp JOB_START.

The number of time units specified by the labeled duration can be negative or 0. Adding a negative number of units is the same as subtracting that number of units. Subtracting a negative number of units is the same as adding that number of units.

The result of the operation is null if the date/time value is null, or if the count in the labeled duration is null.

Incrementing and decrementing dates:
About this task

If the left-hand operand is a date, the right-hand operand must be a labeled duration of years, months, or days. The result is a date.

Adding or subtracting a number of years affects the year part of the date. The month is unchanged, and so is the day unless the result would be February 29 of a non-leap-year. In that case, the day is changed to 28.

Adding or subtracting a number of months affects the month and, if necessary, the year. For the purpose of this operation, a month is the equivalent of a calendar page. Adding or subtracting months is like turning the pages of a calendar. The day part is unchanged unless the result would be invalid (September 31, for example). In that case, the day is changed to the last day of the month.

Adding or subtracting a number of days affects the day and, if necessary, also the month and year.

The result must be within the range of dates.

Incrementing and decrementing times:
About this task

If the left-hand operand is a time, the right-hand operand must be a labeled duration of hours, minutes, seconds, or microseconds. The result is a time with an hour part in the range from 0 to 23.

Any overflow or underflow from the hour part is discarded.

Incrementing and decrementing timestamps:
About this task

If the left-hand operand is a timestamp, the right-hand operand can be a labeled duration of any units. The result is a timestamp with an hour part in the range from 0 to 23.

Adding or subtracting a number of years, months, or days affects the date part as described in “Incrementing and decrementing dates” on page 9-9. Adding or subtracting a number of hours, minutes, seconds, or microseconds may cause an overflow or underflow from the hour part. This overflow or underflow is carried on to the day part and affects the date part in the same way as adding or subtracting a number of days.

The result must be within the range of timestamps.

Concatenation of strings

You can concatenate strings using two vertical bars (||) as an infix operator.

Example

```
JOB_NAME || '01'
```

The result is a character string. The length of the result cannot exceed 254. If any of the strings being concatenated is null, the result is null.

Comparisons

You can compare two values using one of the infix operators equal (=), not equal (<>), greater than (>), less than (<), greater than or equal (>=), less than or equal (<=). The result is a truth value. If one of the compared values is null, the result is *unknown*.

Examples

```
A>10   JOB_NAME<'ABC'   DATE<>'1993.04-15'
```

Numbers must be compared with numbers. Character strings must be compared with character strings and date/time values. Date/time values must be compared with character strings or date/time values of the same type. No other comparisons are allowed.

Numbers are compared by their algebraic value.

- If both numbers are floating-point, they are compared using long floating-point operation of System/390. Two floating-point numbers are considered equal only if their normalized forms have identical bit configurations.
- If one of the numbers is an integer and the other a floating-point number, the integer is converted to a floating-point number. The comparison is then performed with the result of the conversion.

Character strings are compared byte by byte, left to right. If the strings are not the same length, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks so that it has the same length as the other string.

Two strings are equal if they are both empty or if all corresponding bytes are equal. Otherwise, their relation is determined by the comparison of the first unequal pair of bytes.

When a character string is compared with a date/time value, it must be a valid date/time string of the corresponding kind. The string is converted to a date/time value and the comparison is performed on the result.

All comparisons of date/time values are chronological; the value representing the later point of time is considered to be greater.

Because the hour part may range from 0 to 24, certain pairs of different timestamps represent the same time. When such timestamps are compared, the one with a greater date part is considered greater. For example, the result of this comparison is *true*:

```
TIMESTAMP('1985-02-23-00.00.00.000000')>TIMESTAMP('1985-02-22-24.00.00.000000')
```

(However, that the INTERVAL function computes the interval between these timestamps as 0.)

Pattern matching

You can use the infix operator `LIKE` to test whether a string matches a given pattern. This operator can only be used within a lookup expression. (Pattern matching can also be specified in the INCLUDE or EXCLUDE clause of certain statements.)

Examples

```
JOB_NAME  LIKE '%A_CD'
SYSTEM_ID LIKE 'BU%'
```

Both operands must be character strings. The left-hand operand is the string being tested. The right-hand operand is the pattern.

The result is a truth value. The result of the operation is true if the string matches the pattern. The result is false if the string does not match the pattern. The result is unknown if the string or the pattern is null.

The string matches the pattern if it can be divided into substrings of zero or more characters, matching the consecutive characters of the pattern in such a way that:

- Each percent sign in the pattern is matched by a sequence of zero or more arbitrary characters in the string.
- Each underscore character in the pattern is matched by an arbitrary character in the string.
- Any other character in the pattern is matched by an identical character in the string.

If the pattern is an empty string, the only string matching it is the empty string. Also, an empty string matches a pattern consisting of one or more percent signs.

Examples

The string `ABCAXCD` matches the pattern `%A_CD`. The required partition is:

```
ABC  A  X  C  D
 |   |  |  |  |
 %   A  _  C  D
```

Another string matching this pattern is ABCD:

```
   A  B  C  D
|  |  |  |  |
%  A  _  C  D
```

The strings AB, XYZCD, and AXYCD do not match the pattern.

Double-byte characters are not recognized in pattern matching.

Logical operations

You can apply the prefix operator NOT to any truth value. The result is defined as follows for operand p:

Table 9-1. Logical operation NOT

p	**NOT p**
True	False
False	True
Unknown	Unknown

You can apply the infix operators AND, OR to any pair of truth values. The result is defined as follows for operands p and q:

Table 9-2. Logical operations AND and OR

p	**q**	**p AND q**	**p OR q**
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

Testing for null

About this task

Because the result of any comparison involving a null value is unknown, you cannot use a comparison operator to test whether a given value is null. To test whether a value is null, you can code the keywords IS NULL or IS NOT NULL after it.

Examples

```
JOB_NAME   IS NULL
START_TIME IS NOT NULL
```

The result is true or false depending on whether the tested value is null. The value being tested must not be a truth value.

Case expressions

A case expression specifies a value selected by testing one or more conditions. It has this format:

```
            ┌─,──────────────────────────────────┐
            ▼                                    │
►►──CASE────WHEN──condition──THEN──expression───┴──┬──────────────────────┬──END──────►◄
                                                   └──ELSE──expression────┘
```

WHEN *condition* **THEN** *expression*
: Defines one of the possible cases. The case is applicable if the value of condition is true. The result of the case-expression is equal to the result of expression in the applicable case. If several cases are applicable, the result is defined by the first of them. If none of the cases is applicable, the result of case-expression is defined by the ELSE clause.

The expressions in all case definitions must specify values of the same type.

ELSE *expression*
: Defines the result of the case-expression if none of the cases is applicable. The result of the case-expression is then equal to the result of expression. If the ELSE clause is absent, the result is null.

The expression in the ELSE clause must specify a value of the same type as expressions in the case definitions.

An alternative form of case-expression is:

```
                        ┌─,────────────────────────────────────┐
                        ▼                                      │
►►──CASE──expression────WHEN──expression──THEN──expression────┴──────────────────────►

►──┬──────────────────────┬──END─────────────────────────────────────────────────────►◄
   └──ELSE──expression────┘
```

CASE *expression*
: The expression is evaluated and the result is compared with results of expressions appearing in the WHEN clauses.

WHEN *expression* **THEN** *expression*
: Defines one of the possible cases. The case is applicable if the result of the expression following WHEN is equal to the result of the expression appearing in the CASE clause (and neither of them is null). The result of the case-expression is equal to the result of the expression appearing after THEN in the applicable case. If several cases are applicable, the result is defined by the first of them. If none of the cases is applicable, the result of case-expression is defined by the ELSE clause.

All expressions following WHEN must specify values of the same type as the expression in the CASE clause. All expressions following THEN must specify values of the same type (but not necessarily the same as the expressions following WHEN).

ELSE *expression*
: Defines the result of the case-expression if none of the cases is applicable.

The result of the case-expression is then equal to the result of expression. If the ELSE clause is absent, the result is null.

The expression in the ELSE clause must specify a value of the same type as expressions following THEN.

Examples

```
CASE
  WHEN X='A12' THEN 1
  WHEN Y>'BCD' THEN 2
  ELSE 0
END
```

If X has the value `A12`, the result is 1; otherwise, if Y has a value greater than `BCD`, the result is 2; otherwise the result is 0.

```
CASE X
  WHEN 'A12' THEN 1
  WHEN 'B23' THEN 2
END
```

If X has the value `A12`, the result is 1; if X has the value `B23`, the result is 2; otherwise the result is null.

Lookup expressions

A *lookup expression* specifies a value obtained from a table. It has this form:

```
►►—LOOKUP—lookup-column—IN—table-name—┬─────────────────────────┬—WHERE——————►◄
                                      └ORDER BY—order-column—┘
```

lookup-condition:

```
   ┌,──────────────────────────────────────────────┐
   ▼                                               │
├──┬expression—compare-operator—column-name—┬──┴──────────────────────────────┤
   └expression—LIKE—column-name─────────────┘
```

compare-operator:

```
├──┬─ = ──┬─────────────────────────────────────────────────────────┤
   ├─ <> ─┤
   ├─ < ──┤
   ├─ > ──┤
   ├─ <= ─┤
   └─ >= ─┘
```

IN *table-name*
: Identifies the lookup table, that is, the table from which to obtain the value.

LOOKUP *lookup-column*
: Identifies the column of the lookup table from which to obtain the value.

ORDER BY *lookup-column*
: Identifies a column in the lookup table that can be used to specify which entry is selected when several entries match the lookup condition.

WHERE *lookup-condition*
Identifies the row of the lookup table from which to obtain the value.

expression compare-operator column-name
Is a comparison, as described in "Comparisons" on page 9-10. The names in *expression* are names belonging to the context where the lookup expression is used. The *column-name* is the name of a column of the lookup table.

expression **LIKE** *column-name*
Specifies pattern matching as described in "Pattern matching" on page 9-11. The names in *expression* are names belonging to the context where the lookup expression is used. The *column-name* is the name of a column of the lookup table.

AND Is the logical operator defined in "Logical operations" on page 9-12.

How the result is obtained

The result of the lookup expression is defined by this process:

- Replace each *expression* in the lookup condition by its value.
- Evaluate the resulting condition for each row of the lookup table, replacing each *column-name* by a value from that row.
- If the condition is true for exactly one row, obtain the result from that row.
- If the condition is true for several rows, select one of them:
  - If an ORDER BY parameter is specified, select the row containing the lowest value in the order column. If several rows contain the same value in the order column then select the row from among these rows according to the remaining rules below.
  - If the condition does not contain LIKE operators, select any row with a true condition (not defined which).
  - If the condition contains one or more LIKE operators, select among the rows with a true condition, the row with the most specific pattern.

  Obtain the result from the selected row.
- If the condition is not true for any row, the result of the lookup expression is null.

(The described process is just a way of defining the result. The actual method used by the log collector does not necessarily require that all rows are tested.)

Which is the most specific pattern

To see which of two patterns is more specific, you can use this method. Represent each pattern by a string of letters A, U, X, and Z, as follows:

- Represent each percent sign by X.
- Represent each underscore by U.
- Represent each of the remaining characters by A.
- Add Z at the end.

The resulting string is called the *pattern scheme*. The more specific pattern is one whose pattern scheme comes first in alphabetical order.

If the lookup condition contains more than one LIKE operator, compare the patterns obtained by concatenating the right-hand operands of all LIKE operators *in the order they appear in the condition*.

Example A

Suppose the table ACCOUNTING_PERIODS contains this data:

```
START_DATE   END_DATE   PERIOD
------------------------------
2000-01-01  2000-01-28  00/01
2000-01-29  2000-02-25  00/02
2000-02-26  2000-03-25  00/03
     :           :        :
     :           :        :
```

An example of a lookup expression using this table is:

```
LOOKUP PERIOD IN ACCOUNTING_PERIODS
       WHERE SMF72DTE >= START_DATE
         AND SMF72DTE <= END_DATE
```

Assume that SMF72DTE is a field containing the date 2000-01-30. To evaluate the lookup expression, replace SMF72DTE in the lookup condition by its value. The result is:

```
DATE('2000-01-30') >= START_DATE  AND  DATE('2000-01-30') <= END_DATE
```

Now, evaluate this condition for each row of ACCOUNTING_PERIODS, using the values of START_DATE and END_DATE from that row.

The condition is true only for the second row. Obtain the result of the lookup expression from the PERIOD column in that row. The result is `90/02`.

If SMF72DTE contains the date 1999-12-31, the condition is not true for any row, and the result of the lookup expression is null.

Example B

Suppose the table TRANSACTION_CODES contains this data:

```
TRANS  SYSTEM    TRAN
ID     ID        CODE
--------------------
FA21   CICSPROD   T1
B%     CICSPROD   T2
BU%    CICSPROD   T3
F%     CICSPROD   T4
X%     CICSTEST   T5
Y%     CICSTEST   T6
BUS%   %          T7
BUS_   %          T8
```

An example of a lookup expression using this table is:

```
LOOKUP TRAN_CODE IN TRANSACTION_CODES
       WHERE SMF67SYS  LIKE SYSTEM_ID
         AND SMF67TRAN LIKE TRANS_ID
```

If SMF67SYS is a field containing the string `CICSPROD` and SMF67TRAN is a field containing the string `BUS1`, the condition evaluated for each row of the table is:

```
'CICSPROD' LIKE SYSTEM_ID  AND  'BUS1' LIKE TRANS_ID
```

This condition is true for rows 2, 3, 7, and 8. These rows, with their concatenated patterns and pattern schemes (sorted by pattern scheme), are:

```
TRANS  SYSTEM    TRAN   concatenated   pattern
ID     ID        CODE   pattern        scheme
-----  --------  ----   ------------   ------------
BU%    CICSPROD   T3    CICSPRODBU%    AAAAAAAAAAXZ   most specific
B%     CICSPROD   T2    CICSPRODB%     AAAAAAAAAXZ
BUS_   %          T8    %BUS_          XAAAUZ
BUS%   %          T7    %BUS%          XAAAXZ         most general
```

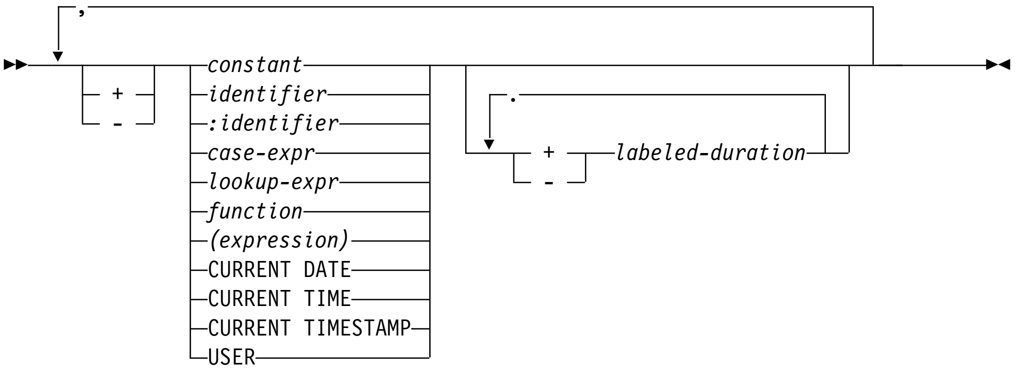
The row with the most specific pattern is one where column TRAN_CODE contains the string T3. The result of the lookup expression is T3.

Important

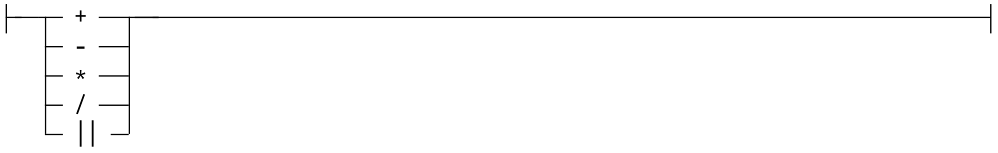
To achieve acceptable performance, lookup tables are read into storage buffers at the start of processing. **This means that the tables should not be too large. It also means that any changes to a lookup table resulting from a collect operation do not affect the result of a lookup before the collect is completed.**

Expressions

This diagram specifies the general form of expression that you can use wherever the syntax specifies an *expression*. The diagram does not reflect all the rules that you must follow when you use operators. You can find these rules in section “Using operators” on page 9-8.



operator:



constant
A constant that explicitly specifies a value. See “Specifying a value explicitly” on page 9-4.

identifier
The name of a value or of something that holds a value. It specifies that value. See “Specifying a value using an identifier” on page 9-4.

:identifier
Specifies the value of a variable. See “Obtaining the value of a variable” on page 9-4.

case-expr
A case expression. See “Case expressions” on page 9-13.

lookup-expr
A lookup expression. See “Lookup expressions” on page 9-14.

function
A function call. It specifies the result of a function. See Chapter 10, “Functions,” on page 10-1.

(*expression***)**
Specifies the value of *expression*.

labeled-duration
A labeled duration. See “Labeled durations” on page 9-7.

CURRENT DATE ;CURRENT TIME ;CURRENT TIMESTAMP
Specify current date, time, or timestamp. See “Obtaining the current date and time” on page 9-5.

USER Specifies current user ID. See “Obtaining the current date and time” on page 9-5.

+ - * / | |
Are operators. Their meaning is specified in “Using operators” on page 9-8.

Precedence of operators

If not specified otherwise by means of parentheses, the prefix plus and minus are applied before multiply and divide. Multiply and divide are applied before infix plus and minus. Operators of the same priority are applied from left to right.

Conditions

The following diagram specifies the general form of expression that you can use wherever the syntax specifies a *condition*. Notice that the diagram does not reflect all the rules that you must follow when you use operators. You can find these rules in “Using operators” on page 9-8.

```
    ┌─,──────────────────────────────────────────────────┐
    ▼                                                    │
►►──┴──┬─────┬──┬─expression──compare-operator──expression─┬─┴──────────►◄
       └─NOT─┘  ├─expression──┬─IS NULL─────┬───────────────┤
                │             └─IS NOT NULL─┘               │
                └─(condition)───────────────────────────────┘
```

logical-operator:

```
├──┬─AND─┬──────────────────────────────────────────────────────────┤
   └─OR──┘
```

compare-operator:

```
├──┬─ =  ─┬──────────────────────────────────────────────────────────────┤
   ├─ <> ─┤
   ├─ <  ─┤
   ├─ >  ─┤
   ├─ <= ─┤
   └─ >= ─┘
```

expression compare-operator expression
Is a comparison, as described in "Comparisons" on page 9-10.

expression **IS NULL ;***expression* **IS NOT NULL**
Is a test for null. See "Testing for null" on page 9-12.

(*condition***)**
Specifies the value of *condition*.

AND, OR, NOT
Are logical operators defined in "Logical operations" on page 9-12.

Precedence of operators

If not specified otherwise by means of parentheses, NOT is applied before AND and OR. The operators AND and OR are then applied from left to right.

Chapter 10. Functions

This chapter describes a special form of expression called a function call. You can use a function call directly in the statements whenever the syntax specifies an expression. You can also use it as a part of more complex expressions.

A function call specifies a value as the result of applying a named operator, called a function, to one or more arguments. It consists of a function name followed by a pair of parentheses enclosing the specification of arguments. Most arguments are values specified by means of expressions. However, some functions use arguments that are not values, for example, a section name or a labeled duration.

The result of a function is null if one of the arguments is null. The only exception to this rule is the VALUE function (see “VALUE” on page 10-19).

This chapter describes, in alphabetical order, all functions available in the log collector language. It describes the purpose, syntax, and result of each function. It also provides examples of how to use the function.

CHAR

The CHAR function obtains a string representation of a date/time value.

Syntax

```
►►──CHAR (expression)──────────────────────────────────────────────►◄
```

The argument must be a date, a time, or a timestamp.

Result

The result is a character string. It is the date/time string that represents the argument.

Example

Assume that:

- X_DATE has the value May 3, 2000.
- X_TIME has the value 5 hours, 17 minutes, and 34 seconds.
- X_TSTAMP has the value 5 hours, 17 minutes, and 34 seconds on May 3, 2000.

The function produces these results:

```
CHAR(X_DATE) = '2000-05-03'
CHAR(X_TIME) =  '05.17.34.000000'
CHAR(X_TSTAMP) = '2000-05-03-05.17.34.000000'
```

DATE

The DATE function obtains a date from a value.

Syntax

```
►►──DATE (expression)──────────────────────────────────────────────►◄
```

The argument must be a date, a timestamp, a number, or a date string.

Result

The result is a date.

- If the argument is a date, the result is that date.
- If the argument is a timestamp, the result is the date part of that timestamp.
- If the argument is a number, consider the integer part of that number as *n*. It must be in the range 1 to 3 652 059. The result of the function is the date of the day with sequential number *n*, counting from January 1, 0001 as day 1.
- If the argument is a date string, the result is the date represented by that string.

Example

Assume that:

- X_DATE has the value April 22, 1993.
- X_TSTAMP has the value 15 hours, 2 minutes, and 1 second on March 6, 1993.
- X_STRING has the value '2000-03-06'.

The function produces these results:

```
DATE(X_DATE) = April 22, 1993
DATE(X_TSTAMP) = March 6, 1993
DATE(727159) = November 23, 1991
DATE('2000-06-15') = June 15, 2000
DATE(X_STRING) = March 6, 2000
```

DAY

The DAY function obtains the day part of a value.

Syntax

```
►►──DAY (expression)───────────────────────────────────────────────►◄
```

The argument must be a date or a timestamp.

Result

The result is an integer between 1 and 31. It is the day part of the date or timestamp.

Example

Assume that:

- X_DATE has the value June 9, 2000.
- X_TSTAMP has the value 15 hours, 2 minutes, and 1 second on February 19, 2000.

The function produces these results:

```
DAY(X_DATE) = 9
DAY(X_TSTAMP) = 19
```

DAYS

The DAYS function obtains the day number corresponding to a date.

Syntax

```
►►──DAYS (expression)──────────────────────────────────────────────────►◄
```

The argument must be a date, a timestamp, a date string, or a timestamp string.

Result

The result is an integer. It is the sequential number of the day represented by the argument, considering January 1, 0001 as day 1.

- If the argument is a date or a date string, the result is the sequential number of the day represented by the date.
- If the argument is a timestamp or a timestamp string, the result is the sequential number of the day represented by the date part of the timestamp.

Example

Assume that:

- X_DATE has the value April 15, 1993.
- X_TSTAMP has the value 11 hours, 33 minutes, and 21 seconds on August 26, 1993.

The function produces these results:

```
DAYS(X_DATE) = 727668
DAYS(X_TSTAMP) = 727801
DAYS('1993-05-03') = 727686
DAYS('1993-05-03-15.45.01.000000') = 727686
```

Usage notes

You can use the DAYS function to obtain the day of week for a given date. To compute the day of week for X_DATE, use this expression:

```
DAYS(X_DATE)-((DAYS(X_DATE)-1)/7)*7
```

The result is a number from 1 through 7, representing Monday through Sunday. If X_DATE has the value April 15, 1993, the result is 4 which represents a Thursday.

You can use the DAYS function, with the DATE function, to obtain the date of Monday in the week containing a given date. To compute the Monday date of the week containing X_DATE, use this expression:

```
DATE((DAYS(X_DATE)/7)*7 + 1)
```

(This expression assumes that weeks start on Monday). If X_DATE has the value April 15, 1993, then the result is April 12, 1993, which is a Monday of the week that includes April 15, 1993.)

DAYTYPE

The DAYTYPE function obtains the day type of a given day.

Syntax

```
►►—DAYTYPE (expression)————————————————————————————————————►◄
```

The argument must be a date.

The result of the function is defined by means of two tables, DRLSYS.DAY_OF_WEEK and DRLSYS.SPECIAL_DAY. These tables are *control tables* and should be set up by your system administrator. (For more information about control tables, refer to the *Administration Guide*).

The table DRLSYS.DAY_OF_WEEK defines an 8-character string, called a *day type code*, for each day of the week. The table on your system might contain the information shown in Figure 10-1. The numbers 1 through 7 identify the days of week, Monday through Sunday. The table defines MON as the day type for Monday, TUE as the day type for Tuesday, and so on.

```
DAY OF  DAY
  WEEK  TYPE
------  --------
     1  MON
     2  TUE
     3  WED
     4  THU
     5  FRI
     6  SAT
     7  SUN
```

Figure 10-1. Example of DRLSYS.DAY_OF_WEEK table

The table DRLSYS.SPECIAL_DAY defines day type codes for certain dates. The table on your system might contain the information shown in Figure 10-2. It defines HOLIDAY as the day type for December 25, 1993 and January 1, 1994.

```
            DAY
DATE        TYPE
----------  --------
1993-12-25  HOLIDAY
1994-01-01  HOLIDAY
```

Figure 10-2. Example of DRLSYS.SPECIAL_DAY table

Notice that day type codes can be different for different installations. The codes for days of week need not be unique. For example, a particular installation might use WEEKDAY as the day type for Monday through Friday, and WEEKEND for Saturday and Sunday.

Result

The result is an 8-character string. It is the day type code for the day represented by the argument.

Let *date* be the argument of DAYTYPE. Let *w* be a number representing the day of week for *date*, the values 1 through 7 standing for Monday through Sunday. The result of DAYTYPE is specified by the following expression.

```
VALUE( LOOKUP DAY_TYPE IN DRLSYS.SPECIAL_DAY
              WHERE date = DATE,
       LOOKUP DAY_TYPE IN DRLSYS.DAY_OF_WEEK
              WHERE w = DAY_OF_WEEK )
```

A null result of this expression is considered an error. It means that day of week number *w* is missing from DRLSYS.DAY_OF_WEEK.

Example

Assume that:

- DRLSYS.DAY_OF_WEEK contains the data shown in Figure 10-1 on page 10-4.
- DRLSYS.SPECIAL_DAY contains the data shown in Figure 10-2 on page 10-4.
- X_DATE has the value December 25, 1993.
- X_TSTAMP has the value 10 hours, 5 minutes, and 21 seconds on December 22, 1993.

The function produces these results:

```
DAYTYPE(X_DATE)   = 'HOLIDAY '
DAYTYPE(X_TSTAMP) = 'WED     '
```

DIGITS

The DIGITS function obtains a character string representation of a number.

Syntax

```
►►──DIGITS (expression)──────────────────────────────────────────────►◄
```

The argument must be an integer.

Result

The result is the string of digits (other than leading zeros) that represents the absolute value of the argument.

Example

```
DIGITS(754) = '754'
DIGITS(00054) = '54'
DIGITS(-54) = '54'
```

FIELD

The FIELD function obtains the contents of a record field in a section specified by its occurrence number.

Syntax

```
                                       ┌─.──────────────┐
►►──FIELD (field-name──┬──────────┬────▼─,──expression──┴─)──────────────────►◄
                       │ ┌─.───┐  │
                       └─▼─,──*─┴─┘
```

The *field-name* must name a field in a repeated section. Each *expression* must specify an integer. These integers are occurrence numbers of nested repeated sections that contain the field named *field-name*.

Result

The result is the value contained in *field-name*.

Example

Assume that you have the record shown in Figure 10-3.

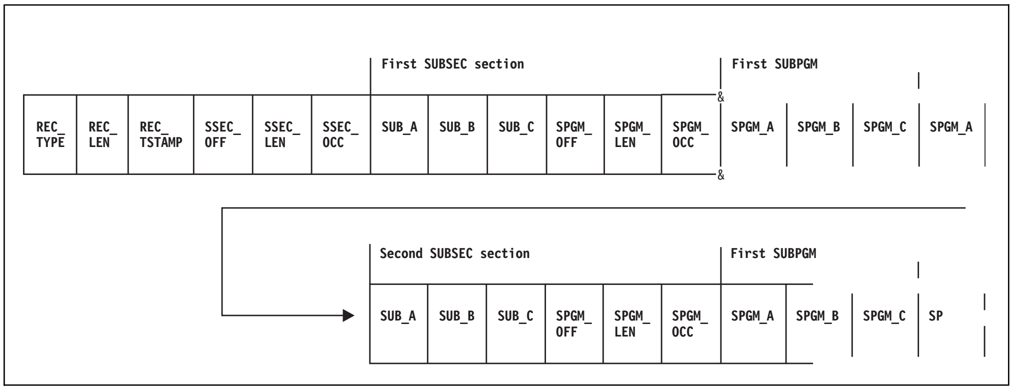


Figure 10-3. Example of a record containing nested sections.

To obtain the data in the field SUB_A in the first occurrence of SUBSEC, write:

```
FIELD(SUB_A,1)
```

To obtain the data in the field SPGM_A in the second occurrence of the SUBPGM section from the first occurrence of the SUBSEC section, write:

```
FIELD(SPGM_A,1,2)
```

For more information about using the FIELD function, see “Accessing specific sections in a record” on page 5-14.

