GENESISsql
Users Guide

June 7, 2023

Trifox Inc.
www.trifox.com

Trademarks

TRIMapp, TRImpl, TRIMqmr, TRIMreport, TRIMtools, GENESISsql, DesignVision, DVapp, DVreport, VORTEX, VORTEXcli, VORTEXc, VORTEXcobol, VORTEXperl, VORTEXjdbc, VORTEX++, VORTEXJava Edition, LIST Manager, VORTEXodbc, VORTEXnet, VORTEXclient/server, VORTEXaccelerator, VORTEXreplicator are all trademarks of Trifox, Inc.

All other brand and product names are trademarks or registered trademarks of their respective owners.

Copyright

Printed in the U.S.A.

Contents

Preface

This guide explains how to use GENESISsql, Trifox's SQL processor for low-level data sources. If you write applications that access low-level data sources and flat files, administer, install, or maintain GENESIS sites, or write custom driver applications, this guide has information for you.

This guide does not discuss managing the database; for instructions and information about database management procedures you must read your database vendor's documentation.

Organization

This document is a guide for using GENESIS. In addition to detailing the SQL and data source-specific command support, it tells you how to use the existing data source drivers.

This document is divided into the following chapters:

- Chapter 1, Basics— Discusses basic issues in accessing flat-file data sources in a client/server environment.
- Chapter 2, GENESIS SQL Support --- Describes the SQL commands and provides examples of their use.
- Chapter 3, GENESIS Built-In Functions— Describes the built-in functions that add power and flexibility to GENESIS access.
- Chapter 4, GENESIS Dictionary - Describes the GENESIS Dictionary structure.
- Chapters 5 and on - Describe the various drivers.

Background

Trifox Inc. has been serving the relational database market since 1984 through consulting and the development of software products. In 1987, Trifox created SQL*QMX for Oracle. This easy-to-use, powerful querying and report writing tool, which is based on IBM's QMF, continues to be used at thousands of sites. In 1989, Trifox created TRIMtools, a family of application and reportwriting tools now known as DesignVision. DesignVision was developed in response to the OLTP requirements of several large application vendors.

Database Access

VORTEX is an integrated family of products that allows nearly any production application to access SQL data:

- On any or all of the major relational databases.

- Across networks.
- Across platforms.
- With a dramatic increase in the number of concurrent users.
- Without any additional hardware.

In a client/server or multi-tier configuration, VORTEX makes it possible for your SQL applications to access data on different platforms over one or more network configurations. Currently it supports only TCP/IP.

Inherent in this approach are services that allow production applications originally written for one relational database (such as Oracle) can access the same data on another database (such as Informix), even if it is spread across different databases.

VORTEX Precompilers for C and COBOL, as well as a variety of program interfaces, allow existing SQL programs to take full advantage of VORTEX services such as performance enhancement, transaction monitoring, and flat-file database access.

With VORTEXaccelerator in your configuration, you dramatically increase the number of concurrent users who can log on to a specific SQL production application. Your users experience faster performance and you won't have to change any programs or add any hardware.

Application and Report Development

DesignVision DVapp lets you design, generate, and maintain forms-based applications. You can easily port the pop-up windows, customizable menus and submenus, and custom keyboard assignments, in fact the entire application, to Windows .NET, Unix, OpenVMS, or HTML5 with no extra effort.

The reportwriter, TRIMreport, lets you create simple reports quickly, or complex reports with absolute confidence in their power.

When you want to write stand-alone applications (including triggers) without a user interface, the TRIMpl 4GL language gives you the freedom you want. The procedural language has over 100 database-specific functions that help you write powerful applications in very little time.

Reaching Legacy Data

GENESISsql is a SQL processor that accesses low-level data sources such as ISAM, SDMS, ADABAS, RMS, and MicroFocus and makes the data accessible to VORTEX clients. You can add GENESIS data sources to a VORTEX system in a matter of days, simplifying what used to be an enormous task.

Conventions

Screen shots in this manual come from the Windows version of our software.

Trifox documentation uses the following conventions for communicating information:

Example	Describes
`CHOOSE REPORT > [F3] >`	Press [F3] on the CHOOSE REPORT menu and ...

Example	Describes
Right-click	Clicking the right mouse button.
Left-click	Clicking the left mouse button.
connect_string	Replace italicized text with your own variable.
vtxnetd	Text in bold typewriter style represents strings that you type exactly as they appear in the manual.

Support

If you have a question about a TRIFOX product that is not answered in the documentation (paper or online), contact the Customer Support Services group at:

- support@trifox.com
- Trifox Customer Support Services
  2959 Winchester Boulevard
  Campbell, CA 95008
  U.S.A.
- 408-796-1590

Revisions

August 1999

Corrected errors in OpenVMS RMS Driver and Synergex SDMS Driver sections.

November 1999

Added listing for SDMS Data Dictionary Utility (wdd0) errors.

Added documentation for SET OPTION *OPTIONTYPE* in the SQL chapter.

Added information for RMS about the DDU and multivalue fields.

Separated single ***"Driver"*** chapter into individual chapters for each datasource.

December 1999

Corrected typographical errors.

Corrected typographical errors.

March 2000

Changed text in ADABAS C Driver chapter from "wdd6" to "gds6" to reflect filename changes.

Added detail about unspecified shortnames in using the ADABAS C FDT utility.

May 2000

Added note for ADABAS C Driver on MVS for AFILE parameter

July 2001

Updated ADABAS C chapter with more exmples and explanations concerning PE and MU fields.

May 2003

Added new RMS field types.

June 2006

Added DesignVision List Driver

Added AcuCorp Vision Driver

September 2007

Added GREATEST, LEAST function definitions

May 2008

Added SKIP, TOP keywords

Added CASE, CAST functions

October 2009

Added BITAND, BITOR, BITXOR, REPLACE, REVERSE functions

Added AM,PM masks to TO_CHAR and TO_DATE functions

Added SQL_BIT datatype

July 2010

Added new Insert and update syntax

February 2011

Added optimizer section to Genesis SQL Support chapter

Added ExtFH chapter

May 2012

Added GENESIS_INITSQL

Removed OJNESTON option

Oct 2012

Updated ADABASC catalog definitions

Nov 2013

Updated SET OPTION MERGESIZE definition

Jul 2016

Added SET OPTION OPTRETRY and TIMEOUT

Added SELECT optimizer index hints

Mar 2018

Updated GRANT (Object privileges)

Updated GRANT (Database privileges)

Modified SET PASSWORD command

Update GENESIS catalog table definitions

Aug 2018

Clarify password case sensitivity and special character usage

Oct 2018

Clarify REVOKE command usage

Jun 2021

Added CURDATETIME, CURRENT_DATE, CURRENT_DATETIME, CURRENT_TIME, CURRENT_TIMESTAMP, CURTIMESTAMP built-in functions

Aug 2022

Update Date, Time, Timestamp function descriptions.

Chapter 1 The Basics

GENESIS is a SQL processor that accesses low-level data sources such as ISAM, ADABAS C, and other flat-file structures. It enables joins across heterogeneous databases and data sources through its partnership with VORTEX.

You can use a number of languages including ODBC, Java, C, C++, Cobol, Perl and our TRIMpl 4GL to create your client applications.

GENESIS currently supports:

- Software AG ADABAS C
- Synergex SDMS ISAM
- Microfocus ExtFH
- OpenVMS RMS
- AcuCorp Vision
- DesignVision Lists

Architecture

GENESISsql has two components: the SQL engine and the data-source specific drivers. The GENESIS engine accepts commands from VORTEX and breaks them down into multiple simple read and write commands to the target database driver.

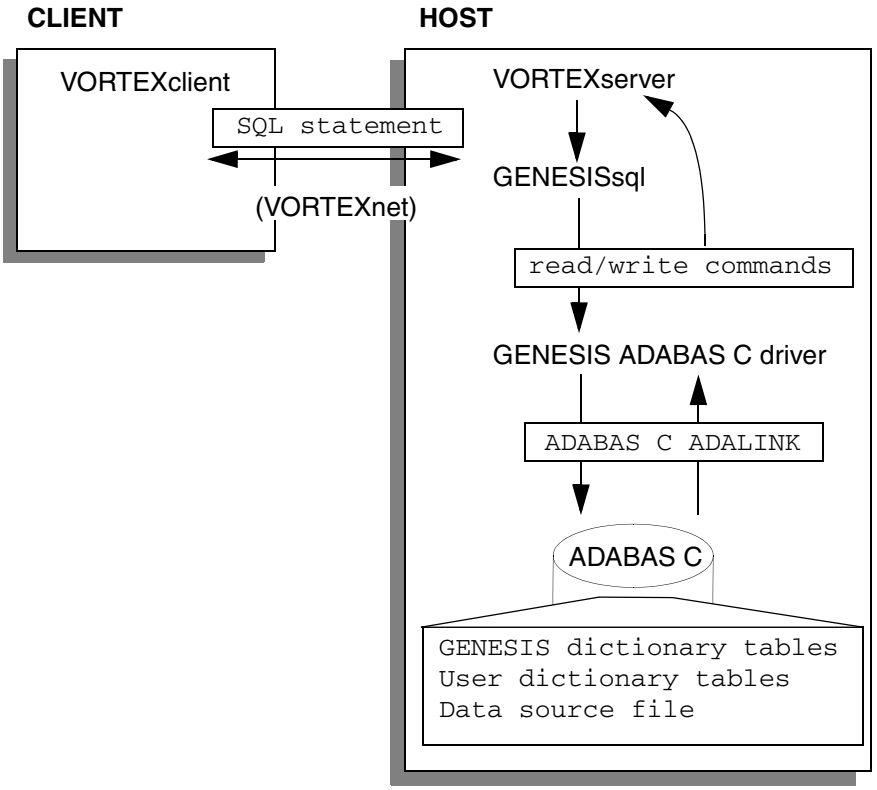
GENESIS maintains a catalog of objects in the target database. There are nine GENESIS catalog tables (discussed in more detail in Chapter 4).

GENESIS_TABLES	Provides the mapping between SQL tables and views and the target database's files.
GENESIS_COLUMNS	Provides the mapping between table columns and the target database's record fields.
GENESIS_INDEXES	Provides the mapping between SQL indexes and the target database's keys.
GENESIS_XCOLUMNS	Provides the mapping between SQL indexes and the target database's keys' segments.
GENESIS_AUTHS	Provides object authentication.
GENESIS_DEPENDS	Keeps track of view dependencies.
GENESIS_FORKEYS	Provides foreign key validation.

GENESIS_USERS	Provides a userid and password security method.
GENESIS_VIEWS	Provides the SQL view definitions.

The first four tables are required and contain header GENESIS information as well as database driver specifics. The header information is stored in the fields with "_" in the column name. The rest of columns contain database-specific information. The other tables are optional and are not applicable to all GENESIS drivers.

Process



All drivers make direct calls to the data source. Neither GENESIS nor VORTEX cache any of the data. You program with SQL statements. GENESIS takes the SQL statements and breaks them down into basic read, write, and update commands, which are translated to data source-specific direct calls by the driver. This example shows the flow to an ADABAS C database on a remote node.

For more information on specific drivers, consult specific driver chapters.

Datasource File

Because most of the work is performed by a generic SQL processor, adding new data sources is a simple task. Data sources are described by a data source file. The file is located by prepending the value of the `GENESIS_HOME` environment variable to the name of the file. For example, on a UNIX system, `GENESIS_HOME` might be set to `/usr2/genesis`. GENESIS would then look in `/usr2/genesis` for the data source file identified in the connect string.

Catalog utility

Each driver has its own GENESIS catalog utility. For example, the ADABAS C driver's is called **gds6init**. It takes two parameters, a data source file and one of:

- -i to initialize the GENESIS dictionary
- -a to add new dictionary entries
- -d to delete dictionary entries

Of course, each operating system has its own method for running the GENESIS catalog utilities. For example, on MVS, the JOBS library contains two jcl scripts: **GDS6INIT** and **GDS6ADD**.

For more information on specific GENESIS catalog utilities, review the specific driver chapters.

Logging

Since GENESIS uses VORTEX you have all the VORTEX logging options available to you on both the client and the server.

On the server side, you can define two environment variables.

- VORTEX_HOST_LOGFILE
- VORTEX_HOST_LOGOPTS

On the client side, you can define two environment variables.

- VORTEX_API_LOGFILE
- VORTEX_API_LOGOPTS

Connecting

The following sample connect strings and URLs illustrate client applets and programs connecting to Genesis ADABAS C on Windows.

```
nthost                  hostname of Windows NT machine
1958                    TCP/IP number that the VORTEXserver
                        is listening on
"C:\\VTX\\BIN\\VTX4"    database driver for GENESIS
"system/manager"        Genesis userid and password
"adabasc:adabas"        Genesis datasource to use
```

PERL example

```
$db->dbConnect("nthost",1958,"C:\\VTX\\BIN\\VTX4",
               "system/manager/adabasc:adabas",
               "VORTEX_HOME=c:\\VTX");
```

JAVA example

```
db.connect("nthost",1958,"C:\\VTX\\BIN\\VTX4",
           "system/manager/adabasc:adabas",
           "VORTEX_HOME=c:\\VTX");
```

JDBC example

```
url="jdbc:vortex//system/manager/adabasc:adabas@1958:nthost!" +
    "C:\\VTX\\BIN\\VTX4,VORTEX_HOME=c:\\VTX";
```

Some VORTEX client methods, such as VORTEXodbc and VORTEXc, use the `net.ini` file located in the `lib` directory under the `VORTEX_HOME` environment variable to fill in any network parameters that are not specified in the connection string. Please refer to the *VORTEX Installation and Usage Guide* for more information.

Initialization SQL Commands

The SET OPTION (page 40) command is used to modify various GENESIS operating parameters. Instead of adding these commands to every application, you can use the GENESIS_INITSQL environment variable to point to a file containing these commands. Each command can be up to 511 characters. All SQL commands except SELECT are valid.

Chapter 2
Genesis SQL Support

Because each driver supports commands that make sense to its target database, not all drivers support all the SQL commands that GENESIS supports. There are also important differences in some of the implementations of SQL commands, especially DDL statements.

There are no rules about how many words you put on a line or where the lines need to break. The following conventions in this document are strictly for readability:

CAPS Capital letters mean the word is a keyword (command).

MIXed Capitals mixed with lower-case letters indicate that the word is a keyword, but you can type either the full word or only the letters in capitals.

lower Lowercase words are variables; use your own.

{ } Curly braces mean you must choose at least one of the enclosed options.

[ ] Brackets mean you can choose one or more of the enclosed options or none of them.

( ) Parentheses are part of the command. Type them just as they appear.

| A vertical bar separates mutually exclusive options. You can only choose one.

, A comma separates multiple options. You can choose as many options as you like. Just be sure to include the commas in the command between choices.

… The ellipses (three dots) mean you can repeat the marked section or commands as many times as you want to.

Summaries

NOTE: *Commands that are not listed are currently unsupported.*

```
CREATE INDEX (UNIQUE) index_name
      ON table_name
         (column_name (ASC | DESC) (,column_name (ASC | DESC)) ...
)

CREATE SYNONYM synonym_name
   FOR table_name

CREATE TABLE table_name
   (column_name datatype (NOT NULL)
      (, column_name datatype (NOT NULL)) ...)

CREATE VIEW view_name
      ((column_name(, column_name) ...))
   AS subselect
   (WITH CHECK OPTION)

DELETE FROM {table_name | view_name} (correlation_name)
   (WHERE search_condition)

DROP INDEX index_name

DROP SYNONYM synonym_name

DROP TABLE table_name

DROP VIEW view_name

GRANT privilege., ..TO { grantee }
    IDENTIFIED by { password }

INSERT INTO table_name
   [ (column_list)]
   { VALUES (constant_list) }

REVOKE { privilege., ..} FROM  { grantee }

    privilege ::=
       { CONNECT
       | DBA
       | RESOURCE }

SELECT [SKIP n] [TOP n] [ALL | DISTINCT ] select_list
   FROM {table_name | view_name} (corr_name)
      (, {table_name | view_name} (corr_name)) ...
   (WHERE search_condition)
   (GROUP BY column_name (, column_name) ...)
   (HAVING search_condition)
```

```
   (ORDER BY {column_name | select_list_number } (ASC | DESC)) ...
)

SET PASSWORD {old_password} {new_password}
UPDATE {table_name | view_name} (correlation_name)
   SET
   column_name = {expression | NULL}
      (column _name = {expression | NULL} ) ...
   (WHERE search_condition)
```

SQL Identifiers

SQL identifiers such as column, table, index names are limited to 30 characters. Identifiers must start with an alphabetic character and can include numbers as well as the "_" character. Avoid using any special characters such as "-" and "+". Using special characters will require that you put "" around identifiers in your SQL statements.

Transaction Management

The transaction management statements, BEGIN WORK, COMMIT WORK and ROLLBACK WORK are not directly supported in GENESISsql. Rather you use the client APIs transaction management functions to perform these operations. For example, VORTEXjdbc has Connection methods called commit() and rollback() which send the appropriate commands to GENESISsql. Also note that not all datasources can actually perform transaction management. For example, the OpenVMS GENESIS RMS driver will simply ignore these commands since RMS itself does not support transaction management.

Predicates

Predicates are expressions that apply comparison operators (comp_op elements in the following syntax) and/or SQL predicate operators (IN, EXISTS, and so on) to values to produce a truth value of TRUE, FALSE, or UNKNOWN.

Predicate can be either a single expression or a combination of any number of expressions using Boolean operators (AND, OR, and NOT) as well as the special SQL operator IS, and parentheses to define the order of evaluation.

Predicates are most often used in the WHERE and HAVING clauses of SELECT statements and subqueries to determine the rows or aggregate groups to select and in UPDATE and DELETE statements to identify the rows on which changes should be made.

Predicates evaluate to TRUE, FALSE, or UNKNOWN. UNKNOWNs arise when NULLs are compared to any value, including other NULLs, since it is impossible to know the value of a data field with NULL value. You can use Boolean operators and SQL IS on UNKNOWN truth values.

Constraints

The ANSI standard defines a list of constraints that can be placed on a table or index. Constraints following the definition of a column apply to that column; those standing alone as table constraints can reference any one or more columns in the table.

At this time GENESIS supports only the NOT NULL constraint. The complete list of possible constraints are:

NOT NULL Forbids NULLs from being entered in a column. According to ANSI standard, this specification can only be a column constraint.

UNIQUE Mandates that every column value, or combination of column values if a table constraint, be unique.

PRIMARY KEY Same effect as UNIQUE, except that none of the columns in a PRIMARY KEY constraint can contain NULLs. You can only issue this constraint once in a given table.

CHECK Followed by a predicate (in parentheses) that uses column values in some expression whose value can be TRUE, FALSE, or in the presence of NULLs, UNKNOWN. The constraint is only violated when the predicate is FALSE.

You can define constraints so that they are not checked until the end of the current transaction. This approach is very useful when, for example, you want to update a table that references itself as a parent key. This operation usually creates intermediate states where referential integrity must be violated. By default, constraints are not deferrable.

SQL Optimization

Genesis optimizes SQL queries using various heuristics applied to the SQL statement and any indexes that are defined on the source tables. You can see what optimizations have occured by using the following options:

```
set option tree on
set option plan on
```

These will display the query tree and optimization plan to stdout. You can use

```
set option logfile 'filename'
```

to send this output to *filename*.