FLOAT

The FLOAT function obtains a floating-point representation of a number.

Syntax

```
►►──FLOAT (expression)─────────────────────────────────────────────────►◄
```

The argument must be a number.

Result

If the argument is an integer, the result is a floating-point representation of that integer. If the argument is a floating-point number, the result is that number.

Example

```
FLOAT(14) = 14.0
FLOAT(25.7) = 25.7
```

GETVAR

The GETVAR function is an internal function designed for the IMS CSQ Feature and obtains the IMS system ID associated with a log data set.

Syntax

```
►►──GETVAR (expression)──────────────────────────────────────────────►◄
```

The only valid argument to the GETVAR function is `'IMSID'`.

Result

An 8 character IMS system ID or `'$UNKNOWN'` is returned.

Example

Assume you have the following IMS system log data sets:

- DRLLOG1 with data for IMSA
- DRLLOG2 with data for IMSB
- DRLLOG3 with data for IMSC

When records from `DRLLOG2` are being processed, the result is `GETVAR('IMSID') = 'IMSB '` .

Usage Notes: If the IMS X'07' records are input from the `DRLIMS07 DD` name, the `GETVAR('IMSID')` returns the value `'$UNKNOWN'`.

HOUR

The HOUR function obtains the hour part of a value.

Syntax

```
►►──HOUR (expression)────────────────────────────────────────────────►◄
```

The argument must be a time or a timestamp.

Result

The result is an integer between 0 and 24. It is the hour part of the argument.

Example

Assume that:

- X_TIME has the value 14 hours, 20 minutes, 55 seconds.
- X_TSTAMP has the value 10 hours, 5 minutes, and 21 seconds on January 3, 1993.

The function produces these results:

```
HOUR(X_TIME) = 14
HOUR(X_TSTAMP) = 10
```

INTEGER

The INTEGER function obtains the integer part of a number.

Syntax

```
►►──INTEGER (expression)──────────────────────────────────────►◄
```

The argument must be a number.

Result

If the argument is an integer, the result is that integer. If the argument is a floating-point number, the result is the integer part of that number.

Example

```
INTEGER(45) = 45
INTEGER(-75.3) = -75
INTEGER(0.0005) = 0
```

INTERVAL

The INTERVAL function obtains the length of a time interval in seconds.

Syntax

```
►►──INTERVAL (expression, expression)─────────────────────────►◄
```

Both arguments must be date/time values of the same type.

Result

The result is a floating-point number.

The result is the interval, in seconds, from the instant designated by the first argument to the instant designated by the second argument.

If the first argument is later than the second, the result is negative.

The result has the maximum precision allowed by its floating-point representation. Therefore, results up to 2283 years have a precision of 1 microsecond.

Example

Assume you have these variables:

- TME1 has the value of 6 hours, 20 minutes, 29 seconds, and 25000 microseconds.
- TME2 has the value of 18 hours, 25 minutes, 20 seconds.
- DAY1 has the value of March 5, 1993.
- DAY2 has the value of March 8, 1993.
- TS1 has the value of 5 hours on March 5, 1993.
- TS2 has the value of 10 hours, 30 minutes on March 11, 1993.

The function produces these results:

```
INTERVAL(TME1, TME2) = 43490.975
INTERVAL(TME2, TME1) = -43490.975
INTERVAL(DAY1, DAY2) = 259200.0
INTERVAL(TS1, TS2) = 538200.0
```

IPCONV

The IPCONV function converts a string that contains the hexadecimal representation of an IP address to the corresponding presentation format.

Syntax

```
►►──IPCONV (expression)──────────────────────────────────────────►◄
```

The argument must be the hexadecimal string that represents the IP address. The hexadecimal string can only contain characters, in the range 0-9 and A-F. The string must be 32 characters long.

For IPV4 addresses, only the following format is supported:

`00000000000000000000FFFF`*`xxxxxxxx`*

Where *xxxxxxxx* are hexadecimal digits.

Result

The result is a character string that is the *presentation format* of the IP address. It can have the following formats:

IPV4 address
`d.d.d.d`

Where *d* are decimal digits, from 0 to 255, with the leading zero omitted.

IPV6 address
`x:x:x:x:x:x:x:x`

Where *x* are groups of four hexadecimal digits, from 0000 to FFFF, with the leading zero omitted, but at least one digit in each group.

Note: Other types of presentation formats, such as an IPV6 address with all zeros omitted (for example, `::`) or an IPV4 address mapped as an IPV6 address (for example, `::FFF`*`d.d.d.d`*) are not supported.

Example

Assume that:

- IPV4 is a string with the following address:

```
IPV4 = '00000000000000000000FFFC50B6B01'
```

- IPV6 is a string with the following address:

```
IPV6 = 'FE800000000000000011019900810106'
```

The function produces these results:

```
IPCONV(IPV4) = '197.11.107.1'
IPCONV(IPV6) = 'FE80:0:0:0:11:199:81:106'
```

LENGTH

The LENGTH function obtains the length of a character string.

Syntax

```
►►──LENGTH (expression)──────────────────────────────────────────►◄
```

The argument must be a character string.

Result

The result is an integer. It is the length of the argument.

Example

Assume X_STRING has the value of 'LOG_NAME'. The function produces these results:

```
LENGTH(X_STRING)  = 8
LENGTH('REC_LOG') = 7
LENGTH(' ') = 1
LENGTH('') = 0
```

MICROSECOND

The MICROSECOND function obtains the microseconds part of a value.

Syntax

```
►►──MICROSECOND (expression)─────────────────────────────────────►◄
```

The argument must be a time or a timestamp.

Result

The result is an integer between 0 and 999 999. It is the microseconds part of the argument.

Example

Assume that:

- X_TIME has the value 14 hours, 20 minutes, 55 seconds and 155 microseconds.
- X_TSTAMP has the value 8 hours, 30 minutes, and 45 seconds on March 25, 1993.

The function produces these results:

```
MICROSECOND(X_TIME) = 155
MICROSECOND(X_TSTAMP) = 0
```

MINUTE

The MINUTE function obtains the minute part of a value.

Syntax

```
►►──MINUTE (expression)──────────────────────────────────────────────►◄
```

The argument must be a time or a timestamp.

Result

The result is an integer between 0 and 59. It is the minute part of the argument.

Example

Assume that:

- X_TIME has the value 17 hours, 16 minutes, 22 seconds and 100,000 microseconds.
- X_TSTAMP has the value 8 hours, 58 minutes, and 19 seconds on April 1, 1993.

The function produces these results:

```
MINUTE(X_TIME) = 16
MINUTE(X_TSTAMP) = 58
```

MONTH

The MONTH function obtains the month part of a value.

Syntax

```
►►──MONTH (expression)───────────────────────────────────────────────►◄
```

The argument must be a date or a timestamp.

Result

The result is an integer between 1 and 12. It is the month part of the argument.

Example

Assume that:

- X_DATE has the value July 22, 2000.
- X_TSTAMP has the value May 3, 2000.

The function produces these results:

```
MONTH(X_DATE) = 7
MONTH(X_TSTAMP) = 5
```

PERIOD

The PERIOD function obtains the name of the period containing a given time instant.

Syntax

```
►►──PERIOD (expression1, expression2, expression3)──────────────────────────────►◄
```

expression1 must be a character string.

expression2 must be a date.

expression3 must be a time.

The result of the function is defined by means of table DRLSYS.PERIOD_PLAN. This table is a *control table* and should be set up by your system administrator. (For more information about control tables, refer to the *Administration Guide* .)

The table DRLSYS.PERIOD_PLAN defines one or more *period plans*. A period plan divides the day into named intervals (such as shifts). These intervals are called *periods*. The table on your system might contain the information shown in Figure 10-4.

```
PERIOD    DAY       START     END       PERIOD
PLAN ID   TYPE      TIME      TIME       NAME
--------  --------  --------  --------  --------
MVS2      MON       00.00.00  24.00.00  SPECIAL
%         MON       00.00.00  08.00.00  NIGHT
%         MON       08.00.00  17.00.00  PRIME
%         MON       17.00.00  24.00.00  NIGHT
%         TUE       00.00.00  08.00.00  NIGHT
%         TUE       08.00.00  17.00.00  PRIME
%         TUE       17.00.00  24.00.00  NIGHT
%         WED       00.00.00  08.00.00  NIGHT
%         WED       08.00.00  17.00.00  PRIME
%         WED       17.00.00  24.00.00  NIGHT
%         THU       00.00.00  08.00.00  NIGHT
%         THU       08.00.00  17.00.00  PRIME
%         THU       17.00.00  24.00.00  NIGHT
%         FRI       00.00.00  08.00.00  NIGHT
%         FRI       08.00.00  17.00.00  PRIME
%         FRI       17.00.00  24.00.00  NIGHT
%         SAT       00.00.00  24.00.00  WEEKEND
%         SUN       00.00.00  24.00.00  WEEKEND
%         HOLIDAY   00.00.00  24.00.00  HOLIDAY
```

Figure 10-4. Example of DRLSYS.PERIOD_PLAN table

Each row in the table describes one period of the period plan. A period plan is defined separately for each day type that may be the result of the DAYTYPE function. That day type is identified in the DAY_TYPE column.

There may be several period plans for the same day, each identified by its *period plan ID,* and each defined by a group of rows in the table. The column PERIOD_PLAN_ID contains a pattern to be matched by period plan ID. The periods specified for each day type and period plan must not overlap and must cover the whole day from 00 hours to 24 hours.

As specified by the table in Figure 10-4 on page 10-12, the plan named MVS2 defines all of Monday to be one period, named SPECIAL. According to any other plan, Monday is divided into three periods named NIGHT, PRIME, and NIGHT, respectively.

Result

The result is an 8-character string.

Let the three arguments of PERIOD be called *plan*, *date*, and *time*, respectively. The result of PERIOD is the name of the period of the plan *plan* that contains the time instant identified by *date* and *time*. It is specified by this lookup expression:

```
LOOKUP PERIOD_NAME IN DRLSYS.PERIOD_PLAN
       WHERE plan LIKE PERIOD_PLAN_ID
       AND   DAYTYPE(date) = DAY_TYPE
       AND   time >= START_TIME
       AND   time <  END_TIME
```

A null result of this lookup expression is regarded as an error.

The PERIOD function is defined using the DAYTYPE function. It is therefore indirectly defined also by the tables DRLSYS.DAY_OF_WEEK and DRLSYS.SPECIAL_DAY that define DAYTYPE.

Example

```
DAY OF  DAY
  WEEK  TYPE
------  --------
     1  MON
     2  TUE
     3  WED
     4  THU
     5  FRI
     6  SAT
     7  SUN
```

Figure 10-5. DRLSYS.DAY_OF_WEEK table

Assume that DRLSYS.PERIOD_PLAN is as shown in Figure 10-4 on page 10-12, and that DAYTYPE is defined by the tables shown in Figure 10-5 and Figure 10-6. Notice that June 7, 1993 is a Monday.

```
            DAY
DATE        TYPE
----------  --------
1993-12-25  HOLIDAY
1994-01-01  HOLIDAY
```

Figure 10-6. DRLSYS.SPECIAL_DAY table

The function produces these results:

```
PERIOD('MVS1',DATE('1993-06-07'),TIME('06.24.19.240000')) = 'NIGHT   '
PERIOD('MVS2',DATE('1993-06-07'),TIME('06.24.19.876050')) = 'SPECIAL '
PERIOD('MVS1',DATE('1993-06-07'),TIME('13.00.00.000000')) = 'PRIME   '
PERIOD('MVS1',DATE('1993-12-25'),TIME('12.34.56.000000')) = 'HOLIDAY '
```

ROUND

The ROUND function rounds a date/time value down to a multiple of the specified number of time units.

Syntax

```
►►──ROUND (expression, labeled-duration)──────────────────────────────────►◄
```

The first argument must be a date, a time, or a timestamp. The second argument must be a labeled duration.

- If the first argument is a date, the second argument must be a labeled duration of years, months, or days.
- If the first argument is a time, the second argument must be a labeled duration of hours, minutes, seconds, or microseconds.
- If the first argument is a timestamp, the second argument can be any labeled duration.

In each case, the labeled duration must specify a number of units greater than 0.

Result

The result is of the same type as the first argument. It is obtained from the first argument by this procedure:

- Select the part that corresponds to the time unit used in the labeled duration. This means, if the labeled duration is *n* YEARS, select the year part; if the labeled duration is *n* MONTHS, select the month part; and so on.
- Round the selected part to a whole multiple of the second argument, in this sense:
  - If the part is an hour, minute, second, or microsecond (and thus has values starting with 0), round it down to the nearest number $k \times n$, where $k \geq 0$ is a whole number, and n is the number of units in the labeled duration.
  - If the part is a month or a day (and thus has values starting with 1), round it down to the nearest number $1 + k \times n$, where $k \geq 0$ is a whole number, and n is the number of units in the labeled duration.
  - If the part is a year, round it down to the closest number $k \times n$, where $k \geq 0$ is a whole number, and n is the number of years in the labeled duration. If the result is 0, replace it by 1.
- Replace all lower-order parts by their lowest values (that is, 1 for month and day, or 0 for other parts).
- Leave the remaining parts unchanged.

Example

```
ROUND(DATE('1993-06-27'),1 MONTH)  = June 1, 1993
ROUND(DATE('1993-06-27'),3 MONTHS) = April 1, 1993
ROUND(DATE('1993-06-27'),6 MONTHS) = January 1, 1993
ROUND(DATE('1993-06-27'),15 DAYS)  = June 16, 1993
ROUND(DATE('1993-06-27'),50 DAYS)  = June 1, 1993
ROUND(TIME('12.47.39.125000'),1 HOUR)     = 12 hours
ROUND(TIME('12.47.39.125000'),60 MINUTES) = 12 hours
ROUND(TIME('12.47.39.125000'),30 MINUTES) = 12 hours, 30 minutes
ROUND(TIME('12.47.39.125000'),20 MINUTES) = 12 hours, 40 minutes
ROUND(TIME('12.47.39.125000'),5 SECONDS)  = 12 hours, 47 minutes, 35 seconds
```

Usage notes

Notice that:

- Rounding a date to 3 months produces the first day of a quarter.
- Rounding a date to 6 months produces the first day of a half-year period.
- Rounding a date to 15 days produces the first day of a 15-day period within a month.
- Rounding to a large number of units is allowed, but it does not affect the higher-order parts. For example, rounding a date to 50 days produces the same effect as rounding to 1 month.

SECOND

The SECOND function obtains the seconds part of a value.

Syntax

```
►►──SECOND (expression)──────────────────────────────────────────────►◄
```

The argument must be a time or a timestamp.

Result

The result is an integer between 0 and 59. It is the seconds part of the argument.

Example

Assume that:

- X_TIME has the value 0 hours, 4 minutes, 25 seconds, and 1 432 microseconds.
- X_TSTAMP has the value 17 hours, 25 minutes, 50 seconds, and 5 microseconds on June 20, 1993.

The function produces these results:

```
SECOND(X_TIME) = 25
SECOND(X_STAMP) = 50
```

SECTNUM

The SECTNUM function obtains the sequential number of a section occurrence.

Syntax

```
►►──SECTNUM (section-name)───────────────────────────────────────────►◄
```

This function is intended for use with an internal record generated from a repeated section (see “Using repeated sections within records” on page 5-2). The *section-name* must identify one of the sections included in that record.

Result

The result is an integer. It is the sequential number (within the containing section) of the occurrence of *section-name* that was used to build the record.

If *section-name* is not a repeated section, the result is 1 if the section is present in the record, or 0 if it is absent.

For more information about using this function, see “Obtaining a section occurrence number” on page 5-13.

Example

Assume that you have a record shown in Figure 10-7:

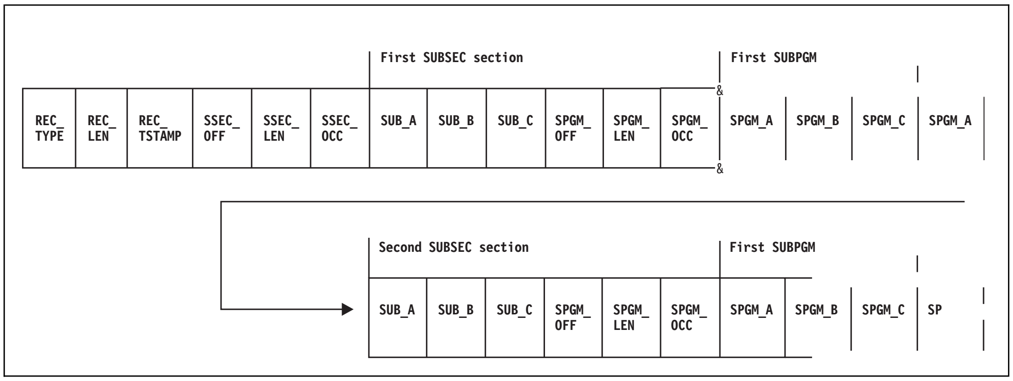


Figure 10-7. Example of a record with nested sections

Assume that SECTNUM is evaluated while processing an internal record generated for the second occurrence of SUBPGM in the first occurrence of SUBSEC.

The function produces these results:

```
SECTNUM(SUBSEC) = 1
SECTNUM(SUBPGM) = 2
```

SUBSTR

The SUBSTR function obtains a substring of a character string.

Syntax

```
►►──SUBSTR (expression, expression──┬──────────────┬─)──────────────────────►◄
                                    └─, expression─┘
```

In this description, the three arguments are called, respectively, *string, start*, and *length*. The *string* must be a character string. The *start* must be an integer in the range 1 to 254. The *length* must be an integer in the range 0 to 255-*start*.

Result

The result is a character string.

If *length* is specified, the result consists of *length* bytes of *string*, starting at the position *start*. The *string* is regarded as extended on the right with the necessary number of blanks so that the specified substring exists.

If *length* is not specified, the result consists of all bytes of *string*, starting at the position *start* and extending up to the end of *string*. If *start* is greater than the length of *string*, the result is an empty string.

Both *start* and *length* are expressed in *bytes*. The SUBSTR function does not recognize double-byte characters, and the result need not be a well-formed character string.

Example

The function produces these results:

```
SUBSTR('SUB_REC',3,2) = 'B_'
SUBSTR('SUB_REC',3) = 'B_REC'
SUBSTR('SUB_REC',3,10) = 'B_REC     '
```

TIME

The TIME function obtains a time from a value.

Syntax

```
►►──TIME (expression)──────────────────────────────────────────────────►◄
```

The argument must be a time, a timestamp, or a time string.

Result

The result is a time.

- If the argument is a time, the result is that time.
- If the argument is a timestamp, the result is the time part of that timestamp.
- If the argument is a time string, the result is the time represented by that string.

Example

Assume that:

- X_TIME has the value 3 hours, 24 minutes, 20 seconds, and 2 microseconds.
- X_TSTAMP has the value 15 hours, 33 minutes, 25 seconds, and 75 microseconds on June 20, 1993.

The function produces these results:

```
TIME(X_TIME) = 3 hours, 24 minutes, 20 seconds, and 2 microseconds
TIME('17.24.13.000025') = 17 hours, 24 minutes, 13 seconds, and 25 microseconds
TIME(X_TSTAMP) = 15 hours, 33 minutes, 25 seconds, and 75 microseconds
```

TIMESTAMP

The TIMESTAMP function obtains a timestamp from a value or a pair of values.

Syntax

```
►►──TIMESTAMP (expression1──┬──────────────┬─)──────────────────────────────►◄
                            └─, expression2─┘
```

Result

The result of the function depends on whether *expression 1* or *expression2* is specified, or both.

If only one argument is specified

expression 1 must be a timestamp or a timestamp string. The result is a timestamp:

- If *expression 1* is a timestamp, the result is that timestamp.
- If *expression 1* is a timestamp string, the result is the timestamp represented by that string.

If both arguments are specified

expression 1 must be a date or a date string. expression2 must be a time or a time string.

The result is a timestamp. It consists of the date and time specified by the arguments.

Example

Assume that:

- X_TIME has the value 3 hours, 24 minutes, 20 seconds, and 2 microseconds.
- X_DATE has the value February 11, 1993.
- X_TSTAMP has the value 15 hours, 33 minutes, 25 seconds, and 75 microseconds on June 20, 1993.

The function produces these results:

```
TIMESTAMP(X_TSTAMP) = 15 hours, 33 minutes, 25 seconds, and 75 microseconds on June 20, 1993
TIMESTAMP('1993-04-17-19.01.25.000000') = 19 hours, 1 minute, 25 seconds on April 17, 1993
TIMESTAMP(X_DATE, X_TIME) = 3 hours, 24 minutes, 20 seconds, and 2 microseconds on February 11, 1993
```

TRANSLATE

The TRANSLATE function translates a character string to another representation.

Syntax

```
►►──TRANSLATE (string──┬───────────┬─)──────────────────────────────►◄
                       └─┤ codes ├─┘
```

codes:

```
├──,──┬─,─from-codes──┬────────────┬──────────┬─────────────────────┤
      │               └─,─pad-byte─┘          │
      └─to-codes──┬──────────────────────────┬┘
                  └─,──┬─,─pad-byte───────────┤
                       └─from-codes──┬────────────┬┘
                                     └─,─pad-byte─┘
```

In this description, the arguments are called, respectively, *string*, *to-codes*, *from-codes*, and *pad-byte*. All arguments must be character strings. The *pad-byte* must be a string of length 1.

Result

The result is a character string. It is a copy of *string* in which some of the bytes have been replaced by others (*string* itself is not altered).

- If *from-codes* is present, each byte of *string* is looked up in *from-codes*. If it is found on position *n* of *from-codes*, it is replaced by the byte appearing on position *n* of *to-codes*; otherwise it is left unchanged.

  If *to-codes* is shorter than *n* or omitted, it is conceptually extended on the right with as many copies of the *pad-byte* as needed. If *pad-byte* is omitted, a blank is used instead.
- If *from-codes* is absent and *to-codes* is present, each byte of *string* is replaced by the byte appearing on position $n = b + 1$ of *to-codes*, where *b* is the binary value of the byte.

  If *to-codes* is shorter than *n*, it is conceptually extended on the right with as many copies of the *pad-byte* as needed. If *pad-byte* is omitted, a blank is used instead.
- If both *from-codes* and *to-codes* are omitted, *string* is translated to uppercase: all occurrences of lowercase letters (a-z) are replaced by their uppercase counterparts (A-Z).

The TRANSLATE function does not recognize double-byte characters, and the result need not be a well-formed character string.

Example

```
TRANSLATE('abcdef') = 'ABCDEF'
TRANSLATE('abcdef', '*#$', 'bde') = 'a*c#$f'
TRANSLATE('abcdef', 'CD', 'acde', '.') = 'CbD..f'
```

VALUE

The VALUE function returns the first argument that is not null.

Syntax

```
►►──VALUE (expression─, expression─)──────────────────────────────────►◄
```

All arguments must have the same data type.

Result

The result has the same data type as the arguments. It is equal to the first argument that is not null. If all arguments are null, the result is null.

Example

Assume that:
- EXPA has the value of 25.
- EXPB has the value of 50.
- EXPC has a null value.

The function produces these results:

```
VALUE(EXPA, EXPB, EXPC) = 25
VALUE(EXPC, EXPB, EXPA) = 50
VALUE(EXPB, EXPA) = 50
```

WORD

The WORD function extracts a word from a character string.

Syntax

```
►►──WORD (expression, expression─┬──────────────┬─)──────────────────────────►◄
                                 └─, expression─┘
```

In this description, the three arguments are called, respectively, *string, n*, and *delimiters*. The first and third arguments must be character strings. The second argument must be an integer.

Result

The result is a character string. If *n* is positive, the result is the *n*th word of *string*. If *n* is negative, the result is the *n*th word of *string*, counting from the end. If *n* is 0, or if the string contains fewer than *n* words, the result is an empty string.

The function treats *string* as a sequence of words separated by delimiters. A delimiter is any byte present in *delimiters*, or a blank if the *delimiters* argument is absent or empty. A word is any substring not containing delimiters, preceded by a delimiter (or start of string), and followed by a delimiter (or end of string).

If a blank is specified as a delimiter, a whole sequence of adjacent blanks is counted as one delimiter.

The WORD function recognizes double-byte characters. Since a delimiter is a one-byte character, it is recognized only within a single-byte sequence.

Example

```
WORD('A   B',2,' ') = 'B'
WORD('A,,,B',2,',') = '
WORD('A,,,B',-1,',') = 'B'
```

YEAR

The YEAR function obtains the year part of a value.

Syntax

```
►►──YEAR (expression)─────────────────────────────────────────────────────────►◄
```

The argument must be a date or a timestamp.

Result

The result is an integer between 1 and 9 999. It is the year part of the argument.

Example

Assume that:

- X_DATE has the value February 11, 1993.
- X_TSTAMP has the value 15 hours, 33 minutes, 25 seconds, and 75 microseconds on June 20, 1993.

The function produces these results:

```
YEAR(X_DATE) = 1993
YEAR(X_TSTAMP) = 1993
```

Chapter 11. Log collector language statements

The log collector language consists of statements that you use to determine how data is collected, processed, and stored. It also provides statements that you can use to maintain data tables and to perform the collection process.

This chapter provides an alphabetical listing of the language statements. For each statement, the chapter describes:

- The purpose of the statement
- The syntax used for the statement
- Parameters (clauses and keywords) that are part of the statement
- Examples of how to use the statement
- Usage notes, if needed, that explain issues to consider when using the statement.

ALTER LOG

Use the ALTER LOG statement to modify a stored log definition. You can add, change, or delete a:

- Header
- Timestamp expression
- First record or last record condition
- Log procedure name and the parameters passed to the log procedure

This description assumes that you are familiar with log definitions and the DEFINE LOG statement. So, it explains only how ALTER LOG modifies the log definition. It does not explain what the modification means. The syntax diagram shows all the clauses that you can specify, but they are explained only as much as it is needed to tell what is altered. See “DEFINE LOG” on page 11-15 for more information.

Syntax

```
                                        ┌─,─────────┐
                                        ▼           │
►►──ALTER LOG──log-name──┬─HEADER───┬─(───┤ field ├─┴─)─┬──────────────────────┬──────►◄
                         │          └─NONE──────────────┘                      │
                         ├─TIMESTAMP───┬─expression─┬────────────────────────────┤
                         │             └─NONE───────┘                            │
                         ├─FIRST RECORD───┬─condition─┬─────────────────────────┤
                         │                └─NONE──────┘                          │
                         ├─LAST RECORD───┬─condition─┬──────────────────────────┤
                         │               └─NONE──────┘                           │
                         └─LOGPROC───┬─procedure-name──┤ procedure-parms ├─┬─────┘
                                     └─NONE────────────────────────────────┘
```

procedure-parms:

```
├──┬───────────────────────────┬──┬─────────────────────────┬───────────────────────┤
   │              ┌─ASM──┐     │  └─PARM───┬─expression─┬──┘
   └─LANGUAGE─────┼─ASML─┼─────┘           └─NONE───────┘
                  └─C────┘
```

field:

```
├──┬─field-name─┬──┬───────────────────────────┬─────────────────────────►
   └─*──────────┘  └─OFFSET──integer-constant──┘

►──┬─────────────────────────────────────────────────────────┬───────────┤
   └─LENGTH──┬─integer-constant─┬──┬────────────────┬────────┘
             └─*────────────────┘  └─field-format───┘
```

Parameters

log-name
Identifies the log definition that you want to alter.

HEADER (*field, ...***)**
Adds or replaces the header definition. The header fields are specified in the same way as in the DEFINE LOG statement. If a header is already defined for the log, the entire header definition is replaced.

HEADER NONE
Deletes all header fields.

TIMESTAMP *expression*
Adds or replaces the timestamp expression.

TIMESTAMP NONE
Deletes the timestamp expression.

FIRST RECORD *condition*
Adds or replaces the first record condition.

FIRST RECORD NONE
Deletes the first record condition.

LAST RECORD *condition*
Adds or replaces the last record condition.

LAST RECORD NONE
Deletes the last record condition.

LOGPROC *procedure-name procedure-parms*
Adds or modifies the LOGPROC clause.

If no log procedure is defined for the log, a log procedure definition is added. You use then *procedure-name* and *procedure-parms* in the same way as in the DEFINE LOG statement. If a log procedure is already defined for the log, you use *procedure-name* and *procedure-parms* to alter the existing log procedure definition:

procedure-name
Replaces the log procedure name. If you do not want to alter the name, you must code here the same name as already defined.

LANGUAGE
Alters the language specification. An omitted LANGUAGE means no change (**not** LANGUAGE ASM).

PARM *expression*
Adds or replaces the parameter expression.

PARM NONE
Deletes the parameter expression.

LOGPROC NONE
Deletes the log procedure definition.