Genesis runs through all the posible optimization plans, creating a score for each plan, and then chooses the plan with the highest score. For very complicated queries with many tables, indexes, and predicates, this process may take a significant amount of time. You can limit the number of generated plans using

```
set option maxoptloop n
```

Indexes

An index will only be considered if at least the first key column of the index is part of the query's predicate. For example,

```
create table staff(id integer,name varchar(10),dept smallint,
                   job char(6), years smallint,
                   salary decimal(8,2),comm decimal(8,2))
create index staff_ix0 on staff(id,name)
```

```
create index staff_ix1 on staff(name,dept)
```

The following queries will use the defined indexes:

```
staff_ix0:
select * from staff where id = 10
select * from staff where id = 10 and name = 'PERNAL'

staff_ix1:
select * from staff where name = 'SANDERS'
select * from staff where name = 'SANDERS' and dept = 15
```

This query will not use an index:

```
select * from staff where dept = 15
```

Even though dept is part of staff_ix1, it is not the first column and so it cannot be used.

In general, the more consequetive index columns used in a query, the better. For example,

```
create index staff_ix2 on staff(id,name,dept)
select * from staff where id = 10 and name = 'SANDERS'
                  and dept = 15
```

This query will use index staff_ix2 instead of staff_ix0 because all three predicate columns fit in the index whereas only two columns fit in staff_ix0.

Boolean operators

Predicate clauses are connected with either AND or OR operators. AND operators are easier to optimize because the values can be pushed to an index. OR operators are much more complicated. For example,

```
select * from staff where id = 10 or id = 20
select * from staff where id in (10,20)
```

Both of these mean the same thing. The IN keyword is simply shorthand. In this case, the optimizer cannot simply use staff_ix0 because the predicate is looking for multiple values of the same column. The optimizer can however still use staff_ix0 by breaking up the query into two portions and merging the results:

```
select * from staff where id = 10
select * from staff where id = 20
```

Genesis keeps track of the records returned so that the same records are not returned for the case where the same values are in multiple OR clauses. For example, Microsoft Access always generates ten OR clauses when it fetches records based on key values. So even if there are only 5 unique key values, it simply repeats the last one to fill out the ten OR clauses. Genesis keeps these record identifiers in its merge buffer. The default size of the merge buffer is 10000 records. This is modified using

```
set option mergesize n
```

where 0 <= *n* <= 65535. If *n* = 0, then the above optimization will not be performed. You will receive a "MERGESIZE overflow" error if your query returns greater than *n* qualifying records. In this case, you can either change the query or set mergesize = 0.

Joins and Subqueries

Predicate values do not have to constants. For example,

```
create table org(deptnumb smallint,deptname varchar(16),
               manager smallint,division char(10),
               location varchar(16))
create index org_ix0 on org(deptnumb)
select * from staff where id = (select manager from org)
```

will use staff_ix0 for the main query and a table scan for the subquery. Adding

```
create index staff_ix3 on staff(id,dept)
select id,name from staff,org where id = manager and
                                dept = deptnumb
select id,name from staff,org where id = manager and
                                dept = deptnumb and
                                deptnumb > 10
```

the first query will use staff_ix3 with both values returned from the org table scan and the second query will use staff_ix3 as well as org_ix1.

CREATE INDEX

Creates an index on a base table.

Syntax

```
CREATE [UNIQUE] INDEX index_name
  ON table_name
   (column_name [ASC | DESC] [,column_name [ASC | DESC]]...)
```

Keywords & Parameters

UNIQUE Creates a unique index.

index_name The name of the index to create.

table_name The name of the table for the index.

column_name Column name to use in the index key. The index key is built using the columns in the order you specify in this list.

Use

GENESIS lets you create an index at any time as long as the base table you want to index exists; however, not all database management systems have this flexibility. In addition, not all databases let you sort columns using the keywords `ASC` and `DESC`.

Consult your database SQL syntax guide before you try to create an index on your table.

Example

```
CREATE UNIQUE INDEX STAFF_IX1 ON STAFF(ID)

CREATE INDEX STAFF_IX2 ON STAFF(DEPT,NAME)
```

CREATE SYNONYM

Creates a synonym for the base table.

Syntax

```
CREATE SYNONYM synonym_name
  FOR table_name
```

Keywords & Parameters

synonym_name The name of the synonym to create.

table_name The name of the table for the synonym.

Use

GENESIS lets you create a synonym for a base table. The synonym can be used in place of [owner.]tablename in SQL statements.

Example

```
CREATE SYNONYM MYSTAFF FOR STAFF
```

CREATE TABLE

Creates a permanent or temporary base table.

Syntax

```
CREATE TABLE table_name
   (column_name datatype (NOT NULL)
      (, column_name datatype (NOT NULL)) ...)
```

Keywords & Parameters

table_name The name of the table to create.

column_name Column name to create in table. Columns are created in the order you specify in this list.

datatype Datatype for the specified column. You must specify a valid datatype for each named column to successfully create the table.

datatype can be:

- CHAR[*n*] — fixed length character string (default: *n = 1, max = 4000)*
- VARCHAR[*n]* — variable-length character string (default: *n = 1, max = 4000)*
- NCHAR[n] -- fixed length UCS2 character string (default: *n = 1, max = 4000)*
- NVARCHAR[*n]* — variable-length UCS2 character string (default: *n = 1, max = 4000)*
- DATETIME — date and time (to the second)
- DECIMAL(*p*[,*s*]) — decimal of precision *p*, scale *s*
- FLOAT — equivalent to DECIMAL(16,6)
- REAL — equivalent to DECIMAL(8,6)
- DOUBLE — equivalent to DECIMAL(16,6)
- INTEGER — four-byte integer
- SMALLINT — two-byte integer

Use

GENESIS supports only the creation of permanent base tables. Tables contain one or more columns, separated by commas, which must also be defined when you create the table.

To define a column you specify a *column_name*, a datatype for the data in that column, and whether the data can be NULL or not.

The order of the columns in this statement determines their order in the table. A column definition must include:

- The name of the column. The column must be named.
- A datatype that applies to all column values.
- NULL or NOT NULL designation.

Some datatypes accept size arguments indicating, for example, the length of a fixed-length character string, or the scale and precision of a decimal number. The meaning and format of these vary with the datatype, but defaults exist.

Default values

If you specify a NOT NULL constraint for a column, every INSERT or UPDATE command on the column must leave it with a specified value.

Ownership and access control

Tables and other database objects are created and owned by authorization IDs, which means users in most contexts. An object's owner controls the privileges others have on it. In a sense, then, all privilege flows from the right to create objects. Tables are grouped into schemas and can only be created by the owner of the schema in which they reside.

Example

```
CREATE TABLE STAFF(ID INTEGER NOT NULL,
             NAME VARCHAR(10) NOT NULL,DEPT INTEGER NOT NULL,
             JOB VARCHAR(6) NOT NULL,YEARS INTEGER,
             SALARY DECIMAL(8,2) NOT NULL,COMM DECIMAL(8,2))
```

CREATE VIEW

Defines a view.

Syntax

```
CREATE VIEW table_name (column_list)
     AS ( SELECT statement );
```

Keywords & Parameters

table_name Name of virtual table, or view.

column_name List of columns to display in view.

statement Criteria by which you identify rows that you want to retrieve.

Use

This statement creates a view, also known as a virtual table. A view is an object that is treated as a table, but whose definition contains a query — a valid SELECT statement. Because the query may access more than one base table, a view may combined data from several tables. Views do not contain their own data. Because the rows of a view are, by definition, unordered, you cannot use ORDER BY when creating a view.

You reference a view in SQL statements just like base tables. When you reference the view in a statement, the output of the query becomes the content of the view for the duration of that statement. In cases where views can be updated, the changes are transferred to the underlying data in the base table(s).

The tables or views directly referenced in a query are called the *simply underlying* tables of the query or view. These combined with all the tables they reference, and all the subsequently referenced tables all the way down to and including the base tables that contain the data, are called the *generally underlying* tables. The base tables — the ones that do not reference any other tables, but actually contain the data — are called the *leaf underlying* tables. View definitions cannot be circular. That is, no view can be among its own generally underlying tables.

Views also cannot contain target specifications or a dynamic parameter specifications. The list of columns is used to provide the columns with names that are used only in given view. You can use it if you do not want to retain the names that the columns have in the underlying base table(s). You must use it whenever:

- Any of the two columns would otherwise have identical names.
- Any of the columns contain computed values or any values other than column values directly extracted from the underlying tables, unless an AS clause is used in the query to name them.
- There are any joined columns with distinct names in their respective tables, unless an AS clause is used in the query to name them.

If you do name the columns, you cannot use the same column name twice in the same view. If you name the columns, you must name all of them, so the number of columns in the name list is the same as the SELECT clause of the contained query. You can use

SELECT * in the query to select all columns; this command is converted internally to a list of all columns so that if a column is added to an underlying table (using ALTER TABLE), your view remains valid.

Views can base their queries on other views, as long as the definition is not circular. Views cannot reference declared temporary tables, although global and created local ones are acceptable.

Inserting, updating, and deleting values in views

When you perform any of these operations on a view the changes are transferred to the base table that contains the data. Such operations are only permitted if the changes that must be made to the underlying table are unambiguous. The principle is that an insertion or change to one row in the view must translate to an insertion or change to one row in the leaf underlying table. If this is the case, the view is said to be updatable. The specific conditions outlined in the standard for a view to be updatable are:

- It must be drawn on one and only one simply underlying table. Joins are not allowed.
- It must contain one and only one query.
- If the simply underlying table is itself a view, that view must also be updatable.
- The SELECT clause of the contained query may only specify column references, not value expressions or aggregate functions, and no column can be referenced more than once.
- The contained query can not specify GROUP BY or HAVING.
- The contained query cannot specify DISTINCT.
- Subqueries are permissible, but only if they do not refer to any of the generally underlying tables on which the view is based.

Example

```
CREATE VIEW STAFF_VIEW (Employee_id, Employee_name, Employee_dept)
            AS SELECT ID,NAME,DEPT FROM STAFF
```

DELETE

Deletes rows from a table.

Syntax

```
DELETE FROM table_name (correlation_name)
       [ (WHERE search_condition) |
       { (WHERE CURRENT OF cursor_name) } ]
```

Keywords & Parameters

table_name Name of table or view from which to delete data rows.

correlation_name Also called range variable or alias, provides alternative name for the table whose name it follows; the definition lasts only for the duration of the statement. *Correlation names* are an option of convenience for base tables and views, but required for tables produced by subqueries.

search_condition Criteria by which you identify rows on which you want to act.

Use

This statement can be coded directly or, in dynamic SQL, be a prepared statement, which is a statement whose text is generated at runtime. The DELETE statement removes rows from permanent base tables, views, or cursors. In the last two cases, the deletions are transferred to the base table from which the view or cursor extracts its data.

The WHERE CURRENT OF form is used for deletions from cursors. The row currently in the cursor is removed. This is called a *positioned deletion*. The WHERE predicate form is used for deletions from base tables or views. All rows that satisfy the predicate are removed at once. This is called a *searched deletion*. If the WHERE clause is absent, it is also a searched deletion, but all rows of the table or view are removed. The following restrictions apply to both types:

- You must have DELETE privilege on the table to delete it.
- If the deletion is performed on a view or cursor, that view or cursor must be updatable.
- The current transaction mode cannot be read-only.

Searched deletions

The predicates used in DELETE statements, like those in SELECT and UPDATE, use one or more expressions — for example, location ='Bahrain' — and test whether they are TRUE, FALSE, or if NULLs exist, UNKNOWN for each row based on the values within that row. Each row for which the predicate is TRUE is deleted.

Positioned deletions

Positioned deletions use cursors and therefore only apply to static or dynamic, not to interactive SQL. You can use a positioned deletion if:

A cursor is within the current module or one of its compilation unit emulations that references the table.

- This cursors has been opened within the current transaction.
- This cursor has had at least one row fetched.
- The cursor has not yet been closed.

The last row fetched is deleted.

Prepared DELETE statements

The PREPARE statement lets you generate the text of dynamic SQL statements at runtime. When you use PREPARE to generate a positioned deletion, you can omit the FROM *table_name* clause of the DELETE statement. The table underlying the cursor is assumed.

Example

Downsize department 15:

```
DELETE FROM STAFF WHERE DEPT = 15
```

DROP INDEX

Drop an index from a base table.

Syntax

```
DROP INDEX index_name;
```

Keywords & Parameters

index_name Name of the index to drop.

Use

This statement is used to drop an index.

Example

```
DROP INDEX STAFF_IX1
```

DROP SYNONYM

Creates a synonym for the base table.

Syntax

```
DROP SYNONYM synonym_name
```

Keywords & Parameters

synonym_name The name of the synonym to drop.

Use

This statement is used to drop a previously defined synonym.

Example

```
DROP SYNONYM MYSTAFF
```

DROP TABLE

Destroys a base table.

Syntax

```
DROP TABLE table_name ;
```

Keywords & Parameters

table_name Name of the table to drop.

Use

This statement is used to drop the same kinds of tables that are created with a CREATE TABLE statement: permanent base tables, global temporary tables, and created local temporary tables. Drop views with the DROP VIEW statement. To drop a table you must own the schema in which the table resides.

The definition of the table is destroyed and all users lose their privileges on that table.

Example

```
DROP TABLE STAFF
```

DROP VIEW

Destroys a view.

Syntax

```
DROP VIEW  view_name ;
```

Keywords & Parameters

view_name Name of the view to drop.

Use

This statement drops a view, which must previously have been created with a CREATE VIEW statement. To drop a view you must own the schema within which the view resides.

Example

DROP VIEW STAFF_VIEW

GRANT (Database privileges)

Gives privileges to users.

Syntax

```
GRANT privilege., ..TO { grantee } [IDENTIFIED by { password }]

privilege ::=
     { CONNECT
     | DBA
     | RESOURCE }
```

Keywords & Parameters

privilege Type of privilege to grant.

grantee User name to allow privilege.

password Password for User name.

Use

Only DBAs can use this GRANT. If you are a DBA, it lets you give grantees (the authorization ID that represents a user) the right to perform specified actions on the database.

`CONNECT` privilege lets grantees connect to the database with the correct password.

`RESOURCE` privilege lets grantees create objects in the database.

`DBA` privilege implies both `CONNECT` and `RESOURCE` and lets grantees connect and read or modify any table in the database.

If the grantee does not already exist, then the IDENTIFIED by {password} clause must be included. If the password is not enclosed in double quotes, then it will be uppercased and connections must present it uppercased. Double quoted passwords are used literally and can contain non-indentifier characters except for ":".

Example

```
GRANT CONNECT,RESOURCE TO CLERK IDENTIFIED BY MY42
```

GRANT (Object privileges)

Gives privileges to users.

Syntax

```
GRANT privilege., ..ON object_name
    TO { grantee., ...} | PUBLIC

privilege ::=
     { ALL PRIVILEGES }
     | { SELECT
         | DELETE
         | INSERT
         | UPDATE }

object name ::=
         [ TABLE ] table name
```

Keywords & Parameters

privilege Type of access, action, or privilege to grant.

object_name Name of the object on which to grant privileges.

grantee User name(s) to allow privilege.

Use

This statement gives grantees (the authorization ID that represents a user) the right to perform specified actions on named objects.

USAGE

To grant a privilege, you (the ''grantor'') must have the privilege itself, with the grant *option* and may grant it with this option, which allows the grantee to further grant the privilege.

ALL PRIVILEGES

ALL PRIVILEGES passes on all applicable privileges that you are entitled to grant. PUBLIC denotes all authorization IDs, present and future.

Other Privileges

SELECT, INSERT, UPDATE, and DELETE let grantees execute the statements of the same names on the object.

Privileges Cascade

Privileges can cascade up; that is, privileges granted on some object can imply grants of privileges on other objects. These situations are covered by the following principles:

- If the grantee owns an updatable view, and is being GRANTed privileges on its leaf underlying table (the base table wherein the data finally resides, regardless of any intervening tables or views), these privileges are GRANTed for the view as well. If specified, the grant option also cascades up. There is only one leaf underlying table for an *updatable* view. (See CREATE VIEW.)
- If the grantee owns an updatable view that immediately references the table on which privileges are being GRANTed (in other words, if the reference appears in the FROM clause without an intervening view), these privileges can also cascade up, including the grant option, if applicable.
- If the grantee owns a view, updatable or not, that grantee already has the SELECT privilege on all tables referenced in its definition as well as on the view itself. If the grantee gains the grant option on SELECT on all the referenced tables, he also acquires the grant option on the SELECT privilege on the view.

For each privilege that is granted, an entry is made in the GENESIS_AUTHS table. The entry indicates:

- The grantee that has received the privilege.
- The privilege itself (the action that can be performed).
- The object on which the privilege is granted.
- The grantor that conferred the privilege.

Multiple identical privilege descriptors are combined, so that a privilege granted twice by the same grantor need only be revoked once. Likewise if two privilege descriptors differ only in that one confers grant option and the other does not, they are merged into a single privilege with grant option. If the grantor lacks the ability to grant the privileges attempted, a completion condition is raised — a warning that privileges were not granted.

Example

```
GRANT SELECT ON STAFF TO CLERK
```

INSERT

Inserts rows into a table.

Syntax

```
INSERT INTO table_name
   [ (column_list)]
   { VALUES (values_list) | SELECT statement}
```

Keywords & Parameters

table_name Name of table into which values are inserted.

column_list Identifies the columns of the table into which the values are inserted. All columns not in the list receive a NULL value. If any such column has a NOT NULL constraint, the INSERT fails.

If you omit the list, all columns of the table are the target of the insert. The number and order in which you list the columns must match the number and order of either the values_list or the the output columns of the query.

values_list A simple list of values to insert.

Use

This statement enters one or more rows into the table named in *table_name*.

You must have INSERT privileges on all named columns to issue an INSERT statement. The table may not be a view.

The SELECT statement can return any number of rows.

Example

```
INSERT INTO STAFF VALUES
(10,'Sanders',15,'Clerk',7,12345.67,543.54)

INSERT INTO STAFF SELECT ID,NAME,DEPT,JOB,YEARS,SALARY,COMM
from OLD_STAFF where ID NOT IN (SELECT A.ID from STAFF A)
```

REVOKE (Database privileges)

Removes the privilege to perform an action.

Syntax

```
REVOKE { privilege., ..} FROM  { grantee }

privilege ::=
     { CONNECT
     | DBA
     | RESOURCE }
```

Keywords & Parameters

privilege Type of access, action, or privilege to grant.

grantee User name(s) to allow privilege.

Use

This statement removes privileges from authorization IDs that have previously received them with the GRANT statement. Only a DBA can execute this statement.

Removing `CONNECT` privilege means that the grantee can no longer access the database. It has no effect on the objects owned by that grantee. Removing `RESOURCE` privilege means that the grantee can no longer create new objects. Be very careful removing `DBA` privilege: If there are no more DBAs then the `GRANT` and `REVOKE` statements can no longer be used.

Example

```
REVOKE CONNECT FROM CLERK
```

REVOKE (Object privileges)

Removes the privilege to perform an action.

Syntax

```
REVOKE
    { ALL PRIVILEGES } | { privilege., ..}
    ON object_name
    FROM PUBLIC | { grantee .,.. }
```

Keywords & Parameters

privilege	Type of access, action, or privilege to grant.
object_name	Name of the object on which to grant privileges.
grantee	User name(s) to allow privilege.

Use

This statement removes privileges from authorization IDs that have previously received them with the GRANT statement. Authorization IDs refer to users. The privileges follow the definitions and rules outlined under GRANT. The GRANT option is the ability to grant the privileges received in turn to others.