Examples

Assume you want to add F_FIELD=1 as the first record condition for a log called SOME_LOG. Use this ALTER LOG statement to add the condition:

```
ALTER LOG SOME_LOG FIRST RECORD F_FIELD=1;
```

Figure 11-1. ALTER LOG statement

Usage

When you use the ALTER LOG statement, you specify only a part of the log definition. You cannot see the complete definition. This makes the change difficult if the definition is complex. It may be more convenient to delete the entire definition using a DROP statement and then store a modified definition using a DEFINE LOG statement.

ALTER RECORD

Use the ALTER RECORD statement to modify a stored record definition. You can add, change, or delete:
- Fields
- Sections
- Condition that identifies the record
- Name of the record procedure that builds the records

This description assumes that you are familiar with record definitions and the DEFINE RECORD statement. Therefore, it explains only how ALTER RECORD modifies the record definition. It does not explain what the modification means. The syntax diagram shows all the clauses that you can specify, but they are explained only as much as it is needed to tell what is altered. See "DEFINE RECORD" on page 11-19 for more information.

Syntax

```
►►──ALTER RECORD──record-name──────────────────────────────────────────►

►──┬─IN LOG──log-name────────────────────────────────────┬──────────►◄
   ├─BUILT BY──┬─procedure-name─┬────────────────────────┤
   │           └─NONE───────────┘                        │
   ├─IDENTIFIED BY──┬─condition─┬────────────────────────┤
   │                └─NONE──────┘                        │
   │                 ┌─,─────────┐                       │
   │                 ▼           │                       │
   ├─ADD FIELDS── (──┤ field ├───┴─)─┬───────────────────────────┬─┤
   │                                 └─IN SECTION──section-name─┘ │
   ├─DELETE FIELD──field-name────────────────────────────┤
   ├─ADD SECTION──┤ section ├────────────────────────────┤
   ├─DELETE SECTION──section-name────────────────────────┤
   └─SECTION──section-name──┤ section-attributes ├────────┘
```

section:

```
├──section-name──┬──────────────────────────┬──┬─────────────────────┬──────────►
                 └─IN SECTION──section-name─┘  └─PRESENT IF──condition─┘

►──┬───────────────────┬──┬───────────────────┬──┬─────────────────────────┬────►
   └─OFFSET──expression─┘  └─LENGTH──expression─┘  └─NUMBER──┬─expression─┬─┘
                                                            └─*──────────┘

►──┬──────────┬──┬──────────────────────────────┬──────────────────────────────┤
   └─REPEATED─┘  │               ┌─,──────────┐ │
                 │               ▼            │ │
                 └─FIELDS──(─────┤ field ├────┴──)─┘
```

section-attributes:

```
├──┬───────────────────────────────┬──┬─────────────────────────────┬──────────►
   └─IN SECTION──┬─section-name─┬──┘  └─PRESENT IF──┬─condition─┬──┘
                 └─NONE─────────┘                   └─NONE──────┘

►──┬─────────────────────────┬──┬─────────────────────────┬────────────────────►
   └─OFFSET──┬─expression─┬──┘  └─LENGTH──┬─expression─┬──┘
             └─NONE───────┘               └─NONE───────┘

►──┬─────────────────────────┬──┬─────────────┬────────────────────────────────┤
   └─NUMBER──┬─expression─┬──┘  ├─REPEATED────┤
             ├─*──────────┤     └─NONREPEATED─┘
             └─NONE───────┘
```

field:

```
├──┬─field-name─┬──┬───────────────────────────┬────────────────────────────────►
   └─*──────────┘  └─OFFSET──integer-constant──┘

►──┬───────────────────────────────────┬──┬──────────────┬──────────────────────┤
   └─LENGTH──┬─integer-constant─┬──────┘  └─field-format─┘
             └─*────────────────┘
```

Parameters

record-name
: Identifies the record definition that you want to alter.

IN LOG *log-name*
: Replaces the IN LOG clause.

BUILT BY *procedure-name*
: Adds or replaces the BUILT BY clause.

BUILT BY NONE
: Deletes the BUILT BY clause.

IDENTIFIED BY *condition*
: Adds or replaces the IDENTIFIED BY condition.

IDENTIFIED BY NONE
: Deletes the IDENTIFIED BY condition.

ADD FIELDS (*field*, ...)
Adds or replaces one or more fields. The fields are specified in the same way as in the DEFINE RECORD statement.

IN SECTION *section-name*
Specifies the section where to add the fields. An omitted IN SECTION means that you want to add fields in the record.

DELETE FIELD *field-name*
Deletes the field *field-name*.

ADD SECTION *section*
Adds a section. The section is specified in the same way as in the DEFINE RECORD statement.

DELETE SECTION *section-name*
Deletes the specified section.

SECTION *section-name*
Modifies the attributes of section *section-name*.

IN SECTION *section-name*
Adds or replaces the IN SECTION attribute.

IN SECTION NONE
Deletes the IN SECTION attribute.

PRESENT IF *condition*
Adds or replaces the PRESENT IF condition.

PRESENT IF NONE
Deletes the PRESENT IF condition.

OFFSET *expression*
Adds or replaces the OFFSET attribute.

OFFSET NONE
Deletes the OFFSET attribute.

LENGTH *expression*
Adds or replaces the LENGTH attribute.

LENGTH NONE
Deletes the LENGTH attribute.

NUMBER *expression* **;NUMBER ***
Adds or replaces the NUMBER attribute.

NUMBER NONE
Deletes the NUMBER attribute.

REPEATED
Specifies the section to be repeated.

NONREPEATED
Specifies the section to be non-repeated.

Examples

Assume that you want to add a 10-byte character field at offset 52 in a section called SUB_SECT. Use the following ALTER RECORD statement to add the field.

```
ALTER RECORD SOME_REC
  ADD FIELDS (NEW_FIELD OFFSET 52 CHAR(10)) IN SECTION SUB_SECT;
```

Figure 11-2. ALTER RECORD statement

For more examples of how to use the ALTER RECORD statement, see "Using the ALTER RECORD statement" on page 3-8.

Usage

When you use the ALTER RECORD statement, you specify only a part of the record definition. You cannot see the complete definition. This makes the change difficult if the definition is complex. It may be more convenient to delete the entire definition using a DROP statement and then store a modified definition using a DEFINE RECORD statement.

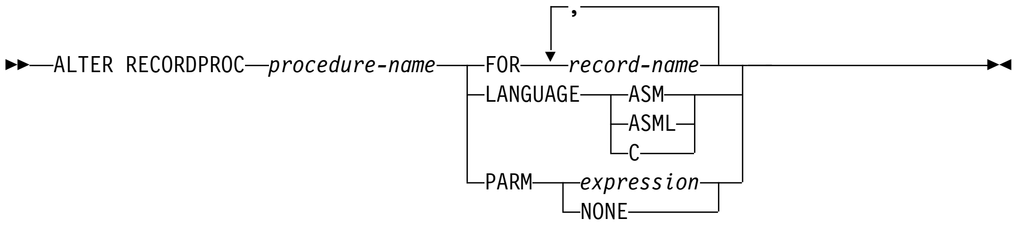
ALTER RECORDPROC

Use the ALTER RECORDPROC statement to modify a stored record procedure definition. You can change the:

- Record types for which the record procedure applies
- Language in which the record procedure is written
- Parameters that are passed to the record procedure

This description assumes that you are familiar with record procedures and the DEFINE RECORDPROC statement. So, it explains only how ALTER RECORDPROC modifies the definition of a record procedure. It does not explain what the modification means. See "DEFINE RECORDPROC" on page 11-28 for more information.

Syntax



Parameters

procedure-name
: Identifies the record procedure definition that you want to alter.

FOR *record-name* **, ...**
: Replaces the list of records processed by the procedure.

LANGUAGE
: Replaces the language specification of the procedure.

PARM *expression*
: Adds or replaces the parameter expression.

PARM NONE
: Deletes the parameter expression.

Examples

Assume that you want to change the record procedure definition DRL2CIC1 so the log collectorlog collector will invoke the exit with the C language interface. Use this ALTER RECORDPROC statement to change the language interface:

```
ALTER RECORDPROC DRL2CIC1 LANGUAGE C;
```

Figure 11-3. ALTER RECORDPROC statement

Usage

When you use the ALTER RECORDPROC statement, you specify only a part of the record procedure definition. You cannot see the complete definition. This makes the change difficult if the definition is complex. It may be more convenient to delete the entire definition using a DROP statement and then store a modified definition using a DEFINE RECORDPROC statement.

ALTER UPDATE

Use the ALTER UPDATE statement to modify a stored update definition. You can add, change, or delete:

- The SECTION clause
- The WHERE clause
- Parts of the LET, GROUP BY, or SET clauses
- The APPLY SCHEDULE, DISTRIBUTE, or MERGE clauses

This description assumes that you are familiar with update definitions and the DEFINE UPDATE statement. So, it explains only how ALTER UPDATE modifies the update definition. It does not explain what the modification means. The syntax diagram shows all the clauses that you can specify, but they are explained only as much as it is needed to tell what is altered. See “DEFINE RECORDPROC” on page 11-28 for more information.

Syntax

```
►►──ALTER UPDATE──update-name───┬─SECTION──┬─section-name─┬─────────────────┬──────►◄
                                │          └─NONE─────────┘                 │
                                ├─WHERE──┬─condition─┬──────────────────────┤
                                │        └─NONE──────┘                      │
                                ├─┤ apply-schedule-clause ├─────────────────┤
                                ├─APPLY SCHEDULE NONE───────────────────────┤
                                ├─┤ distribute-clause ├─────────────────────┤
                                ├─DISTRIBUTE NONE───────────────────────────┤
                                ├─LET identifier = ──┬─expression─┬─────────┤
                                │                    └─NONE───────┘         │
                                ├─GROUP BY column-name = ──┬─expression─┬───┤
                                │                          └─NONE───────┘   │
                                ├─SET column-name = ──┬─┤ accumulation ├─┬──┤
                                │                     └─NONE─────────────┘  │
                                ├─┤ merge-clause ├──────────────────────────┤
                                └─MERGE NONE────────────────────────────────┘
```

accumulation:

```
├──┬─SUM (expression)────────────────────────────────────────┬──────────────────────┤
   ├─MIN (expression)────────────────────────────────────────┤
   ├─MAX (expression)────────────────────────────────────────┤
   ├─COUNT (expression)──────────────────────────────────────┤
   ├─FIRST (expression)──────────────────────────────────────┤
   ├─LAST (expression)───────────────────────────────────────┤
   ├─AVG (expression, column-name)───────────────────────────┤
   └─PERCENTILE (expression, column-name, integer-constant)──┘
```

apply-schedule-clause:

```
├──APPLY SCHEDULE──expression──TO column-name, column-name, column-name──────────►

►─STATUS──identifier────────────────────────────────────────────────────────────┤
```

distribute-clause:

```
                ┌─,─────────────┐
                ▼               │
├──DISTRIBUTE───┬─field-name──┬─┴──BY──expression──START──expression────────────►
                └─column-name─┘

►─END──expression──TIMESTAMP──identifier──INTERVAL──identifier──────────────────┤
```

merge-clause:

```
├──MERGE──(──column-name = expression──,──column-name = expression──,───────────►

►─column-name = expression──,──column-name = expression──)──────────────────────┤
```

Parameters

update-name
Identifies the update definition that you want to alter.

SECTION *section-name*
Adds or replaces the SECTION clause.

SECTION NONE
Deletes the SECTION clause.

WHERE *condition*
Adds or replaces the WHERE condition.

WHERE NONE
Deletes the WHERE condition.

apply-schedule-clause
Adds or replaces the APPLY SCHEDULE clause. The clause is specified as in the DEFINE UPDATE statement.

APPLY SCHEDULE NONE
Deletes the APPLY SCHEDULE clause.

distribute-clause

Adds or replaces the DISTRIBUTE clause. The clause is specified as in the DEFINE UPDATE statement.

DISTRIBUTE NONE

Deletes the DISTRIBUTE clause.

LET *identifier* = *expression*

Adds or replaces the specification of *identifier* in the LET clause.

LET *identifier* = **NONE**

Deletes the specification of *identifier* from the LET clause.

GROUP BY *column-name* = *expression*

Adds or replaces the specification of column *column-name* in the GROUP BY clause.

GROUP BY *column-name* = **NONE**

Deletes the specification of column *column-name* from the GROUP BY clause.

SET *column-name* = *accumulation*

Adds or replaces the specification of column *column-name* in the SET clause.

SET *column-name* = **NONE**

Deletes the specification of column *column-name* from the SET clause.

merge-clause

Adds or replaces the MERGE clause. The clause is specified as in the DEFINE UPDATE statement.

MERGE NONE

Deletes the MERGE clause.

Examples

The update definition UPD_WKLD applies to records of type WKLD_REC, which contain a field WRKFLD. Assume that you want to change UPD_WKLD so that it will only apply to WKLD_REC records when WRKFLD is not equal to 120. Use this ALTER UPDATE statement to change the condition to which UPD_WKLD applies:

```
ALTER UPDATE UPD_WKLD
  WHERE WRKFLD <> 120;
```

Figure 11-4. ALTER UPDATE statement

For more information about using the ALTER UPDATE statement, see "Changing and deleting update definitions" on page 5-27.

Usage

When you use the ALTER UPDATE statement, you specify only a part of the update definition. You cannot see the complete definition. This makes the change difficult if the definition is complex. It may be more convenient to delete the entire definition using a DROP statement and then store a modified definition using a DEFINE UPDATE statement.

COLLECT

Use the COLLECT statement to collect log data. The processing is controlled by the stored definitions of the log, records, and updates.

Syntax

```
►►──COLLECT──log-name──┬──────────────────────┬──┬───────────────────────┬──────────────►
                       └─FROM──file-name──────┘  └─WHERE──condition──────┘

►──┬──────────────────────────────────────────────────┬───────────────────────────────►
   │              ┌─,──────────────────────────┐     │
   │              ▼                            │     │
   └─INCLUDE──────┬─table-name─────────────────┴─────┘
                  └─LIKE──string-constant──────┘

►──┬──────────────────────────────────────────────────┬──┬─────────────┬──────────────►
   │              ┌─,──────────────────────────┐     │  └─REPROCESS───┘
   │              ▼                            │     │
   └─EXCLUDE──────┬─table-name─────────────────┴─────┘
                  └─LIKE──string-constant──────┘

►──┬───────────────────────────────────┬──┬─────────────┬──┬──────────┬──┬────────┬───►
   └─ON TIMESTAMP OVERLAP──┬─SKIP─┬────┘  └─PARTITION───┘  └─DIRECT───┘  └─SCAN───┘
                           └─STOP─┘

►──┬──────────────────────────────────────────────────────┬───────────────────────────►
   │                ┌─BUFFER FULL─────────────────────┐   │
   └─COMMIT AFTER───┼─END OF FILE─────────────────────┼───┘
                    ├─integer-constant──RECORDS───────┤
                    └─BUFFER FULL ONLY────────────────┘

►──┬────────────────────────────────────────────┬──┬──────────────────────────────┬──►◄
   │                                  ┌─M─┐     │  │                ┌─BREAK────┐  │
   └─BUFFER SIZE──integer-constant────┴─K─┴─────┘  └─ON OVERFLOW────┴─CONTINUE─┴──┘
```

Parameters

log-name
: Is the name of a stored log definition. It identifies the type of log to be collected.

FROM *file-name*
: Names the DD statement that specifies the log data set to be collected. The default is DRLLOG.

WHERE *condition*
: Limits processing to those records for which the *condition* is true. Any identifiers used in the *condition* must be names of log header fields. If there is a log procedure, the condition applies to the records produced by the log procedure. Otherwise, the condition is applied to the records from the log. The condition is not applied to the records produced by record procedures.

INCLUDE
: Limits collect to update only the specified tables. If you specify INCLUDE, the log collectorlog collector does not update any other tables associated with this log definition.

table-name
Is the name of a table to be included.

LIKE *string-constant*
Specifies a group of tables to be included. The tables are those with names matching the pattern specified as the *string-constant*. The pattern matching rules are defined in “Pattern matching” on page 9-11. If the pattern contains a period (.), the table prefix must match the part before the period, and the rest of the table name must match the part after the period. For example, the pattern DRL.CICS% includes all tables whose names start with CICS and have DRL as the prefix.

If the pattern does not contain a period, the prefix must be the current user ID, and the rest of the name must match the whole pattern.

EXCLUDE
Prevents collect from updating the specified tables. If you specify EXCLUDE, the log collectorlog collector updates all other tables associated with this log definition.

table-name
Is the name of a table to be included.

LIKE *string-constant*
Specifies a group of tables to be excluded, using the same rules as for INCLUDE.

REPROCESS
Instructs the log collectorlog collector to collect data from the log data set in its entirety even if that data set has already been partially or completely processed.

ON TIMESTAMP OVERLAP SKIP
Instructs the log collector to collect data from a log with a matching DATASET_NAME entry in the DRLLOGDATASETS system table, skipping records included in the range indicated by the FIRST_TIMESTAMP and LAST_TIMESTAMP entries in DRLLOGDATASETS. This option can only be specified if a TIMESTAMP expression was specified in the definition for the log being collected.

Consider using this option if the log management procedures at your site allow for the possibility of logs with duplicate data set names being created with later logs supplementing or replacing the data from earlier logs, and you want the data in the later logs to be processed automatically.

For example, suppose log SMF.DAILY.D001 is created, but is incomplete, and only contains data from 8:00am to 11:00am. After the data has been collected by TDS, SMF.DAILY.D001 is re-created containing a full day's data.

If you collect this re-created log without specifying ON TIMESTAMP OVERLAP SKIP then the collect will succeed because the logs do not have matching first records, but data will be duplicated for the period that the logs overlap. If you specify ON TIMESTAMP OVERLAP SKIP when you collect the re-created log then the collect will succeed, but the log records from 8:00 am and 11:00 am will not be reprocessed.

If both REPROCESS and ON TIMESTAMP OVERLAP SKIP are specified then the log will be processed as if only REPROCESS had been specified.

ON TIMESTAMP OVERLAP STOP

Instructs the log collector to stop processing a log if the first record identified has a timestamp that is earlier than the LAST_TIMESTAMP entry in the DRLLOGDATASETS system table for a previously collected log with the same data set name.

Consider using this option if the log management procedures at your site allow for the possibility of logs with duplicate data set names being created that will not contain matching first records, and you want this to be treated as an error condition.

Note:

1. When ON TIMESTAMP OVERLAP SKIP is specified the log collector simply skips the records for any time period that overlaps a previously collected log with the same data set name. The results of this process may not always match exactly the results obtained when the same overall set of records is collected in a single pass.
2. ON TIMESTAMP OVERLAP SKIP cannot be used in conjunction with collection methods that use REPROCESS with INCLUDE/EXCLUDE to process the same log data set in multiple passes. If ON TIMESTAMP OVERLAP SKIP is used in this situation then skip processing will be performed successfully for the first recollect of the log, but if that collect is successful then all subsequent collects will skip over the entire log.
3. ON TIMESTAMP OVERLAP SKIP is mutually exclusive with REPROCESS. The REPROCESS option is intended for reprocessing a previously processed set of data. ON TIMESTAMP OVERLAP is intended for processing (recreated) data that overlaps previously processed data.

PARTITION

Specifies when COLLECT must collect on the PARTITIONING feature only. It allows jobs which fill in different partitions of the same tables to run in parallel to improve COLLECT performance. Do not use this parameter in COLLECT jobs that fill in NON-PARTITIONED tables.

Specifying the PARTITION parameter also instructs the log collector to use the DIRECT algorithm for database updates. If you set both the PARTITION and SCAN parameters, the SCAN parameter is overridden and the DIRECT algorithm is used.

DIRECT or SCAN

Instructs the log collector to use either the DIRECT or the SCAN algorithm to update the DB2 database. This means that the log collector does not run a query to select the algorithm to be used, and performance is therefore improved.

To decide which parameter to specify, refer to message DRL0356I in the output log of previous collects that were run without specifying either DIRECT or SCAN.

If you set both the SCAN and PARTITION parameters, the SCAN parameter is overridden and the DIRECT algorithm is used.

If you do not specify DIRECT (or PARTITION) or SCAN then the log collector will run a query to select which algorithm to use. In this case if the rows in the collect buffer do not match with any rows in the database for a table, an insert algorithm is used.

COMMIT AFTER

Specifies when the log collectorlog collector should execute COMMIT to make the database updates permanent. (The log collectorlog collector always writes its internal buffer to the DB2 database before issuing COMMIT, but it may write the buffer to the database without committing the updates.)

If a collect abends after a commit, at least part of the data set has been successfully processed. The log collectorlog collector can automatically resume collecting data from the log at the point where the log collectorlog collector made the commit.

The possible options follow.

BUFFER FULL

Commit in each of these situations:
- After the internal buffer was filled and written to the database
- At the end of each concatenated log data set
- After the entire input was processed

END OF FILE

Commit only after the entire input has been processed.

integer-constant **RECORDS**

Commit only after processing the number of records specified by the *integer-constant*. This option results in the longest execution time, compared with other COMMIT AFTER options.

BUFFER FULL ONLY

Commit in each of these situations:
- After the internal buffer was filled and written to the database
- After the entire input was processed

This option will normally result in a shorter execution time when processing concatenated log data sets.

BUFFER SIZE *integer-constant*

Specifies the size (in bytes) of the internal collect buffer. The default is 10M bytes. The minimum allowed value is 10K bytes. The maximum size of the internal collect buffer is limited to the virtual storage available when the log collectorlog collector executes. If you specify a BUFFER SIZE that exceeds the available virtual storage, the log collectorlog collector abends.

Note: The log collector sometimes requires more buffer space than you specify. It abends if it cannot obtain the extra space.

ON OVERFLOW

Specifies the action to be taken in case of an overflow. An overflow is a situation when a numeric value accumulated in a table column becomes too large for that column. The possible options follow.

BREAK

Stop data collection. Do not update the database.

CONTINUE

Reset the column to 0, write the lost value to the DRLDUMP file, and continue data collection.

Examples

Assume that you want to update only two tables, NETWORK_SESSIONS and DB2_ACCOUNTING, with data for the MVS1 system from an SMF log data set named by the SMFLOG DD statement. You also want to write the internal buffer to the database and commit the change after every 5 000 records in the log data set have been processed.

```
COLLECT SMF
  FROM SMFLOG
  WHERE SMFSID = 'MVS1'
  INCLUDE NETWORK_SESSIONS, DB2_ACCOUNTING
  COMMIT AFTER 5000 RECORDS;
```

Figure 11-5. COLLECT statement

For more information about using the COLLECT statement, see Chapter 6, “Collecting log data,” on page 6-1.

Usage

You can specify both INCLUDE and EXCLUDE on a COLLECT statement. For example, INCLUDE LIKE 'DRL.CICS%' EXCLUDE DRL.CICS_APPL_H includes all CICS tables except DRL.CICS_APPL_H.

COMMENT ON

Use the COMMENT ON statement to add or replace comments in stored definitions. You can add or replace comments for:

- Log definitions
- Record definitions
- Record procedure definitions
- Update definitions

Syntax

```
►►──COMMENT ON──┬─LOG──log-name──────────────────┬──IS──string-constant──────────►◄
                ├─RECORD──record-name────────────┤
                ├─FIELD──record-name.field-name──┤
                ├─RECORDPROC──procedure-name─────┤
                └─UPDATE──update-name────────────┘
```

Parameters

LOG *log-name*
: Specifies that the comment applies to the log definition *log-name*.

RECORD *record-name*
: Specifies that the comment applies to the record definition *record-name*.

FIELD *record-name.field-name*
: Specifies that the comment applies to the field *field-name* in the record definition *record-name*.

RECORDPROC *procedure-name*
: Specifies that the comment applies to the record procedure definition *procedure-name*.

UPDATE *update-name*
Specifies that the comment applies to the update definition *update-name*.

IS *string-constant*
Specifies the comment text, which can be any character string up to 254 characters long.

Examples

Assume you want to add a comment to the field WRKLD in the record type HOUR_REC. Use this COMMENT ON statement to add the comment:

```
COMMENT ON FIELD HOUR_REC.WRKLD
   IS 'New definition for the WRKLD field';
```

Figure 11-6. COMMENT ON statement

Usage

- The comments stored using the COMMENT ON statement can be viewed using the administration dialog. (See the , SH19-6816.)
- To store comments for the DB2 tables and their columns, use the COMMENT ON statement that is part of SQL.
- Notice that the double minus sign (--) comments and the slash asterisk (/*) asterisk slash (*/) comments are **not** stored with your definitions.

DEFINE LOG

Use the DEFINE LOG statement to define a log type.

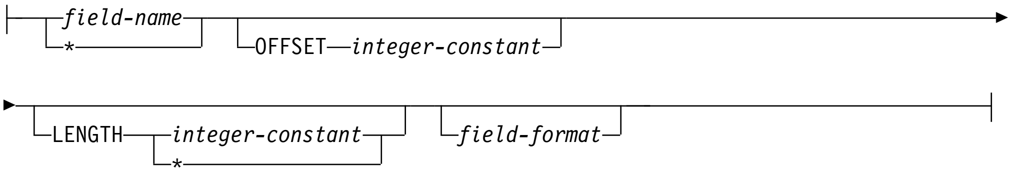
Syntax

```
►►──DEFINE LOG──log-name──┬──────────────────────────┬──────────────────────►
                          └─VERSION──string-constant─┘

►──┬───────────────────────┬──┬──────────────────────┬──────────────────────►
   │               ┌─,───┐ │  └─TIMESTAMP──expression─┘
   │               ▼     │ │
   └─HEADER──(─────field─┴─)─┘

►──┬─────────────────────────┬──┬────────────────────────┬──────────────────►
   └─FIRST RECORD──condition─┘  └─LAST RECORD──condition─┘

►──┬──────────────────────────────────────────────────┬───────────────────►◄
   └─LOGPROC──procedure-name──┤ procedure-parms ├─┘
```

procedure-parms:

```
├──┬──────────────────────┬──┬─────────────────┬──────────────────────────┤
   │           ┌─ASM──┐   │  └─PARM──expression─┘
   └─LANGUAGE──┼─ASML─┼───┘
               └─C────┘
```

field:



Parameters

log-name
The name of the log type being defined. It must be an identifier, at most 16 bytes long. All log types defined to the log collectorlog collector must have distinct names.

VERSION *string-constant*
The string specified by the *string-constant* is stored together with the definition, to identify the statement that was used to create the definition. The string can be at most 18 bytes long. Omitted VERSION means the same as specifying `VERSION '`.

If the stored definition is later altered by means of an ALTER LOG statement, its version identification is changed to 'ALTERED'.

Version names are used by the Tivoli Decision Support for z/OS installation program to decide which definitions should be replaced. All definitions supplied by IBM have version names starting with 'IBM'. To ensure correct installation of new releases, do not use such names for your own definitions. See Figure 11-7 on page 11-18 for an example of how VERSION is used by IBM. Refer to the *Administration Guide and ReferenceAdministration Guide and Reference* for information on how to use VERSION.

HEADER (*field***, ...)**
Defines the fields that are common to all records. There is a limit of 2 000 fields in a header definition.

field Defines one field.

This general rule applies to all fields.

The LENGTH and OFFSET (explicit or default) define a field as an area so many bytes long, starting at a specific place in the record. If the record is too short to contain ALL bytes of a field, the field is considered absent and a reference to it produces null value.

The above rule has one exception: LENGTH *. The asterisk length means that the field extends up to the end of the record. The field is absent if the record is too short to contain the first byte of the field.

field-name
The name of the field. It can be any identifier. Field names must be unique within a header.

OFFSET *integer-constant*
Defines the offset of the field in the record. Notice that offsets in varying-length records (record format V, VB, or VBS) include the 4-byte record descriptor word.

If you omit OFFSET, the field starts at the end of the field defined just before it. The preceding field cannot have an asterisk length. If you omit the offset for the first field in the list, that field begins at offset 0.

LENGTH *integer-constant*

Defines the length of the field in bytes. The allowed lengths depend on the format of the field. See Table 11-1 on page 11-23; the Length column states the possible length(s) of the field.

If you omit LENGTH, the log collectorlog collector uses the default length depending on the field format. If the Length column in Table 11-1 on page 11-23 specifies a single value, this is the default. Otherwise the default is stated in the column.

LENGTH *

Indicates that the field extends up to the end of the record.

field-format

Specifies the format of the data contained in the field. The possible values of *field-format* are listed in Table 11-1 on page 11-23, in the Field format column. The Data type column states the data type to which the log collectorlog collector automatically converts the content of the field when it uses that field.

If you omit the field format, the field format is HEX.

TIMESTAMP *expression*

Describes how the timestamp of the records is derived from the fields in the header. The log collectorlog collector prints the timestamp of the first and last processed records in the log data set (and saves these timestamps in a system table) to identify which time period the log covers.

The result of *expression* must be a timestamp. Any identifiers used in the *expression* must be names of log header fields.

FIRST RECORD *condition*

Specifies a condition that the first record in the log data set should satisfy. If this condition is not met, the log collectorlog collector gives a warning message.

Any identifiers used in the *condition* must be names of log header fields.

LAST RECORD *condition*

Specifies a condition that the last record in the log data set should satisfy. If this condition is not met, the log collectorlog collector gives a warning message.

Any identifiers used in the *condition* must be names of log header fields.

LOGPROC *procedure-name*

Identifies the log procedure for the log. The log procedure must be a load module, available in a load library under the name *procedure-name*.

LANGUAGE

Specifies the interface to the procedure: the language, linkage convention, and parameters.

ASM The procedure is written in Assembler and is called using standard System/390 linking conventions. It returns length of the output record in a field within the record.

ASML The procedure is written in Assembler and is called using standard System/390 linking conventions. It returns length of the output record in a parameter.

C The procedure is written in C and is called using persistent C environment. It returns length of the output record in a field within the record.

See Appendix A, “Log and record procedures,” on page A-1 for details.

PARM *expression*

Specifies an *expression* that the log collectorlog collector evaluates and passes to the log procedure the first time it is called. The procedure must understand the format and interpret the value derived from the *expression*.

The *expression* cannot contain identifiers. The result of *expression* must be an integer, a floating-point number, or a character string.

Examples

Define a log type named SMF, identifying fields where the log collectorlog collector can obtain timestamp information. Also, tell the log collectorlog collector how to determine the first and last records in this log.

```
DEFINE LOG XMP
  VERSION 'IBM.120'
  HEADER(XMPLEN LENGTH 2 BINARY,
         XMPSEG LENGTH 2 BINARY,
         XMPFLG LENGTH 1 BIT,
         XMPRTY LENGTH 1 BINARY,
         XMPTME          TIME(1/100S),
         XMPDTE          DATE(0CYYDDDF),
         XMPSID          CHAR(4),
         XMPSSI          CHAR(4),
         XMPSTY LENGTH 2 BINARY)
  TIMESTAMP TIMESTAMP(XMPDTE,XMPTME)
  FIRST RECORD XMPRTY = 2
  LAST  RECORD XMPRTY = 3;
```

Figure 11-7. DEFINE LOG statement

For more information about using the DEFINE LOG statement, see “Defining a log” on page 2-3 and “Verifying log data sets during data collection” on page 6-4.

DEFINE PURGE

Use the DEFINE PURGE statement to store a purge condition. The stored condition is used by the PURGE statement to determine which data should be purged (see “PURGE” on page 11-47).

Syntax

```
►►──DEFINE PURGE──┬──────────────────────────┬──FROM──table-name──────────────►
                  └─VERSION──string-constant─┘

►──┬─────────────────────┬─────────────────────────────────────────────────────►◄
   └─WHERE──sql-condition─┘
```

Parameters

VERSION *string-constant*
The string specified by the *string-constant* is stored together with the purge condition, to identify the statement that was used to create the condition. The string can be at most 18 bytes long. Omitted VERSION means the same as specifying `VERSION '`.

Version names are used by the Tivoli Decision Support for z/OS installation program to decide which definitions should be replaced. All definitions supplied by IBM have version names starting with 'IBM'. To ensure correct installation of new releases, do not use such names for your own definitions. See Figure 11-8 for an example of how VERSION is used by IBM. Refer to the *Administration Guide and ReferenceAdministration Guide and Reference* for information on how to use VERSION.

FROM *table-name*
Specifies the name of the table to which the purge condition applies.

WHERE *sql-condition*
Specifies which data in the table should be deleted by PURGE. The *sql-condition* is executed by the database manager.

The *sql-condition* must be a valid SQL search condition for the table *table-name*, and its individual tokens must be recognized by the log collectorlog collector.

Examples

Write a DEFINE PURGE statement to define conditions so that the log collectorlog collector deletes data in the IMS_TRANSACTIONS_H table when the APPLICATION is not equal to ACCT and the date in the data is more than 7 days old, or when the APPLICATION is ACCT and the data is more than 14 days old:

```
DEFINE PURGE
  VERSION 'IBM.120'
  FROM IMS_TRANSACTIONS_H
  WHERE APPLICATION <> 'ACCT' AND DATE < CURRENT DATE - 7 DAYS
     OR APPLICATION =  'ACCT' AND DATE < CURRENT DATE - 14 DAYS;
```

Figure 11-8. Example of the DEFINE PURGE statement

Usage

You can have at most one purge condition for each table. If you execute a DEFINE PURGE statement for a table that already has a purge condition, the new condition replaces the old condition.

DEFINE RECORD

Use the DEFINE RECORD statement to define a record type.

Syntax

```
►►──DEFINE RECORD──record-name──┬───────────────────────────┬──IN LOG──log-name──────►
                                └─VERSION──string-constant──┘

►──┬───────────────────────────┬──┬─────────────────────────┬─────────────────────────►
   └─BUILT BY──procedure-name──┘  └─IDENTIFIED BY──condition─┘

►──┬───────────────────────────────┬──┬──────────────────────────┬────────────────────►◄
   │            ┌─,─────────┐      │  └─SECTION──┤ section ├─────┘
   │            ▼           │      │
   └─FIELDS──(──┤ field ├───┴──)───┘
```

section:

```
├──section-name──┬──────────────────────────────┬──┬──────────────────────┬───────────►
                 └─IN SECTION──section-name─────┘  └─PRESENT IF──condition─┘

►──┬────────────────────┬──┬────────────────────┬──┬──────────────────────────┬───────►
   └─OFFSET──expression─┘  └─LENGTH──expression─┘  └─NUMBER──┬─expression─┬──┘
                                                             └─*──────────┘

►──┬──────────┬──┬───────────────────────────────┬────────────────────────────────────┤
   └─REPEATED─┘  │            ┌─,─────────┐      │
                 │            ▼           │      │
                 └─FIELDS──(──┤ field ├───┴──)───┘
```

field:

```
├──┬─field-name─┬──┬──────────────────────────┬────────────────────────────────────────►
   └─*──────────┘  └─OFFSET──integer-constant─┘

►──┬─────────────────────────────────────┬──┬──────────────┬─────────────────────────┤
   └─LENGTH──┬─integer-constant─┬────────┘  └─field-format─┘
             └─*────────────────┘
```

Parameters

record-name

The name of the record type being defined. It can be any identifier, except that it cannot both start and end with an asterisk (*). All record types defined to the log collector must have distinct names.

VERSION *string-constant*

The string specified by the *string-constant* is stored together with the definition, to identify the statement that was used to create the definition. The string can be at most 18 bytes long. Omitted VERSION means the same as specifying `VERSION '`.

If the stored definition is later altered by means of an ALTER RECORD statement, its version identification is changed to 'ALTERED'. Version names are used by the Tivoli Decision Support for z/OS installation program to decide which definitions should be replaced. All definitions supplied by IBM have version names starting with 'IBM'. To ensure correct installation of new releases, do not use such names for your own definitions. See Figure 11-9 on page 11-27 for an example of how VERSION is used by IBM. Refer to the *Administration Guide and Reference* for information on how to use VERSION.

IN LOG *log-name*

Indicates that this record is encountered when processing log data sets of type *log-name* (and only such logs). The record is one of the records in the log, or is built by a record procedure from one of the records in the log (or from other records built from these).

BUILT BY *procedure-name*

Indicates that this record does not appear in the log data set, but is built by the record procedure *procedure-name*. An omitted BUILT BY clause means that the record appears in the log data set.

IDENTIFIED BY *condition*

Tells how to distinguish records of this type from other records. A record is of the type *record-name* if the *condition* is true. A specific record may satisfy the IDENTIFIED BY condition of several record definitions. The log collector uses then only one of these definitions (undefined which one).

Any identifiers used in the *condition* must be names of fields defined directly in the record (not in the sections within the record). An omitted IDENTIFIED BY clause is equivalent to specifying a condition that is true for every record.

SECTION *section*

Defines one section. There is a limit of 300 sections in a record. These clauses describe a section:

section-name

The name of the section. It can be any identifier. Section names must be unique within a record type.

IN SECTION *section-name*

Indicates that the section being defined is a subsection of the section named *section-name*. If you omit the IN clause, the section is a section of the record. The section *section-name* must be defined earlier in this record definition.

PRESENT IF *condition*

Indicates that the section is optional. The section is absent if the *condition* is not true. The section may be absent even if the condition is true, if the containing section (or record) is too short to contain the first byte of the section.

Any identifiers used in the *condition* must be names of fields in the section being defined, in the containing sections, in the record, or in previously defined non-repeated subsections of these.

OFFSET *expression*

Defines the offset of the section within the containing section (or record). Notice that offsets in varying-length records (record format V, VB, or VBS) include the 4-byte record descriptor word.

The *expression* must specify an integer ≥ 0. Any identifiers used in the *expression* must be names of fields in the containing sections, in the record, or in previously defined non-repeated subsections of these.

If you omit OFFSET, the section starts at the end of the most recently defined section with the same IN SECTION attribute. That section cannot be a repeated section. If no section with the same IN SECTION attribute has been previously defined, an omitted OFFSET means offset 0.

LENGTH *expression*

Defines the length of the section.

The *expression* must specify an integer > 0. Any identifiers used in the *expression* must be names of fields in the section being defined, in the containing sections, in the record, or in previously defined non-repeated subsections of these. If you omit LENGTH, the log collector assumes the minimum length needed to contain all named fields specified for this section.

If the containing section (or record) is too short to contain the whole section, the log collector assumes that the section extends up to the end of the containing section (or record). If the containing section (or record) is too short to contain the first byte of the section, the section is absent.

NUMBER *expression*

Defines the number of occurrences of the section.

The *expression* must specify an integer ≥ 0. Any identifiers used in the *expression* must be names of fields in the containing sections, in the record, or in previously defined non-repeated subsections of these.

An omitted NUMBER clause means the same as NUMBER 1.

NUMBER *

Defines the number of occurrences of the section to be as many occurrences as the containing section (or record) can hold.

REPEATED

Means that the section is repeated. If you omit REPEATED, the section is not repeated.

FIELDS (*field***, ...)**

Defines all fields of the record or section. There is a limit of 2 000 fields in a record.

field Defines one field.

This general rule applies to all fields. The LENGTH and OFFSET (explicit or default) define a field as an area so many bytes long, starting at a specific place in the record (or section). If the record (or section) is too short to contain ALL bytes of a field, the field is considered absent and a reference to it produces null value.

The above rule has one exception: LENGTH *. The asterisk length means that the field extends up to the end of the record (or section). The field is absent if the record (or section) is too short to contain the first byte of the field.

field-name

The name of the field. It can be any identifier. Field names must be unique within a record type.

OFFSET *integer-constant*

Defines the offset, in bytes, of the field in the record (or section). Notice that offsets in varying-length records (record format V, VB, or VBS) include the 4-byte record descriptor word.

If you omit OFFSET, the field starts at the end of the field defined just before it. The preceding field cannot have an asterisk length. If you omit the offset for the first field in the list, that field begins at offset 0.

LENGTH *integer-constant*

Defines the length of the field in bytes. The allowed lengths depend on the format of the field. See Table 11-1; the Length column states the possible length(s) of the field.

If you omit LENGTH, the log collector uses the default length depending on the field format. If the Length column in Table 11-1 specifies a single value, this is the default. Otherwise the default is stated in the column.

LENGTH *

Indicates that the field extends up to the end of the containing structure (record or section).

field-format

Specifies the format of the data contained in the field. The possible values of *field-format* are listed in Table 11-1, in the Field format column. The Data type column states the data type to which the log collector automatically converts the content of the field when it uses that field.

If you omit the field format, the field format is HEX.

Table 11-1. Field formats

Field format	Contents	Length in bytes	Data type
BINARY BINARY SIGNED BINARY UNSIGNED	Binary integer represented according to System/390 architecture. The default is SIGNED for lengths 2, 4, and 8, and UNSIGNED for lengths 1 and 3.	1,2,3,4,8 default 4	Integer for SIGNED of length≤4 and UNSIGNED of length≤3; otherwise floating-point
EXTERNAL HEX	A string of bits representing an integer in hexadecimal characters.	2,4,8 default 8	String
EXTERNAL INTEGER	A string of characters, representing an integer in the same format as for integer constants. Optional sign, leading and trailing blanks are allowed.	1 - 32 default 8	Integer
DECIMAL(*p*,*s*) where 1 ≤ *p*≤ 31 and 0 ≤ *s*≤ *p*	Packed decimal number of System/390 architecture, with precision *p* and scale *s*. The precision is the total number of decimal digits. Odd *p* means a signed number; even *p* means an unsigned number. The scale is the number of digits after the decimal point.	Integer part of (*p*+1)/2	Integer if *s*=0 and *p*≤9; otherwise floating-point

Table 11-1. Field formats (continued)

Field format	Contents	Length in bytes	Data type
ZONED(*p*,*s*) where 1 ≤ *p*≤ 31, and 0 ≤ *s*≤ *p*	Unsigned zoned decimal number of System/390 architecture, with precision *p* and scale *s*. The precision is the total number of decimal digits. The scale is the number of digits after the decimal point.	*p*	Integer if *s*=0 and *p*≤9; otherwise floating-point
FLOAT	A floating-point number of System/390 architecture, short (4 bytes) or long (8 bytes).	4, 8 default 8	Floating-point
EXTERNAL FLOAT	A string of characters expressing a floating-point number in the same format used for floating-point constants. Leading and trailing blanks are allowed.	1 - 32 default 8	Floating-point
CHAR	A string of characters. May include sequences of double-byte characters, enclosed between shift-out and shift-in characters.	1 - 254 default 1	String
CHAR(*n*) where 1 ≤ *n*≤ 254	A string of characters occupying *n* bytes. May include sequences of double-byte characters, enclosed between shift-out and shift-in characters.	*n*	String
CHAR(*)	A string of characters, extending up to the end of the containing structure. If the string is longer than 254 bytes, it is truncated. This format is only allowed with LENGTH *.	* (1-254)	String
VARCHAR	A string of characters including length information. The first two bytes contain the length *l* of the data as a binary integer; the remaining bytes contain the data itself. The length *l* may be 0, and cannot exceed the length of the field minus 2. The data portion of the string may include sequences of double-byte characters, enclosed between shift-out and shift-in characters.	3-256 default 8	String
BIT	A string of bits. Converted to string of characters "0" and "1" representing individual bits.	1 - 31 default 1	String

Table 11-1. Field formats (continued)

Field format	Contents	Length in bytes	Data type
BIT(*n*) where 8 ≤ *n*≤ 248, *n* multiple of 8	A string of *n* bits. Converted to string of characters "0" and "1" representing individual bits.	*n*/8	String
HEX	A string of bits. Converted to string of characters "0" through "F" representing the string in hexadecimal notation.	1 - 127 default 1	String
DATE(0CYYDDDF)	Date in the format *0cyydddF* (packed), where *c* indicates the century (0=1900, 1=2000), *yy* is the year within the century, *ddd* is the day within the year, and *F* can have any value. (*F* is ignored and is not checked to be a valid decimal sign).	4	Date
DATE(YYYYDDDF)	Date in the format *yyyydddF* (packed), where *yyyy* is the year, *ddd* is the day within the year, and *F* can have any value. (*F* is ignored and is not checked to be a valid decimal sign).	4	Date
DATE(YYDDDF)	Date in the format *yydddF* (packed), where *yy* is the year, *ddd* is the day within the year, and *F* can have any value. (*F* is ignored and is not checked to be a valid decimal sign).	3	Date
DATE(CYYMMDDF)	Date in the format *cyymmddF* (packed), where *c* indicates the century (0=1900, 1=2000), *yy* is the year within the century, *mm* is the month, *dd* is the day of month, and *F* can have any value. (*F* is ignored and is not checked to be a valid decimal sign).	4	Date
DATE(YYMMDD)	Date as character string *yymmdd*, where *yy* is the year, *mm* is the month, and *dd* is the day. *yy*≥50 means year 19*yy*; *yy*<50 means year 20*yy*.	6	Date
DATE(MMDDYY)	Date as character string *mmddyy*, where *mm* is the month, *dd* is the day, and *yy* is the year.	6	Date
DATE(MMDDYYYY)	Date as character string *mmddyyyy*, where *mm* is the month, *dd* is the day, and *yyyy* is the year.	8	Date

Table 11-1. Field formats (continued)

Field format	Contents	Length in bytes	Data type
TIME(1/100S)	A 32-bit binary integer representing time in hundredths of a second elapsed since hour 0.	4	Time
TIME(HHMMSSTF)	Time in the format *hhmmsstF* (packed), where *hh* is hours, *mm* is minutes, *ss* is seconds, *t* is tenths of a second, and *F* can have any value. (*F* is ignored and is not checked to be a valid decimal sign).	4	Time
TIME(0HHMMSSF)	Time in the format *0hhmmssF* (packed), where *hh* is hours, *mm* is minutes, *ss* is seconds, and *F* can have any value. (*F* is ignored and is not checked to be a valid decimal sign).	4	Time
TIME(HHMMSSXF)	Time in the format *hhmmssxF* (packed), where *hh* is hours, *mm* is minutes, *ss* is seconds, *x* is sixteenths of a second, and *F* can have any value. (*F* is ignored and is not checked to be a valid decimal sign).	4	Time
TIME(HHMMSSTH)	Time in the format *hhmmssth* (packed), where *hh* is hours, *mm* is minutes, *ss* is seconds, and *th* is hundredths of a second.	4	Time
TIME(HHMMSSU6)	Time in the format *hhmmssuuuuuu* (packed), where *hh* is hours, *mm* is minutes, *ss* is seconds, and *uuuuuu* is microseconds.	6	Time
TIME(HHMMSS)	Time as character string *hhmmss*, where *hh* is hours, *mm* is minutes, and *ss* is seconds.	6	Time
INTV(MMSSTTTF)	Time duration in the format *mmsstttF* (packed), where *mm* is minutes of duration, *ss* is seconds, *ttt* is milliseconds, and *F* can have any value. The duration is converted to milliseconds and expressed as an integer. (*F* is ignored and is not checked to be a valid decimal sign).	4	Integer

Table 11-1. Field formats (continued)

Field format	Contents	Length in bytes	Data type
TIMESTAMP(TOD)	Date and time in System/390 time-of-day (TOD) clock format: the number of microseconds since the start of year 1900, expressed as a binary number, with the highest bit position representing 2^{51}.	4,8 default 8	Timestamp

Examples

Figure 11-9 shows a DEFINE RECORD statement for a simple record without sections.

```
DEFINE RECORD XMPACCT_01
  VERSION 'IBM.120'
  IN LOG VMACCT
  IDENTIFIED BY XMPCODE='01'
  FIELDS
   (XMPUSER  OFFSET  0 LENGTH 8 CHAR,
    XMPNUM   OFFSET  8 LENGTH 8 CHAR,
    XMPDATE  OFFSET 16 LENGTH 6 DATE(MMDDYY),
    XMPTIM   OFFSET 22 LENGTH 6 TIME(HHMMSS),
    XMPCONT  OFFSET 28 LENGTH 4 BINARY,
    XMPTIME  OFFSET 32 LENGTH 4 BINARY,
    XMPVTIM  OFFSET 36 LENGTH 4 BINARY,
    XMPPGRD  OFFSET 40 LENGTH 4 BINARY,
    XMPPGWT  OFFSET 44 LENGTH 4 BINARY,
    XMPIOCT  OFFSET 48 LENGTH 4 BINARY,
    XMPPNCH  OFFSET 52 LENGTH 4 BINARY,
    XMPLINS  OFFSET 56 LENGTH 4 BINARY,
    XMPCRDS  OFFSET 60 LENGTH 4 BINARY,
    XMPVECTM OFFSET 64 LENGTH 4 BINARY,
    XMPVVECT OFFSET 68 LENGTH 4 BINARY,
    *        OFFSET 72 LENGTH 6 CHAR,
    XMPCODE  OFFSET 78 LENGTH 2 CHAR);
```

Figure 11-9. Example of a DEFINE RECORD statement

For more information about using the DEFINE RECORD statement, see “Defining a record” on page 2-3 and “Defining sections within a record” on page 3-3.

Usage

Using facilities of the log collector language, you can process date/time formats other than those supported in the DEFINE RECORD statement. As an example, suppose that your records contain date in the form of a character string *yyddd*, where *yy* are the last two digits of year, and *ddd* is day number within the year. The date starts at offset 36 within the record. To process the date, specify these fields in your DEFINE RECORD statement:

```
YY  OFFSET 36 LENGTH 2 CHAR,
DDD OFFSET 38 LENGTH 3 EXTERNAL INTEGER,
```

To obtain the date, use this expression in your DEFINE UPDATE statement:

```
DATE('19' || YY || '-01-01') + (DDD-1) DAYS
```

Another example of processing an unsupported field format is given in “Example log procedures” on page A-8.

DEFINE RECORDPROC

Use the DEFINE RECORDPROC statement to specify record procedure, that is, a procedure that the log collector calls each time it processes a record of a particular type.

Syntax

```
►►──DEFINE RECORDPROC──procedure-name──┬─────────────────────────────┬──FOR──────────►
                                        └─VERSION──string-constant─┘

   ┌─,──────────┐
   ▼            │
►────record-name──┴──┤ procedure-parms ├──────────────────────────────────────────►◄
```

procedure-parms:

```
├──┬────────────────────────────┬──┬──────────────────────┬─────────────────────────┤
   │              ┌─ASM──┐      │  └─PARM──expression──┘
   └─LANGUAGE─────┼─ASML─┼──────┘
                  └─C────┘
```

Parameters

procedure-name
: The name of the record procedure. The record procedure must be a load module, available in a load library under the name *procedure-name*.

VERSION *string-constant*
: The string specified by the *string-constant* is stored together with the definition, to identify the statement that was used to create the definition. The string can be at most 18 bytes long. Omitted VERSION means the same as specifying `VERSION '`.

If the stored definition is later altered by means of an ALTER RECORDPROC statement, its version identification is changed to 'ALTERED'. Version names are used by the Tivoli Decision Support for z/OS installation program to decide which definitions should be replaced. All definitions supplied by IBM have version names starting with 'IBM'. To ensure correct installation of new releases, do not use such names for your own definitions. See Figure 11-10 on page 11-29 for an example of how VERSION is used by IBM. Refer to the *Administration Guide and Reference* for information on how to use VERSION.

FOR *record-name* **, ...**
: Enumerates record types that this record procedure processes.

LANGUAGE
: Specifies the interface to the procedure: the language, linkage convention, and parameters.

ASM The procedure is written in Assembler and is called using standard System/390 linking conventions. It returns length of the output record in a field within the record.

ASML The procedure is written in Assembler and is called using standard System/390 linking conventions. It returns length of the output record in a parameter.

C The procedure is written in C and is called using persistent C environment. It returns length of the output record in a field within the record.

See Appendix A, “Log and record procedures,” on page A-1 for details.

PARM *expression*

Specifies an *expression* that the log collectorlog collector evaluates and passes to the procedure the first time it is called. The procedure must understand the format and interpret the value derived from the *expression*.

The *expression* cannot contain identifiers. The result of *expression* must be an integer, a floating-point number, or a character string.

Examples

Identify an assembly language program, DRL2CICS, that processes record types SMF_110_1 and SMF_110_V2. The program requires a variable name CICS_OPTION. Write this DEFINE RECORDPROC statement to identify the record procedure:

```
        DEFINE RECORDPROC DRL2CICS
          VERSION 'IBM.120'
          FOR SMF_110_1, SMF_110_V2
          LANGUAGE ASM
          PARM &CICS_OPTION;
```

Figure 11-10. DEFINE RECORDPROC statement

For more information about using the DEFINE RECORDPROC statement, see “Specifying log and record procedures” on page A-2.

DEFINE UPDATE

Use the DEFINE UPDATE statement to specify how to process data from a given record type or a given data table (the source of the update), and how to store it the result in another data table (the target of the update).

Syntax

```
►►──DEFINE UPDATE──update-name──┬───────────────────────────┬──────────────────►
                                └─VERSION──string-constant─┘

►─FROM──source-name──┬──────────────────────────┬──┬────────────────────┬──────►
                     └─SECTION──section-name─┘  └─WHERE──condition─┘

►─TO──table-name──┬────────────────────────┬──┬────────────────────┬──────────►
                  └─apply-schedule-clause─┘  └─distribute-clause─┘

►──┬────────────┬──┬─────────────────┬──┬──────────────┬───────────────────────►◄
   └─let-clause─┘  └─group-by-clause─┘  ├─set-clause───┤
                                        └─merge-clause─┘
```

Parameters

update-name

The name of the update being defined. It can be any identifier. All updates defined to the log collector must have distinct names.

VERSION *string-constant*

The string specified by the *string-constant* is stored together with the definition, to identify the statement that was used to create the definition. The string can be at most 18 bytes long. Omitted VERSION means the same as specifying `VERSION '`.

If the stored definition is later altered by means of an ALTER UPDATE statement, its version identification is changed to 'ALTERED'. Version names are used by the Tivoli Decision Support for z/OS installation program to decide which definitions should be replaced. All definitions supplied by IBM have version names starting with 'IBM'. To ensure correct installation of new releases, do not use such names for your own definitions. For an example of how VERSION is used by IBM, see Figure 11-11 on page 11-31. For information about using VERSION, refer to the *Administration Guide and Reference* .

FROM *source-name*

Identifies the source for the update definition. Must be a record type name or a table name.

> **SECTION** *section-name*
>
> Specifies that the source of the update is a repeated section *section-name* of the record *source-name*. As explained in “Using repeated sections within records” on page 5-2, the log collectorlog collector then generates an internal record for each occurrence of the repeated section. The source of the update is that internal record.
>
> If the record *source-name* has repeated sections, and you omit the SECTION clause, the update can only use the data that is outside the repeated sections.

WHERE *condition*

Limits the update to only those source records or rows for which the *condition* is true. Any identifier used in the *condition* must be the name of a field in the source record or of a column in the source table.

TO *table-name*

Names the table to be updated.

apply-schedule-clause **;***distribute-clause* **;***let-clause* **;***group-by-clause* **;***set-clause* **;***merge-clause*

These clauses specify the processing to be done. You can think of them as instructions, executed in the order they appear in the statement. They are described in detail in separate sections.

You will normally use only two or three of these clauses in one update definition. For the update definition to make sense, you must specify at least one of these: GROUP BY clause, SET clause, or MERGE clause. The first clause that you specify uses as its input the source records or rows from the table *source-name*. Each of the subsequent clauses uses the result of the preceding clause as its input. The result of the APPLY SCHEDULE and DISTRIBUTE clauses is a temporary internal table. The LET clause only defines more names, and passes the internal table (if any) to the next clause. The result of GROUP BY clause are groups of records or rows. The result of SET and MERGE clauses are updates to the target table. Any identifier used in an expression in any of the clauses must be the name of a field (or column) in the source record (or table), or a name introduced in one of the preceding clauses. Notice that the APPLY SCHEDULE clause introduces one name (that of status column), the DISTRIBUTE introduces two names (these of timestamp and interval column), and the LET clause can introduce any number of names.

Examples

Figure 11-11 shows a DEFINE UPDATE statement that tells the log collector how to enter data from VM accounting records (record type VMACCT_01) into table VM_ACCOUNTING_D.

```
DEFINE UPDATE MKTVACC_01D
  VERSION 'IBM.120'
  FROM VMACCT_01
  TO   VM_ACCOUNTING_D
  GROUP BY
   (DATE            = ACODATE,
    USER_ID         = ACOUSER,
    ACCOUNT_NUMBER  = ACONUM)
  SET
   (CONNECT_TIME    = SUM(ACOCONT),
    PROCESSOR_TIME  = SUM(ACOTIME/1000),
    VIRTPROC_TIME   = SUM(ACOVTIM/1000),
    PAGE_READS      = SUM(ACOPGRD),
    PAGE_WRITES     = SUM(ACOPGWT),
    IO_COUNT        = SUM(ACOIOCT),
    PUNCH_CARDS     = SUM(ACOPNCH),
    PRINT_LINES     = SUM(ACOLINS),
    READER_CARDS    = SUM(ACOCRDS),
    VECTOR_TIME     = SUM(ACOVECTM/1000),
    VECTOR_OVERHEAD = SUM(ACOVECTT/1000));
```

Figure 11-11. DEFINE UPDATE statement

APPLY SCHEDULE clause

This clause applies a specified schedule to the source. The source may be a record type or a table. For the purpose of this description, it is assumed to be a table.

The source must contain availability data. That means each row must represent an interval described by three items: interval type, interval start, and interval end. The row can also contain other data.

The schedule is obtained from the table DRLSYS.SCHEDULE.