In any case, the revoker of the privilege is the same authorization ID that granted it, and all dependent privileges may be revoked. A privilege (privilege A) depends directly on another (privilege B) if either of the following sets of conditions is met:

1. Privilege A is grantable (has GRANT option)

   and

2. The grantee of A is PUBLIC or the same as the grantee of B

   and

3. A and B are both privileges for the same action on the same object.

OR

1. The actions of the two privileges are the same

   and

2. The grantee of A owns the object (which must be a table) on which the privileges exist.

   and

3. Either privilege B is on a view referencing a table on which privilege A is the SELECT privilege (if it is a read-only view) or the privilege at hand (if it is an updatable one).

Example

```
REVOKE SELECT ON STAFF FROM CLERK
```

SELECT

Syntax

```
SELECT [SKIP n] [TOP n] [ALL | DISTINCT ] select_list
   FROM [table_name | subquery] (corr_name) (index hints)
      (, [table_name | subquery] (corr_name) (index hints)) ...
   (INNER or OUTER JOINS) ...
   (WHERE search_condition)
   (GROUP BY column_name (, column_name) ...)
   (HAVING search_condition)
   (ORDER BY {column_name | select_list_number } (ASC | DESC)) ...
```

Keywords & Parameters

select_list List of columns on which to act.

table_name Name of table in which the columns reside.

correlation_name Also called range variable or alias, provides alternative name for the table whose name it follows; the definition lasts only for the duration of the statement. *Correlation names* are an option of convenience for base tables and views, but required for tables produced by subqueries.

search_condition Criteria by which you identify rows on which you want to act.

column_name Name of the column to evaluate with the search_condition.

select_list_number Position in the select list of the column to evaluate with the search_condition.

Use

This is the statement used to formulate queries — requests for information from the database. To issue this statement you must have the SELECT privilege on all tables accessed. Queries may be stand-alone or used in the definitions of views and cursors. In addition, you can use them as subqueries, to produce values that are used within other statements including the SELECT statement itself. Sometimes a subquery is evaluated separately for each row processed by the outer query. Values from that outer row are used in the subquery. Queries of this type are called *correlated subqueries*.

The output of a query is itself a table, and the SELECT clause defines the columns of that table (the *output columns*).

Clauses of the SELECT statement are evaluated in the following order:

1. FROM
2. INNER/OUTER JOIN
3. WHERE
4. GROUP BY
5. HAVING
6. SELECT

7. ORDER BY

SELECT list (SELECT statement)

The SELECT list appears as the first clause in a SELECT statement, but it is not the first logical step. The other clauses produce a set of rows, the *source rows*, from which the output is derived.

The SELECT list determines which columns from these rows are output. It may directly output these columns, or it may use them in aggregate functions or value expressions. Value expressions can be NUMERIC, STRING, or DATETIME; they may include aggregate functions and subqueries.

If SKIP n is specified, then the first n rows of the result set are discarded. If TOP n is specified, then only the first n rows are returned. If DISTINCT is specified, the rows are compared and if any duplicate rows are found, only one copy appears in the output. The SELECT clause may contain any of the following:

- ***Aggregate functions*** — Functions that extract single values from groups of column values — for example, SUM or COUNT.
- ***An asterisk (*)*** — All the columns of all tables listed in the FROM clause are output in the order in which they appear in the FROM clause.
- ***qualifier.**** — Where the *qualifier* is the table or correlation name referenced in the FROM clause. All columns of that (possibly derived) table are output, excluding common columns of joined tables.
- ***A value expression*** — Normally is (or includes) a column name from one of the tables identified in the FROM clause. Either the column's value is directly output or it becomes part of some expression, such as `AMOUNT * 3`.
- ***A specified column name*** — If the output columns are directly taken from one and only one column referenced in the FROM clause, it inherits the name of that column by default. You can override this name by using the AS clause. The names of columns not directly taken from input columns are implementation-dependent. You are not required to name any output columns by the SELECT clause, but may be required to by the context of the way the output columns are to be used (for example, in a view). It doesn't make any difference whether you include the word AS — if omitted, it is implied.

If aggregate functions and value expressions are mixed, all the value expressions must be specified in a GROUP by clause.

FROM clause (SELECT statement)

The FROM clause names the source tables for the query. These tables may be:

- Tables or views named and accessed directly.
- Derived on the spot with a subquery.
- Explicit joins.

The FROM clause determines the one or more tables from which the data is taken or derived. These sources can include temporary or permanent base tables, views, or the results of a subqueries and other operations that return tables.

You can use correlation names to qualify ambiguous column references in the rest of the statement. You can choose to join a table to itself, which is treated as a join of two identical tables; in this case, you must use correlation names to distinguish the two ‘‘copies.” They prefix the *column_name* separated by a period. The column name lists here are for renaming columns, just as they are in the SELECT clause.

The names used here, however, are not for the output; they are for references to the columns made in the remainder of the statement, particularly in the WHERE clause. They are optional, but may be required to clarify column references in some cases.

Index hints are used to force which indexes the optimizer considers when optimizing the predicates for a table. GENESIS currently supports two index hint phrases: USE INDEX and IGNORE INDEX. Both phrases can be used and the order is not important. The indexes are specified in a comma separated list within parenthesis following the phrase. The index names can be found either by using the SET OPTION PLAN ON command or by querying the GENESIS_INDEXES and GENESIS_XCOLUMNS tables. For example, table STAFF has two indexes: STAFF01 on column ID and STAFF02 on columns ID and NAME. Using

```
SELECT * FROM STAFF USE INDEX(STAFF01) WHERE ID=42
```

directs the optimizer to consider only the STAFF01 index when optimizing this query. If the statement is

```
SELECT * FROM STAFF USE INDEX(STAFF01) WHERE NAME='SANDERS'
```

then the optimizer will attempt to use only the STAFF01 index but since this index cannot be used, the optimizer will create a table scan which is not a good solution. The USE INDEX phrase eliminates all indexes that are not specified from optimizer consideration, similar to specifying them in the IGNORE INDEX phrase. Index hints can help the optimizer create better plans based on your knowledge of the data in your tables but you must be careful not to inadvertently direct it to use table scans.

Joins

If more than one table is named in the FROM clause, they are all implicitly joined. This means that every possible combination of rows (one from each table) is, in effect, derived. In addition, this concatenation is the table on which the rest of the query operates. The concatenated table is called a *Cartesian product* or *cross join*.

Another method of adding extra join tables is by using the INNER join syntax:

(INNER) JOIN <tablename> ON <Column1> <op> <Column2>

where <column1> is a column in <tablename> and <column2> is a column in another FROM or INNER/OUTER clause table. Multiple conditions can be added similar to the WHERE clause defined below.

Outer Joins

GENESIS supports OUTER joins, including LEFT, RIGHT, and FULL. The outer joins follow the FROM clause and are of the form:

<LEFT | RIGHT | FULL> OUTER JOIN <tablename> ON <column1> = <column2>

where <column1> is a column in <tablename> and <column2> is a column in another FROM or INNER/OUTER clause table. Multiple conditions can be added similar to the WHERE clause defined below.

WHERE clause (SELECT statement)

The WHERE clause defines the criteria that rows must meet in order to be used for deriving the output.

The WHERE clause contains a predicate, which is a set of one or more expressions that can be TRUE, FALSE, or UNKNOWN. Values are compared according to:

NULLS	Compared to any value, including other NULLs produces UNKNOWNs.
Character string types	Collating sequence
Numeric types	Numerical order
Date-time types	Chronological order

These comparisons are expressed using the following operators: =, <, <=, >, =>, and < > (does not equal).

Operators such as * (multiplication) or || (concatenation) maybe applied depending on the datatype. In most situations, row value constructors may be used instead of simple value expressions.

In addition to the standard comparison operators, SQL provides the following special predicate operators. Assume that B and C are all value expressions, which can be column names or direct expressions (possibly using column names or aggregate functions) in the appropriate datatype:

B BETWEEN A AND C	Equal to (A <= B) AND (B < = C). A and C must be specified in ascending order. B BETWEEN C AND A is interpreted as (C < =B) AND (B < =A) which is FALSE if the first expression is TRUE, unless all three values are the same. If any of the values is NULL the predicate is UNKNOWN.
A IN (C, D,., …)	This is true if A equals any value in the list.
A LIKE 'string'	This assumes that A is a character string and searches for the specified substring. Fixed and varying-length wildcards can be used.
A IS NULL	Specifically tests for NULLs. Unlike most other predicates, it can only be TRUE or FALSE, not UNKNOWN.
A *comp op* SOME | ANY *subquery*	SOME and ANY have equivalent meanings. The subquery produces a set of values. If, for any value V so produced, A *comp op* V is TRUE, then the ANY predicate is TRUE.
A *comp op* ALL *subquery*	Similar to ANY, except that all the values produced by the subquery have to make A *comp op* V true.

EXISTS *subquery* Evaluates to TRUE if the *subquery* produces any rows and FALSE otherwise. It is never UNKNOWN. To be meaningful, this phrase must use a correlated subquery.

These predicates are combined using the conventional Boolean operators AND, OR, and NOT. For TRUE and FALSE values, these have the conventional results. The rows selected by the WHERE clause, whether direction extracted from tables or based on Cartesian products, are the ones that go on to be processed by subsequent clauses.

GROUP BY clause (SELECT statement)

The GROUP BY clause groups the output over identical values in the named columns. If you use this clause, every value expression in the output column that includes a table column must be named in it unless it is an argument to aggregate functions. GROUP BY is used to apply aggregate functions to groups of rows defined by having identical values in specified columns.

If you don't use GROUP BY, either all or none of the output columns in the SELECT clause must use aggregate functions. If all of them use aggregate functions, all rows satisfying the WHERE clause (if any) or all rows produced by the FROM clause (if there is no WHERE clause) are treated as a single group for deriving the aggregates.

The GROUP BY clause defines groups of output rows to which aggregate functions (COUNT, MIN, AVG, and so on) can be applied. If you do not use this clause and elect to use aggregate functions, the column names in the SELECT clause must all be contained in aggregate functions and the functions are applied to all rows to satisfy the query.

Otherwise, each column referenced in the SELECT list outside an aggregate function must be a grouping column and be referenced in this clause. All rows output from the query that have all grouping column values equal constitute a group. (For the purposes of GROUP BY all NULLs are considered equal). The aggregate function is applied to each such group.

HAVING clause (SELECT statement)

The HAVING clause defines criteria that the groups of rows defined in the GROUP BY clause must satisfy to be output by the query.

Just as the WHERE clause defines a predicate to filter rows, HAVING is applied after grouping to define a similar predicate to filter the groups based on the aggregate values. It is needed to test for aggregate function values, as these are not derived from single rows of the Cartesian product defined by the FROM clause, but from groups of such rows and therefore cannot be tested in a WHERE clause.

ORDER BY clause (SELECT statement)

ORDER BY forces the output of the one or more queries to emerge in a particular sequence.

The ORDER BY clause sorts the output. The rows are sorted according to the values in the columns listed here; the first column listed gets the highest priority, and the second column determines the order within duplicate values of the first, the third within duplicate values of the second, and so on. You can specify ASC (for ascending, the default) or DESC (descending) independently for each column.

Character sets are sorted according to their collations. You can also use integers rather than names to indicate columns. The integers refer to the placement of the column among those in the output, so that the first column is indicated with a 1, the fifth with a 5, and so on. If any output columns are unnamed, you must use a number.

Possibly Nondeterministic Queries

In some cases the same query can produce different output tables on different implementations because of subtle implementation-dependent behaviors. Such queries are called *possibly nondeterministic queries*. A query is possibly nondeterministic if any of the following is true:

- It specifies DISTINCT and the datatype of at least one column of the source row is character string.
- One column of the source rows is of a character string datatype and is used in either the MIN or the MAX aggregate function.
- A character set column is used as a grouping column or in a UNION.
- A HAVING clause uses a character string column within a MIN or MAX function.
- It uses UNION without specifying ALL.

Possibly nondeterministic queries cannot be used in constraints.

Examples

```
SELECT DEPT,AVG(SALARY) FROM STAFF
GROUP BY DEPT
```

SET OPTION

Syntax

```
SET OPTION option  param1 [param2]
```

Keywords & Parameters

Option	Param1	Param2	Description
COMPSORT	ON | OFF		Set sort page compression (default ON).
DATETIME	[n] 'string'		Allows the user to modify the DATETIME formatting string. The default is DD-MON-RR and the maximum size for the string is 64 bytes. The user can define up to four DATETIME formatting strings. These are identified by the optional ***n*** parameter (default: 0).
ERROR	ON | OFF		Sets internal error dumping.
EXPR	ON | OFF		Sets internal expression dumping.
HASH	ON | OFF		Sets internal hash dumping.
HEAPBLOCKSIZE	bytes		Allows the user to set the heap block size used in allocating memory. Larger sizes require less CPU overhead but may result in excessive memory usage. The range is 0 to 1000000. Setting 0 means that the exact required size will be used.
LOGFILE	'filename'		Sets the name of the debugging logfile.
MAXOPTLOOP	count		Sets the maximum number of loops the optimizer will execute when determining an access plan (default unlimited).

Option	Param1	Param2	Description
MERGESIZE	records		Sets the number of records used to check for duplicates while processing multiple OR operators, an IN or UNION clause. The default is 1000. Increase this value if you get a "MERGESIZE overflow" error. The maximum value is 500,000 however values over 100,000 will likely cause performance degradation. In this case, either modify the query or use a value of 0 which reduces query optimization but will improve performance.
MKEYOP	ON | OFF		Allows the user to control whether or not to use multi-segment key optimization (default ON).
OPTRETRY	count		Directs the optimizer to create count alternate query plans. If TIMEOUT is set and the timeout occurs, then the next alternate query plan is used.
OPTIMIZE	ON | OFF		Allows the user to control whether or not (default ON).
PLAN	ON | OFF		Sets query plan dumping.
PREOPT	ON | OFF		Allows the user to control whether or not to use preoptimization (default ON).
RECORD	ON | OFF		Sets internal tree dumping to a file. The files are dumped in the cwd as VTXn.zno and VTXn.zcu. These files can be used by the zgen utility to debug optimizer problems.

Option	Param1	Param2	Description
SORTPAGES	totalpages	mempages	Allows the user to modify the amount of disk and memory storage used for sort operations. The *totalpages* parameter is the total number of 4096 byte pages to use and *mempages* is the number of these pages kept in memory. ***totalpages*** must be greater than or equal to ***mempages***. The default values are 10000 and 1000. The default values are fine for most users. If you get a message indicating that Virtual Memory has been exceeded, then increase the totalpages value.
SORTPAGESIZE	page size		The sort *page size* must be large enough to hold at least one record. The range is 4096 to 65535, the default size is 8192.
TIMEOUT	seconds		The number of seconds to wait for the first resultset record to return. If no record is returned before the timeout value and OPTRETRY is not zero, then the next alternate plan is executed and the timeout is restarted. If OPTRETRY is not set or all plans have been tried and the timeout occurs, then a "Fetch timed out" error is returned. Once the first record is returned, the timeout is cancelled for the rest of the query.
TIMEOUT_FETCH	seconds		The number of seconds to wait for any resultset record to be returned. If no record is returned before the timeout value, then a "Fetch timed out" error is returned. Unlike TIMEOUT above, TIMEOUT_FETCH is set for every fetch operation.
TMPINDEX	ON | OFF		Allow use of temporary indexes (default OFF, SDMS only).
TRACE	ON | OFF		Sets internal trace dumping.
TREE	ON | OFF		Sets internal tree dumping.
XML	ON | OFF		Use XML when dumping tree.

Use

This statement is used to set a number of options.

NOTE: *Using Set Option closes any currently open cursors.*

Examples

```
SET OPTION SORT 8000 2000
SET OPTION DATETIME DD-MM-YYYY
SET OPTION LOGFILE mylogfile
SET OPTION ERROR ON
```

SET PASSWORD

Syntax

```
SET PASSWORD new_password {old_password | FOR username}
```

Keywords & Parameters

new_password New password.

old_password Current password.

Use

This statement is used to change the currently connected user's password. A DBA can change another user's password by specifying "FOR username". If the new_password is not enclosed in double quotes, then it will be uppercased and connections must present it uppercased. Double quoted passwords are used literally and can contain non-indentifier characters except for ":".

Examples

```
SET PASSWORD newpass FOR otheruser
SET PASSWORD newpass oldpass
SET PASSWORD "N@#eWpas%" oldpass
```

UPDATE

Changes the data in a table.

Syntax

```
UPDATE table_name (correlation_name)
   SET column_name = {expression | NULL}
      (column_name = {expression | NULL} )
      ((column_name,...) = (subquery)) ...
   (WHERE search_condition)
```

Keywords & Parameters

table_name The name of an existing table that you can access. May include the owner's name, if it is not you. For example, `owner.table`.

correlation_name Also called an alias. Used to relabel the name of the reference in other clauses in the statement.

column_name A column within the table. Parentheses are only required if the column list contains more than one column.

expression The operation or function to execute on the specified column_names.

search_condition A valid condition that evaluates to TRUE, FALSE, or UNKNOWN.

subquery A query that returns only one row with as many columns as column_names in the target set.

Use

This statement changes one or more column values in an existing row of a table. The table may be a base table or view. You can set any number of columns to values and follow the whole *column_name = value_expression* clause with a comma if there is another such to follow. As an alternative to an explicit value, you can set the column to NULL.

You can use the *value_expression* to refer to the current values in the table being updated. Any such references refer to the values of all of the columns before any of them were updated. This allows you to do such things as double all column values (if numeric) by specifying

```
column_name = column_name * 2
```

You can also swap values between columns. Value expressions also can use subqueries.

The UPDATE is applied to all rows that fulfill the WHERE clause, which is one of two types. The WHERE predicate form is like the WHERE predicate clause in the SELECT statement; it uses an expression that can be TRUE, FALSE or UNKNOWN for each row of the table to be updated, and the UPDATE is performed wherever it is TRUE.

Be careful of omitting the WHERE clause; if you do, the UPDATE is performed on every row in the table. You can use the WHERE CURRENT OF form in static or dynamic SQL if the cursor direction is updatable (in other words, not through views) and provided the target table is open and positioned on a row. The UPDATE is applied to the row on

which it is positioned. When using WHERE CURRENT OF in dynamic SQL, you can omit the table name from the UPDATE clause; the table in the cursor is implied.

In either case, for the UPDATE to be successful, the following conditions must be met:

- The statement issuer must have the UPDATE privilege on each column of the table being set.
- If the target table is a view, it must be updatable.
- If the current transaction is read-only, the target table must be temporary.
- If the UPDATE is done through a cursor that specifies ORDER BY, it may not set the values of any columns specified in the ORDER BY clause.
- If the target table is a view, the *value_expression* in the SET clause must not, directly, or though views, reference its leaf-underlying table (the base table where the data ultimately resides).
- The *value_expression* may not use aggregate functions except in subqueries.
- Each column of the target table can only be altered once by the same UPDATE statement.
- If the UPDATE is on a cursor that specified FOR UPDATE, each column being set must have been specified or implied by that FOR UPDATE.
- If the UPDATE is made through a view, it may be constrained by a WITH CHECK OPTION clause.