The result of the APPLY SCHEDULE clause is a temporary internal table. It is a copy of the source table, with these modifications:

- The intervals are split on boundaries between schedule periods. The rows resulting from the split contain the same data as the original row, except the interval type, interval start, and interval end, which are modified.
- A *status* column is added. The column contains equal sign (=) if the interval is within the schedule, or the letter X if the interval is outside the schedule.

See “Comparing actual availability to scheduled availability” on page 5-24 for more information about using the APPLY SCHEDULE clause.

The syntax of the APPLY SCHEDULE clause is:

```
►►──APPLY SCHEDULE──expression──TO column-name-1, column-name-2, column-name-3────►

►─STATUS──identifier──────────────────────────────────────────────────────────────►◄
```

expression
Specifies the name of the schedule to use. The *expression* must specify a character string.

column-name-1
Names the source column that contains interval type code. It must be a character column of length 3.

column-name-2
Names the source column that contains the interval start. It must be a timestamp column.

column-name-3
Names the source column that contains the interval end. It must be a timestamp column.

STATUS *identifier*
Specifies the name for the status column in the resulting internal table.

DISTRIBUTE clause

This clause distributes input values over specified time periods. The input to this clause is the source record type, or source table, or the internal table that is the result of the preceding APPLY SCHEDULE clause. For the purpose of this description, it is assumed to be a table.

Each input row must contain data related to a time interval.

The result of the DISTRIBUTE clause is a temporary internal table. It is a copy of the source, with these modifications:

- The intervals are split on boundaries between the time periods. The rows resulting from the split contain the same data as the original row, except for the columns specified to be distributed. The data in these columns is distributed in proportion to the length of the interval.
- A *timestamp* and *interval* columns are added. The timestamp column contains the start of the interval represented by the row. The interval column contains the length of the interval in seconds.

See “Distributing measurements” on page 5-16 for more information about using the DISTRIBUTE clause.

The syntax of the DISTRIBUTE clause is:

```
                  ┌─,──────────────┐
                  ▼                │
►►──DISTRIBUTE────┬─field-name──┬──┴──BY──expression──START──expression──────────►
                  └─column-name─┘

►─END──expression──TIMESTAMP──identifier──INTERVAL──identifier───────────────────►◄
```

field-name ;column-name
Names a field or column to be distributed. The field or column must

contain a numeric value. The corresponding column in the resulting internal table contains floating-point numbers.

BY *expression*
Specifies the length of the distribution period in seconds. The periods start at midnight and are all of the same length, except possibly the last one before next midnight. The *expression* must specify an integer.

START *expression*
Identifies the start time of the interval represented by the input row. The *expression* must specify a timestamp.

END *expression*
Identifies the end time of the interval represented by the input row. The *expression* must specify a timestamp.

TIMESTAMP *identifier*
Specifies the name of the timestamp column in the resulting internal table.

INTERVAL *identifier*
Specifies the name of the interval column in the resulting internal table.

LET clause

Using this clause, you can give names to expressions that are frequently used in the next clauses. This saves you writing, but also speeds up the processing. For example, if you use data from a record field, the log collector reads and converts the contents of the field every time you specify the name of that field. If you give a name to the value from the field, and then use that name, the conversion will be done only once.

The syntax of the LET clause is:

```
              ┌─,─────────────────────────┐
              ▼                           │
►►──LET──(────identifier = expression─────┴──)──────────────────────────────►◄
```

expression
The log collector evaluates this expression and assigns the specified name to the result. The expression can use the names defined earlier in the same LET clause.

identifier
Specifies the name assigned to the result of the expression.

The name can be any identifier distinct from the names of fields (or columns) in the source, the name of the status column specified by APPLY SCHEDULE (if APPLY SCHEDULE was used), the names of timestamp and interval columns specified by DISTRIBUTE (if DISTRIBUTE was used), and the names defined earlier in the same LET clause.

GROUP BY clause

This clause groups records or rows by specified grouping values. The input to this clause is the source record type, or source table, or the internal table that is the result of the preceding APPLY SCHEDULE or DISTRIBUTE clause. For the purpose of this description, it is assumed to be a table.

The result of GROUP BY are groups of the input rows, such that all rows within each group have the same grouping values. All grouping values must be non-null. A row that has null as any of its grouping values is not included in any group.

If you omit the GROUP BY clause, all input rows are treated as one group.

See “Understanding the GROUP BY clause” on page 2-4 for more information about using the GROUP BY clause.

The syntax of the GROUP BY clause follows.

```
                  ┌─,──────────────────────────┐
                  ▼                            │
►►──GROUP BY──(───column-name = expression─────┴──)──────────────────────────►◄
```

expression
: Specifies one grouping value.

column-name
: Names the column where to store the grouping value. It must be the name of a column in the target table. The column cannot be a decimal or long string column.

SET clause

This clause summarizes groups of records or rows resulting from GROUP BY. For the purpose of this description, the groups are assumed to consist of rows.

The SET clause represents each group by one row in the target table. In that row, the grouping values are stored in the columns specified in the GROUP BY clause. The values of other columns are derived from all rows in the group, as specified in the SET clause. The columns not named in the GROUP BY and SET clauses are set to null.

See “Understanding the SET clause” on page 2-5 for more information about using the SET clause.

The syntax of the SET clause is:

```
             ┌─,─────────────────────────────────┐
             ▼                                   │
►►──SET──(───column-name = ─┤ accumulation ├─────┴──)────────────────────────►◄
```

accumulation:

```
├──┬─SUM (expression)──────────────────────────────────┬──────────────────────┤
   ├─MIN (expression)──────────────────────────────────┤
   ├─MAX (expression)──────────────────────────────────┤
   ├─COUNT (expression)────────────────────────────────┤
   ├─FIRST (expression)────────────────────────────────┤
   ├─LAST (expression)─────────────────────────────────┤
   ├─AVG (expression, column-name)─────────────────────┤
   └─PERCENTILE (expression, column-name, integer-constant)─┘
```

column-name
Names a column of the target table. The *accumulation* specifies how to derive the value to be stored in that column.

SUM(*expression***)**
Evaluates the *expression* for each row in the group. The value of SUM is the sum of all non-null values thus obtained. If the value of *expression* is null for all records (or rows) in the group, the value of SUM is null.

The *expression* must specify a numeric value. If that value is not of the same type as the column *column-name*, it is converted to the type of the column, and the sum is computed for the converted values.

MAX(*expression***)**
Evaluates the *expression* for each row in the group. The value of MAX is the greatest of all non-null values thus obtained. If the value of *expression* is null for all records (or rows) in the group, the value of MAX is null.

MIN(*expression***)**
Evaluates the *expression* for each row in the group. The value of MIN is the least of all non-null values thus obtained. If the value of *expression* is null for all rows in the group, the value of MIN is null.

COUNT(*expression***)**
Evaluates the *expression* for each row in the group. The value of COUNT is the number of non-null values thus obtained.

The result is an integer.

FIRST(*expression***)**
Evaluates the *expression* for each row in the group, in the order they are processed. The result is the first non-null value of *expression*. If the value of *expression* is null for all records (or rows) in the group, the value of FIRST is null.

LAST(*expression***)**
Evaluates the *expression* for each row in the group, in the order they are processed. The result is the last non-null value of *expression*. If the value of *expression* is null for all records (or rows) in the group, the value of LAST is null.

AVG(*expression,column-name***)**
Evaluates the *expression* for each row in the group. The result is the average or weighted average of the values thus obtained, depending on *column-name*.

The *column-name* must name a column whose value is specified in the same SET clause. The value of *column-name* must be specified using either COUNT or SUM. If *column-name* is specified by means of COUNT, its value must be equal to the number of non-null values of *expression*. The result of AVG is then the average of all non-null values of *expression* in the group. If the value of *expression* is null for all records in the group, the value of AVG is null. If *column-name* is specified by means of SUM, the result of AVG is the weighted average of all non-null values of *expression* in the group. The argument of SUM obtained for the same row is used as the weight. If the value of *expression* is null for all records in the group, or the sum of all weights is 0, the value of AVG is null. The *expression* must specify a numeric value. The result of AVG must be stored in a floating-point column.

PERCENTILE(*expression,column-name,integer-constant***)**

Evaluates the *expression* for each row in the group. The value of PERCENTILE is a value *p* such that *integer-constant* percent of all non-null values resulting from evaluating the *expression*s is lower than *p*, and 100-*integer-constant* percent is higher than *p*. The *integer-constant* value must be in the range 1-99.

The *column-name* must name a column whose value is specified in the same SET clause. Its value must be specified using COUNT, and must be equal to the number of non-null values of *expression*. The *expression* must specify a numeric value. The result of PERCENTILE must be stored in a floating-point column.

PERCENTILE can be used only if the source of the update is a record. If the value of a column is specified by means of PERCENTILE, it can be specified in only one update definition.

MERGE clause

This clause merges availability intervals. The input to this clause are groups of records or rows resulting from GROUP BY. For the purpose of this description, the groups are assumed to consist of rows.

The parameters of the MERGE clause specify how to derive from each row a piece of evidence about availability of some resource. It is described by four parameters: start and end times of a time interval, the *interval type* that identifies the status of the resource during the interval, and the *quiet interval*.

The MERGE clause combines this evidence and produces a set of rows in the target table that describes the status of the resource at different times. Each of these rows represents an interval described by means of start and end times, interval type, and quiet interval. Besides this information, each row contains the grouping values, in the columns specified by the GROUP BY clause.

See “Understanding the MERGE clause” on page 5-23 for more information about using the MERGE clause.

The syntax of the MERGE clause is:

```
►►──MERGE──(──column-name-1 = expression-1──,──column-name-2 = expression-2──,────►

►─column-name-3 = expression-3──,──column-name-4 = expression-4──)──────────────►◄
```

expression-1

Specifies the interval type. It must be one of these character strings: `|==`, `===`, `==|`, `|=|`, `|XX`, `XXX`, `XX|`, or `|X|`. For information about their meaning, see Table 5-19 on page 5-21.

expression-2

Specifies the start time of the interval. It must be a timestamp.

expression-3

Specifies the end time of the interval. It must be a timestamp greater than or equal to the timestamp specified by *expression2*.

expression-4

Specifies the quiet interval in seconds. It must be a non-negative integer.

column-name-1
Names the column of the target table where to store the interval type. It must be a character column of length 3.

column-name-2
Names the column of the target table where to store the start time. It must be a timestamp column.

column-name-3
Names the column of the target table where to store the end time. It must be a timestamp column.

column-name-4
Names the column of the target table where to store the quiet interval. It must be an integer or small-integer column.

How data is obtained from DB2 tables

These rules apply when the log collector obtains data from a DB2 table:

- The result of a reference to an integer or small integer column is an integer.
- The result of a reference to a floating-point column is a floating-point number. Numbers from single-precision columns are extended with binary zeros on the right.
- The result of a reference to a decimal column is a floating-point number. If the number from the table cannot be represented exactly as a floating-point number, it is rounded to the nearest floating-point value.
- The result of a reference to a character column is a character string. If the string from the table is longer than 254 bytes, it is truncated to 254 bytes. The truncation does not take into account any double-byte characters that might be present in the string.
- The result of a reference to a graphic column is a character string. This character string is obtained by adding shift-out and shift-in characters at the end of the graphic string from the table. If the string from the table is longer than 252 bytes, it is truncated to 252 bytes before adding the shift characters.
- The result of a reference to a date/time column is a date/time value of the same type. The time values from the table are extended with the microseconds part of 0.

How data is stored in DB2 tables

These rules apply when the log collector stores data in SQL tables:

- Numbers can be stored in numeric columns. Character strings can be stored in character columns or graphic columns. Date/time strings can be stored in date/time columns of the corresponding type. Date/time values can be stored in date/time columns of the same type and in character columns.
- When a floating-point number is stored in an integer or small integer column, only the integer part of the number is stored. The fractional part is discarded.
- When a floating-point number is stored in a single-precision floating-point column, it is rounded to the nearest single-precision value.
- When a floating-point number is stored in a decimal column, it is rounded to the nearest value with the required scale.
- When a character string is stored in a character column, it is truncated or padded with blanks if needed.

- When a character string is stored in a graphic column, it must have shift-out and shift-in characters at its ends. These characters are removed. If needed, the result is truncated or padded with blanks.
- When a date/time string is stored in a date/time column, it is converted to the date/time value represented by the string.
- When a date/time value is stored in a character string column, it is converted to a date/time string representing that value, and padded with blanks at the end if needed. Truncation is not allowed. The column must be wide enough to hold the whole string.
- When a time value is stored in a time column, the microseconds part is discarded.

DROP

Use the DROP statement to delete a stored definition. You can drop:

- A log definition
- A record definition
- A record procedure definition
- An update definition
- A purge condition for a table

Syntax

```
►►──DROP──┬─LOG──log-name──────────────┬─────────────────────────────────►◄
          ├─RECORD──record-name────────┤
          ├─RECORDPROC──procedure-name─┤
          ├─UPDATE──update-name────────┤
          └─PURGE FROM──table-name─────┘
```

Parameters

LOG *log-name*
: Specifies to drop the log definition *log-name*.

RECORD *record-name*
: Specifies to drop the record definition *record-name*.

RECORDPROC *procedure-name*
: Specifies to drop the record procedure definition *procedure-name*.

UPDATE *update-name*
: Specifies to drop the update definition *update-name*.

PURGE FROM *table-name*
: Specifies to drop the purge condition for the table *table-name*.

Examples

Assume that you want to delete the log definition named SOME_LOG, the record type SOME_REC, and the purge condition for the data table SOME.DATA_.TABLE. Use these drop statements to remove the log, record, and purge condition definitions:

```
DROP LOG SOME_LOG;

DROP RECORD SOME_REC;

DROP PURGE FROM SOME.DATA_.TABLE;
```

Figure 11-12. DROP statement

For more information about using the DROP statement, see “Using the DROP statement to delete a record definition” on page 3-8 and “Using the DROP statement to delete an update definition” on page 5-27.

GENERATE INDEX

Use the GENERATE INDEX statement to create a new index on a data table. This statement is converted to an SQL 'CREATE INDEX' statement. GENERATE INDEX allows index customization without the need to change the definition members. The created index will be a Unique, Primary index. If you are using a table space type of RANGE for your component installation the index will be a data-partitioned secondary index.

Syntax

```
►►──GENERATE INDEX──index-name──ON──table-name──PROFILE──string-constant──────────►◄
```

Parameters

INDEX *index-name*
Names the index to create.

ON *table-name*
Names the table to create the index on.

PROFILE *string-constant*
Specifies the name of a profile in the GENERATE_PROFILES system table. The information in this profile is used when building the SQL statement that creates the index.

Example

The following example shows a GENERATE INDEX statement.

```
GENERATE INDEX &PREFIX.SOME_TABLE_IX
 ON &PREFIX.SOME_TABLE PROFILE 'SMF';
```

GENERATE PARTITIONING

Use the GENERATE PARTITIONING statement to specify range partitioning for a table. This statement is converted to an SQL 'ALTER TABLE ADD PARTITION BY RANGE' statement. GENERATE PARTITIONING allows partitioning customization without the need to change the definition members.

GENERATE PARTITIONING only creates a partitioning scheme if the table space used by the table is set up with a partitioning type of RANGE in the

GENERATE_PROFILES system table. In all other cases the statement is ignored and does not generate any SQL statements.

The GENERATE PARTITIONING statement does not create a partitioning scheme on an already existing table: it will only create partitioning if its executed at the same time that the table is created.

Syntax

```
►►──GENERATE PARTITIONING ON──table-name──PROFILE──string-constant──────────────►◄
```

Parameters

ON *table-name*
Names the table to create the partitioning on.

PROFILE *string-constant*
Specifies the name of a profile in the GENERATE_KEYS system table. The information in this profile is used when building the SQL statement that creates the index.

Example

The following example shows a GENERATE PARTITIONING statement.

```
GENERATE PARTITIONING ON &PREFIX.DB2_SYS_PARAMETER
 PROFILE 'SMF'
```

GENERATE TABLESPACE

Use the GENERATE TABLESPACE statement to create a table space. The statement is converted to an SQL 'CREATE TABLESPACE' statement. This statement allows a table space to be created as either partitioned or non-partitioned without changing the definition members.

The type of table space created will depend on the value of the field TABLESPACE_TYPE in the GENERATE_PROFILES for the profile specified on the GENERATE statement.

Table 11-2. Table space type

TABLESPACE_TYPE value	Type of Table
RANGE Range Partitioned universal tablespace with NUMPARTS determined by the number of entries in the GENERATE_KEYS system table.	GROWTH Partition-by-growth tablespace with MAXPARTS as specified in the GENERATE_PROFILES system table.
SEGMENTED Segmented tablespace. If SEGSIZE is not specified in the GENERATE_PROFILES system table, it defaults to SEGSIZE = 4	Anything else Partition-by-growth tablespace

Syntax

```
►►──GENERATE TABLESPACE──identifier──PROFILE──string-constant───────────────►◄
```

Parameters

TABLESPACE *identifier*
Names the table space to create. This identifier cannot exceed 8 bytes. It must consist of uppercase letters, digits, and underscore characters. It must start with a letter and must be distinct from all SQL reserved words.

PROFILE *string-constant*
Specifies the name of a profile in the GENERATE_PROFILES system table. The information in this profile is used when building the SQL statement that creates the index.

Example

The following is an example of the GENERATE TABLESPACE statement.

```
GENERATE TABLESPACE DRLSTBSP PROFILE 'SMF';
```

LIST RECORD

Use the LIST RECORD statement to produce reports directly from a log data set without going through the collect process. This statement is useful when you want to produce detailed reports that cover only a short time period.

The log collector presents the output in one of these formats:

1. As a readable file. This format is useful when the log collector produces the report in batch or when no specific report formatting is required.
2. As an IXF file. QMF can display and format this file. QMF can also load this file into a DB2 table.

You can use LIST RECORD statement to list several record types from the same log. Each record type is then listed in a separate output file.

LIST RECORD works the same way as update definitions during collect. LIST RECORD has the same summarization and grouping concepts, and handles repeated sections in the same way as collect.

If LIST RECORD needs to summarize or group the data, the log collector performs data buffering (similar to collect). Otherwise the writes the result immediately to the output file(s). If the buffer fills up, the log collector writes the data accumulated so far, and terminates LIST RECORD with a warning message.

LIST RECORD labels the columns in the listing with the field name if the expression is a single field, or with the first part of the expression if the expression consists of more than one field.

Syntax

```
              ┌─,─────────────────────┐
              ▼                       │
►►──LIST──────┤ list-specification ├──┴──┬──────────────────────┬──────────────►
                                         └─LOGFILE──file-name──┘
```

```
►─┬───────────────────────────────┬──┬─────────────────────────────────┬──────────►◄
  └─BUFFER SIZE──integer-constant─┘  └─ON OVERFLOW──┬─BREAK────┬──┘
                                                    └─CONTINUE─┘
```

list-specification:

```
├──RECORD──record-name──┬─────────────────────────┬──────────────────────────►
                        └─SECTION──section-name───┘
```

```
          ┌─,──────────────────────────┐
          ▼                            │
►─FIELDS────┤ column-specification ├───┴──┬───────────────────┬──────────────►
                                          └─WHERE──condition──┘
```

```
►──┬─────────────────────────────────┬───────────────────────────────────────►
   │             ┌─,──────────┐      │
   │             ▼            │      │
   └─GROUP BY──────expression─┴──────┘
```

```
►──┬─────────────────────────────────────────────────────┬─┬──────────────────────┬──►
   │             ┌─.────────────────────────────────┐    │ └─LISTFILE──file-name──┘
   │             ▼                      ┌─ASC──┐    │    │
   └─ORDER BY──────┬─integer-constant─┬─┼──────┼────┴────┘
                   └─field-name───────┘ └─DESC─┘
```

```
►──┬──────────────────────────┬──────────────────────────────────────────────┤
   │          ┌─LIST─┐        │
   └─FORMAT───┴─IXF──┴────────┘
```

column-specification:

```
├──┬─expression───────────┬──────────────────────────────────────────────────┤
   └─┤ accumulation ├─────┘
```

accumulation:

```
├──┬─SUM (expression)──────────────────────────────┬─────────────────────────┤
   ├─MIN (expression)──────────────────────────────┤
   ├─MAX (expression)──────────────────────────────┤
   ├─COUNT (expression)────────────────────────────┤
   ├─FIRST (expression)────────────────────────────┤
   ├─LAST (expression)─────────────────────────────┤
   ├─AVG (expression)──────────────────────────────┤
   └─PERCENTILE (expression, integer-constant)─────┘
```

Parameters

list-specification

Specifies how to list one record type. You can think of the different clauses of *list-specification* as being interpreted in the order in which they are described below.

RECORD *record-name*

Names the record type to be listed. If several records are to be listed (more than one list specification is coded), all *record-name*s coded in the RECORD clauses must belong to the same log. This optional clause is available:

SECTION *section-name*

Specifies listing of a repeated section *section-name* of record *record-name*. As in the case of update processing, (explained in "Using repeated sections within records" on page 5-2), the log collectorlog collector generates an internal record for each occurrence of the repeated section. LIST RECORD lists data from that internal record.

If the record *record-name* has repeated sections and you omit the SECTION clause, you can only list data that is outside the repeated sections.

Any identifiers used in the *expressions* and *condition* within this list specification must be names of fields in the record being listed.

WHERE *condition*

Limits the log collectorlog collector to list only those records for which the *condition* is true.

GROUP BY *expression***, ...**

Specifies grouping of records. LIST RECORD produces one row for each unique combination of values of *expression*s.

If you omit the GROUP BY clause, all list specifications in the FIELDS clause must be of the same kind: either all expressions or all accumulations. The result depends on which is the case:

- If all list specifications are expressions, LIST RECORD does not perform any grouping and produces one row for each record.
- If all list specifications are accumulations, LIST RECORD produces one row that summarizes all records.

FIELDS *column-specification***, ...**

Defines one row of the output list. Each *column-specification* defines the value to be listed in one column. As the syntax diagram shows, each *column-specification* is either an expression or an accumulation.

If you specify GROUP BY, all expressions listed in GROUP BY must appear as column specifications in the FIELDS clause. All the remaining column specifications (if any) must be accumulations.

If you do not specify GROUP BY, all column specifications must be of the same kind, either all expressions or all accumulations.

expression

Specifies a value to be listed. If records are not grouped, it is obtained from the record listed in the row. If records are grouped, it is a value common to all records in the group.

accumulation

Specifies a value obtained from all records in the group. You can use it only if records are grouped. The value is obtained in a similar way as in the SET clause of the DEFINE UPDATE statement (see “SET” on page 11-53). Note that AVG and PERCENTILE have a syntax that differs from the SET clause:

AVG(*expression***)**

Calculates the value of *expression* for each input row that is grouped together to form an output row. LIST RECORD adds these values and divides by the number of input rows to the group.

PERCENTILE(*expression,integer-constant***)**

Calculates the value of *expression* for each input row to a group, places the values in a buffer and sorts them. Before LIST RECORD writes the grouped output row, it determines the output value (percentile) where *integer-constant* percent of the values in the group are smaller than the output value, and 100-*integer-constant* percent of the values are larger.

ORDER BY

Orders the output produced by LIST RECORD. If you omit ORDER BY, the order remains the same as the records in the input log data set.

The output is ordered by the values of the columns you identify. You can identify columns by their numbers. If the value of a column is specified by means of an expression (not an accumulation), and this expression is a single field name, you can use that field name to identify the column.

If you identify more than one column, the output is ordered by the values of the first column you identify, then by the values of the second column, and so on.

integer-constant

Identifies a column to be used for ordering. It is the number of the column, counting from the left.

field-name

Identifies a column to be used for ordering. The *field-name* must appear as one of column specifications in the FIELDS clause.

ASC Orders the data in ascending sequence. This is the default.

DESC Orders the data in descending sequence.

LISTFILE *file-name*

Names the output file where the log collectorlog collector writes the data. The default is DRLLST1 for the first list-specification, DRLLST2 for the second, and so on. The same *file-name* may not be used in different list specifications.

FORMAT

Specifies the format of the list file:

LIST Write the output in a readable list format. This is the default.

IXF Write the output in the IXF format.

LOGFILE *file-name*

Names the input log ddname. The default is DRLLOG.

BUFFER SIZE *integer-constant*

Specifies the size (in bytes) of the internal buffer used if the LIST RECORD statement includes a GROUP BY clause or an ORDER BY clause. The default is 10 000 000 bytes. The minimum allowed value is 10 000 bytes.

ON OVERFLOW

Specifies the action to be taken when an overflow occurs during LIST RECORD processing when buffering is in use. The overflow is a situation when an accumulated value exceeds the range allowed for this type of values. The possible options follow.

BREAK

Stop the processing. The log collectorlog collector does not produce any LIST RECORD output.

CONTINUE

Reset the accumulated value to 0, write the lost value to the DRLDUMP file, and continue the processing.

Examples

List the jobs that ran between 2 and 2:30 p.m.

```
LIST
  RECORD SMF_030
  FIELDS SMF30TME,                                -- Time
         SMF30JBN,                                -- Job name
         SMF30CLS,                                -- Class
         INTEGER(INTERVAL(SMF30SIT,SMF30TME)),    -- Elapsed time (seconds)
         (SMF30CPT+SMF30CPS) / 100.0              -- CPU time (seconds)
  WHERE SUBSTR(SMF30JNM,1,3) = 'JOB'
    AND HOUR(SMF30TME) >= 14
    AND HOUR(SMF30TME) <= 15;
```

Figure 11-13. LIST RECORD statement

Figure 11-14 shows messages that the log collectorlog collector might write to the DRLOUT DD statement as a result of the LIST RECORD statement in Figure 11-13.

```
DRL0300I List started at 2000-06-23-14.30.12
DRL0302I Processing SMF.DATA.SET on VOL001
DRL0341I The first-record timestamp is 2000-06-22-04.02.27.
DRL0380I 102347 records read from the input log
DRL0342I The last-record timestamp is 2000-06-22-22.35.18.
DRL0315I Records read from the log or built by log procedure:
DRL0317I Record name        │    Number
DRL0318I -------------------│----------
DRL0319I SMF_030            │      2489
DRL0318I -------------------│----------
DRL0321I Total              │      2489
DRL0381I 42 records written to DRLLST1 file
DRL0301I List ended at 2000-06-23-14.32.15
```

Figure 11-14. Messages from the LIST RECORD statement

Figure 11-15 on page 11-46 illustrates the LIST RECORD statement output in the DRLLST1 file.

```
SMF30TME  SMF30JBN  SMF30CLS  INTEGER(INTERVAL(SMF3  (SMF30CPT+SMF30CPS) /
--------  --------  --------  ---------------------  ---------------------
14.00.51  TEST4            A                     48   6.20000000000000E+00
14.02.14  XYZ123           A                    518   3.90000000000000E+02
14.02.19  ABBJ             C                    111   2.20000000000000E+01
:
```

Figure 11-15. Results from the LIST RECORD statement

LOGSTAT

Use the LOGSTAT statement to print the number of records of different types found in a log data set, and the number of records built by record procedures.

Syntax

```
►►──LOGSTAT──log-name──┬───────────────────┬──────────────────────────────►◄
                       └─FILE──file-name───┘
```

Parameters

log-name

Is the name of a stored log definition. It identifies the type of log to be processed.

FILE *file-name*

Names the input DD statement that refers to the log data set. The default *file-name* is DRLLOG.

Example

Assume that you want to print the different record types in an IMS log data set referred to in the JCL DD statement IMSLOG1.

```
LOGSTAT IMS FILE IMSLOG1;
```

Figure 11-16. LOGSTAT statement

Figure 11-17 on page 11-47 shows the messages produced by the log collectorlog collector.

```
DRL0300I Logstat started at 2000-06-23-14.29.41.
DRL0302I Processing IMS.DATA.SET on VOL001
DRL0341I First record timestamp is 2000-06-22-04.02.27
DRL0342I Last record timestamp is 2000-06-22-22.35.18
DRL0003I
DRL0315I Records read from the log or built by log procedure:
DRL0317I Record name        |    Number
DRL0318I -------------------|----------
DRL0319I IMS_000            |         0
DRL0319I IMS_006            |       191
DRL0319I IMS_007            |         0
DRL0319I IMS_030            |      2489
DRL0319I IMS_039            |         0
DRL0319I IMS_070            |        51
DRL0319I IMS_071            |        51
DRL0319I IMS_072_1          |       918
DRL0320I Unrecognized       |      5518
DRL0318I -------------------|----------
DRL0321I Total              |      9218
DRL0003I
DRL0316I Records built by record procedures:
DRL0317I Record name        |    Number
DRL0318I -------------------|----------
DRL0319I IMS_X              |        36
DRL0318I -------------------|----------
DRL0321I Total              |        36
DRL0301I Logstat ended at 2000-06-23-14.30.12
```

Figure 11-17. Messages from the LOGSTAT statement

PURGE

Use the PURGE statement to delete data from data tables based on the stored purge conditions (see “DEFINE PURGE” on page 11-18). Only the tables with specified purge conditions are purged.

Syntax

```
>>──PURGE──┬─────────────────────────────────────────┬────────────────>
           │            ┌─,─────────────────────┐   │
           │            ▼                       │   │
           └─INCLUDE────┬─table-name────────────┬┴──┘
                        └─LIKE──string-constant─┘

>──┬─────────────────────────────────────────┬────────────────────────><
   │            ┌─,─────────────────────┐   │
   │            ▼                       │   │
   └─EXCLUDE────┬─table-name────────────┬┴──┘
                └─LIKE──string-constant─┘
```

Parameters

INCLUDE

Specifies the tables for which the purge applies. If you specify INCLUDE, the log collectorlog collector purges only the specified tables.

table-name

Is the name of a table to be included.

LIKE *string-constant*

Specifies a group of tables to be included. The tables are those with names matching the pattern specified as the *string-constant*. The pattern matching rules are defined in “Pattern matching” on page 9-11. If the pattern contains a period (.), the table prefix must match the part before the period, and the rest of the table name must match the part after the period. For example, the pattern DRL.CICS% includes all tables whose names start with CICS and have DRL as prefix.

If the pattern does not contain a period, the prefix must be the current user ID, and the rest of the name must match the whole pattern.

EXCLUDE

Specifies the tables to be excluded from the purge. If you specify EXCLUDE, the log collectorlog collector purges all tables with defined purge condition except the specified tables.

table-name

Is the name of a table to be excluded.

LIKE *string-constant*

Specifies a group of tables to be excluded, using the same rules as for INCLUDE.

Example

Assume that you want to apply the purge definition to all tables that have a prefix of DRL and begin with CICS except DRL.CICS_APPL1. Use this PURGE statement to include all CICS tables except DRL.CICS.APPL1:

```
PURGE INCLUDE LIKE 'DRL.CICS%'
      EXCLUDE DRL.CICS_APPL1;
```

Figure 11-18. PURGE statement

For more information about using the PURGE statement, see “Deleting data” on page 4-7.

Usage

You can specify both INCLUDE and EXCLUDE on a PURGE statement. For example, INCLUDE LIKE 'DRL.CICS%' EXCLUDE DRL.CICS_APPL_H includes all CICS tables except DRL.CICS_APPL_H.

RECALCULATE

Use the RECALCULATE statement to update one or more data tables with information derived from another table.

The table used as the source of data in the RECALCULATE statement is called the *base table*. The tables being updated with information derived from the base table are called the *dependent tables*. Each dependent table must have an update definition either from the base table or from another dependent table. This update definition is used to calculate the new contents of the dependent table.

You can choose between two alternative ways of updating the tables:

- Recalculation.

  The log collector replaces data in the dependent tables by new values, calculated from the current contents of the base table. The contents of the dependent tables after RECALCULATE reflect the contents of the base table (with some exceptions resulting from the rule that RECALCULATE never deletes rows from dependent tables). You choose this alternative by specifying RECALCULATE FROM.
- Propagation of changes.

  You specify a change to the base table. The log collector makes this change and propagates it to the dependent tables. For example, you increase by 2 the value in some column of the base table. If the dependent table contains a sum computed from that column, that sum is also increased by 2. But, it need not be equal to the sum of values currently present in the base table (which is the normal situation if data were purged).

  You choose this alternative by specifying RECALCULATE DELETE, INSERT, or UPDATE.

Note: The RECALCULATE function has been implemented for small DB2 changes or column adjustments. It is not technically and logically correct to utilize that function in place of the COLLECT function of the log collector. For large updates and inserts in DB2 tables you must use the COLLECT function of the log collector. The RECALCULATE function does not allow for commits to be done until the job completes and uses a pre-allocated buffer of 200 megabytes. If this buffer size cannot handle your amount of data, an abend U0005 can occur. In this case you should perform the RECALCULATE in different steps, using the WHERE condition to change only a group of rows at a time.

Syntax

```
►►──RECALCULATE──┬──────────────────┬──┤ from-clause ├──┬───────────────────►◄
                 │  ┌─,──────────┐  │  ┤ delete-clause ├┤
                 │  ▼            │  │  ┤ insert-clause ├┤
                 └───table-name──┴──┘  ┤ update-clause ├┘
```

from-clause:

```
├──FROM──table-name──┬────────────────────────┬─────────────────────────┤
                     └─WHERE──sql-condition───┘
```

delete-clause:

```
├──DELETE FROM──table-name──┬────────────────────────┬──────────────────┤
                            └─WHERE──sql-condition───┘
```

insert-clause:

```
├──INSERT INTO──table-name──┬──────────────────────────┬────────────────►
                            │    ┌─,─────────────┐     │
                            │    ▼               │     │
                            └─(───column-name────┴──)──┘
```

```
             ┌─,────────┐
             ▼          │
►─VALUES──(───constant──┴──)──────────────────────────────────────────────┤
```

update-clause:

```
                                    ┌─,──────────────────────────┐
                                    ▼                            │
├──UPDATE──table-name──SET──(───column-name = expression──┴──)──────────────►

►──┬──────────────────────┬────────────────────────────────────────────────┤
   └─WHERE──sql-condition─┘
```

Parameters

table-name, **...**
: Lists the dependent tables.

  Omitted list means the same as listing all tables that have their contents derived from the base table by means of update definitions, either direct or cascaded.

FROM *table-name*
: Identifies the table *table-name* as the base table. Indicates that you want to recalculate the contents of the dependent tables from the contents of the base table.

  The log collector calculates first new rows for all dependent tables that have update definitions from the base table. It uses these update definitions to calculate the rows. If a new row has the same GROUP BY values as a row already present in the table, the log collector replaces the old row by the new. The data from the old row is lost. If the table does not contain a row with the same GROUP BY values, the log collector inserts the new row into the table.

  The procedure is repeated with tables that have update definitions from the tables thus recalculated, and so on. No rows are deleted in the process. As a result, an updated table may contain old rows that no longer reflect any data from the underlying table. These old rows are **not** used to calculate new rows for the next table.

  It is possible to calculate the new data using only selected rows from the base table. You select the rows using this clause:

  WHERE *sql-condition*
  : Specifies which rows to select. The selected rows are those for which the *sql-condition* is true. If you omit WHERE, all rows are selected. The *sql-condition* must be a valid SQL search condition for the table *table-name*, and its individual tokens must be recognized by the log collector.

DELETE FROM *table-name*
: Identifies the table *table-name* as the base table. Indicates that you want to delete rows from the base table and propagate the change to the dependent tables.

The change to the dependent tables consists of changing data in the existing rows. No rows are deleted from the dependent tables.

You specify the rows to be deleted using this clause:

WHERE *sql-condition*

Specifies the rows to be deleted from the base table. The log collector deletes the rows for which the *sql-condition* is true. If you do not specify a WHERE clause, all rows in the table are deleted. The *sql-condition* must be a valid SQL search condition for the table *table-name*, and its individual tokens must be recognized by the log collector.

INSERT INTO *table-name*

Identifies the table *table-name* as the base table. Indicates that you want to insert a row into the base table and propagate the change to the dependent tables.

If the base table already contains a row with the same GROUP BY values as the row being inserted, the log collector does not insert a new row. It updates instead the existing row with the specified values. The log collector uses for this purpose the accumulation functions specified in update definitions for the table.

The change to the dependent tables can consist of inserting new rows or changing data in the existing rows. No rows are deleted from the dependent tables. Insertion of rows with duplicate GROUP BY values follows the same rule as for the base table. You specify the row to be inserted using these clauses:

(*column-name***, ...)**

Lists the columns for which you specify values. Notice that you must specify values for all GROUP BY columns.

Omitted list of columns means that you specify values for all columns.

VALUES(*constant***, ...)**

Specifies the values in the row. If the list of columns is present, the *constant*s specify values for the columns in the order they appear in the list. The number of *constant*s must be the same as the number of *column name*s.

If the list of columns is omitted, the *constant*s specify values for the columns in the order they appear in the table. The number of *constant*s must be the same as the number of columns in the table.

UPDATE *table-name*

Identifies the table *table-name* as the base table. Indicates that you want to change one or more rows in the base table and propagate the change to the dependent tables.

If one or more GROUP BY values in a row is changed, the resulting row can have the same GROUP BY values as a row already present in the table. The log collector merges then the two rows: it uses data from the changed row to update the existing row and deletes the changed row. In this process, the log collector uses the accumulation functions specified in update definitions for the table.

The change to the dependent tables can consist of inserting new rows or changing data in the existing rows. No rows are deleted from the dependent tables. Insertion of rows with duplicate GROUP BY values follows the same rule as for INSERT INTO. You specify the change to the base table using these clauses:

SET (*column-name* **=** *expression***, ...)**
Specifies new values for the named columns.

WHERE *sql-condition*
Specifies the rows to be updated. The log collector updates only the rows for which the *sql-condition* is true. If you do not specify a WHERE clause, all rows in the table are updated. The *sql-condition* must be a valid SQL search condition for the table *table-name*, and its individual tokens must be recognized by the log collector.

Example

Assume that you want to change an account number. The account number is stored in the column ACCOUNT_NO of the table ACCOUNT.INFO_TABLE.

Use this RECALCULATE statement to change the account number in the table:

```
RECALCULATE
  UPDATE ACCOUNT.INFO_TABLE
    SET (ACCOUNT_NO = '880503')
    WHERE ACCOUNT_NO = '880502';
```

Figure 11-19. RECALCULATE statement

Because you omitted the list of dependent tables, your change is propagated to all tables that contain information based on ACCOUNT.INFO_TABLE.

For more information about using the RECALCULATE statement, see “Changing data within tables” on page 4-8.

Usage

Be careful when you specify WHERE for a RECALCULATE FROM. Suppose you have three tables:

- TABLE_H, containing hourly data
- TABLE_D, containing data from TABLE_H summarized by day
- TABLE_M, containing data from TABLE_D summarized by month.

Suppose you execute this statement:

```
RECALCULATE TABLE_D,TABLE_M FROM TABLE_H WHERE DATE='2000-05-21';
```

The data for the specified day in TABLE_D is recalculated correctly. But, the rows for other days in TABLE_D are treated as old rows, that are left in the table because of the rule that RECALCULATE does not delete rows. They are not used to calculate the data for TABLE_M. As a result, the data for May 2000 in TABLE_M is derived from data for only one day: May 21.

To avoid this problem, use a separate statement for each table:

```
RECALCULATE TABLE_D FROM TABLE_H WHERE DATE='2000-05-21';
RECALCULATE TABLE_M FROM TABLE_D WHERE MONTH=5;
```

SET

Use the SET statement to define a named character string. Such a named string is called a *variable*. The string itself is called the *value* of the variable.

You can also use the SET statement to change the value of an existing variable, that is, to replace a named string by another with the same name.

The variable remains defined until the end of the log collector run.

Syntax

```
►►──SET──variable-name = string-constant────────────────────────────────────►◄
```

Parameters

variable-name
The name of the variable.

string-constant
The value of the variable.

Examples

Assume you want to create a variable named PREFIX with a value of STROMBK. Use this SET statement to create the variable:

```
   SET PREFIX = 'STROMBK';
```

Figure 11-20. SET statement

If the variable PREFIX already exists, the statement changes its value to STROMBK.

Usage

You can use variables to modify your statements with the help of *variable markers* (see "Using variables to modify your text" on page 8-7) and *variable references* (see "Obtaining the value of a variable" on page 9-4). You can also use variables to control certain diagnostic functions. (See , SH19-6902.)

Part 3. Report definition language guide

Chapter 12. Introducing the report definition language

You can use the report definition language to write definitions for producing reports from the data you collect using the log collector language.

Note: To use the report definition language, you need to have QMF installed on your system.

The report definition language contains statements that let you:

- Create report definitions
- Create group definitions that are a logical collection of reports about a related topic
- Delete report and group definitions

You can create report definitions using the report definition language, or you can use the reporting dialog. The predefined reports that are supplied with Tivoli Decision Support for z/OS are defined with the report definition language. Users wanting to create new reports usually do this with the reporting dialog. For more information about using the reporting dialog, refer to the *Guide to ReportingGuide to Reporting*.

The syntax used to write report and group definition statements is similar to that used to write log collector language statements. When you execute the report definition program, it stores the report and group definitions that you create. Then you can use the definitions to produce reports without having to write the definition each time.

Chapter 13, “Implementing the report definition language,” on page 13-1 describes how to use the report definition language to write a report definition. It explains how to create tabular and graphical reports from data collected with the log collector.

Chapter 14, “Report definition language elements,” on page 14-1 describes the elements associated with the report definition language.

Chapter 15, “Report definition language statements,” on page 15-1 describes each of the statements that you can use in the report definition language.

Chapter 13. Implementing the report definition language

About this task

You use the report definition language to define reports and report groups. To do this, you:

- Create a QMF query that determines how the data in the data table is accessed. Export the query to a data set.
- Optionally, create a QMF form that specifies how the data is formatted, and export the form to a data set.
- Write a report definition that identifies the query, form, and other options. You might also write a group definition to identify a set of reports.
- Execute the report definition language program to store the report definition.

This chapter uses an example to describe how to create a simple QMF query and QMF form. It also describes how to write a report definition and group definition, and how to submit JCL to store the definitions and produce reports.

Getting started with the report definition language

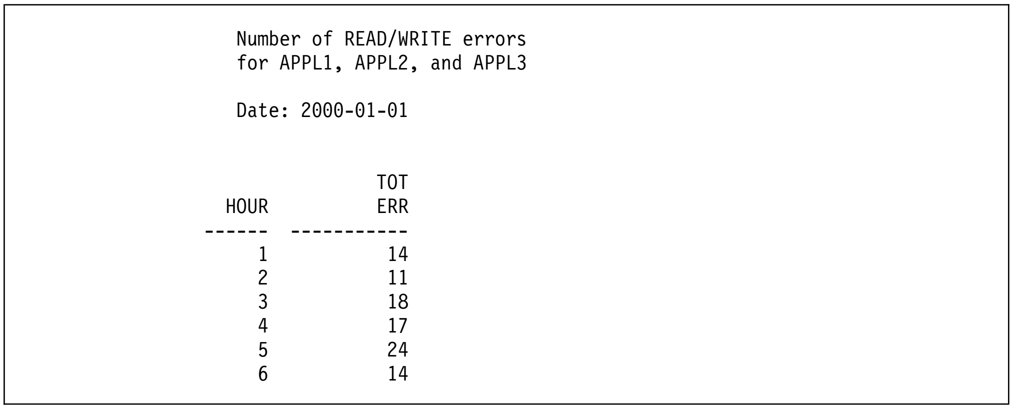
About this task

Assume that you have collected data and stored it in a data table called DRL.RWSTAT. Table 13-1 shows the contents of DRL.RWSTAT.

Table 13-1. Contents of DRL.RWSTAT data table

DATE	HOUR	R_ERR	W_ERR	TOT_ERR
2000-01-01	1	6	8	14
2000-01-01	2	7	4	11
2000-01-01	3	7	11	18
2000-01-01	4	6	11	17
2000-01-01	5	7	17	24
2000-01-01	6	8	6	14

You can use the report definition language to produce reports based on this data. You can create a tabular report like the one shown in Figure 13-1 on page 13-2.



```
                    Number of READ/WRITE errors
                    for APPL1, APPL2, and APPL3

                    Date: 2000-01-01


                                TOT
                   HOUR         ERR
                ------  -----------
                     1           14
                     2           11
                     3           18
                     4           17
                     5           24
                     6           14
```

Figure 13-1. Tabular report produced from DRL.RWSTAT

You can also produce a graphic report like the one shown in Figure 13-2.

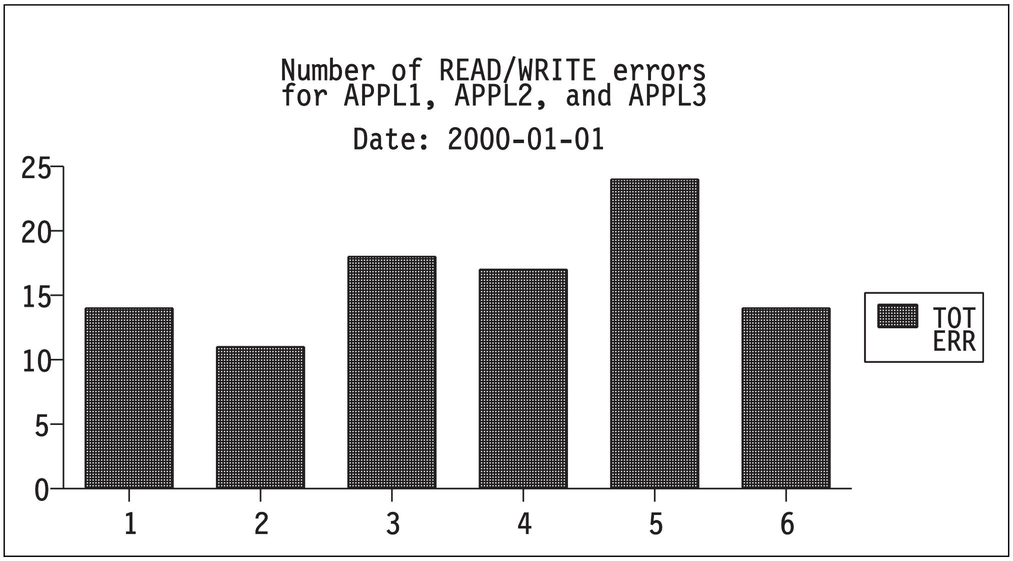


Figure 13-2. Graphic report produced from DRL.RWSTAT

Creating a QMF query and form

About this task

To create reports from a data table, you must create a QMF query to access the data and, optionally, a QMF form to define the format of the report. If you do not define a form, QMF will use a default form.

If you export the query and form to different data sets, you must allocate them according to the QMF requirements for such objects.

You can access QMF to create queries and forms using the reporting dialog. For more information about using the reporting dialog, refer to the *Guide to Reporting*. For more information about using QMF to create queries and forms, refer to the *Query Management Facility Learner's Guide*.

Writing a group definition

About this task

Before you create the actual report definition, you must first determine whether the report will be part of a group. In this example, the report is going to be part of a report group called ACCESS_ERRORS.

To create the group definition for the group called ACCESS_ERRORS, edit a new member of DRL.LOCAL.DEFS called RSTATS and type the DEFINE GROUP statement shown in Figure 13-3.

```
-- Define a report group called ACCESS_ERRORS
DEFINE GROUP ACCESS_ERRORS
   DESC 'Access errors for APPL1, APPL2, and APPL3';
```

Figure 13-3. Using the DEFINE GROUP statement

In Figure 13-3, you specify a name and a description for the group. When you display this group through the reporting dialog, the description you use on the DESC clause is displayed.

You assign a report to this group using the DEFINE REPORT statement.

Writing a report definition

About this task

Use the DEFINE REPORT statement to identify the query, form, and any options. To create the reports shown in Figure 13-1 on page 13-2 and Figure 13-2 on page 13-2, you must create one report definition for a tabular report and one for a graphic report.

Writing a definition for a tabular report

About this task

To create a definition for a tabular report, type the DEFINE REPORT statement shown in Figure 13-4 in DRL.LOCAL.DEFS(STATTAB).

```
-- Define tabular report for READ/WRITE errors from APPL1, APPL2, APPL3
DEFINE REPORT TAB_RPT
   DESC 'Table of RD/WR errors for APPL1, APPL2, APPL3'
   QUERY RWQUERY
   FORM RWFORM
   FILE RWTABOUT
   BATCH PRINT SAVE DAILY
   GROUPS ACCESS_ERRORS;
```

Figure 13-4. Using the DEFINE REPORT statement for a tabular report

In Figure 13-4, you define the report name as TAB_RPT and specify the QMF query and form you created earlier. The query and form are imported into QMF when the statement is executed.

The FILE and BATCH clauses define batch processing options for the report. When the report is produced in batch it will be printed, and then saved in the member RWTABOUT in the data set allocated to DRLREP.

Using the GROUPS clause, you assign this report to the report group called ACCESS_ERRORS.

Writing a definition for a graphic report

About this task

You can use the DEFINE REPORT statement to specify that the report be produced in chart format.

Type the DEFINE REPORT statement shown in Figure 13-5 in a member of DRL.LOCAL.DEFS called STATGRA to define a bar chart for this data.

```
-- Define bar chart for READ/WRITE errors from APPL1, APPL2, and APPL3
DEFINE REPORT CHART_RPT
   DESC 'Bar chart for RD/WR errors'
   QUERY RWQUERY
   FORM RWFORM
   FILE RWCHAOUT
   CHART BAR
   BATCH PRINT SAVE DAILY
   GROUPS ACCESS_ERRORS;
```

Figure 13-5. Using the DEFINE REPORT statement for a chart

Creating the report definition for a chart is similar to creating the report definition for a tabular report. You specify the name of the report definition, CHART_RPT, and assign a description to the report definition. You specify the CHART clause to identify the GDDM-ICU format used for this graphic report. (BAR is a predefined QMF format; otherwise you would have to create a chart format.)

When the report is produced in batch it is printed and then saved in the member RWCHAOUT in the data set allocated to ADMGDF.

Storing report definitions

About this task

After creating the group definition and the report definition, you can store these definitions in batch using JCL.

Storing definitions in batch

About this task

Figure 13-6 on page 13-5 shows the JCL you can use to run jobs for storing group and report definitions.

```
//jobname    JOB parameters
//RDEF       EXEC PGM=IKJEFT01
//SYSPROC    DD DISP=SHR,DSN=DRL180.SDRLEXEC
//DRLIN      DD DISP=SHR,DSN=DRL.LOCAL.DEFS(RSTATS)
//           DD DISP=SHR,DSN=DRL.LOCAL.DEFS(STATTAB)
//           DD DISP=SHR,DSN=DRL.LOCAL.DEFS(STATGRA)
//DRLOUT     DD SYSOUT=*
//DRLDEFS1   DD DISP=SHR,DSN=DRL.LOCAL.DEFS
//...
//... QMF and DB2 libraries
//...
//SYSTSIN    DD *
%DRLERDEF SYSTEM=DSN SYSPREFIX=DRLSYS PREFIX=DRL MODE=BATCH QMF=YES
```

Figure 13-6. JCL for storing report definitions in batch

You may need to modify these parameters submitted with DRLERDEF:

SYSTEM=DSN
The SYSTEM parameter specifies the name of the DB2 subsystem, which is DSN in Figure 13-6.

SYSPREFIX=DRLSYS
The SYSPREFIX parameter specifies the prefix of the Tivoli Decision Support for z/OS system tables, which is DRLSYS in Figure 13-6.

PREFIX=DRL
The PREFIX parameter specifies the prefix of all other tables, which is DRL in Figure 13-6

SHOWSQL YES/NO
The SHOWSQL parameter specifies whether SQL statements should be shown (for debugging).

QMF=YES/NO
The QMF parameter specifies whether QMF is used, which is YES in Figure 13-6

For more information about the parameters used in the JCL, see Appendix B, “JCL for the log collector language and report definition language,” on page B-1.

You can also store definitions from the reporting dialog. For more information about using the reporting dialog, refer to the *Guide to Reporting*.

Generating reports

About this task

After report definitions are stored, you can use them to produce reports. You can produce reports from the reporting dialog, or use JCL in batch. For more information about producing reports, refer to the *Guide to Reporting*.

Part 4. Report definition language reference

Chapter 14. Report definition language elements

This chapter describes the elements that are common to Report definition language elements. It describes how to code report definition language elements, and discusses variables and constants.

Input format

When you enter report definition language elements, you can enter them in any format you choose. You need not begin elements at a particular column within your input data set. Instead, you can use any column between 1 and 72.

For example, this DEFINE REPORT statement:

```
DEFINE REPORT NEW_REPORT
    DESC 'New report definition'
    QUERY SQLQUERY
    FORM QMFFORM
    BATCH PRINT DAILY
    GROUPS ALL_REPORTS;
```

can also be entered in this format:

```
DEFINE REPORT NEW_REPORT DESC 'New report definition'
  QUERY SQLQUERY        FORM QMFFORM
  BATCH PRINT DAILY
  GROUPS ALL_REPORTS;
```

Identifiers

Identifiers are used as names or components of names. Identifiers can be either long or short.

A long identifier is an identifier that has a maximum length of 18 bytes. If you use a delimited long identifier, quotation marks are not included in the 18-byte length restriction unless they are part of the name.

You can also use sequences of double-byte characters in long delimited identifiers. Each sequence must begin with a shift-out character and end with a shift-in character. The shift-out and shift-in characters are considered part of the identifier.

A short identifier has a maximum length of 8 bytes and must follow the MVS rules for member names.

Comments

A comment provides documentation within the report definition language. Comments can be:

- A sequence of characters starting with a double minus sign (--) and ending at the end of the line.

  If you begin a comment with a double minus sign (--), the comment is ended at the end of a line. To create multiple-line comments, you must specify double minus signs (--) at the beginning of each comment line.

  For example:

```
-- A comment line must be on one line only
-- But, you can have multiple-line comments
```

- A sequence of characters starting with slash asterisk (/*) and ending with asterisk slash (*/).

  If you begin a comment with /*, the comment is not ended until a */ is encountered. Therefore, if you want multiple-line comments, you begin a comment with /* and create lines of comments. At the end of the last line of comments, you add */. For example:

  ```
  /*  This comment line stretches over more
  than one line */
  ```

Character string constants

A character string constant is a sequence of characters that starts and ends with an apostrophe.

You can include double-byte characters in character string constants. Each string of double-byte characters must be enclosed between shift-out and shift-in characters.

To include an apostrophe in a character string constant, use two consecutive apostrophes.

Examples of character string constants include:

’2000-5-15’

’32’

’DON’T CHANGE’

Chapter 15. Report definition language statements

The report definition language consists of statements that you can use to write report definitions and to write group definitions for sets of reports. You can also delete report and group definitions that you have created and stored.

This section provides an alphabetical listing of the report definition language statements. For each statement, this section describes:
- The purpose of the statement
- The syntax used for the statement
- Parameters (clauses and keywords) that you can specify for the statement
- Examples of how to use the statement

For more information about how to read the syntax diagrams shown in this chapter, see Chapter 7, “How to read the syntax diagrams,” on page 7-1.

DEFINE GROUP

Purpose

Use the DEFINE GROUP statement to assign an ID to a set of reports that you plan to create later. You can also specify a group owner and a description for the group.

Note: You specify which reports are assigned to this group using the GROUPS clause of the DEFINE REPORT statement.

Format

```
                                 ┌─,─────────────────────────┐
                                 ▼                           │
►►──DEFINE GROUP──group-identifier──┬─────────────────────────┬┴──────────────►◄
                                    ├─VERSION──string-constant─┤
                                    ├─OWNER──user-identifier───┤
                                    └─DESC──'text'─────────────┘
```

Parameters

group-identifier
: Specifies a long identifier for the ID of this group of reports.

VERSION *string-constant*
: The string specified by the *string-constant* is stored together with the definition, to identify the statement that was used to create the definition. The string can be at most 18 bytes long. Omitted VERSION means the same as specifying `VERSION '`.

OWNER *user-identifier*
: Specifies a short identifier for the owner of the group. If you specify a group owner, only that owner can view and modify the group of reports.

If you do not specify OWNER, the report group is public (it is accessible by all users).

DESC '*text*'

Specifies a description to be used for this report group, where *text* can be any character string up to 50 characters long (characters over the 50-character maximum are truncated). You can include double-byte character set (DBCS) characters in *text*.

The group description appears in the reporting dialog when the group definition is displayed.

Examples

Assume that you want to create a report group called CICS that will contain all reports produced from CICS data. Use the DEFINE GROUP statement shown in Figure 15-1.

```
DEFINE GROUP CICS
  DESC 'CICS Reports';
```

Figure 15-1. DEFINE GROUP statement

DEFINE REPORT

Purpose

Use the DEFINE REPORT statement to create a report. You can specify how the report is generated and identify the groups to which the report belongs.