Example

The following statement updates every salary in department 15:

```
UPDATE STAFF SET SALARY = SALARY * 1.10 WHERE DEPT = 15
UPDATE STAFF SET SALARY = SALARY * 1.10,
                (COMM) = (SELECT max(COMM) from STAFF)
                WHERE DEPT = 15
```

Chapter 3
Built-In Functions

GENESIS has a number of built-in functions that help you create your SQL statements. You can use them in the select-list to modify the result table or in the WHERE clause to limit the number of qualifying rows. They are also valid in INSERT/UPDATE/DELETE statements.

NOTE: *The LEFT and RIGHT built-in functions are also SQL keywords. You must enclose them in {fn <LEFT | RIGHT>(parameters)}.*

ABS

```
ABS(expr)
```

Returns the absolute value of the expr.

ASCII

```
ASCII(char)
```

Returns the integer value of `char`.

BITAND

```
BITAND(expr1,expr2)
```

Returns the bitand of expr1 and expr2.

BITOR

```
BITOR(expr1,expr2)
```

Returns the bitor of expr1 and expr2.

BITXOR

```
BITXOR(expr1,expr2)
```

Returns the bitxor of expr1 and expr2.

CASE

NOTE: *The CASE expression is defined in the built-in functions chapter even though it is not strictly a function.*

```
CASE
  WHEN search-condition1 THEN result1
  ...
  WHEN search-conditionn THEN resultn
  ELSE resultx
END
```

All the $result_i$ values must have comparable datatypes. The ELSE $result_x$ is optional and is set to ELSE NULL if not specified. The search-conditions are in the form of

```
  operand operator operand
```

For example,

```
  CASE
    WHEN dept = 'HQ' THEN 'Headquarters'
    WHEN dept = 'FC' THEN 'Factory'
    ELSE 'Somewhere else'
  END
```

A shorthand syntax is also permitted:

```
CASE valuet
  WHEN value1 THEN result1
  ...
  WHEN valuen THEN resultn
  ELSE resultx
END
```

CAST

```
CAST({ expr | NULL } AS { datatype })
```

Returns the expr or NULL as datatype. Valid datatypes are the same as for the CONVERT() function as well as any valid datatypes used in a CREATE TABLE statement. Note that no trunctation occurs if you use the CREATE TABLE datatypes; any precision such as varchar(n) is ignored.

CHAR_LENGTH

```
CHAR_LENGTH(expr)
```

Returns the length of the character expression.

CHR

```
CHR(n)
```

Returns the character value of n.

CONCAT

```
CONCAT(char1,char2)
```

Returns the concatentation of char1 and char2.

CONVERT

```
CONVERT(expr1,datatype)
```

Returns the value of expr1 converted into datatype which is one of the following:

SQL_BIGINT

SQL_BINARY

SQL_BIT

SQL_CHAR

SQL_DATE

SQL_DECIMAL

SQL_DOUBLE

SQL_FLOAT

SQL_INTEGER

SQL_LONGVARBINARY

SQL_LONGVARCHAR

SQL_NUMERIC

SQL_REAL

SQL_SMALLINT

SQL_TIME

SQL_TIMESTAMP

SQL_TINYINT

SQL_VARBINARY

SQL_VARCHAR

CURDATE

```
CURDATE(), CURDATE
```

Returns the current date when the statement started.

CURDATETIME

```
CURDATETIME(), CURDATETIME
```

Returns the current date and time when the statement started.

CURRENT_DATE

```
CURRENT_DATE(), CURRENT_DATE
```

Returns the current date when the statement started.

CURRENT_DATETIME

```
CURRENT_DATETIME(), CURRENT_DATEIME
```

Returns the current date and time when the statement started.

CURRENT_TIME

```
CURRENT_TIME(), CURRENT_TIME
```

Returns the current time when the statement started

CURRENT_TIMESTAMP

```
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP
```

Returns the current timestamp including microseconds when the statement started.

CURTIME

```
CURTIME(), CURTIME
```

Returns the current time when the statement started.

CURTIMESTAMP

```
CURTIMESTAMP(), CURTIMESTAMP
```

Returns the current timestamp including microseconds when the statement started.

DATABASE

```
DATABASE()
```

Returns the name of the database.

DAYNAME

```
DAYNAME(expr)
```

Returns the name of the day of the week for the date specified by expr.

DECODE

```
DECODE(expr,value,result[,value,result]...,default)
```

Compares `expr` with each `value` and returns either the first matching `value`'s `result` or the `default` value if no `values` match.

GREATEST

```
GREATEST(expr1,expr2[,...])
```

Returns the greatest value of the expressions. The returned datatype is based on the datatype of the first expression.

HOUR

```
HOUR(expr)
```

Returns the hour for the datetime specified by expr.

IFNULL

```
IFNULL(expr1,expr2)
```

Returns expr2 if expr1 is NULL, otherwise it returns expr1.

INSTR

```
INSTR(char1,char2[,n[,m]])
```

Returns the position of char2 within char1. If n is specified and positive, then the search begins n chars into char1. If n is negative, then the search begins n chars from the end of char1. If m is specified, then the mth occurrence of char2 in char1 is located. If char2 does not exist within char1, then 0 is returned.

LCASE

```
LCASE(expr)
```

Returns the lowercase representation of expr.

LEAST

```
LEAST(expr1,expr2[,...])
```

Returns the least value of the expressions. The returned datatype is based on the datatype of the first expression.

LEFT

```
LEFT(expr,n)
```

Returns the first n characters of expr.

LENGTH

```
LENGTH(expr)
```

Returns the length of the character representation of expr.

LOCATE

```
LOCATE(char1,char2[,n[,m]])
```

Returns the position of char1 within char2. If n is specified and positive, then the search begins n chars into char2. If n is negative, then the search begins n chars from the end of char2. If m is specified, then the mth occurrence of char1 in char2 is located. If char1 does not exist within char2, then 0 is returned.

LTRIM

```
LTRIM(expr)
```

Returns the character representation of expr trimmed of leading blanks.

NOW

```
NOW()
```

Returns the current date and time for every invocation.

```
NOW
```

Returns the current date and time when the statement started.

NVL

```
NVL(expr1,expr2)
```

Returns expr2 if expr1 is NULL, otherwise it returns expr1.

POSITION

POSITION(char1 IN char2)

Returns the position of char1 within char2. If char1 does not exist within char2, then 0 is returned.

REPLACE

```
REPLACE(char,from,to)
```

Returns a string where the instances of from in char are replaced with to.

REVERSE

```
REVERSE(char)
```

Returns a string where the characters in char are in reverse order.

RIGHT

```
RIGHT(expr,n)
```

Returns the last n characters of expr.

ROUND

```
ROUND(n[,m])
```

Returns the rounded value of `n` based on the value of `m`. If `n` is a numeric value, it can be either positive or negative.

A positive value `m` specifies the digits to the right of the decimal point. A negative value `m` specifies the digits to the left of the decimal point. If `m` is 0 or not specified, the value is rounded at the decimal point.

If `n` is a date value, then `m` can be one of the following:

SCC,CC	Century
SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Year

Q	Quarter
MONTH, MON, MM	Month
WW,W	Start of Week
DDD,DD,J (default)	Day
DAY, DY, D	Nearest Sunday
HH, HH12, HH24	Hour
MI	Minute

RTRIM

```
RTRIM(expr)
```

Returns the character representation of expr trimmed of trailing blanks.

SQRT

```
SQRT(n)
```

Returns the square root of `n`.

SUBSTR

```
SUBSTR(char,m[,n])
```

Returns a substring of `char`, beginning at position `m` for `n` characters. If you specify `m=0`, the whole string is returned. If you specify a negative number, the function returns the number of characters specified from the end of the string. If you don't specify `n`, the default is to return all characters starting from `m`.

SUBSTRING

```
SUBSTRING(char,m[,n])
```

See SUBSTR().

SYSDATE

```
SYSDATE()
```

Returns the current system date and time for every invocation.

```
SYSDATE
```

Returns the current system date and time when the statement started.

TO_CHAR

```
TO_CHAR(expr[,fmt])
```

Returns the character representation of `expr` based on the `fmt` string or the default for `expr`'s datatype. If `expr` is already a character string, then `expr` is not converted.

If `expr` is a numeric value, then `fmt` can be:

%	Percent sign at right of number.
$	Dollar sign at left of number.
B	Display zero as blank.
0	Display leading zeros.
9	A digit position.
other	Delimiting character (not leading)

The default mask is as many ***9***s as required for the number's precision and scale.

If `expr` is a date value, then `fmt` can be:

YYYY	Four digit year.
YY	Two digit year.
RR	Two digit year in another century.
MM	Two digit month of year (01-12)
MON	Three character month (all uppercase).
mon	Three character month (all lowercase)
Mon	Three character month (initial cap)
MONTH	Fully named month (all uppercase)
month	Fully named month (all lowercase)
Month	Fully named month (initial cap)
DDD	Three digit day of year (001-356)
DD	Two digit day of month (01-31)
D	Single digit day of week (1-7)
DY	Three character day (all upper case)
dy	Three character day (all lowercase)
Dy	Three character day (initial cap)
DAY	Fully named day (all uppercase)
day	Fully named day (all lowercase)

Day	Fully named day (initial cap)
HH12	Two digit hour (00-11)
HH,HH24	Two digit hour (00-23)
MI	Two digit minutes (00-59)
SS	Two digit seconds (00-59)
SSSSS	Seconds past midnight (0000-86399)
J	Julian day
Q	Single digit quarter of year (0-4)
W	Single digit week of month (1-4). The week begins on Sunday.
WW	Two digit week of year (01-52)
AM,PM	Interpret hour value as AM or PM, respectively
other	Delimiting character

To add character extensions to the value that represent counting, such as *ST*, *ND*, *RD*, or *TH*, simply add *th* to any uppercase digit mask. The function correctly interprets the extension based on the last digit and the case based on the mask's case.

Put embedding characters that are valid masks inside double quotes (").

The default mask is **DD-MON-YY**.

TO_DATE

```
TO_DATE(expr[,fmt])
```

Returns the datetime representation of `expr` based on the `fmt` string or the default for `expr`'s datatype. If `expr` is already a datetime, then the value is not converted.

The `expr` can be an integer or numeric value. If it is an integer value, it represents the number of days since year 0. If `expr` is a numeric, then the integer portion represents the number of days since year 0 and the fractional portion represents the part of the last day.

If `expr` is a char value, then `fmt` can be::

YYYY	Four digit year.
YY	Two digit year.
RR	Two digit year in another century.
MM	Two digit month of year (01-12)
MON	Three character month (all uppercase).
mon	Three character month (all lowercase)
Mon	Three character month (initial cap)
DDD	Three digit day of year (001-356)

DD	Two digit day of month (01-31)
HH12	Two digit hour (00-11)
HH,HH24	Two digit hour (00-23)
MI	Two digit minutes (00-59)
SS	Two digit seconds (00-59)
SSSSS	Seconds past midnight (0000-86399)
J	Days since 1/1/1
AM,PM	Interpret hour value as AM or PM, respectively
other	Delimiting character

TO_NUMBER

```
TO_NUMBER(expr[,fmt])
```

Returns the numeric representation of `expr` based on the `fmt` string or the default for `expr`'s datatype.

If `expr` is already a numeric, then the value is not converted. If `expr` is a char value, then `fmt` can be:

%	Percent sign at right of number.
$	Dollar sign at left of number.
B	Display zero as blank.
0	Display leading zeros.
9	A digit position.
other	Delimiting character (not leading)

The default mask is as many **9**s as required for the number's precision and scale.

If `expr` is a date value, then the returned numeric represents the number of days since year 0 and the portion of the last day.

TRANSLATE

```
TRANSLATE(char,from,to)
```

Returns `char` with all characters in `from` replaced with the corresponding ones in `to`. If the number of characters in `to` is a multiple of the number of characters in `from`, for each character in `from`, that multiple of characters from `to` replaces that single character. If `to` is empty, all characters found in `from` are deleted.

TRUNC

```
TRUNC(value)
```

Returns the truncated `value`. If `value` is of type datetime then the hours, minutes, and seconds are set to zero. Otherwise it is treated as a number and the fractional part is removed.

UCASE

```
UCASE(expr)
```

Returns the uppercase representation character representation of expr.

USER

```
USER()
```

Returns the username of the current connection.

Chapter 4
GENESIS Dictionary

Introduction

GENESISsql consists of two parts:

- The GENESISsql engine.
- The driver that communicates with the database.

You use GENESISsql to convert SQL statements from your client application into a command structure that flat-file databases (typically called “legacy data sources”) can understand.

The process begins when you issue SQL statements from the client. The GENESISsql engine converts the SQL statements into discrete read and write operations and passes them on to the database-specific driver. This driver, using a dictionary (also called catalog) to map the translation, converts the engine’s read/write operations into commands that the database understands. Then, the process is reversed when the database sends data to the client.

This chapter describes the dictionary, the utility you can use to build and load the dictionary, and a procedure for completing that task for each of the drivers on each supported platform.

GENESIS dictionary

The GENESIS dictionary contains four required tables that describe the structure of the flat file database and five optional tables that control access and views. The initial structure of the first three tables, up to the comments field, is the same for all databases; the fourth is the same for all. Some databases have fields beyond the comments field that are used to further define the file structures. The dictionary itself is stored in the target database.

- GENESIS_TABLES
- GENESIS_COLUMNS
- GENESIS_INDEXES
- GENESIS_XCOLUMNS
- GENESIS_AUTHS (optional)
- GENESIS_DEPENDS (optional)
- GENESIS_FORKEYS (optional)
- GENESIS_USERS (optional)
- GENESIS_VIEWS (optional)

GENESIS_TABLES

This required table contains the definition for the relation. You can read and query the following header fields, but not modify them.

T_DATABASE Database name.

T_OWNER Table owner.

T_NAME Table name.

T_TYPE Table type (S - System, T - User, V - View).

T_COMMENT Comments.

GENESIS_COLUMNS

This required table contains the columns for each entry in GENESIS_TABLES. You can read and query the following header fields, but not modify them.

C_DATABASE Database name

C_OWNER Table owner

C_TABLE Table name

C_NAME Column name

C_POSITION Column position.

C_TYPE Column datatype

C_LENGTH Column length

C_PRECISION Column precision

C_SCALE Column scale

C_NULLS Column NULLS (Y/N)

C_COMMENT Comments.

GENESIS_INDEXES

This required table contains the indexes for each entry in GENESIS_TABLES. You can read and query the following header fields, but not modify them. While you can define as many indexes as you wish for a given table, GENESIS will only consider the first 15 in alphabetical order, using **I_DATABASE** + **I_OWNER** + **I_TABLE** + **I_NAME** as the key.

I_DATABASE Database name

I_OWNER Table owner

I_TABLE Table name

I_NAME Index name

I_TYPE Unique ('U'/'D')

I_COMMENT Comments.

GENESIS_XCOLUMNS

This required table contains the indexes for each entry in GENESIS_TABLES. You can read and query the following header fields, but not modify them.

X_DATABASE Database name

X_OWNER Table owner

X_TABLE Table name

X_INDEX Index name

X_NAME Column name

X_POSITION Key column position

X_DIRECTION Key column direction ('A'/'D')

GENESIS_AUTHS

This optional table describes object privileges. The privileges are DELETE, INSERT, SELECT, and UPDATE. PUBLIC tables bypass GENESIS_AUTHS checking.

A_USER User ID

A_DATABASE Database name

A_OWNER Object owner

A_TABLE Obejct name (Table or View)

A_SELECT Select authority ('Y'/'N')

A_INSERT Insert authority ('Y'/'N')

A_UPDATE Update authority ('Y'/'N')

A_DELETE Delete authority ('Y'/'N')

GENESIS_DEPENDS

This optional table defines the view dependencies. You manage it with the CREATE VIEW and DROP VIEW SQL commands.

D_DATABASE Database name

D_OWNER View owner

D_NAME View name

D_DATABASE2 Database name

D_OWNER2 View owner

D_NAME2 View/table name

GENESIS_FORKEYS

This optional table defines the foreign key dependencies. You manage it with direct SQL or via the catalog management program.

F_F_DATABASE Foreign key table database name

F_F_OWNER Foreign key table owner name

F_F_TABLE Foreign key table name

F_F_COLUMN Foreign key column name

F_P_DATABASE Primary key table database name

F_P_OWNER Primary key table owner name

F_P_TABLE Primary key table name

F_P_COLUMN Primarh key column name

F_SEQ Rule sequence number

F_UPDRULE Update rule

F_DELRULE Delete rule

F_P_NAME Primary key name

F_F_NAME Foreign key name

GENESIS_USERS

This optional table describes user privileges. You use it to control access to the database with the GRANT and REVOKE SQL commands. The password is encoded using a SHA512 encoding scheme.

U_NAME User ID

U_PASSWORD User password

U_DBA DBA authority ('Y'/'N')

U_CONNECT Connect authority ('Y'/'N')

U_RESOURCES Resource authority ('Y'/'N')

GENESIS_VIEWS

This optional table defines the views. You manage it with the CREATE VIEW and DROP VIEW SQL commands.

V_DATABASE Database name

V_OWNER View owner

V_NAME View name

V_SEQ View text sequence number

V_TEXT View text

Drivers by Type

The following chapters describe the steps you take to create a sample table definition for each of the supported data sources:

- ADABAS C
- Synergex
- AcuCobol Vision
- OpenVMS RMS
- DesignVision Lists

These low-level translators take commands from the GENESISsql engine and turn them into database-specific controls so that you have access to your "legacy" data from any SQL application.

The instructions assume that VORTEX is installed and working at your site and GENESISsql resides on your database server machine.

Briefly summarized the steps for creating and loading table definitions are:

1. Create data source file.
2. Create dicitonary file(s).
3. Initialize dictionary and load GENESIS tables.
4. Build user table definitions.
5. Load user table definitions.

Chapter 5
ADABAS C Driver

Introduction

The GENESIS ADABAS C driver, available on Windows, UNIX, OpenVMS and MVS, uses direct ADABAS C calls to provide very high performance with SQL access.

This section explains how you create and load sample table definitions. It assumes that VORTEX, GENESISsql, and the driver have all been installed according to the instructions for your platform.

The sample instructions use the following values. You must replace them with the correct values for your site.

- dbid = 235
- file = 20

Creating the Dictionary File

First, you must create an ADABAS C dictionary file for the GENESIS dictionary tables. This dictionary contains the tables that the driver consults for mapping instructions.

1. Find the `gds6init.fdu` in your distribution (in the `lib` directory under $GENESIS_HOME).
2. Modify the **dbid** and **file** so they have the values appropriate for your site. For example:

```
dbid   = 235
file   = 20
```

Use the ADABAS adarep utility to find an unused file number. For example,

```
adarep db=235 contents
```

displays a list of the currently used file numbers. Once you have modified the `gds6init.fdu` file, you run the ADABAS C `adafdu` utility to create the ADABAS C file.

All the GENESIS dictionary tables are stored in one ADABASC file. Thus, each record (or row) contains columns for every column in every table but only data for the columns for one table. The columns for other tables in that record are null.