Format

```
▸▸──DEFINE REPORT──long-identifier────────────────────────────────►
```

```
   ┌─.──────────────────────────────────────────┐
   ▼                                            │
►──┴──┬─VERSION──string-constant──────────────┬─┴──────────────────────────────►◄
      ├─OWNER──user-identifier────────────────┤
      ├─DESC──'text'──────────────────────────┤
      ├─TYPE──┤ choice ├──────────────────────┤
      ├─BATCH──┬───────┬──┬──────┬──┬────────┬┤
      │        └─PRINT─┘  └─SAVE─┘  ├─DAILY──┤│
      │                             ├─WEEKLY─┤│
      │                             └─MONTHLY┘│
      ├─QUERY──query-name─────────────────────┤
      ├─FORM──form-name───────────────────────┤
      ├─CHART──format-name────────────────────┤
      ├─FILE──member-name─────────────────────┤
      │              ┌─,───────────────┐      │
      │              ▼                 │      │
      ├─ATTRIBUTES────long-identifier──┴──────┤
      │          ┌─,───────────────┐          │
      │          ▼                 │          │
      ├─GROUPS────group-identifier─┴──────────┤
      │            ┌─,───────────┐            │
      │            ▼             │            │
      └─VARIABLES────┤ variable ├┴────────────┘
```

choice

```
├──def.QUERY──TABDATA──GRAPHDATA──────────────────────────────────────┤
```

variable:

```
├──long-identifier──┬────────────┬──┬──────────┬──┬────────────────┬──────────┤
                    ├─CHAR───────┤  └─REQUIRED─┘  └─DEFAULT──'text'─┘
                    ├─NUMERIC────┤
                    ├─DATE───────┤
                    ├─TIME───────┤
                    └─TIMESTAMP──┘
```

Parameters

long-identifier
: Specifies a long identifier as the ID of the report.

VERSION *string-constant*
: The string specified by the *string-constant* is stored together with the definition, to identify the statement that was used to create the definition. The string can be at most 18 bytes long. Omitted VERSION means the same as specifying `VERSION '`.

OWNER *user-identifier*
: Specifies a short identifier as the owner of the report. If you specify OWNER, only that owner can view or modify the report. If you do not specify OWNER, the report is public (accessible by all users).

DESC '*text*'
: Specifies a description to be used for this report, where *text* can be any

character string up to 50 characters long (characters over the 50-character maximum are truncated). You can include double-byte character set (DBCS) characters in *text*.

The report description appears in the reporting dialog when the report definition is displayed.

TYPE Specifies the kind of report, where TYPE can be:

QUERY
Specifies that the report is the displayed output of a QMF query. If you specify `TYPE QUERY`, you must also specify the QUERY clause. The default TYPE is QUERY.

TABDATA
Specifies that the report is a saved tabular report. If you specify `TYPE TABDATA`, you must also specify the FILE clause.

GRAPHDATA
Specifies that the report is a saved graphic report. If you specify `TYPE GRAPHDATA`, you must also specify the FILE clause.

BATCH
Specifies that the report is produced in batch. If you specify the BATCH keyword, you must also specify `TYPE QUERY` and the QUERY clause.

You can specify these options on the BATCH keyword:

PRINT
Specifies that the report is printed when produced in batch.

SAVE Specifies that the report is saved in the member specified by the FILE option when the report is produced in batch.

DAILY
Specifies that the report is produced daily.

WEEKLY
Specifies that the report is produced weekly.

MONTHLY
Specifies that the report is produced monthly.

QUERY *query-name*
Identifies the data set member that contains the QMF query used for the report, where *query-name* must be a previously defined QMF query. This query will be imported into QMF when the statement is executed.

FORM *form-name*
Identifies the data set member that contains the QMF form used for this report, where *form-name* must be a previously defined QMF form. This form is imported into QMF when the statement is executed.

CHART *format-name*
Specifies that the report is a graphic report, where *format-name* is the name of the GDDM-ICU format used for this chart.

Note: If you do not specify the CHART clause, a tabular report is assumed.

FILE *member-name*
Specifies the member that a saved report is retrieved from. It is also used by batch reporting for saving the report. (*member-name* must be a short identifier.)

ATTRIBUTES *long-identifier*, **...**

Identifies the attributes of the report. When you display reports using the reporting dialog, you can display all the reports that have the same attribute.

GROUPS *group-identifier*, **...**

Specifies the group or groups to which this report belongs, where *group-identifier* is a long identifier that corresponds to a previously defined group.

VARIABLES *long-identifier*, **...**

Specifies variables (besides the variables defined in the QMF query) that must be supplied before the report is produced. (The maximum length for the variables is 17.) If the report will be produced online from the reporting dialog, the dialog prompts you to provide these variables. If the report is produced in batch, you must include these variables in the job used to produce the report.

Variables are also extracted from the query, so this clause is optional if you do not want special checks or functions.

For each variable, you can specify these parameters:

CHAR | NUMERIC | DATE | TIME | TIMESTAMP

Specifies the data type of the variable.

REQUIRED

Specifies that the variable is required. You must specify a value for this variable before the report is produced.

DEFAULT '*text***'**

Specifies a default value for the variable, where *text* can be up to 40 characters.

Examples

Assume that you have created (and exported) a QMF query called DRLQCIEX and a QMF form called DRLFCIEX. You want to create a report that uses this query and form. It will have attributes of CICS, PERFORMANCE, and EXCEPTION. It will also belong to groups CICS and MGMT.

To write a report definition based on this information, use the DEFINE REPORT statement shown in Figure 15-2.

```
DEFINE REPORT CICS_EXC
  DESC 'CICS Exceptions'
  QUERY DRLQCIEX
  FORM  DRLFCIEX
  ATTRIBUTES CICS, PERFORMANCE, EXCEPTION
  GROUPS CICS, MGMT;
```

Figure 15-2. DEFINE REPORT statement

DROP GROUP

Purpose

Use the DROP GROUP statement to delete a report group definition that you have previously created.

Format

The syntax of the DROP GROUP statement is:

```
►►──DROP GROUP──group-identifier──┬──────────────────────┬──────────────►◄
                                  └─OWNER──user-identifier─┘
```

Parameters

You can specify these parameters for the DROP GROUP statement:

group-identifier
Specifies the name of the group that you want to drop.

OWNER *user-identifier*
Identifies the owner of the group that you want to drop. If you drop a public group (a group accessible by all users), you need not specify the OWNER clause.

Examples

Assume that you created and stored a report group named CICS. To delete this group definition, use the DROP GROUP statement shown in Figure 15-3.

```
DROP GROUP CICS;
```

Figure 15-3. DROP GROUP statement

DROP REPORT

Format

Use the DROP REPORT statement to delete a report definition that you previously created and stored.

The syntax of the DROP REPORT statement is:

```
►►──DROP REPORT──long-identifier──┬──────────────────────┬──────────────►◄
                                  └─OWNER──user-identifier─┘
```

Parameters

You can specify these parameters with the DROP REPORT statement:

long-identifier
Specifies the name of the report that you want to drop.

OWNER *user-identifier*
Specifies the owner of the report that you want to drop. If you drop a public report (a report that is accessible to all users), you need not specify the OWNER clause.

Examples

Assume that your user identifier is USER1 and you have a private report called USER_EXC that you want to delete. To delete this report, use the DROP REPORT statement shown in Figure 15-4.

```
DROP REPORT USER_EXC OWNER USER1;
```

Figure 15-4. DROP REPORT statement

Part 5. Appendixes

Appendix A. Log and record procedures

This appendix contains Product-sensitive Programming Interface and Associated Guidance Information.

Although the log collector provides extensive processing capabilities, you might decide to create your own procedures to process data before it is processed using the stored definitions.

In particular, you must use own procedures to:
- Process data in unusual formats.
- Cross-reference data between parallel repeated sections and between records.

You write these procedures in assembler or C. There are two types of procedure:

Log procedures
Invoked for all records read from the log data set.

Record procedures
Invoked only for the type of records you specify.

Figure A-1 shows an example of the processing that occurs when you use log and record procedures.

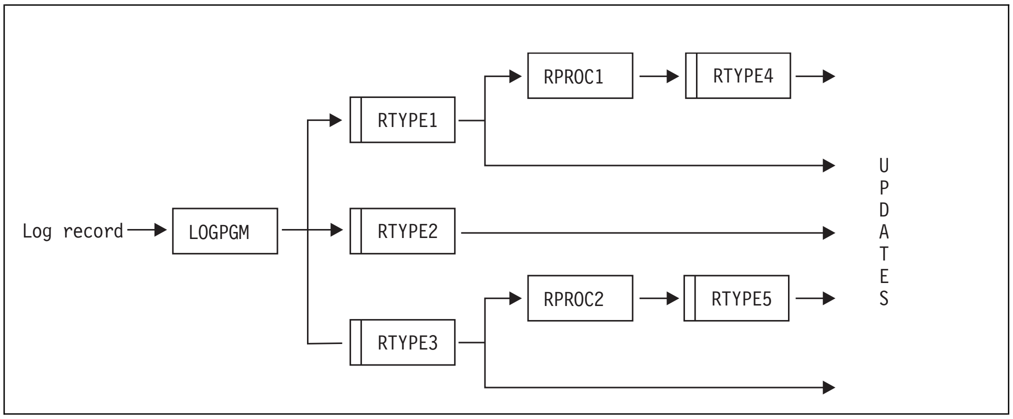


Figure A-1. Processing for log and record procedures

As shown in the figure, the log procedure LOGPGM processes each record from the log data set and produces three internal records of types RTYPE1, RTYPE2, and RTYPE3, respectively. Each of these record types must be defined to the log collector by means of a DEFINE RECORD statement, and can be specified as the source of a DEFINE UPDATE statement. In the example, each of the three record types is so specified, and the data from each of them is used to update the data tables. (Note that the procedure is not limited to producing only one internal record of each type; there can be any number within each type.)

An internal record produced by one procedure may be used as input to another record procedure. In the example of Figure A-1, the internal records of type RTYPE1, besides being used in an update, are processed by a record procedure RPROC1. This procedure produces internal records of type RTYPE4. Again, this record type must be defined by means of a DEFINE RECORD statement, and the data from the record are used to update the data tables via an update definition.

In a similar way, internal records of type RTYPE3 are processed by a record procedure RPROC2 that produces internal records of type RTYPE5.

This scheme may be made as complex as needed; however, the output of a record procedure must not be used, directly or indirectly, as input to the same procedure.

Specifying log and record procedures

About this task

You must define your log and record procedures to the log collector using the log collector language. To define a log procedure, use the LOGPROC clause of the DEFINE LOG statement. This is an example of a log procedure definition:

```
DEFINE LOG TST_LOG
  LOGPROC LOGPGM
    LANGUAGE ASM;
```

Figure A-2. Defining a log procedure

The log procedure LOGPGM is called for every record that occurs in a log of type TST_LOG. LOGPGM is a program written in assembler.

To define a record procedure, use the DEFINE RECORDPROC statement. This is an example of a record procedure definition:

```
DEFINE RECORDPROC RPROC1
   FOR RTYPE1
   LANGUAGE C;
```

Figure A-3. Defining a record procedure

The record procedure RPROC1 is called for every record of type RTYPE1. RPROC1 is a program written in C. You must define the record type RTYPE1 before you execute this DEFINE RECORDPROC statement.

When you run the log collector, you must ensure that load modules LOGPGM and RPROC1 containing the specified procedures are present in an accessible load library.

Calling log and record procedures

About this task

Before the log collector begins processing a log data set, it passes control to each of the defined log and record procedures. The procedure does not read data at this time and no output records are generated. Instead, the procedure performs any required initialization, such as buffer and work area allocation. The procedure can return the address of the work area in a parameter. Then, each time the procedure is called, the log collector passes this address to it.

When the log collector processes the log, it calls the log procedure for each record from the log, and the record procedure for each record of the type specified in the DEFINE RECORDPROC statement. The procedure processes the record and passes a return code back to the log collector.

The return code indicates how many records were produced by the procedure. If the procedure did not produce output records, the log collector continues processing the next input record. If the procedure produced one output record, it returns a pointer to it. The log collector processes the output record, and continues with the next input record. If the procedure produced more than one output record, it returns a pointer to the first of them. The log collector processes the output record, and calls the procedure again *with the same input*. The procedure returns then a pointer to the next output record. This is repeated until the procedure indicates that there are no more output records.

The log or record procedures can determine that the input record is not valid. In which case, the log collector writes information to the DRLDUMP file. The procedure can also immediately terminate processing.

The log collector calls each log procedure and record procedure each time after committing the database updates.

After processing all records, the log collector calls the procedure again to perform any required termination tasks (such as freeing work areas). This step occurs even if the procedure specified that processing terminate immediately. During this last call to the procedure, no input records are provided. But the procedure can generate one or more output records.

If you specified a PARM expression in your DEFINE LOG or DEFINE RECORDPROC statement, the procedure receives the value of that expression as a parameter every time it is called. For example, the log procedure LOGPGM defined by this statement receives the string that is the current value of the variable UPDT_EXP:

```
DEFINE LOG TST_LOG
  LOGPROC LOGPGM
    PARM :UPDT_EXP;
```

Figure A-4. Supplying a parameter using the PARM option

Notice that the PARM expression is evaluated only once, before the first call to the procedure. The resulting value is then passed to the procedure on all calls.

A log or record procedure may contain SQL calls. A procedure containing SQL calls must be precompiled, and the DBRM must be bound together with the log collector DBRM.

Calling assembler procedures

About this task

The call to a procedure written in assembler follows the standard linkage conventions of System/390:
- R15 contains the entry address.
- R14 contains the return address.
- R13 points to a 72-byte save area.

- R1 points to a parameter list.

The procedure is invoked in 31-bit mode. Before the procedure returns control, it must perform all administrative tasks, such as restoring registers.

You can choose between two alternative interfaces to the procedure, each using a different parameter list. You choose the interface by specifying LANGUAGE ASM or LANGUAGE ASML on your DEFINE LOG or DEFINE LOGPROC statement.

Using LANGUAGE ASM interface

The parameter list for a log or record procedure using the interface specified as LANGUAGE ASM has this layout:

```
PARMLIST        DSECT
RESERVED        DS    A   Not used by the procedure
P_CALL_TYPE     DS    A   Address of CALL_TYPE
P_RETURN_CODE   DS    A   Address of RETURN_CODE
PP_IN_RECORD    DS    A   Address of P_IN_RECORD
PP_OUT_RECORD   DS    A   Address of P_OUT_RECORD
PP_WORK_AREA    DS    A   Address of P_WORK_AREA
PP_PARM         DS    A   Address of P_PARM
```

CALL_TYPE

A fullword. Indicates the type of call. It can have one of these values:

0 First call
1 Normal call
2 Last call
3 Commit call.

RETURN_CODE

A fullword. Receives the return code from the procedure. The return code can have one of these values:

0 No record has been built.
1 A record has been built, and there is no more output for this input.
2 A record has been built, and there are more output records for this input.
3 No record has been built because the input record is not correct.
4 No record has been built and processing should end immediately.

The values of return code expected from different call types are shown in Table A-1 on page A-8.

P_IN_RECORD

A fullword containing a pointer to the input record. The pointer is supplied only on those call types that receive an input record. Otherwise it is 0.

The procedure must not modify the input record or the pointer to it.

P_OUT_RECORD

A fullword. Receives a pointer to the output record built by the procedure. The log collector uses this pointer only if return code indicates that a record has been built.

The value of `P_OUT_RECORD` on entry to the procedure is undefined. It is normally **not** the same as set by the preceding call to the procedure.

The first two bytes of the output record must contain the length of the record as an unsigned binary integer. The length includes these two bytes.

P_WORK_AREA
A fullword. Provides a way for preserving information between the consecutive calls to the procedure. The value placed in P_WORK_AREA by the first call is supplied there on all subsequent calls to the procedure. You will normally use it as a pointer to a work area that contains all data that you want to preserve between the calls.

P_PARM A fullword. Contains a pointer to the result of the expression specified using the PARM option. The format of the result depends on data type of the expression, and may be an integer, an 8-byte floating-point number, or a character string preceded by a two-byte length field. (The designer of the procedure must know what format to expect and how to interpret the value.)

If the result of PARM expression is null, the value passed to the procedure is a zero or an empty string, depending on data type of the expression. If PARM was not specified for the procedure, P_PARM is 0.

Table A-1 on page A-8 summarizes the use of parameters in different types of calls.

Using LANGUAGE ASML interface

The interface specified as LANGUAGE ASML differs from that specified as LANGUAGE ASM by the method of returning the length of the output record. Instead of the length being returned in a field within the record, it is returned in a separate parameter. Another difference is the absence of RESERVED as the first parameter. The parameter list has this layout:

```
PARMLIST        DSECT
P_CALL_TYPE     DS    A   Address of CALL_TYPE
P_RETURN_CODE   DS    A   Address of RETURN_CODE
PP_IN_RECORD    DS    A   Address of P_IN_RECORD
PP_OUT_RECORD   DS    A   Address of P_OUT_RECORD
P_OUT_LENGTH    DS    A   Address of OUT_LENGTH
PP_WORK_AREA    DS    A   Address of P_WORK_AREA
PP_PARM         DS    A   Address of P_PARM
```

OUT_LENGTH
A fullword. Receives length of the output record.

Other parameters are the same as for the ASM interface. Notice the absence of RESERVED.

Calling C procedures

The procedures written in C require Version 2 of C/370™ and are run using a persistent C environment (HOTC). The environment is established before the initial call to the procedure and terminated after the last call. The environment is implemented by modules EDCXHOTL, EDCXHOTU, and EDCXHOTT. These modules are a part of C/370 and are not supplied with Tivoli Decision Support for z/OS. To execute procedures written in C, you must link edit these modules with the Tivoli Decision Support for z/OS module DRL2CTOP.

Figure A-5 shows sample JCL that you can use to link edit DRL2CTOP. (See also DRLJCLNK in DRL180.SDRLCNTL.) Check the names of C/370 libraries on your installation. If they are not EDC.V2R1M0.SEDCSPC and EDC.V2R1M0.SEDCBASE, use your names instead.

```
//jobname  JOB parameters
//LKED     EXEC PGM=IEWL,
//         PARM='DCBS,MAP,LIST,LET,TEST,RENT,XREF,REUS,RMODE(ANY),AMODE(31)'
//SYSLIB   DD DISP=SHR,DSN=EDC.V2R1M0.SEDCSPC
//         DD DISP=SHR,DSN=EDC.V2R1M0.SEDCBASE
//SYSLMOD  DD DISP=OLD,DSN=DRL180.SDRLLOAD
//DD1      DD DISP=SHR,DSN=DRL180.ADRLLOAD
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,DISP=(NEW,DELETE),
//            SPACE(32000,(30,30))
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
  INCLUDE DD1(DRLPCIBM)
  INCLUDE SYSLMOD(DRL2CTOP)
  INCLUDE DD1(DRLPXPRO)
  INCLUDE DD1(DRLPXEPI)
  ENTRY   DRL2CTOP
  NAME    DRL2CTOP(R)
```

Figure A-5. Sample JCL for linking the DRL2CTOP module

The persistent C environment requires that you specify the size of the stack to be used by the C procedures. (The size of the stack is passed as a parameter to initialization routine EDCXHOTC. Refer to *IBM C/370 Programming Guide* for more information.)

The log collector specifies for you the stack size of 4 096 bytes. You can request a different stack size using the log collector variable CSTACK. The value of the variable specifies the size of the stack in bytes or Kbytes, like this:

```
SET CSTACK ='8192';
SET CSTACK ='2K';
```

The first SET statement specifies a stack of 8 192 bytes. The second specifies a stack of 2 048 bytes. The log collector does not check the value that you specify. Specifying a value that is too large may cause an unpredictable result. The stack is always allocated above the 16MB line.

Using LANGUAGE C interface

A log or record procedure written in C must conform to this declaration:

```
       #pragma linkage(name,OS)

       void name(int          reserved,
                 int          call_type,
                 int          return_code,
                 in_record  **p_in_record,
                 out_record **p_out_record,
                 work_area  **p_work_area,
                 parm_type  **p_parm)
```

name The name of the procedure, as specified by the DEFINE LOG or DEFINE RECORDPROC statement.

reserved
This parameter is not used by the procedure.

call_type
Indicates the type of call. It can have one of these values:

0 First call
1 Normal call
2 Last call
3 Commit call.

return_code
Receives the return code from the procedure. The return code can have one of these values:

0 No record has been built.
1 A record has been built, and there is no more output for this input.
2 A record has been built, and there are more output records for this input.
3 No record has been built because the input record is not correct.
4 No record has been built and processing should end immediately.

The values of return code expected from different call types are shown in Table A-1 on page A-8.

***p_in_record**
A pointer to the input record. It is supplied only on those call types that receive an input record. Otherwise it is null.

in_record
A structure representing the layout of the input record. The procedure must not modify the input record or the pointer to it.

***p_out_record**
Receives a pointer to the output record built by the procedure. The log collector uses this pointer only if return code indicates that a record has been built.

The value of `*p_out_record` on entry to the procedure is undefined. It is normally **not** the same as set by the preceding call to the procedure.

out_record
A structure representing the layout of the output record. This layout must be independently defined to the log collector by means of a DEFINE RECORD statement.

The first two bytes of the record must contain the length of the record in the form of an unsigned short integer.

***p_work_area**
Provides a way for preserving information between the consecutive calls to the procedure. The value returned in this parameter by the first call is supplied in this parameter on all subsequent calls to the procedure. You will normally use it as a pointer to a work area that contains all data that you want to preserve between the calls.

work_area
A structure containing the data that you want to preserve between the calls to the procedure.

***p_parm**

A pointer to the result of the expression specified by the PARM option.

parm_type

Data type of the result of PARM expression. It depends on data type of the expression, and may be an integer, a floating-point number, or a character string preceded by a two-byte length. Depending on what is expected, parm_type should be one of these:

```
int
double
struct
  {
    short  lgth;
    char   parm[n];
  }
```

where n is not less than the maximum expected length of the string. (The designer of the procedure must know what type to expect and how to interpret the value.)

If the result of PARM expression is null, the value passed to the procedure is a zero or an empty string, depending on the type of the expression. If PARM was not specified for the procedure, p_parm is null.

Table A-1 summarizes the use of parameters in different types of calls.

Table A-1. Input and output of log and record procedures

Input				Output			
CALL_TYPE ;call_type	**P_IN_RECORD ;*p_in_record**	**P_WORK_AREA ;*p_work_area**	**P_PARM ;*p_parm**	**RETURN_CODE ;return_code**	**P_OUT_RECORD ;*p_out_record**	**OUT_LENGTH ;(ASML only)**	**P_WORK_AREA ;*p_work_area**
0 ;(first)	null	null	&parm or null	0	not used	not used	&work_area
				4	not used	not used	&work_area
1 ;(normal)	&in_record	&work_area	&parm or null	0	not used	not used	&work_area
				1	&out_record	out length	&work_area
				2	&out_record	out length	&work_area
				3	not used	not used	&work_area
				4	not used	not used	&work_area
2 ;(last)	null	&work_area	&parm or null	0	not used	not used	&work_area
				1	&out_record	out length	&work_area
				2	&out_record	out length	&work_area
3 ;(commit)	null	&work_area	&parm or null	0	not used	not used	&work_area
				4	not used	not used	&work_area

Example log procedures

Suppose you want to process a log XMP where each record contains a date in the format exemplified by 14MAR00. This date format is not supported by the log collector. You might define the three parts of the date as three fields in CHAR format, use the facilities of the log collector language to build from them a date string 00-03-14, and then convert that string to date using the DATE function. You would have to do this in every update definition that uses the date. If you have many such update definitions, it may be simpler to convert the date using a log procedure.

An example of a log procedure that you might write to perform this conversion is shown in “Example C log procedure.” An Assembler version of the same procedure is shown in “Example Assembler log procedure” on page A-12.

In order to illustrate the use of the parameter specified via PARM clause, the example assumes that the month part of the date in the input record may sometimes be missing or invalid. In such a case, you want to replace it by a default supplied by means of PARM.

```
DEFINE LOG XMP
  LOGPROC XMPPROC
    LANGUAGE C
    PARM :DEF_MONTH;
```

The default month is supplied via the variable DEF_MONTH that must be set every time you work with the log XMP. The value of the variable should be a two-digit string from 01 through 12.

The input record from the log data set is represented in the procedure by the structure in_record. The first seven bytes contain date as described above, and the next six bytes contain a transaction count represented as an external integer.

The structure out_record represents a record built by the procedure. The record starts with a two-byte length field, followed by the converted date in the format DATE(YYMMDD), followed by a copy of transaction count from the input record. You define this record to the log collector like this:

```
DEFINE RECORD XMP_REC
  IN LOG XMP
  FIELDS
   (RECLEN       OFFSET 0  LENGTH 2  BINARY,
    DATE         OFFSET 2  LENGTH 6  DATE(YYMMDD),
    TRANS_COUNT  OFFSET 8  LENGTH 6  EXTERNAL INTEGER);
```

You cannot build the output record in a variable that is local to your procedure, because it must be accessible to the log collector after a return from the procedure. So, the first call to the procedure must allocate a buffer for the output record that will remain allocated until it is freed by the last call.

The output buffer is placed in a data area that is allocated by the first call and freed by the last call. The layout of this area is represented by the structure work_area. In this example, the work area contains only the output buffer. In general, you include there all information that you need to store between the calls to the procedure.

Other details of the procedure are explained by comments.

Example C log procedure

This section shows an example C log procedure. An Assembler version of the same procedure is shown in “Example Assembler log procedure” on page A-12.

```
/*=================================================================*/
/*  Example of a log procedure written in C                        */
/*=================================================================*/
#pragma linkage(XmpProc,OS)
```

Example log procedures

```
  #include <stdio.h>
  #include <stdlib.h>

  typedef struct                        /* Input record              */
    {  char  day[2];                    /*   Day, DD                 */
       char  month[3];                  /*   Month, MMM              */
       char  year[2];                   /*   Year, YY                */
       char  trans_count[6];            /*   Transaction count       */
    } in_record;

  typedef struct                        /* Output record             */
    {  short  lgth;                     /*   Length                  */
       char  year[2];                   /*   Year, YY                */
       char  month[2];                  /*   Month, MM               */
       char  day[2];                    /*   Day, DD                 */
       char  trans_count[6];            /*   Transaction count       */
    } out_record;

  typedef struct                        /* Work area                 */
    {  out_record out_buffer;           /*   Buffer for output record */
    } work_area;

  typedef struct                        /* Parm string               */
    {  short  lgth;                     /*   Length                  */
        char  parm[2];                  /*   Default month, MM       */
    } parm_string;

  /*------------------------------------------------------------------*/
  /*  Start of procedure                                              */
  /*------------------------------------------------------------------*/
  void XmpProc
   (int           reserved,
    int           call_type,
    int           return_code,
    in_record   **p_in_record,
    out_record  **p_out_record,
    work_area   **p_work_area,
    parm_string **p_parm)

  {
    /*----------------------------------------------------------------*/
    /*  Local variables                                               */
    /*----------------------------------------------------------------*/
    in_record   *pi;                    /* Local ptr to input record  */
    work_area   *pw;                    /* Local ptr to work area     */
    parm_string *pp;                    /* Local ptr to parm string   */
    int         m;                      /* Index in month table       */
    char        *month_in[12] =         /* Month codes in             */
                  {
                   "JAN","FEB","MAR","APR","MAY","JUN",
                   "JUL","AUG","SEP","OCT","NOV","DEC"
                  };
    char        *month_out[12] =        /* Month codes out            */
                  {
                   "01","02","03","04","05","06",
                   "07","08","09","10","11","12"
                  };
switch (call_type)
{
/*----------------------------------------------------------------*/
/*  First call                                                    */
/*----------------------------------------------------------------*/
case (0):

  /* Allocate work area. */
  pw = (work_area *)malloc(sizeof(work_area));
  /* Set output record length. */
  pw->out_buffer.lgth = sizeof(out_record);

  /* Save work area ptr and return indicating success. */
  *p_work_area = pw;
  return_code = 0;
  break;

/*----------------------------------------------------------------*/
/*  Normal call                                                   */
/*----------------------------------------------------------------*/
case (1):

  /* Set local pointer to input record. */
  /* Set local pointer to parm value.   */
  /* Retrieve pointer to work area.     */
  pi = *p_in_record;
  pp = *p_parm;
  pw = *p_work_area;

  /* Copy year, day, and transaction count to output record. */
  /* Set month to default.                                   */
  strncpy(pw->out_buffer.year,pi->year,2);
  strncpy(pw->out_buffer.month,pp->parm,2);
  strncpy(pw->out_buffer.day,pi->day,2);
  strncpy(pw->out_buffer.trans_count,pi->trans_count,6);

  /* Find the three-letter month code in the table.            */
  /* If found, set corresponding month number in output record. */
  for (m=0; m<12; m++)
  {
   if (strncmp(month_in[m],pi->month,3)==0)
   {
    strncpy(pw->out_buffer.month,month_out[m],2);
    break;
   }
  }

  /* Return pointer to output record and indicate 1 record built. */
  *p_out_record = &(pw->out_buffer);
  return_code = 1;
  break;

    /*----------------------------------------------------------------*/
    /*  Last call                                                     */
    /*----------------------------------------------------------------*/
    case (2):

      /* Free work area. */
      free(*p_work_area);

      /* Return indicating no record built. */
      return_code = 0;
      break;

    /*----------------------------------------------------------------*/
    /*  Commit call                                                   */
    /*----------------------------------------------------------------*/
    case (3):

      /* No action: indicate success. */
      return_code = 0;
      break;

    /*----------------------------------------------------------------*/
    /*  Invalid call type                                             */
    /*----------------------------------------------------------------*/
    default:

      /* Request termination. */
      return_code = 4;
      break;
   }
  }
```

Example Assembler log procedure

This section shows an Assembler version of the C log procedure shown in "Example C log procedure" on page A-9.

```
*======================================================================
        *  Example of a log procedure written in assembler (ASM interface)
        *======================================================================
        *----------------------------------------------------------------------
        *  Standard prologue
        *----------------------------------------------------------------------
        XMPPROC  CSECT ,
        XMPPROC  AMODE 31
        XMPPROC  RMODE ANY
                 DS    0H
                 USING *,R15
                 B     PROLOG               Branch around identification.
                 DC    AL1(16)
                 DC    C'XMPPROC     95.150' Standard module identification
                 DROP  R15
        PROLOG   STM   R14,R12,12(R13)      Save registers.
                 LR    R12,R15              Set R12 as base for the module.
                 USING XMPPROC,R12          Use R12 as base.
        *
                 SR    R15,R15              | Get dynamic storage
                 LA    R0,DYNSIZE           | for the module.
                 GETMAIN  RU,LV=(0),SP=(15) | Address to R1.
        *
                 LR    R15,R13              |
                 LR    R13,R1               | Link save areas.
                 USING DYNSTOR,R13          | Use R13 as base
                 ST    R15,4(,R13)          | for dynamic storage.
                 ST    R13,8(,R15)          |
        *
                 L     P_LIST,24(R15)       Retrieve parameter list address
                 USING PARMLIST,P_LIST      Use it to address parameter list
        *
                 USING IN_RECORD,PI         Use PI to address IN_RECORD
                 USING WORK_AREA,PW         Use PW to address WORK_AREA
                 USING OUT_RECORD,PW        Record buffer is first in WORK_AREA
                 USING PARM_STRING,PP       Use PP to address PARM_STRING
        *----------------------------------------------------------------------
        *  Branch according to call type
        *----------------------------------------------------------------------
                 L     R1,P_CALL_TYPE       R1 = addr of CALL_TYPE
                 L     R1,0(,R1)            R1 = CALL_TYPE
                 BM    INVALID_CALL         If CALL_TYPE<0
                 LA    R0,3
                 CR    R1,R0
                 BH    INVALID_CALL         If CALL_TYPE>3
                 SLL   R1,2                 R1 = CALL_TYPE * 4
                 B     BRANCH_TABLE(R1)     Branch according to CALL_TYPE
        *
        BRANCH_TABLE   DS   0H
                 B     FIRST_CALL
                 B     NORMAL_CALL
                 B     LAST_CALL
                 B     COMMIT_CALL
```

```
*----------------------------------------------------------------------
*  First call
*----------------------------------------------------------------------
FIRST_CALL     DS   0H
*
*  Allocate work area
*
         SR    R15,R15              | Get storage
         LA    R0,WORKSIZE          | for work area.
         GETMAIN  RU,LV=(0),SP=(15) | Address to R1.
*
         LR    PW,R1                PW = address of work area.
*
*  Set output record length
*
         LA    R0,OUTSIZE
         STH   R0,OUT_LGTH          OUT_LGTH = length of OUT_RECORD
*
*  Save work area address and return indicating success
*
         L     R1,PP_WORK_AREA
         ST    PW,0(,R1)            P_WORK_AREA = PW
         SR    R0,R0
         L     R1,P_RETURN_CODE
         ST    R0,0(,R1)            RETURN_CODE = 0
         B     EPILOG
*----------------------------------------------------------------------
*  Normal call
*----------------------------------------------------------------------
NORMAL_CALL    DS   0H
*
*  Set local pointer to input record.
*  Set local pointer to parm value.
*  Retrieve pointer to work area.
*
         L     R1,PP_IN_RECORD
         L     PI,0(,R1)            PI = address of IN_RECORD
         L     R1,PP_PARM
         L     PP,0(,R1)            PP = address of PARM_STRING
         L     R1,PP_WORK_AREA
         L     PW,0(,R1)            PW = address of WORK_AREA
*
*  Copy year, day and transaction count to output record.
*  Set month to default.
*
         MVC OUT_YEAR,IN_YEAR
         MVC OUT_MONTH,PARM
         MVC OUT_DAY,IN_DAY
         MVC OUT_TRANS_COUNT,IN_TRANS_COUNT        *
*  Find the three-letter month code in table.
*  If found, set corresponding month number in output record.
*
         LA    M,1                  M = 1
*
LOOP     LR    R1,M                 R1 = M
         SLL   R1,2                 R1 = M * 4
         LA    R1,MONTHS-4(R1)      R1 = address of MONTHS(M)
         CLC   0(3,R1),IN_MONTH
         BE    FOUND                If MONTH(M)=IN_MONTH
         LA    M,1(,M)              M = M + 1
         LA    R2,12
         CR    M,R2
         BNH   LOOP                 If M<=12
         B     READY                If M>12
*
FOUND    LR    R1,M                 R1 = M
         SLL   R1,1                 R1 = M * 2
         LA    R1,DEC_M-2(R1)       R1 = address of DEC_M(M)
         MVC   OUT_MONTH,0(R1)      OUT_MONTH = DEC_M(M)
*
*  Return pointer to output record and indicate one record built.
*
READY    EQU   *
         L     R1,PP_OUT_RECORD
         ST    PW,0(,R1)            P_OUT_RECORD = PW
         LA    R0,1
         L     R1,P_RETURN_CODE
         ST    R0,0(,R1)            RETURN_CODE = 1
         B     EPILOG
*----------------------------------------------------------------------
*  Last call
*----------------------------------------------------------------------
LAST_CALL      DS   0H
*
*  Free work area
*
         L     R1,PP_WORK_AREA
         L     R1,0(,R1)            R1 = address of WORK_AREA
         LA    R15,0
         LA    R0,WORKSIZE          R0 = length of WORK_AREA
         FREEMAIN RU,LV=(0),A=(1),SP=(15)
*
*  Return code indicating no record built
*
         SR    R0,R0
         L     R1,P_RETURN_CODE
         ST    R0,0(,R1)            RETURN_CODE = 0
         B     EPILOG
*----------------------------------------------------------------------
*  Commit call
*----------------------------------------------------------------------
COMMIT_CALL    DS   0H
*
*  No action: indicate success.
*
         SR    R0,R0
         L     R1,P_RETURN_CODE
         ST    R0,0(,R1)            RETURN_CODE = 0
         B     EPILOG

*----------------------------------------------------------------------
*  Invalid call type
*----------------------------------------------------------------------
INVALID_CALL   DS   0H
*
*  Request termination.
*
         LA    R0,4
         L     R1,P_RETURN_CODE
         ST    R0,0(,R1)            RETURN_CODE = 4
         B     EPILOG
*----------------------------------------------------------------------
*  Standard epilogue
*----------------------------------------------------------------------
EPILOG   DS    0H
         LR    R1,R13               | R1 = address of dynamic storage
         L     R13,4(,R13)          | Restore R13 from save area
*
         LA    R15,0                | Free dynamic storage
         LA    R0,DYNSIZE
         FREEMAIN RU,LV=(0),A=(1),SP=(15)
*
         LM    R14,R12,12(R13)      Restore registers
         BR    R14                  Return to caller
```

```
*----------------------------------------------------------------------
*  Table of three-letter month codes
*----------------------------------------------------------------------
MONTHS   DC    CL4'JAN'
         DC    CL4'FEB'
         DC    CL4'MAR'
         DC    CL4'APR'
         DC    CL4'MAY'
         DC    CL4'JUN'
         DC    CL4'JUL'
         DC    CL4'AUG'
         DC    CL4'SEP'
         DC    CL4'OCT'
         DC    CL4'NOV'
         DC    CL4'DEC'
*----------------------------------------------------------------------
*  Table of month numbers
*----------------------------------------------------------------------
DEC_M    DC    CL2'01'
         DC    CL2'02'
         DC    CL2'03'
         DC    CL2'04'
         DC    CL2'05'
         DC    CL2'06'
         DC    CL2'07'
         DC    CL2'08'
         DC    CL2'09'
         DC    CL2'10'
         DC    CL2'11'
         DC    CL2'12'

*----------------------------------------------------------------------
*  Dynamic storage for the module
*----------------------------------------------------------------------
DYNSTOR         DSECT
                DS    18F           Save area
                DS    0D
DYNSIZE         EQU   *-DYNSTOR
*
*----------------------------------------------------------------------
*  Parameter list
*----------------------------------------------------------------------
PARMLIST        DSECT
RESERVED        DS    A             Not used by the procedure
P_CALL_TYPE     DS    A             Address of CALL_TYPE
P_RETURN_CODE   DS    A             Address of RETURN_CODE
PP_IN_RECORD    DS    A             Address of P_IN_RECORD
PP_OUT_RECORD   DS    A             Address of P_OUT_RECORD
PP_WORK_AREA    DS    A             Address of P_WORK_AREA
PP_PARM         DS    A             Address of P_PARM
*
*----------------------------------------------------------------------
*  Input record
*----------------------------------------------------------------------
IN_RECORD       DSECT
IN_DAY          DS    CL2           Day, DD
IN_MONTH        DS    CL3           Month, MMM
IN_YEAR         DS    CL2           Year, YY
IN_TRANS_COUNT  DS    CL6           Transaction count
*
*----------------------------------------------------------------------
*  Output record
*----------------------------------------------------------------------
OUT_RECORD      DSECT
OUT_LGTH        DS    H             Length
OUT_YEAR        DS    CL2           Year, YY
OUT_MONTH       DS    CL2           Month, MM
```

```
OUT_DAY         DS    CL2           Day, DD
OUT_TRANS_COUNT DS    CL6           Transaction count
OUTSIZE         EQU   *-OUT_RECORD
*----------------------------------------------------------------------
*  Work area
*----------------------------------------------------------------------
WORK_AREA       DSECT
OUT_BUFFER      DS    CL14          Buffer for output record
                DS    0D            Doubleword alignment
WORKSIZE        EQU   *-WORK_AREA
*----------------------------------------------------------------------
*  Result of PARM expression
*----------------------------------------------------------------------
PARM_STRING     DSECT
PARM_LGTH       DS    H             Length
PARM            DS    CL2           Parameter string

*----------------------------------------------------------------------
*  Registers
*----------------------------------------------------------------------
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
*
P_LIST   EQU   R7                   Pointer to parameter list
PI       EQU   R8                   Local pointer to input record
PW       EQU   R9                   Local pointer to work area
PP       EQU   R10                  Local pointer to parm string
M        EQU   R11                  Month number
         END
```

Appendix B. JCL for the log collector language and report definition language

When you use JCL to submit batch jobs for either the log collector language or the report definition language, you can specify a number of different parameters based on your installation. This appendix describes the parameters that you can specify.

JCL for the log collector language

Figure B-1 shows sample JCL that you can use when running the log collector in batch. (See also DRLJCOLL in DRL180.SDRLCNTL.)

```
//jobname  JOB parameters
//LC       EXEC PGM=DRLPLC,PARM=('SYSPREFIX=DRLSYS SYSTEM=DSN'),
//         REGION=1M
//STEPLIB  DD DISP=SHR,DSN=DRL180.SDRLLOAD
//DRLIN    DD *
           COLLECT ERR_EXMP;
//DRLLOG   DD DISP=SHR,DSN=ABC.ERROR.EXAMPLE
//DRLOUT   DD SYSOUT=*
//DRLDUMP  DD SYSOUT=*
```

Figure B-1. Sample JCL for the log collector

You use the **PARM= parameter** to define log collector variables. The string supplied with **PARM=**consists of one or more variable definitions separated by blanks or commas. A variable definition has one of these forms:

variable-name=string
&variable-name=string

It creates a variable named *variable-name* with a value *string*. The *variable-name* must be an identifier and *string* can be any character string. If *string* contains blanks or commas, it must be enclosed in quotation marks (").

For information about how variables are used, see “Using variables to modify your text” on page 8-7 and “Obtaining the value of a variable” on page 9-4.

These four variables have a special meaning to the log collector

SYSTEM
Specifies the name of the DB2 subsystem that manages the tables that make up the Tivoli Decision Support for z/OS database. If you do not define this variable, it is defined by default with the value DSN.

PLAN Specifies the name of the DB2 application plan as defined in the bind job for Tivoli Decision Support for z/OS. If you do not define this variable, it is defined by default with the value DRLPLAN.

SYSPREFIX
Specifies the prefix of system table names. If you do not define this variable, it is defined by default with the value DRLSYS.

SHOWINPUT
By defining this variable with value NO, you suppress copying of log

collector statements to the DRLOUT file. But, keep in mind that you may then have problems in interpreting the messages issued by the log collector.

There are also other variables that have a special meaning to the log collector. They have names starting with ZZ, and are used to control certain diagnostic functions. See the , SH19-6902 book for their description.

You can use these DD names in the job:

DRLIN DD
Specifies the data set that contains log collector language statements. You can use an asterisk (*) to include the statements directly in the JCL jobstream. It can contain fixed-length or varying-length records of any length, but the log collector reads a maximum of 72 bytes from each record.

DRLOUT DD
Specifies the data set that will contain messages generated by the log collector. It can have fixed length or varying-length records of any length, but the log collector assumes a length of at least 80 bytes for formatting. Lines that are no longer than the specified record length are wrapped to the next line. DRLOUT is allocated as RECFM=F and LRECL=80 if no DCB attributes are specified.

DRLLOG DD
Specifies the log data set you want to process. You can specify concatenated data sets with different attributes. This statement is required only for log collector statements that process log data. The date set attributes are determined by the creator of the log.

DRLLST*x*DD
Specifies the data sets that will contain output from the LIST RECORD statement. This statement is required only if you use LIST RECORD.

DRLDUMP DD
Specifies the data set that the log collector will use to write diagnostic information. It can have fixed-length or varying-length records of any length, but the log collector assumes a length of at least 80 bytes for formatting. Lines that are longer than the specified record length are wrapped to the next line. DRLDUMP is allocated as RECFM=F and RECL=80 if no DCB attributes are specified.

JCL for the report definition language

Figure B-2 on page B-3 shows sample JCL you can use when running the report definition language in batch. (See also DRLJRDEF in DRL180.SDRLCNTL.)

```
//*-----------------------------------------------------------------*
//EPDMRDEF EXEC PGM=IKJEFT01
//STEPLIB  DD DISP=SHR,DSN=DRL180.SDRLLOAD
//         DD DISP=SHR,DSN=qmfloadlibrary
//         DD DISP=SHR,DSN=db2loadlibrary
//SYSPROC  DD DISP=SHR,DSN=DRL180.SDRLEXEC
//         DD DISP=SHR,DSN=qmfclistlibrary
//SYSEXEC  DD DISP=SHR,DSN=qmfexeclibrary
//*-----------------------------------------------------------------*
//*  QMF allocations                                                *
//*-----------------------------------------------------------------*
//DSQDEBUG DD DUMMY
//DSQUDUMP DD DUMMY
//DSQPNLE  DD DISP=SHR,DSN=QMFDSQPNLxlibrary
//DSQSPILL DD DSN=&&SPILL,DISP=(NEW,DELETE),UNIT=SYSDA,
//  SPACE=(CYL,(1,1),RLSE),DCB=(RECFM=F,LRECL=4096,BLKSIZE=4096)
//DSQEDIT  DD DSN=&&EDIT,UNIT=SYSDA,SPACE=(CYL,(1,1),RLSE),
//  DCB=(RECFM=FBA,LRECL=79,BLKSIZE=4029)
//DSQPRINT DD SYSOUT=*,DCB=(RECFM=FBA,LRECL=133,BLKSIZE=1330)
//ADMGGMAP DD DISP=SHR,DSN=ADMGGMAPlibrary
//ADMCFORM DD DISP=SHR,DSN=ADMCFORMlibrary
//DSQUCFRM DD DISP=SHR,DSN=DRL180.SDRLFENU
//*-----------------------------------------------------------------*
//*  Performance Reporter allocations                               *
//*-----------------------------------------------------------------*
//DRLIN    DD DISP=SHR,DSN=DRL180.SDRLRENU(DRLOSAMP)
//         DD DISP=SHR,DSN=DRL180.SDRLRENU(DRLORACF)
//DRLOUT   DD SYSOUT=*
//DRLDEFS1 DD DISP=SHR,DSN=DRL.LOCAL.DEFS
//DRLDEFS2 DD DISP=SHR,DSN=DRL180.SDRLRENU
//DRLDEFS3 DD DISP=SHR,DSN= ....
//*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 %DRLERDEF SYSTEM=DSN SYSPREFIX=DRLSYS PREFIX=DRL -
           MODE=BATCH SHOWINPUT=YES
```

Figure B-2. JCL for defining reports in batch

You can use these DD names in the job:

DRLIN DD
Specifies the input data set that contains report definition language statements. You can use an asterisk (*) to include the report definition language statements directly in the JCL jobstream.

DRLOUT DD
Specifies the data set that will contain messages generated during processing by Tivoli Decision Support for z/OS.

DRLDEFS1, DRLDEFS2, DRLDEFS3 DD
Specifies the data sets containing the QMF queries and forms defined in the report definition statements. DRLDEFS1 is searched first and must exist. The DRLDEFS2 and DRLDEFS3 data sets are optional and are searched only if a defined query or form is not found in DRLDEFS1.

Reporting definition language exec

The reporting definition language uses an exec called DRLERDEF to process report definition language statements. You can specify these parameters for the DRLERDEF exec:

SYSTEM=*db2-system*
db2-system is the name of the DB2 subsystem that manages the DB2 tables that make up the Tivoli Decision Support for z/OS database. The default name is DSN.

SYSPREFIX=*sysprefix*
sysprefix identifies the prefix (owner) of the system tables. The default is DRLSYS.

PREFIX=*prefix*
prefix identifies the prefix (owner) of the queries and forms that are imported to QMF during Tivoli Decision Support for z/OS report definition processing. The default is DRL.

SHOWINPUT=*YES***/NO**
Specifies whether the statements read from the input file (determined from DRLIN) are written to the output message file (determined by DRLOUT).

MODE=BATCH/ONLINE
Specifies how the processing is to occur. MODE=BATCH must be specified when you wish to process Tivoli Decision Support for z/OS report definition statements in batch.

SHOWSQL YES/*NO*
The SHOWSQL parameter specifies whether SQL statements should be shown (for debugging).

QMF=*YES***/NO**
The QMF parameter specifies whether QMF is used.

Appendix C. Support information

If you have a problem with your IBM software, you want to resolve it quickly. This section describes the following options for obtaining support for IBM software products:

- "Searching knowledge bases"
- "Obtaining fixes"
- "Receiving weekly support updates" on page C-2
- "Contacting IBM Software Support" on page C-2

Searching knowledge bases

Searching the information center

IBM provides extensive documentation that can be installed on your local computer or on an intranet server. You can use the search function of this information center to query conceptual information, instructions for completing tasks, and reference information.

Searching the Internet

If you cannot find an answer to your question in the information center, search the Internet for the latest, most complete information that might help you resolve your problem.

To search multiple Internet resources for your product, use the **Web search** topic in your information center. In the navigation frame, click **Troubleshooting and support** ▸ **Searching knowledge bases** and select **Web search**. From this topic, you can search a variety of resources, including the following:

- IBM technotes
- IBM downloads
- IBM developerWorks®
- Forums and newsgroups
- Google

Obtaining fixes

A product fix might be available to resolve your problem. To determine what fixes are available for your IBM software product, follow these steps:

1. Go to the IBM Software Support Web site at http://www.ibm.com/software/support/.
2. Click **Downloads and drivers** in the **Support topics** section.
3. Select the **Software** category.
4. Select a product in the **Sub-category** list.
5. In the **Find downloads and drivers by product** section, select one software category from the **Category** list.
6. Select one product from the **Sub-category** list.
7. Type more search terms in the **Search within results** if you want to refine your search.

8. Click **Search**.
9. From the list of downloads returned by your search, click the name of a fix to read the description of the fix and to optionally download the fix.

For more information about the types of fixes that are available, see the *IBM Software Support Handbook* at http://www-304.ibm.com/support/customercare/sas/f/handbook/home.html.

Receiving weekly support updates

To receive weekly e-mail notifications about fixes and other software support news, follow these steps:

1. Go to the IBM Software Support Web site at http://www.ibm.com/support/us/.
2. Click **My support** in the upper right corner of the page.
3. If you have already registered for **My support**, sign in and skip to the next step. If you have not registered, click **register now**. Complete the registration form using your e-mail address as your IBM ID and click **Submit**.
4. Click **Edit profile**.
5. In the **Products** list, select **Software**. A second list is displayed.
6. In the second list, select a product segment, for example, **Application servers**. A third list is displayed.
7. In the third list, select a product sub-segment, for example, **Distributed Application & Web Servers**. A list of applicable products is displayed.
8. Select the products for which you want to receive updates, for example, **IBM HTTP Server** and **WebSphere® Application Server**.
9. Click **Add products**.
10. After selecting all products that are of interest to you, click **Subscribe to email** on the **Edit profile** tab.
11. Select **Please send these documents by weekly email**.
12. Update your e-mail address as needed.
13. In the **Documents** list, select **Software**.
14. Select the types of documents that you want to receive information about.
15. Click **Update**.

If you experience problems with the **My support** feature, you can obtain help in one of the following ways:

Online
Send an e-mail message to erchelp@ca.ibm.com, describing your problem.

By phone
Call 1-800-IBM-4You (1-800-426-4968).

Contacting IBM Software Support

IBM Software Support provides assistance with product defects.

Before contacting IBM Software Support, your company must have an active IBM software maintenance contract, and you must be authorized to submit problems to IBM. The type of software maintenance contract that you need depends on the type of product you have:

- For IBM distributed software products (including, but not limited to, Tivoli, Lotus®, and Rational® products, as well as DB2 and WebSphere products that run on Windows, or UNIX operating systems), enroll in Passport Advantage® in one of the following ways:

  Online
  Go to the Passport Advantage Web site at http://www.lotus.com/services/passport.nsf/ WebDocs/Passport_Advantage_Home and click **How to Enroll**.

  By phone
  For the phone number to call in your country, go to the IBM Software Support Web site at http://techsupport.services.ibm.com/guides/contacts.html and click the name of your geographic region.

- For customers with Subscription and Support (S & S) contracts, go to the Software Service Request Web site at https://techsupport.services.ibm.com/ssr/login.
- For customers with IBMLink, CATIA, Linux, S/390®, iSeries, pSeries, zSeries, and other support agreements, go to the IBM Support Line Web site at http://www.ibm.com/services/us/index.wss/so/its/a1000030/dt006.
- For IBM eServer™ software products (including, but not limited to, DB2 and WebSphere products that run in zSeries, pSeries, and iSeries environments), you can purchase a software maintenance agreement by working directly with an IBM sales representative or an IBM Business Partner. For more information about support for eServer software products, go to the IBM Technical Support Advantage Web site at http://www.ibm.com/servers/eserver/techsupport.html.

If you are not sure what type of software maintenance contract you need, call 1-800-IBMSERV (1-800-426-7378) in the United States. From other countries, go to the contacts page of the *IBM Software Support Handbook* on the Web at http://techsupport.services.ibm.com/guides/contacts.html and click the name of your geographic region for phone numbers of people who provide support for your location.

To contact IBM Software support, follow these steps:

1. "Determining the business impact"
2. "Describing problems and gathering information" on page C-4
3. "Submitting problems" on page C-4

Determining the business impact

When you report a problem to IBM, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting. Use the following criteria:

Severity 1
The problem has a *critical* business impact. You are unable to use the program, resulting in a critical impact on operations. This condition requires an immediate solution.

Severity 2
The problem has a *significant* business impact. The program is usable, but it is severely limited.

Severity 3

The problem has *some* business impact. The program is usable, but less significant features (not critical to operations) are unavailable.

Severity 4

The problem has *minimal* business impact. The problem causes little impact on operations, or a reasonable circumvention to the problem was implemented.

Describing problems and gathering information

When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?
- Do you have logs, traces, and messages that are related to the problem symptoms? IBM Software Support is likely to ask for this information.
- Can you re-create the problem? If so, what steps were performed to re-create the problem?
- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, and so on.
- Are you currently using a workaround for the problem? If so, be prepared to explain the workaround when you report the problem.

Submitting problems

You can submit your problem to IBM Software Support in one of two ways:

Online

Click **Submit and track problems** on the IBM Software Support site at http://www.ibm.com/software/support/probsub.html. Type your information into the appropriate problem submission form.

By phone

For the phone number to call in your country, go to the contacts page of the *IBM Software Support Handbook* at http://techsupport.services.ibm.com/guides/contacts.html and click the name of your geographic region.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
2Z4A/101
11400 Burnet Road
Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information in softcopy form, the photographs and color illustrations might not display.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Glossary

administration
A Tivoli Decision Support for z/OS task that includes maintaining the database, updating environment information, and ensuring the accuracy of data collected.

administration dialog
A set of host windows used to administer Tivoli Decision Support for z/OS.

asterisk length
The length of a field that extends to the end of the containing structure.

cascaded update
Occurs during data collection, when information entered in one table is further processed and entered in another table.

case expression
An expression that specifies a value as being dependent on a given condition.

collect A process used by Tivoli Decision Support for z/OS to read data from input log data sets, interpret records in the data set, and store the data in DB2 tables in the Tivoli Decision Support for z/OS database.

component
An optionally-installable part of a Tivoli Decision Support for z/OS feature.Specifically in Tivoli Decision Support for z/OS, a component refers to a logical group of objects used to collect log data from a specific source, to update the Tivoli Decision Support for z/OS database using that data, and to create reports from data in the database.

control table
A predefined Tivoli Decision Support for z/OS table that controls results returned by some log collector functions.

data table
A Tivoli Decision Support for z/OS table that contains performance data used to create reports.

environment information
All of the information that is added to the log data to create reports. This information can include data such as performance groups, shift periods, installation definitions, and so on.

exit anchor
A parameter provided to a log or record procedure. An exit anchor is used upon the first call of the procedure to store a work area location, which is then used in subsequent calls to that procedure.

grouping value
Value that is used to sort records into groups.

key columns
The columns of a DB2 table that together constitute the key.

log collector
A Tivoli Decision Support for z/OS program that processes log data sets and provides other Tivoli Decision Support for z/OS services.

log collector language
Tivoli Decision Support for z/OS statements used to supply definitions to and invoke services of the log collector.

log data set
Any sequential data set that is used as input to Tivoli Decision Support for z/OS.

log definition
The description of a log data set processed by the log collector.

log procedure
A program module that is used to process all record types in certain log data sets.

lookup expression
An expression that specifies how a value is obtained from a lookup table.

lookup table
A Tivoli Decision Support for z/OS DB2 table that contains grouping, translation, or substitution information.

nested section
A section of a record that is location within another section.

Tivoli Decision Support for z/OS database
A set of DB2 tables that includes data tables, lookup tables, system tables, and control tables.

purge condition
Instruction for purging old data from the Tivoli Decision Support for z/OS database.

record definition
The description of a record type contained in the log data sets used by Tivoli Decision Support for z/OS, including detailed record layout and data formats.

record procedure
A program module that is called to process some types of log records.

record type
The classification of records in a log data set.

repeated section
A section of a record that occurs more than once, with each occurrence adjacent to the previous one.

report definition language
Tivoli Decision Support for z/OS statements used to define reports and report groups.

report group
A collection of Tivoli Decision Support for z/OS reports that can be referred to by a single name.

reporting dialog
A set of host or workstation windows used to request reports.

section
A structure within a record that contains one or more fields and may contain other sections.

source
In an update definition, the record or DB2 table that contains the data used to update a Tivoli Decision Support for z/OS DB2 table.

system table
A DB2 table that stores information that controls log collector processing, Tivoli Decision Support for z/OS dialogs, and reporting.

target
In an update definition, the DB2 table in which Tivoli Decision Support for z/OS stores data from the source record or table.

update definition
Instructions for entering data into DB2 tables from records of different types or from other DB2 tables.

Bibliography

TDS publications

, SH19-6816, SH19-6816

, SH19-4495, SH19-4495

, SH19-4019, SH19-4019

, SH19-6820, SH19-6820

, SH19-4018, SH19-4018

, SH19-6842, SH19-6842

, SH19-6825, SH19-6825

, SH19-6817, SH19-6817

, SH19-6902, SH19-6902

, SH19-6901, SH19-6901

, SH19-6822, SH19-6822

, SH19-6821, SH19-6821

, SH19-6818, SH19-6818

, SH19-6819, SH19-6819

, SH19-4494, SH19-4494

, SC23-7966, SC23-7966

Index

Printed in USA

SH19-6817-13