NOTE: *Using any other tools, such as Software AG's NATURAL, to manipulate the GENESIS tables is likely to corrupt the entire GENESIS dictionary.*

Unix

C shell

1. Type `setenv FDUFDT $GENESIS_HOME/lib/gds6init.fdt`
2. Type `adafdu < gds6init.fdu`

Bourne shell

Create the dictionary file by

1. Type `FDUFDT=$GENESIS_HOME/lib/gds6init.fdt`
2. Type `export FDUFDT`
3. Type `adafdu < gds6init.fdu`

Windows

Create the dictionary file by

1. Type `set FDUFDT=gds6init.fdt`
2. Type `adafdu < gds6init.fdu`

OpenVMS

Create the dictionary file by

1. Type `set def GENESIS_HOME:[lib]`
2. Type `def FDUFDT gds6init.fdt`
3. Type `def sys$input gds6init.fdu`
4. Type `adafdu`
5. Type `deassign sys$input`

MVS

To create the ADABAS dictionary file on MVS, modify the LODCATLG JCL in the JOBS dataset of your distribution:

```
…
//DDCARD    DD   *
ADARUN PROG=ADALOD,MODE=MULTI,SVC=237,DEVICE=3380,DBID=235
/*                                                ( Correct DBID)
//DDKARTE   DD   *
ADALOD LOAD FILE=20  <== Correct file
ADALOD NAME='GENESIS-CATALOG'
ADALOD MAXISN=2000,DSSIZE=20B,NISIZE=20B,UISIZE=20B
ADALOD ISNREUSE=YES,DSREUSE=YES
ADALOD TEMPSIZE=15,SORTSIZE=15
```

Creating the data source file

The data source file tells the driver which ADABAS C database and dictionary file to use. You create this file with a text editor. Be sure to put it in the `$GENESIS_HOME` directory.

You must include entries for database and dictionary. Five other entries, four related to logging, are optional:

Keyword	Format	Description
database	integer	(Required) ADABAS C database number
dictionary	integer	(Required) ADABAS C dictionary file number
fast_count	yes/no	Use L9 command to return table row count instead of fetching every record. **The first table index is used and if this key contains NULL values, the count will be incorrect.**
logfile	string	Fully-qualified logfile name. For MVS, this must include the high-level qualifier.
logformat	string	File format (MVS only). Please refer to the "IBM C/C++ Runtime Library Reference", fopen() section.
loglevel	integer	Decimal value of one or more of the following hexadecimal values added together: 0x01 - Messages 0x02 - Errors 0x04 - Command timing 0x08 - Data dumping 0x10 - Data conversions 0x20 - ADABAS control block 0x40 - GENESIS catalogs For example, if you want to log Messages and Errors, set loglevel 3
logmulti	yes/no	yes - generate a unique logfile name suffix.
readonly	yes/no	yes - disallow any database modifications.

Initializing the dictionary and loading GENESIS definitions

You are now ready to initialize and load the GENESIS ADABAS C dictionary using the Data Dictionary Utility, `gds6init`. This creates the GENESIS dictionary tables and loads their own definitions into the GENESIS dictionary.

Unix, Windows, and OpenVMS

1. Type `gds6init data_source_file -i`

MVS

1. Edit the GDS6INIT job in the JOBS dataset of your distribution to match your installation, specifically the AFILE card:

```
//GDS6INIT JOB ADA,CLASS=A,MSGCLASS=X
//INITCAT  EXEC GDS6PROC,
// AFILE='DB235 -I',
// ROPTS='ENVAR(_CEE_ENVFILE=DD:EV)'
//GDS6INIT.EV DD DISP=SHR,DSN=TRIFOX.LIB(ENVFILE)  ENVIRONMENT VARS
//GDS6INIT.DDCARD DD DISP=SHR,DSN=TRIFOX.JOBS(ADARUN)  ADARUN CARDS
//
```

NOTE: *The AFILE parameter "DB235" must match the name of the data source file that you created in the previous step. The "-I" means initialization.*

2. Edit WDD6PROC JCL to match your installation, specifically the ADABAS LOADLIB definition:

```
//*
//* INITIALIZE AND MAINTAIN GENESIS CATALOG
//*
//GDS6PROC PROC AFILE=,                  < INPUT ... REQUIRED
//  ROPTS=                               < RUNTIME OPTIONS
//*--------------------------------------------------------------------
//* GSD6INIT STEP
//*--------------------------------------------------------------------
//GDS6INIT EXEC PGM=GDS6INIT,
// PARM='&ROPTS/&AFILE'
//STEPLIB  DD DISP=SHR,DSN=TRIFOX.LOAD
//         DD DISP=SHR,DSN=SAG.ADA622.LOAD         ADABAS LOADLIB
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//DDCARD   DD DUMMY
//EV       DD DUMMY
//
```

3. Submit the GDS6INIT job.

☞ *Building and loading user table definitions*

There are three steps to loading your existing ADABAS file definitions into the GENESIS dictionary. Each of these steps uses a utility program:

1. Create the synonym file (gds6syn)
2. Create the gds6init input file (gds6fdt)
3. Insert the table definitions in to the dictionary (gds6init)

Building the synonym file

The first step is to create a synonym file which provides a descriptive SQL name to the ADABAS two character shortnames. While it is not strictly necessary to do this, it will make your SQL statements much more understandable.

There are two ways to build the synonym file. The first is to simply use a text editor and the other is to use the `gds6syn` utility.

NOTE: *The gds6syn utility is not yet available on MVS however it can be run on a supported operating system and its output transferred back to MVS. Please contact Trifox support if you want to do this.*

The `gds6syn` utility reads a Natural DDM file and extracts the tablename and column definitions from it. The syntax is

```
gds6syn <DDMfile> <synfile> [owner [table]]
```

where *owner* is the SQL owner of the table and *table* is the SQL name of the table. The owner defaults to PUBLIC and the tablename defaults to the name given in the DDM file header. These defaults are usually fine for most applications.

The format of the synonym file is as follows:

```
OWNER
TABLE
shortname,synonym[,keyword]
...
```

The possible keywords are:

Keyword	Description
DATE	A DATE field with a P4 decimal.
FKEY	A foreign key field for defining PE and/or MU subtables
READONLY	Specifies a read-only field
SCALE=*n*	Define the scale of a numeric value.
SKIP	Skip this field when building the `gds6fdt` output file.
TIMESTAMP	A TIMESTAMP field with a P7 decimal

Date and Timestamp

The ADABAS C GENESIS driver needs to know the field type for date and timestamp information so it can correctly map those datatypes between the fields and the client applications. Since ADABAS C users have historically used packed decimal fields to store

date and timestamp information the corresponding keywords use this type as the default.

Fkey

The ADABAS C GENESIS driver needs to know which field(s) to use when looking up a particular MU or PE occurrence. The FKEY keyword marks those fields as being part of the key used to identify the correct record.

Readonly

Readonly fields are those that cannot be updated. Typically these are used for superdescriptors that use partial fields. SQL does not have a standard method for specifying partial fields in predicate clauses. By marking a field as readonly, the `gds6fdt` utility will create a field that cannot be updated but can be used to specify an indexed lookup.

Scale

As there is no scale information stored in the ADABAS fdt, the user must tell GENESIS what scale to use for packed and unpacked numbers.

Skip

This tells the `gds6fdt` utility to skip this field when building the output file. You can accomplish the same thing by simply leaving the field out of the synonym file.

Because the Natural DDM does not have any information concerning these keywords, it is necessary for the user to modify the `gds6syn` output. In our example, we will use the VEHICLES file that comes with ADABAS. The Natural DDM for the VEHICLES file is shown below:

```
0001DB: 000 FILE: 012  - VEHICLES                          DEFAULT SEQUENCE:
0002
0003T L DB Name                             F Leng  S D Remark
0004- - -- --------------------------------  - ----  - - ----------------------
0005  1 AA REG-NUM                           A   15  N D  LOCAL FORMAT
0006*   LOCAL FORMAT
0007*   CAR'S REGISTRATION NUMBER
0008  1 AB CHASSIS-NUM                       B    4  F    NUMERIC
0009*   NUMERIC
0010*   UNIQUE MANIFACTURER NO FOR C
0011  1 AC PERSONNEL-ID                      A    8    D  INTERN.
0012*   INTERN.
0013*   IDENTIFIER OF CAR USER/OWNER
0014G 1 CD CAR-DETAILS                        0          LOCAL/GENERIC
0015       HD=CAR DETAILS
0016*   LOCAL/GENERIC
0017*   DESCRIPTION OF THE CAR
0018  2 AD MAKE                              A   20  N D  LOCAL/GENERIC
0019*   LOCAL/GENERIC
0020*   NAME OF CAR MAKE/MANUFACTURE
0021  2 AE MODEL                             A   20  N    LOCAL/GENERIC
0022*   LOCAL/GENERIC
```

```
0023*   NAME OF CAR MODEL
0024  2 AF COLOR                              A   10  N D  LOCAL/GENERIC
0025*   LOCAL/GENERIC
0026*   CAR MODEL'S COLOR NAME
0027  2 AF COLOUR                             A   10  N D  LOCAL/GENERIC
0028*   LOCAL/GENERIC
0029*   CAR MODEL'S COLOR NAME
0030  1 AG YEAR                               N  4.0  N    INTERN.
0031*   INTERN.
0032*   YEAR OF THE CAR'S MANUFACTUR
0033  1 AH CLASS                              A    1  F D  INTERN.
0034*   INTERN.
0035*   CODE FOR OWNERSHIP (P/C)
0036  1 AI LEASE-PUR                          A    1  F    INTERN.
0037*   INTERN.
0038*   CODE HOW CAR WAS AQUIRED
0039  1 AJ DATE-ACQ                           N  8.0  N    INTERN.
0040*   INTERN.
0041*   DATE WHEN CAR WAS ACQUIRED
0042  1 AL CURR-CODE                          A    3  N    INTERN.
0043*   INTERN.
0044*   CURRENCY OF MAINTENANCE COST
0045M 1 AM MAINT-COST                         P  7.0  N    INTERN.
0046*   INTERN.
0047*   MAINTENANCE COST IN CURR.UNI
0048  1 AO MODEL-YEAR-MAKE                    A   24    S  LOCAL/GENERIC
0049*   LOCAL/GENERIC
0050*   CAR MAKE + MODEL YEAR
```

Running

```
gds6syn vehicles.nsd vehicles.syn
```

creates the following synonym file:

```
PUBLIC
VEHICLES
AA,REG-NUM
AB,CHASSIS-NUM
AC,PERSONNEL-ID
AD,MAKE
AE,MODEL
AF,COLOUR
AG,YEAR
AH,CLASS
AI,LEASE-PUR
AJ,DATE-ACQ
AL,CURR-CODE
AM,MAINT-COST
AO,MODEL-YEAR-MAKE
```

To simplify this example, we will leave out the MU field, MAINT-COST, for the moment. We will also change the "-" to "_" as "-' is an invalid character for a SQL identifier. We will also add the readonly attribute to the MODEL-YEAR-MAKE field. This produces

PUBLIC
VEHICLES
AA,REG_NUM
AB,CHASSIS_NUM
AC,PERSONNEL_ID
AD,MAKE
AE,MODEL
AF,COLOUR
AG,YEAR
AH,CLASS
AI,LEASE_PUR
AJ,DATE_ACQ
AL,CURR_CODE
AM,MAINT_COST,SKIP
AO,MODEL_YEAR_MAKE,READONLY

As you can see, the synonym file could easily have been created by hand however the `gds6syn` program creates a good starting point.

Building the gds6init input file

As with the synonym file, it is possible to build the `gds6init` input file by hand however this is a laborious process and must be perfect otherwise the ADABAS driver will most likely fail. For these reasons, it is better to use the `gds6fdt` utility.

The `gds6fdt` utility reads the FDT information directly from the ADABAS C file. It also uses an optional synonym file to insert descriptive column and index names in place of the two character ADABAS shortnames.

The syntax is

```
gds6fdt <datasource file> <fnum> [-p password] [outfile] [synfile]
```

where *datasource file* is the name of the file you created on page 67, *fnum* is the ADABAS file number, *password* is the ADABAS file password if any, *outfile* is the name of the `gds6fdt` output file (default is in*fnum*) and *synfile* is the name of the synonym file.

Continuing with the VEHICLES file, the following command will create the `gds6init` input file:

```
gds6fdt data_source_file 12 vehicles.in vehicles.syn
```

For MVS systems, use the following procedure:

1. Modify the GDS6FDT JCL in the JOBS dataset of your distribution:

```
//GDS6FDT JOB ADA,CLASS=A,MSGCLASS=X
//ADABAS EXEC GDS6FPRC,
// AFILE='DB235 12',
// ROPTS='ENVAR(_CEE_ENVFILE=DD:EV)'
//GDS6FDT.EV DD DISP=SHR,DSN=TRIFOX.LIB(ENVFILE)   ENVIRONMENT VARS
//GDS6FDT.DDCARD   DD DISP=SHR,DSN=TRIFOX.JOBS(ADARUN)  ADARUN CARDS
//GDS6FDT.GDS6FSYN DD DISP=SHR,DSN=TRIFOX.MISC(VEHCLSYN)
//GDS6FDT.GDS6FOUT DD DISP=SHR,DSN=TRIFOX.MISC(VEHCLIN)
//
```

2. Modify the GDS6FPRC JCL in the JOBS dataset of your distribution making sure that the ADABAS LOADLIB is properly defined:

```
.
.
.
//*
//* GENERATE GENESIS DD CARDS FROM ADABAS FDT
//*
//GDS6FPRC PROC AFILE=,                  < INPUT ... REQUIRED
//  ROPTS=                               < RUNTIME OPTIONS
//*--------------------------------------------------------------------
//* GDS6FDT STEP
//*--------------------------------------------------------------------
//GDS6FDT EXEC PGM=GDS6FDT,
// PARM='&ROPTS/&AFILE'
//STEPLIB  DD DISP=SHR,DSN=TRIFOX.LOAD
//         DD DISP=SHR,DSN=SAG.ADA622.LOAD         ADABAS LOADLIB
//SYSPRINT DD SYSOUT=*
//SYSTERM  DD SYSOUT=*
//GDS6FSYN DD DUMMY
//GDS6FOUT DD DUMMY
//DDCARD   DD DUMMY
//EV       DD DUMMY
//
```

3. Submit the GDS6FDT job. It creates the output file, VEHCLIN.

The output from `gds6fdt` looks like this:

```
; Input file machine generated
; GENESIS ADABAS C FDT Utility
; 1.0.13
TABLES
,PUBLIC,VEHICLES,T,0,12,12,,0,,0
COLUMNS
,PUBLIC,VEHICLES,         REG_NUM, 1,1,15,15,0,Y,0,AA,0,0
,PUBLIC,VEHICLES,     CHASSIS_NUM, 2,99, 4, 4,0,Y,0,AB,0,0
,PUBLIC,VEHICLES,    PERSONNEL_ID, 3,1, 8, 8,0,Y,0,AC,0,0
,PUBLIC,VEHICLES,            MAKE, 4,1,20,20,0,Y,0,AD,0,0
,PUBLIC,VEHICLES,           MODEL, 5,1,20,20,0,Y,0,AE,0,0
,PUBLIC,VEHICLES,          COLOUR, 6,1,10,10,0,Y,0,AF,0,0
,PUBLIC,VEHICLES,            YEAR, 7,5, 4, 4,0,Y,0,AG,0,0
,PUBLIC,VEHICLES,           CLASS, 8,1, 1, 1,0,Y,0,AH,0,0
,PUBLIC,VEHICLES,       LEASE_PUR, 9,1, 1, 1,0,Y,0,AI,0,0
,PUBLIC,VEHICLES,        DATE_ACQ,10,5, 8, 8,0,Y,0,AJ,0,0
,PUBLIC,VEHICLES,       CURR_CODE,11,1, 3, 3,0,Y,0,AL,0,0
,PUBLIC,VEHICLES,TRIFOX_RO_FLD_0,12,99,22,22,0,Y,0,AO,16,0
INDEXES
,PUBLIC,VEHICLES,         TRIFOX1,U,0,1,AA,0,0,0
,PUBLIC,VEHICLES,         TRIFOX3,D,0,1,AC,0,0,0
,PUBLIC,VEHICLES,         TRIFOX4,D,0,1,AD,0,0,0
,PUBLIC,VEHICLES,         TRIFOX6,D,0,1,AF,0,0,0
,PUBLIC,VEHICLES,         TRIFOX8,D,0,1,AH,0,0,0
,PUBLIC,VEHICLES,MODEL_YEAR_MAKE,D,0,1,AO,0,0,0
XCOLUMNS
```

```
,PUBLIC,VEHICLES,         TRIFOX1,         REG_NUM,0,A,0,0
,PUBLIC,VEHICLES,         TRIFOX3,    PERSONNEL_ID,0,A,0,0
,PUBLIC,VEHICLES,         TRIFOX4,            MAKE,0,A,0,0
,PUBLIC,VEHICLES,         TRIFOX6,          COLOUR,0,A,0,0
,PUBLIC,VEHICLES,         TRIFOX8,           CLASS,0,A,0,0
,PUBLIC,VEHICLES,MODEL_YEAR_MAKE,TRIFOX_RO_FLD_0,0,A,0,0
```

NOTE: *The VEHICLES table is defined slightly differently on various ADABAS platforms so your file may differ from this one.*

There are four distinct sections in the `gds6fdt` output file: TABLES, COLUMNS, INDEXES and XCOLUMNS. These correspond to the GENESIS_TABLES, GENESIS_COLUMNS, GENESIS_INDEXES and GENESIS_XCOLUMNS tables in the GENESIS dictionary. The entries are described in the following section. While it is highly unlikely that you will ever modify the output file, the definitions are included for your information.

User table description

The TABLES section, which only has one line in our example, contains entries for GENESIS_TABLES in the format:

T_Database Database name (none)

T_Owner Table owner (PUBLIC)

T_Name Table name (VEHICLES)

T_Type Table type (T)

T_Comment Comment (0)

Columns column count (12)

Filenum ADABAS file number (12)

PEname PE group name for this table (none)

PE/MU count Count of PE and MU entries (0)

Password ADABAS C file password (0)

Internal checksum (automatically calculated) (0)

The COLUMNS section shows a line for each of the columns as specified in the TABLES section. It contains the entries for GENESIS_COLUMNS in the format (the examples in parentheses are for the first column only):

C_Database Database name (none)

C_Owner Table owner (PUBLIC)

C_Table Table name (VEHICLES)

C_Name Column name (REGNUM)

C_Sequence One-based column sequence number (1)

C_Datatype Column datatype (1)

ADABAS C type	GENESIS type
A	1
U	5
B	99
P	4
G	8
F	0

C_Length Column length (15)

C_Precision Precision (15)

C_Scale Scale (0)

ADABAS C type/length (Description)	GENESIS type	Precision/Scale
U / *len* (unpacked)	5	*len* / User defined
P / *len* (packed)	4	(2 * *len* -1) User defined
P / 4 (date)	12	7 / 4
P / 7 (timestamp)	12	13 / 7
G / *len* (float)	8	(2 * *len*) / 2
F / 1 (integer)	0	3 / 0
F / 2 (integer)	0	5 / 0
F / 4 (integer)	0	10 / 0

C_Nulls Nulls allowed ('Y'/'N') (Y)

C_Comment Comment (0)

ADABAS name ADABAS short name (AA).

You can optionally provide a length and data type override separated by "/" (a slash) up to 10 bytes long (for example, ID/2/F). If you choose this method, however, make sure the datatype, length, precision, and scale entries match the remapping. See "Periodic Groups and Multivalue fields" on page 79.

Flag Column flags. If you don't use a flag, then zero is assumed. The flags column has one or more of the following values OR'd as necessary:

Description	Decimal Value
Foreign key	1
Sequence number	2
MU field	4
PE field	8
Read-only field	16
Nulls & Nulls allowed ('Y'/'N')	Y

Internal checksum (automatically calculated) (0)

The INDEXES section contains entries for GENESIS_INDEXES in the format:

I_Database Database name (none)

I_Owner Table owner (PUBLIC)

I_Table Table name (VEHICLES)

I_Index Index name (TRIFOX1)

I_Type 'U' — unique, 'D' — duplicates (U)

I_Comment Comment

Columns Column count (1)

ADABAS name ADABAS short name (AA)

Options Index options(0) (Used by ODBC catalog functions)

Keynum Key number(0) (Used by ODBC catalog functions)

internal checksum (automatically calculated)

The XCOLUMNS section contains entries for GENESIS_XCOLUMNS in the format:

X_Database Database name (none)

X_Owner Table owner (PUBLIC)

X_Table Table name (VEHICLES)

X_Index Index name (TRIFOX1)

X_Name Column name (REGNUM)

X_Position Column index position, zero-based (0)

X_Direction 'A' - ascending, 'D' - descending (A)

Filler	Not used (0)
Internal	checksum (automatically calculated)

☞ *Loading the dictionary*

You are now ready to load the table definition into the GENESIS catalog.

Unix, Windows, and OpenVMS

Type `gds6init data_source_file -avehicles.in`

MVS

Edit the GDS6ADD job in the JOBS dataset of your distribution, specifically the AFILE card. Make sure that it is refering to the data source file that you created earlier (DB235 is just an example):

```
//GDS6ADD JOB ADA,CLASS=A,MSGCLASS=X
//STAFF EXEC GDS6PROC,
// AFILE='DB235 -A''''TRIFOX.MISC(VEHCLIN)''''',
// ROPTS='ENVAR(_CEE_ENVFILE=DD:EV)'
//GSD6INIT.EV DD DISP=SHR,DSN=TRIFOX.LIB(ENVFILE)  ENVIRONMENT VARS
//GDS6INIT.DDCARD DD DISP=SHR,DSN=TRIFOX.JOBS(ADARUN)  ADARUN CARDS
//
```

The procedure is now complete. You can now connect to the GENESIS database and query the VEHICLES table!

☞ *Deleting table definitions*

If you make a mistake or simply want to delete a table definition, you can use either SQL or **gds6init** to do so. If your GENESIS database is up and you can connect, simply use the "DROP TABLE <owner>.<table>" command from whatever client access method you are using to drop the definitions. Using this SQL command does NOT delete the actual data stored in the ADABAS C file; it merely removes the GENESIS catalog definitions related to the table.

If your definition is incorrect or you cannot connect with a client method, use the **gds6init -d** option to drop the table definition. For example, to drop the definitions for the PUBLIC.VEHICLES table, type

```
gds6init data_source_file -d.PUBLIC.VEHICLES
```

You can specify database.owner.tablename for the target. You must however include the "." separator even if you omit the value.

Customization

Some ADABAS C concepts do not translate easily into SQL. If you want to use any of them, including packed decimal Date and Timestamp fields, super/sub descriptor columns, or PE/MU fields, you must either modify the `gds6init` input files as described in this section or specify the correct synonym file for `gds6fdt`.

Date and Timestamp fields

NATURAL programmers have traditionally stored date and timestamp data in packed decimal fields. To correctly convert these to SQL datetime fields, you must modify the GENESIS ADABAS C Data Dictionary Utility input file.

For the following ADABAS C file:

```
1, DT,  4, P     ; DATE
1, TI,  7, P     ; TIME
```

The correct description entries are:

```
,PUBLIC,TIMEDATE,DATEFIELD,1,12,7,4,4,Y,0,DT,0,0
,PUBLIC,TIMEDATE,TIMEFIELD,2,12,7,7,7,Y,0,TI,0,0
```

The datatype (the 5th field) is set to 12 which tells GENESIS that this is a datetime field and the length is 7. For a Date field, the precision is 6 and the scale is 4. For a Timestamp field, the precision is 13 and the scale is 7.

NOTE: *The **gds6fdt** utility automates this definition process for you if you give it the appropriate synonym files, i.e. use the DATE and TIMESTAMP keywords. It is recommended that you use the **gds6fdt** utility to define your DATE and TIMESTAMP fields rather than attempting this manually. The ADABAS C GENESIS driver is very sensitive to the accuracy of the GENESIS catalog. Errors in setting up these tables will most likely result in driver failure.*

Read-only fields

Read-only fields enable you to use descriptor values that are made up of partial pieces of fields. SQL does not readily support such indexes. To use them, create a read-only field for the super/sub-descriptor and use it for passing in search criteria. Assume that the super/sub-descriptor is `IX` and consists of parts of the alphanumeric partnumber and color fields:

```
IX=PN(5,10),CO(2,4)
```

Create a GENESIS_COLUMNS entry as follows:

```
.
.
.
,PUBLIC,INVENTORY,PNUMCOLOR,11,1,7,0,0,Y,0,IX,16,0
```

Remember to increment the GENESIS_TABLES record's column count. Then create GENESIS_INDEXES and GENESIS_XCOLUMNS entries as follows:

```
,PUBLIC,INVENTORY,RDINDEX,D,,1,IX,0
,PUBLIC,INVENTORY,RDINDEX,PNUMCOLOR,1,A,0,0
```

Now you can use a SQL statement such as

```
SELECT PARTNUMBER,COLOR,SIZE,QUANTITY
   from INVENTORY where PNUMCOLOR = 'BD67RED'
```

and the GENESIS optimizer will use the IX descriptor to find the correct record(s). If the various pieces of the super/sub-descriptor are not all alphanumeric, then you must set the column datatype to 99 (binary) instead of 1.

NOTE: *The **gds6fdt** utility automates this definition process for you if you give it the appropriate synonym files, i.e. use the READONLY keyword. It is recommended that you use the **gds6fdt** utility to define your READONLY fields rather than attempting this manually. Please refer to the synonym file example on page 71. The ADABAS C GENESIS driver is very sensitive to the accuracy of the GENESIS catalog. Errors in setting up these tables will most likely result in driver failure.*

Periodic Groups and Multivalue fields

SQL does not support the concept of an array field. GENESISsql treats Periodic Groups (PE) and Multivalue (MU) fields as separate tables connected to the "main" table with a primary key. A record with MU fields would use two tables, the main one and the MU sub-table, as would a record with a PE group of fields. A record with a PE group that includes MU fields would use three tables - one main table, the PE group sub-table, and the MU field's sub-table, all connected via the same primary key.

Consider the ADABAS VEHICLES demo file:

```
  1, AA, 15, A, DE,UQ,NU
  1, AB,  4, B, FI
  1, AC,  8, A, DE
  1, AD, 20, A, DE,NU
  1, AE, 20, A, NU
  1, AF, 10, A, DE,NU
  1, AG,  4, U, NU
  1, AH,  1, A, DE,FI
  1, AI,  1, A, FI
  1, AJ,  8, U, NU
  1, AL,  3, A, NU
  1, AM,  4, P, NU,MU
AO=AG(1,2),AD(1,20)
```

NOTE: *Your FDT structure may be slightly different from this depending upon your platform.*

One way for SQL to represent the same data is with the following two tables:

VEHICLES	REG_NUM, CHASSIS_NUM, PERSONNEL_ID, MAKE, MODEL, COLOUR, YEAR, CLASS, LEASE_PUR, DATE_ACQ, CURR_CODE

VEHICLES_MAINTENANCE REG_NUM, MAINT_COST

Each record of the ADABAS VEHICLES demo file has:

```
   Column name         Number of occurences
============== ====================================
 REG_NUM       1 -- Primary key
 CHASSIS_NUM    1
 PERSONNEL_ID   1
 MAKE           1
 MODEL          1
 COLOUR         1
 YEAR           1
 CLASS          1
 LEASE_PUR      1
 DATE_ACQ       1
 CURR_CODE      1
 MAINT_COST     1 to n -- MU Field
```

The `gds6fdt` synonym files used to define the tables are:

```
PUBLIC
VEHICLES
AA,REG_NUM
AB,CHASSIS_NUM
AC,PERSONNEL_ID
AD,MAKE
AE,MODEL
AF,COLOUR
AG,YEAR
AH,CLASS
AI,LEASE_PUR
AJ,DATE_ACQ
AL,CURR_CODE
AM,MAINT_COST,SKIP
AO,MODEL_YEAR_MAKE,READONLY

PUBLIC
VEHICLES_MAINTENANCE
AA,REG_NUM,FKEY
AM,MAINT_COST
```

NOTE: *In the case of PE sub-tables and MU sub-tables that are part of a parent PE table, the PE group "field" is specified in the synonym file. If you do not do so, `gds6fdt` will exit with an error. Also in each of the sub-tables, the foreign key and PE or MU fields are the only fields in the synonym file. It is important that you do not include non-foreign key or non-PE/MU fields in the sub-table definition. Doing so may cause unreliable results or driver failure.*

The GENESIS ADABAS C Data Dictionary Utility input file for these descriptions is shown below. Note that you can specify several table definitions in one file:

```
TABLES
; Input file machine generated
; GENESIS ADABAS C FDT Utility
; 1.0.13
TABLES
,PUBLIC,VEHICLES,T,0,12,12,,0,,0
,PUBLIC,VEHICLES_MAINTENANCE,T,0,3,12,,1,,0
COLUMNS
,PUBLIC,VEHICLES,        REG_NUM, 1,1,15,15,0,Y,0,AA,0,0
,PUBLIC,VEHICLES,    CHASSIS_NUM, 2,99, 4, 4,0,Y,0,AB,0,0
,PUBLIC,VEHICLES,   PERSONNEL_ID, 3,1, 8, 8,0,Y,0,AC,0,0
,PUBLIC,VEHICLES,           MAKE, 4,1,20,20,0,Y,0,AD,0,0
,PUBLIC,VEHICLES,          MODEL, 5,1,20,20,0,Y,0,AE,0,0
,PUBLIC,VEHICLES,         COLOUR, 6,1,10,10,0,Y,0,AF,0,0
,PUBLIC,VEHICLES,           YEAR, 7,5, 4, 4,0,Y,0,AG,0,0
,PUBLIC,VEHICLES,          CLASS, 8,1, 1, 1,0,Y,0,AH,0,0
,PUBLIC,VEHICLES,      LEASE_PUR, 9,1, 1, 1,0,Y,0,AI,0,0
,PUBLIC,VEHICLES,       DATE_ACQ,10,5, 8, 8,0,Y,0,AJ,0,0
,PUBLIC,VEHICLES,      CURR_CODE,11,1, 3, 3,0,Y,0,AL,0,0
,PUBLIC,VEHICLES,TRIFOX_RO_FLD_0,12,99,22,22,0,Y,0,AO,16,0
,PUBLIC,VEHICLES_MAINTENANCE,   REG_NUM, 1,1,15,15,0,Y,0,AA,1,0
,PUBLIC,VEHICLES_MAINTENANCE,    SEQNO1, 2,0, 4,10,0,N,0,,3,0
,PUBLIC,VEHICLES_MAINTENANCE,MAINT_COST, 3,4, 4, 7,0,Y,0,AM%d,4,0
INDEXES
,PUBLIC,VEHICLES,        TRIFOX1,U,0,1,AA,0,0,0
,PUBLIC,VEHICLES,        TRIFOX3,D,0,1,AC,0,0,0
,PUBLIC,VEHICLES,        TRIFOX4,D,0,1,AD,0,0,0
,PUBLIC,VEHICLES,        TRIFOX6,D,0,1,AF,0,0,0
,PUBLIC,VEHICLES,        TRIFOX8,D,0,1,AH,0,0,0
,PUBLIC,VEHICLES,MODEL_YEAR_MAKE,D,0,1,AO,0,0,0
,PUBLIC,VEHICLES_MAINTENANCE,TRIFOX1,U,0,1,AA,0,0,0
XCOLUMNS
,PUBLIC,VEHICLES,        TRIFOX1,        REG_NUM,0,A,0,0
,PUBLIC,VEHICLES,        TRIFOX3,   PERSONNEL_ID,0,A,0,0
,PUBLIC,VEHICLES,        TRIFOX4,           MAKE,0,A,0,0
,PUBLIC,VEHICLES,        TRIFOX6,         COLOUR,0,A,0,0
,PUBLIC,VEHICLES,        TRIFOX8,          CLASS,0,A,0,0
,PUBLIC,VEHICLES,MODEL_YEAR_MAKE,TRIFOX_RO_FLD_0,0,A,0,0
,PUBLIC,VEHICLES_MAINTENANCE,TRIFOX1,REG_NUM,0,A,0,0
```

NOTE: *For MVS, the unique key field(s) can be descriptors or non-descriptors. For other platforms, the unique key field(s) must be ADABAS C descriptors.*

Individual occurrences of PE and MU fields must also be uniquely identified. The pseudo-columns SEQNO1 and SEQNO2 identify which PE or MU instance is being referenced. For example,

```
SELECT REG_NUM,SEQNO1,MAINT_COST FROM
VEHICLES_MAINTENANCE
WHERE REG_NUM = 'DA-C 3371'
```

returns a set of rows with each maintenance cost entry preceded by its sequence number:

```
DA-C 3371 1 1288
DA-C 3371 2 322
```

```
DA-C 3371 3 899
DA-C 3371 4 33
```

Note the syntax of the ADABAS C shortname in the input file. You must end the shortname for PE fields and MU fields that are not within a PE with %d. End MU fields within a PE with a %d(%d) appended to the shortname. These markers are used to identify the correct MU and PE instance.

The flag field in the GENESIS_COLUMNS record identifies the various important columns in a MU or PE sub-table (please refer to the *User Table Description* section above for the flag values).

The SEQNO*n* pseudo-columns are also considered foreign key fields, so their flag field is `0x01 | 0x02` or 3.

> ***NOTE:*** *The `gds6fdt` utility automates this definition process for you if you give it the appropriate synonym files. It is recommended that you use the `gds6fdt` utility to build your sub-tables rather than attempting this by hand. The ADABAS C GENESIS driver is very sensitive to the accuracy of the GENESIS catalog and this is vital in the case of PE and MU sub-tables. Errors in setting up these tables will most likely result in driver failure.*

Now you have two tables mapped onto the file. Use standard SQL join syntax to join the two tables as necessary. You can use the SEQNOn pseudo-columns just like any other column. Use the SEQNO*x* fields to insert and update the correct PE and MU fields. You should use the main table as the "driver" for querying any subtables. This ensures that you get the data returned in the expected order.

For a more complex example of MU and PE fields, consider the ADABAS EMPLOYEES demo file:

```
; EMPLOYEES PE and MU example
  1, AA,  8, A, DE,UQ    ; PERSONNEL_ID
  1, AB,                  ; FULL_NAME          Group field
  2, AC, 20, A, NU        ;   FIRST_NAME
  2, AE, 20, A, DE        ;   NAME
  2, AD, 20, A, NU        ;   MIDDLE_NAME
  1, AF,  1, A, FI        ; MAR_STAT
  1, AG,  1, A, FI        ; SEX
  1, AH,  6, U, DE        ; BIRTH
  1, A1,                  ; FULL_ADDRESS       Group field
  2, AI, 20, A, MU,NU     ;   ADDRESS_LINE     MU 1 - 6 occurences
  2, AJ, 20, A, DE,NU     ;   CITY
  2, AK, 10, A, NU        ;   ZIP
  2, AL,  3, A, NU        ;   COUNTRY
  1, A2,                  ; TELEPONE           Group field
  2, AN,  6, A, NU        ;   AREA_CODE
  2, AM, 15, A, NU        ;   PHONE
  1, AO,  6, A, DE        ; DEPT
  1, AP, 25, A, DE,NU     ; JOB_TITLE
  1, AQ,         PE       ; INCOME             PE 1 - 4 occurences
```

```
  2, AR,  3, A, NU       ;   CURR_CODE
  2, AS,  5, P, NU       ;   SALARY
  2, AT,  5, P, MU,NU    ;   BONUS            MU 1 - 4 occurences
  1, A3,                 ; LEAVE_DATA         Group field
  2, AU,  2, U,          ;   LEAVE_DUE
  2, AV,  2, U, NU       ;   LEAVE_TAKEN
  1, AW,        PE       ; LEAVE_BOOKED       PE 1 - 4 occurences
  2, AX,  6, U, NU       ;   LEAVE_START
  2, AY,  6, U, NU       ;   LEAVE_END
  1, AZ,  3, A, DE,MU,NU ; LANG               MU 1 - 4 occurences
H1=AU(1,2),AV=(1,2)      ; LEAVE_LEFT         Super descriptor
S1=AO(1,4)               ; DEPARTMENT         Sub descriptor
S2=AO(1,6),AE=(1,20)     ; DEPT_PERSON        Super descriptor
S3=AR(1,3),AS(1,9)       ; CURRENCY_SALARY    Super descriptor
PH=PHON(AE)              ; PHON_NAME          Phonetic descriptor
end
```

You can represent this data in SQL in six tables::

EMPLOYEES	PERSONNEL_ID,FIRST_NAME,NAME, MIDDLE_NAME, MAR_STAT,SEX,BIRTH,CITY,ZIP, COUNTRY,AREA_CODE, PHONE,DEPT, JOB_TITLE, LEAVE_DUE,LEAVE_TAKEN
EMPLOYEES_ADDRESS_LINE	PERSONNEL_ID,SEQNO1,ADDRESS_LINE
EMPLOYEES_INCOME	PERSONNEL_ID,SEQNO1,CURR_CODE,SALARY
EMPLOYEES_INCOME_BONUS	PERSONNEL_ID,SEQNO1,SEQNO2,BONUS
EMPLOYEES_LEAVE_BOOKED	PERSONNEL_ID,SEQNO1,LEAVE_START, LEAVE_END
EMPLOYEES_LANG	PERSONNEL_ID,SEQNO1,LANG

Each record of the ADABAS EMPLOYEES demo file has:

```
   Column name          Number of occurences
============== ====================================
 PERSONNEL_ID   1 -- Primary key
 FIRST_NAME     1
 NAME           1
 MIDDLE_NAME    1
 MAR_STAT       1
 SEX            1
 BIRTH          1
 ADDRESS_LINE   1 to 6 -- MU Field
 CITY           1
 ZIP            1
 COUNTRY        1
 AREA_CODE      1
 PHONE          1
 DEPT           1
```

```
JOB_TITLE      1
INCOME         1 to 4 -- PE Group
  SALARY
  BONUS        1 to 4 -- MU Field within a PE group
LEAVE_DUE      1
LEAVE_TAKEN    1
LEAVE_BOOKED   1 to 4 -- PE Group
  LEAVE_START
  LEAVE_END
LANG           1 to 4 -- MU Field
```

The `gds6fdt` synonym files used to define the tables are:

```
PUBLIC
EMPLOYEES
AA,PERSONNEL_ID
AC,FIRST_NAME
AE,NAME
AD,MIDDLE_NAME
AF,MAR_STAT
AG,SEX
AH,BIRTH
AJ,CITY
AK,ZIP
AL,COUNTRY
AN,AREA_CODE
AM,PHONE
AO,DEPT
AP,JOB_TITLE
AU,LEAVE_DUE
AV,LEAVE_TAKEN

PUBLIC
EMPLOYEES_ADDRESS_LINE
AA,PERSONNEL_ID,FKEY
A1,GROUP
AI,ADDRESS_LINE

PUBLIC
EMPLOYEES_INCOME
AA,PERSONNEL_ID,FKEY
AQ,GROUP
AR,CURR_CODE
AS,SALARY

PUBLIC
EMPLOYEES_INCOME_BONUS
AA,PERSONNEL_ID,FKEY
AT,BONUS

PUBLIC
EMPLOYEES_LEAVE_BOOKED
AA,PERSONNEL_ID,FKEY
```

```
A3,GROUP
AX,LEAVE_START
AY,LEAVE_END

PUBLIC
EMPLOYEES_LANG
AA,PERSONNEL_ID,FKEY
AZ,LANG
```

The GENESIS ADABAS C Data Dictionary Utility input file for these descriptions is shown below:

```
TABLES
,PUBLIC,EMPLOYEES,T,0,16,1,,0,0,0
,PUBLIC,EMPLOYEES_ADDRESS_LINE,T,0,3,1,,1,0,0
,PUBLIC,EMPLOYEES_INCOME,T,0,4,1,AQ,1,0,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,T,0,4,1,AQ,2,0,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,T,0,4,1,AW,1,0,0
,PUBLIC,EMPLOYEES_LANG,T,0,3,1,,1,0,0
COLUMNS
,PUBLIC,EMPLOYEES,   PERSONNEL_ID, 1,1, 8, 8,0,Y,0,AA,0,0
,PUBLIC,EMPLOYEES,     FIRST_NAME, 2,1,20,20,0,Y,0,AC,0,0
,PUBLIC,EMPLOYEES,           NAME, 3,1,20,20,0,Y,0,AE,0,0
,PUBLIC,EMPLOYEES,    MIDDLE_NAME, 4,1,20,20,0,Y,0,AD,0,0
,PUBLIC,EMPLOYEES,       MAR_STAT, 5,1, 1, 1,0,Y,0,AF,0,0
,PUBLIC,EMPLOYEES,            SEX, 6,1, 1, 1,0,Y,0,AG,0,0
,PUBLIC,EMPLOYEES,         BIRTH , 7,5, 6, 6,0,Y,0,AH,0,0
,PUBLIC,EMPLOYEES,           CITY, 8,1,20,20,0,Y,0,AJ,0,0
,PUBLIC,EMPLOYEES,            ZIP, 9,1,10,10,0,Y,0,AK,0,0
,PUBLIC,EMPLOYEES,        COUNTRY,10,1, 3, 3,0,Y,0,AL,0,0
,PUBLIC,EMPLOYEES,      AREA_CODE,11,1, 6, 6,0,Y,0,AN,0,0
,PUBLIC,EMPLOYEES,         PHONE ,12,1,15,15,0,Y,0,AM,0,0
,PUBLIC,EMPLOYEES,           DEPT,13,1, 6, 6,0,Y,0,AO,0,0
,PUBLIC,EMPLOYEES,      JOB_TITLE,14,1,25,25,0,Y,0,AP,0,0
,PUBLIC,EMPLOYEES,      LEAVE_DUE,15,5, 2, 2,0,Y,0,AU,0,0
,PUBLIC,EMPLOYEES,    LEAVE_TAKEN,16,5, 2, 2,0,Y,0,AV,0,0
,PUBLIC,EMPLOYEES_ADDRESS_LINE,PERSONNEL_ID,1,1, 8, 8,0,Y,0,AA,1,0
,PUBLIC,EMPLOYEES_ADDRESS_LINE,      SEQNO1,2,0, 4,10,0,N,0,,  3,0
,PUBLIC,EMPLOYEES_ADDRESS_LINE,ADDRESS_LINE,3,1,20,20,0,Y,0,AI%d,4,0
,PUBLIC,EMPLOYEES_INCOME,PERSONNEL_ID, 1,1, 8, 8,0,Y,0,AA,1,0
,PUBLIC,EMPLOYEES_INCOME,      SEQNO1, 2,0, 4,10,0,N,0,,  3,0
,PUBLIC,EMPLOYEES_INCOME,   CURR_CODE, 3,1, 3, 3,0,Y,0,AR%d,8,0
,PUBLIC,EMPLOYEES_INCOME,      SALARY, 4,4, 5, 9,0,Y,0,AS%d,8,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,PERSONNEL_ID,1,1,8, 8,0,Y,0,AA,1,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,      SEQNO1,2,0,4,10,0,N,0,,  3,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,      SEQNO2,3,0,4,10,0,N,0,,  3,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,       BONUS,4,4,5, 9,0,Y,0,AT%d(%d),4,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,PERSONNEL_ID,1,1, 8, 8,0,Y,0,AA,1,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,      SEQNO1,2,0, 4,10,0,N,0,,  3,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED, LEAVE_START,3,5, 6, 6,0,Y,0,AX%d,8,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,   LEAVE_END,4,5, 6, 6,0,Y,0,AY%d,8,0
,PUBLIC,EMPLOYEES_LANG, PERSONNEL_ID,1,1,8, 8,0,Y,0,AA,1,0
,PUBLIC,EMPLOYEES_LANG,       SEQNO1,2,0,4,10,0,N,0,,  3,0
,PUBLIC,EMPLOYEES_LANG,         LANG,3,1,3, 3,0,Y,0,AZ%d,4,0
INDEXES
,PUBLIC,EMPLOYEES,TRIFOX1,U,0,1,AA,0,0,0
,PUBLIC,EMPLOYEES,TRIFOX3,D,0,1,AE,0,0,0
,PUBLIC,EMPLOYEES,TRIFOX7,D,0,1,AH,0,0,0
```

```
,PUBLIC,EMPLOYEES,TRIFOX8,D,0,1,AJ,0,0,0
,PUBLIC,EMPLOYEES,TRIFOX13,D,0,1,AO,0,0,0
,PUBLIC,EMPLOYEES,TRIFOX14,D,0,1,AP,0,0,0
,PUBLIC,EMPLOYEES,LEAVE_LEFT,D,0,2,H1,0,0,0
,PUBLIC,EMPLOYEES,DEPT_PERSON,D,0,2,S2,0,0,0
,PUBLIC,EMPLOYEES,CURRENCY_SALARY,D,0,2,S3,0,0,0
,PUBLIC,EMPLOYEES_ADDRESS_LINE,TRIFOX1,U,0,1,AA,0,0,0
,PUBLIC,EMPLOYEES_INCOME,TRIFOX1,U,0,1,AA,0,0,0
,PUBLIC,EMPLOYEES_INCOME,DEPT_PERSON,D,0,2,S2,0,0,0
,PUBLIC,EMPLOYEES_INCOME,CURRENCY_SALARY,D,0,2,S3,0,0,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,TRIFOX1,U,0,1,AA,0,0,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,DEPT_PERSON,D,0,2,S2,0,0,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,CURRENCY_SALARY,D,0,2,S3,0,0,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,TRIFOX1,U,0,1,AA,0,0,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,DEPT_PERSON,D,0,2,S2,0,0,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,CURRENCY_SALARY,D,0,2,S3,0,0,0
,PUBLIC,EMPLOYEES_LANG,TRIFOX1,U,0,1,AA,0,0,0
,PUBLIC,EMPLOYEES_LANG,TRIFOX3,D,0,1,AZ,0,0,0
XCOLUMNS
,PUBLIC,EMPLOYEES,TRIFOX1,   PERSONNEL_ID,0,A,0,0
,PUBLIC,EMPLOYEES,TRIFOX3,            NAME,0,A,0,0
,PUBLIC,EMPLOYEES,TRIFOX7,          BIRTH ,0,A,0,0
,PUBLIC,EMPLOYEES,TRIFOX8,            CITY,0,A,0,0
,PUBLIC,EMPLOYEES,TRIFOX13,            DEPT,0,A,0,0
,PUBLIC,EMPLOYEES,TRIFOX14,      JOB_TITLE,0,A,0,0
,PUBLIC,EMPLOYEES,LEAVE_LEFT,       LEAVE_DUE,0,A,0,0
,PUBLIC,EMPLOYEES,LEAVE_LEFT,    LEAVE_TAKEN,1,A,0,0
,PUBLIC,EMPLOYEES,DEPT_PERSON,            DEPT,0,A,0,0
,PUBLIC,EMPLOYEES,DEPT_PERSON,            NAME,1,A,0,0
,PUBLIC,EMPLOYEES,CURRENCY_SALARY,      CURR_CODE,0,A,0,0
,PUBLIC,EMPLOYEES,CURRENCY_SALARY,         SALARY,1,A,0,0
,PUBLIC,EMPLOYEES_ADDRESS_LINE,TRIFOX1,   PERSONNEL_ID,0,A,0,0
,PUBLIC,EMPLOYEES_INCOME,TRIFOX1,   PERSONNEL_ID,0,A,0,0
,PUBLIC,EMPLOYEES_INCOME,DEPT_PERSON,           DEPT,0,A,0,0
,PUBLIC,EMPLOYEES_INCOME,DEPT_PERSON,           NAME,1,A,0,0
,PUBLIC,EMPLOYEES_INCOME,CURRENCY_SALARY,      CURR_CODE,0,A,0,0
,PUBLIC,EMPLOYEES_INCOME,CURRENCY_SALARY,         SALARY,1,A,0,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,TRIFOX1,   PERSONNEL_ID,0,A,0,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,DEPT_PERSON,           DEPT,0,A,0,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,DEPT_PERSON,           NAME,1,A,0,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,CURRENCY_SALARY,      CURR_CODE,0,A,0,0
,PUBLIC,EMPLOYEES_INCOME_BONUS,CURRENCY_SALARY,         SALARY,1,A,0,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,TRIFOX1,   PERSONNEL_ID,0,A,0,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,DEPT_PERSON,           DEPT,0,A,0,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,DEPT_PERSON,           NAME,1,A,0,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,CURRENCY_SALARY,      CURR_CODE,0,A,0,0
,PUBLIC,EMPLOYEES_LEAVE_BOOKED,CURRENCY_SALARY,         SALARY,1,A,0,0
,PUBLIC,EMPLOYEES_LANG,TRIFOX1,   PERSONNEL_ID,0,A,0,0
,PUBLIC,EMPLOYEES_LANG,TRIFOX3,           LANG,0,A,0,0
```

Because a PE is a repeating group of fields within the same record, you must ensure that there is a unique key that identifies the row. In this case, the unique key is PERSONNEL_ID.

Now you have six tables mapped onto the file. To get a list of salaries for all employees in department TECH88, use the following SQL:

```
SELECT DEPT,A.PERSONNEL_ID,SALARY FROM EMPLOYEES
A,EMPLOYEES_INCOME B
WHERE DEPT = 'TECH88'
AND A.PERSONNEL_ID = B.PERSONNEL_ID
```

To get a list of bonuses for all employees in department TECH88, use the following SQL:

```
SELECT DEPT,A.PERSONNEL_ID,BONUS FROM EMPLOYEES
A,EMPLOYEES_INCOME_BONUS B
WHERE DEPT = 'TECH88'
AND A.PERSONNEL_ID = B.PERSONNEL_ID
```

To get a list of salaries and bonuses for all employees, use the following SQL:

```
SELECT A.PERSONNEL_ID,SALARY,BONUS FROM EMPLOYEES_INCOME A,
EMPLOYEES_INCOME_BONUS B
WHERE A.PERSONNEL_ID = B.PERSONNEL_ID
AND A.SEQNO1 = B.SEQNO1
```

In each of these examples, you are ‘‘joining’’ two tables which are in fact the same record. Likewise you use the SEQNO*x* fields to insert and update the correct PE and MU fields

Chapter 6
Synergex SDMS Driver

Introduction

The GENESIS Synergex SDMS driver, available on Windows, Unix, and OpenVMS, uses direct SDMS calls to provide very high performance with SQL access.

This section explains how you create the data source file and load Synergy/DE Repository definitions. It assumes that VORTEX, GENESISsql, and the driver have all been installed according to the instructions for your platform.

NOTE: *You must obtain the Synergex SDMS driver from Synergex (www.synergex.com). Trifox developed the Synergex SDMS driver exclusively for Synergex.*

Making it Work

☞ *Creating the data source file*

The data source file tells the driver how to find the GENESIS dictionary and database files. You create this file with a text editor. Be sure to put it in the `$GENESIS_HOME` directory.

You must include entries for the database and dictionary. Two other entries, related to logging, are optional:

datasource	string	(Required) Directory path(s) to SDMS database files, delimited by ";".
dictsource	string	(Required) Directory path to GENESIS dictionary SDMS database files.
logfile	string	Fully-qualified logfile name.
loglevel	integer	Logging level (0 - none, 1 - control blocks/data, 2 - conversion output)

In our example, the dictionary directory is "`/usr2/genesis/sdms`" and the SDMS database files are stored in "`/usr2/genesis/sdms0`" and "`/usr2/genesis/sdms1`":

```
datasource      ;/usr2/genesis/sdms0;/usr2/genesis/sdms1;
dictsource      /usr2/genesis/sdms
```

The GENESIS SDMS driver uses the multiple datasource directory entries to locate the database files. It searches the entries in the order they are listed.

Initializing the dictionary and loading GENESIS definitions

You are now ready to initialize and load the GENESIS SDMS dictionary using the Data Dictionary Utility (DDU).

1. Change directory to your dictionary directory
2. Type

```
gds0init -c
```

Loading existing Synergy/DE Repository definitions

The DDU reads the definitions stored in the Repository for information on how to build the user table entries. It finds the repository files either from the command line or from the RPSMFIL and RPSTFIL environment variables.

If these are not defined, then it uses "`RPSDAT:rpsmain.ics`" and "`RPSDAT:rpstext.ics`".

The DDU reads in every file definition and goes through the structures defined for that file. The structure name becomes the table name. If there are multiple structures defined in the file, DDU reads the tag information and it with the table definition. Field definitions are inserted into the GENESIS_COLUMNS table. User defined fields have optional TYPE, CLASS, and DATA definitions which are also stored in the GENESIS_COLUMNS record for that field. Array fields are identifed by the field name appended with the dimension character and the count. The dimension character is read from the GENESIS_WDD_DIMCHAR environment variable and defaults to "#". For example, if address is an array field, the columns will be "ADDRESS", "ADDRESS#1", and "ADDRESS#2".

Indexes are also read from the Repository. If the index is unamed, the default name is "INDEX_*n*" where *n* is the index number.

Chapter 7
OpenVMS RMS Driver

Introduction

The GENESIS OpenVMS RMS driver, available only on AXP OpenVMS, uses direct RMS calls to provide very high performance with SQL access.

This section explains how you create the data source file and load RDF definitions. It assumes that VORTEX, GENESISsql, and the driver have all been installed according to the instructions for your platform.

Making it Work

☞ *Creating the data source file*

The data source file tells the driver how to find the GENESIS dictionary and database files. You create this file with a text editor. Be sure to put it in the `GENESIS_HOME` directory.

You must include entries for the database and dictionary. The other entries are optional:

Variable	Data Type	Description
datasource	string	(Required) Directory path(s) to RMS database files, delimited by ";".
dictsource	string	(Required) Directory path to GENESIS dictionary RMS database files.
bit_char	Yes/No	If Yes, then all bit fields will be described as CHAR(1) in GENESIS_COLUMNS queries.
bit_decode	string	A two character string, e.g. NY, which is applied to bit fields on fetch and insert/update.
logfile	string	Fully-qualified logfile name.
loglevel	integer	Logging level (0 - none, 1 - control blocks/data, 2 - conversion output)

In the example, the dictionary directory is `dkb500:[usr.rms]` and the new RMS database files are stored in `dkb500:[usr.rms0]`:

```
datasource  ;dkb500:[usr.rms0]
dictsource  dkb500:[usr.rms]
…
```

The GENESIS RMS driver uses the datasource directory entry to locate the database files.

Initializing the dictionary and loading GENESIS definitions

You are now ready to initialize and load the GENESIS RMS dictionary using the Data Dictionary Utility (DDU). Use the data source file described to create the GENESIS data dictionary files in the `dkb500:[usr.rms]` directory with the following command:

gds8init *data_source_file* **-i**|**-a***rdffile* [*options*]

where

Variable	Description
data_source_file	Is a file located in `$GENESIS_HOME`.
-i	Initializes the GENESIS directory.
-a*rdffile*	Adds a new dictionary entry described in *rdffile.*

The following options are available:

Option	Description
-o*name*	Uses *name* as the object owner review.
-t*name*	Use *name* as the object table name.

Loading existing RMS table definitions

You can define existing RMS files in the GENESIS data dictionary. For example, to load a file (described later in this section) called `dkb500:[usr.rms]indexed-abc.rdf`, type

`gds8init` *data_source_file* `-adkb500:[usr.rms]indexed-abc.rdf`

The format for RDF files is similar to that of a FDL file. For example, a three column, keyed, variable length record file with two indexes is shown here:

```
IDENT    5-MAY-1999 17:34:08.14
AXP/OpenVMS RDF Routine

SYSTEM
         SOURCE                                AXP/OpenVMS

FILE
         NAME                                  ABC
         ORGANIZATION                          indexed

RECORD
         FORMAT                                variable
         SIZE                                  20

FIELD 0
         NAME                                  lastname
```

```
        TYPE                            string
        LENGTH                          10
        POSITION                        0
        KEY FIELD                       yes
        KEY ID                          0
        KEY CHANGES                     no
        KEY DUPLICATES                  no

FIELD 1
        NAME                            age
        TYPE                            int2
        LENGTH                          2
        POSITION                        10
        KEY FIELD                       no

FIELD 2
        NAME                            salary
        TYPE                            decimal
        LENGTH                          8
        POSITION                        12
        KEY FIELD                       yes
        KEY ID                          1
        KEY CHANGES                     yes
        KEY DUPLICATES                  yes
        KEY ALT NUL                     yes
        KEY ALT NUL VAL                 0
```

You define a RDF file with any editor using the keywords defined below.

The table name is ABC and comes from the FILE->NAME keyword. If you specify the *-o* option, it defines the table owner. Otherwise, the table owner is set to PUBLIC.

FILE keywords	Definition
FILE	Name of the RMS file. This can be either: a. Logical:filename.dat b. device:[dir]filename.dat c. filename d. filename.[ext] In the first two cases, the exact filename is used to open the RMS file. In the thirdcase, the file has .DAT appended. In the last case, if there is just a dot, then it is discarded; otherwise the .ext is used as is. In these two cases, the file is searched for in the dictsource and then the datasource directories.

ORGANIZATION	One of: INDEXED SEQUENTIAL RELATIVE

RECORD keywords	Definition
FORMAT	One of: FIXED VARIABLE
LENGTH	Length of the RMS records
TABLENAME	Name of the table. If not specified, then FILE is used as the tablename.

FIELD keywords	Definition
NAME	Name of the field
TYPE	Datatype of the field. One of [D]BINn - Unsigned Integer, n = 1, 2, 4, 8 [D]BIT - Bit field [D]COLLATED - Collated key [D]DATE - Date in DATEFORMAT character format. [D]DECIMAL - Packed Decimal [D]DOUBLE - VAX D_Float (8 bytes) [D]DOUBLE2 - VAX D_Float (8 bytes) with implied scale 2 stored as a whole number [D]FLOAT - VAX F_Float (4 bytes) [D]GFLOAT - VAX G_Float format (8 bytes) [D]INTn - Signed integer, n = 1, 2, 4, 8 [D]STRING - Character string [D}STRINGV - Variable length string [D]ZONED - Zoned Decimal [D] indicates a descending key if the field is a key.

DATEFORMAT	Describes the format of the date. One of DDMMYYYY MMDDYYYY YYYYMMDD
LENGTH	Length of the field.
SCALE	Scale for Packed or Zoned decimals.
POSITION	Starting record position (0 based)
BIT n	For BIT fields, n is the bit position (0-7)
KEY FIELD	yes/no
KEY ID	RMS index number
KEY CHANGES	Can the key field(s) be modified? yes/no
KEY DUPLICATES	Is this a non-unique key? yes/no
KEY ALT NUL	Does this key have an alternate NULL value? yes/no
KEY ALT NUL VAL	Character to use to signify null in the index
KEY POSITION	Index segment position. If this is not specified, then it defaults to 0.
NULL	yes/no

NOTE: *All floating point data is assumed to be stored in RMS in VAX format.*

The FIELD keywords must be used in the order in which they are defined in this table. Not all of the keywords are required, e.g. the KEY keywords are not used if this is not a key field. The KEY keywords have been superseded by the INDEX keywords defined below.

INDEX keywords	**Definition**
INDEX n	Begins index definition. n is the RMS key number.
NAME	Name of the index.
FIELD [DESC | ASC]	Name of the field. Fields are defined in their key segment order. The optional DESC and ASC keywords override the field datatype definition.
DUPLICATES	Defines a non-unique index. The default is unique.

There are two methods of defining indexes. The first is to use the FIELD keywords and is appropriate for very simple RMS files, i.e. those that have simple indexes without fields that belong in multiple indexes. The second method is to use the INDEX keywords. These can be used after all the fields have been defined and are used in lieu of the FIELD KEY keywords. It is not a good idea to mix the two methods.

An example an index defined using the INDEX keywords is:

```
INDEX 0
   NAME INDEX_01
   FIELD ID
   FIELD NAME
   FIELD LOCATION
```

This defines a three segment key called INDEX_01 consisting of the ID, NAME, and LOCATION fields. It is a UNIQUE index by default and sorted in ascending order.

Many users may find this method simpler and more efficient to set up than the FIELD KEY keywords. Both are supported for backward compatibility.

Multivalue fields

You can define Multivalue (MU) or array fields in the GENESIS data dictionary.

The MU field(s) must be the last one(s) in the table definition. There are two types of MU fields, fixed and null suppressed. Both of them define a maximum number of entries. Null suppressed MU fields have a one byte count at the field's record offset whereas fixed MU fields always display the maximum number of entries.

Null suppressed MU fields never contain a NULL value. For example, if entries 0, 1, and 2 have values, then the count is 3. If you update entry 1 to NULL, the value in entry 2 is shifted over and the count is decremented to 2.

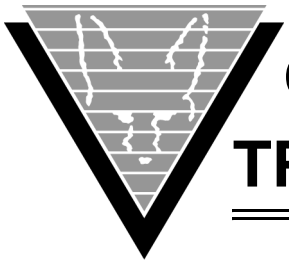
FIELD -> MU keywords	Definition
MU COUNT	If String, then the MU count is stored as a character string; otherwise it is integer.
MU FIELD	Yes/No
MU FORMAT	Fixed/Null-suppressed
MU MAXIMUM	Maximum number of values
MU VALUE SIZE	Size of each value

Tagged records

Tagged records are a method of defining a table with an implied WHERE clause. This is done when different record types are stored in the same file. For example a RMS file may have records for both the ORDER and ORDER_DETAIL table. By specifying that the ORDER records have RECORD_TYPE = 0 and ORDER_DETAIL records have RECORD_TYPE = 1, for example, an SQL query to the ORDER table will not return any ORDER_DETAIL records even though they are in the same file.

Tagged records are defined by using the TAG keywords:

TABLE -> TAG keywords	Definition
TAG n	Identifies the order of this tag definition
FIELD	Name of the tag field
VALUE	Value to compare (maximum 64 bytes). The value can also be specified using hex 0x notation, e.g. 0x010521.
BOOLOP	AND or OR



Chapter 8
TRIMpl List Driver

Introduction

The GENESIS TRIMp list driver implements a very high performance read-only database using the TRIMpl list feature combined with shared memory. You can also use the TRIMpl list driver to create a read-write database using either a file-based catalog with local lists or a completely dynamic in-memory database. Such a database exists only for the life of the connection and so any tables created during the connection are lost when the connection is released.

This section explains how you create the GENESIS catalog and load the list table definitions. It assumes that VORTEX, GENESISsql, and the driver have all been installed according to the instructions for your platform.

Making it Work

☞ *Creating the data source file*

The data source file gives the driver certain behavioral options. You create this file with a text editor. Be sure to put it in the `GENESIS_HOME` directory.

All the entries are optional:

Variable	Data Type	Description
dictsource	string	The directory in which the GENESIS catalog files are stored. If not given, then look for a shared memory catalog.
heap_block_size	integer	The minimum heap block size in bytes. The default is 16384 bytes.
logfile	string	Fully-qualified logfile name.
loglevel	integer	Logging level (0 - none, 1 - control blocks/data, 2 - conversion output)
shmem_seg_size	integer	The shared memory segment size in Kb. The default is 128 Kb
vtx_shm_addr	string	Fully-qualified VORTEX_SHM_ADDR filename. If not given, the driver will look for a VORTEX_SHM_ADDR env var.

Variable	Data Type	Description
vtx_shm_file	string	Fully-qualified VORTEX_SHM_FILE filename. If not given, the driver will look for a VORTEX_SHM_FILE env var.

The logging options are useful to help debug access problems.

If you specify "memory" as the data source filename in the connect string, then the GENESIS list driver creates a completely dynamic in-memory database. The driver will look for a file called "memory" in GENESIS_HOME and if found, it will use the logfile, loglevel, and heap_block_size keywords.

Initializing the dictionary and loading GENESIS definitions

You are now ready to initialize and load the GENESIS List dictionary using the Data Dictionary Utility (DDU). Create the GENESIS catalog in shared memory with the following command:

```
gds9init
```

This creates the GENESIS catalog in shared memory and also creates GENESIS catalog entries for all the lists currently stored in shared memory.

If you want to use a file-based catalog instead of shared memory, type

```
gds9init -f [directory]
```

If [directory] is not specified, the files are built in the current directory. This only builds the GENESIS catalog, no user lists are loaded. Any DDL statements executed in this mode will create memory-based catalog entries and tables which will exist only for the life of the connection.

Chapter 9
AcuCobol Vision Driver

Introduction

The GENESIS AcuCobol Vision driver, available on Windows and Unix, uses direct Vision calls to provide very high performance with SQL access.

This section explains how you create the data source file and load Vision XFD definitions. It assumes that VORTEX, GENESISsql, and the driver have all been installed according to the instructions for your platform.

NOTE: *You must obtain the AcuCobol Vision driver from Micro Focus, Inc (www.microfocus.com). Trifox developed the AcuCobol Vision driver exclusively for AcuCobol.*

Making it Work

Creating the data source file

The data source file tells the driver how to find the GENESIS dictionary and database files. You create this file with a text editor. Be sure to put it in the `$GENESIS_HOME` directory.

You must include entries for the database and dictionary. Two other entries, related to logging, are optional:

debug_logfile	string	Fully-qualified debugging logfile name.
debug_loglevel	integer	Debugging logging level (0 - none, 1 - control blocks/data, 2 - conversion output)
dictsource	string	(Required) Directory path to GENESIS dictionary Vision database files.
file_prefix	string	(Required) Directory path(s) to Vision database files, delimited by ";".
file_suffix	string	Suffix appended to file names stored in GENESIS_TABLES.
logging	yes/no	Turns on Vision transaction logging.
log_encrypt	yes/no	Turns on Vision transaction logfile encryption.

locks_per_file	integer	The maximum number of locks per file.
log_buffer_size	integer	The size of the log buffer.
log_device	yes/no	Turns on device logging.
log_dir	string	The directory where to store the transaction logfiles. The default is the current working directory.
log_file	string	The name of the transaction logfile.
max_files	integer	The maximum number of open Vision files.
max_locks	integer	The maximum number of locks.
read_only	yes/no	Disallows any database modifications.
v_buffers	integer	The number of Vision data buffers.

In our example, the dictionary directory is "`/usr2/genesis/vision`" and the Vision database files are stored in "`/usr2/genesis/vision0`" and "`/usr2/genesis/vision1`":

```
file_prefix     ;/usr2/genesis/vision0;/usr2/genesis/vision1;
dictsource      /usr2/genesis/vision
```

The GENESIS Vision driver uses the multiple datasource directory entries to locate the database files. It searches the entries in the order they are listed.

Initializing the dictionary and loading GENESIS definitions

You are now ready to initialize and load the GENESIS Vision dictionary using the Data Dictionary Utility (DDU).

1. Change directory to your dictionary directory or use the -d option to specify a dictionary directory.
2. Type

```
gds4init -c
```

Loading existing Vision XFD definitions

The DDU reads the definitions stored in the XFD file(s) for information on how to build the user table entries. You can specify multiple XFD files.

The DDU reads in every file definition and creates the GENESIS catalog entries to describe the Vision data files.

Chapter 9 Micro Focus ExtFH Driver

Introduction

The GENESIS Micro Focus ExtFH driver, available on Windows and Unix, uses direct ExtFH calls to provide very high performance with SQL access.

This section explains how you create the data source file and load ExtFH XFD definitions. It assumes that VORTEX, GENESISsql, and the driver have all been installed according to the instructions for your platform.

NOTE: *You must obtain the Micro Focus ExtFH driver from Micro Focus, Inc (www.microfocus.com). Trifox developed the Micro Focus ExtFH driver exclusively for Micro Focus.*

Making it Work

☞ *Creating the data source file*

The data source file tells the driver how to find the GENESIS dictionary and database files. You create this file with a text editor. Be sure to put it in the `$GENESIS_HOME` directory.

You must include entries for the database and dictionary. Two other entries, related to logging, are optional:

debug_logfile	string	Fully-qualified debugging logfile name.
debug_loglevel	integer	Debugging logging level (0 - none, 1 - control blocks/data, 2 - conversion output)
dictsource	string	(Required) Directory path to GENESIS dictionary ExtFH database files.
file_prefix	string	(Required) Directory path(s) to ExtFH database files, delimited by ";".
file_suffix	string	Suffix appended to file names stored in GENESIS_TABLES.
logging	yes/no	Turns on ExtFH transaction logging.

log_encrypt	yes/no	Turns on ExtFH transaction logfile encryption.
locks_per_file	integer	The maximum number of locks per file.
log_buffer_size	integer	The size of the log buffer.
log_device	yes/no	Turns on device logging.
log_dir	string	The directory where to store the transaction logfiles. The default is the current working directory.
log_file	string	The name of the transaction logfile.
max_files	integer	The maximum number of open ExtFH files.
max_locks	integer	The maximum number of locks.
read_only	yes/no	Disallows any database modifications.
v_buffers	integer	The number of ExtFH data buffers.

In our example, the dictionary directory is "`/usr2/genesis/mfextfh`" and the ExtFH database files are stored in "`/usr2/genesis/mfextfh0`" and "`/usr2/genesis/mfextfh1`":

```
file_prefix     ;/usr2/genesis/mfextfh0;/usr2/genesis/mfextfh1;
dictsource      /usr2/genesis/mfextfh
```

The GENESIS ExtFH driver uses the multiple datasource directory entries to locate the database files. It searches the entries in the order they are listed.

☞ *Initializing the dictionary and loading GENESIS definitions*

You are now ready to initialize and load the GENESIS ExtFH dictionary using the Data Dictionary Utility (DDU).

1. Change directory to your dictionary directory or use the -d option to specify a dictionary directory.
2. Type

```
gds10init -c
```

☞ *Loading existing Vision XFD definitions*

The DDU reads the definitions stored in the XFD file(s) for information on how to build the user table entries. You can specify multiple XFD files.

The DDU reads in every file definition and creates the GENESIS catalog entries to describe the ExtFH data files.

Chapter 10
Messages & Codes

Generic GENESISsql Messages

***position* End of buffer reached**

Cause: The SQL statement ended prematurely.

Action: Check the syntax for the command you are using.

***position* Illegal character**

Cause: The SQL statement contains an illegal character at the given position.

Action: Check the statement for legality.

***position* Identifier too long**

Cause: The SQL statement contains an identifier that is too long. Identifiers are limited to 30 characters.

Action: Rename the identifier.

***position* Ending quote missing**

Cause: The SQL statement is missing an ending quote.

Action: Add the missing quote.

***position* String too long**

Cause: The SQL statement contains a string constant that is too long.

Action: Use a bind variable.

position* yacc: *msg

Cause: The SQL statement cannot be parsed correctly.

Action: Check for invalid keywords.

Table *name* undefined

Cause: The SQL statement references a table that is not defined in the catalog.

Action: Correct the statement to reference a defined table or define the table in the catalog.

Number of columns does not match number of values

Cause: The SQL INSERT statement's values do not match the number of columns defined for the table or listed in the column list.

Action: Correct the statement.

Illegal number of parameters for builtin function

Cause: The SQL statement has the wrong number of parameters for the builtin function.

Action: Correct the statement.

Character array too big (max: *number*)

Cause: The SQL statement contains a character array that is too big.

Action: Correct the statement.

Too many tables in SELECT (max: *number*)

Cause: The SQL SELECT statement contains too many tables.

Action: Correct the statement.

Column *name* undefined

Cause: The SQL statement references a column that is not defined in the catalog.

Action: Correct the statement or define the column/table in the catalog.

Non aggregates require a GROUP BY expression

Cause: The SQL SELECT statement contains aggregate and non aggregate select list items and this requires a GROUP BY expression.

Action: Edit the statement and use GROUP BY or change the statement's structure.

to_char/date/ number's format mask must be a constant string

Cause: The SQL statement uses a data conversion function with a non-constant format mask string.

Action: Correct the statement.

Too many sort columns (max: number)

Cause: The SQL SELECT statement has too many columns in the ORDER BY clause.

Action: Reduce the number of columns in the ORDER BY clause.

Sort column *name* out of range (1 - *number*)

Cause: The SQL SELECT statement's ORDER BY clause references a column number that is out of range.

Action: Correct the statement.

Column *name* already defined

Cause: The SQL CREATE TABLE/VIEW statement has duplicate column names.

Action: Correct the statement.

Too many columns specified

Cause: The SQL statement has too many columns defined.

Action: Correct the statement.

Too many sub-queries at level *number* (max: *number*)

Cause: The SQL statement contains too many sub-queries.

Action: Correct the statement.

Sub-query must return a single column

Cause: The SQL statement contains a sub-query whose select list has more than one column.

Action: Correct the statement.

Operation requires *named* authorization

Cause: The SQL statement requires the specified authorization.

Action: Ensure that you have authority to issue the statement. Contact your DBA.

Invalid password

Cause: The connection password is incorrect.

Action: Ensure that you are using a valid password. Contact your DBA.

Data truncation (max: *number*)

Cause: Data was truncated.

Action: Notify Trifox support.

Create view column count mismatch (create: *number*, select *number*)

Cause: The SQL CREATE VIEW statement's column list does not match the number of columns in the SELECT statement's select list.

Action: Correct the statement.

NULL not allowed for column

Cause: The SQL INSERT/UPDATE statement is using a NULL value for a not-NULL column.

Action: Correct the statement.

If any numeric operand is NULL then only '==' and '!=' are valid

Cause: The SQL statement's WHERE clause is using an invalid operator with a NULL value.

Action: Correct the statement.

Invalid predicate result (NULL or invalid datatype)

Cause: The SQL statement's WHERE clause returned a NULL or invalid datatype result.

Action: Correct the statement.

Notify Trifox support.

Cause: Internal error.

Action: Notify Trifox support.

Too many cursors opened

Cause: Your application has too many concurrently opened cursors.

Action: Explicitly close cursors when they are no longer needed. If this action does not solve the problem, notify Trifox support.

Function *name* not implemented yet

Cause: Not implemented yet.

Action: Contact Trifox with an enhancement request.

number: Unknown node (type: *name*)

Cause: Internal error.

Action: Notify Trifox support.

Unknown error code

Cause: Internal error.

Action: Notify Trifox support.

Catalog table '*name*' corrupted or out of date

Cause: The specified GENESIS catalog table cannot be read. It has either been modified directly or it is from a different version.

Action: Rebuild the GENESIS catalog with the correct catalog utility.

'GENESIS_HOME' environment variable not found

Cause: The GENESIS_HOME environment variable is not set.

Action: Set the GENESIS_HOME environment variable, following instructions in the installation procedure.

Cannot open file '*filename*'

Cause: The specified file cannot be opened. The path may be wrong, it may not exist, or the permissions are not set correctly.

Action: Check the spelling and verify the location of *filename*.

No datasource specified

Cause: The connect string does not contain a data source specification.

Action: Refer to the chapter, *Connecting Your Application to VORTEX* in the ***Trifox Resource Manual*** for detailed instructions.

Invalid parameter

Cause: The VORTEX COMMAND command sent an invalid parameter for the specified command.

Action: Verify that the parameters you use are all valid.

Too many columns *number* (max: *number*)

Cause: The SQL statement references a table with too many columns.

Action: Correct the statement.

Only '=' is allowed with ROWID

Cause: The SQL statement's WHERE clause contains an invalid ROWID predicate.

Action: Verify that all your ROWID predicates are correct.

Column 'name', integer overflow

Cause: An overflow occurred while converting a number to an integer.

Action: Correct the GENESIS catalog entry for the column.

Column 'name', 8 byte integer not supported on this platform

Cause: The GENESISsql platform does not support eight-byte integers.

Action: Remove references to 8-byte integers from the GENESIS catalog.

Synergex SDMS and SQL

SQL statements have several limitations when you try to use them with Synergex' SDMS system.

- ***Creating Tables*** —Tables are stored in Synergex SDMS files. Tables created using the CREATE TABLE statement create files named `owner%tablename`.
- ***Dropping Tables*** — The DROP TABLE command removes all GENESIS catalog entries for the table. It also deletes the file. Be careful issuing the DROP TABLE. ***If the file contains data for multiple tables, they will all be lost when you delete the file.***

Creating Tables

Since SDMS files must have all indexes defined before the file is actually created, you do not see the `owner%tablename` file after you issue the CREATE TABLE statement. Because the actual file creation is deferred until the first table access, you have the opportunity to define indexes. For example, to create the STAFF table with two indexes, perform the following:

```
CREATE TABLE STAFF (ID INTEGER,NAME CHAR(10),DEPT INTEGER,JOB CHAR(6),
                    YEARS INTEGER,SALARY NUMERIC(8,2),COMM NUMERIC(8,2));
CREATE INDEX STAFF_IX1 ON STAFF(ID);
CREATE INDEX STAFF_IX2 ON STAFF(NAME,DEPT);
```

The table and index definitions are stored in the GENESIS catalog but the file is not created. The first SELECT/INSERT/UPDATE/DELETE statement that references the STAFF table forces the file creation.

Tables that are defined using the GENESIS SDMS Data Dictionary Utility are stored in the filename defined at create-time. In addition, you can use store data for multiple SQL tables in the same SDMS file by using tag fields.

Synergex SDMS-Specific Messages

No data source

Cause: The connect string does not specify a data source.

Action: Refer to the chapter, *Connecting Your Application to VORTEX* in the ***Trifox Resource Manual*** for detailed instructions.

No directory defined

Cause: The data source file does not contain the `datasource` directive.

Action: Verify the data source file contents.

No dictionary source directory defined

Cause: The data source file does not contain the `dictsource` directive.

Action: Verify the data source file contents.

SDMS-1: Authorization failure
SDMS-2: Dictionary access failure
SDMS-3: No license available
SDMS-4: Out of memory

Cause: An SDMS authorization error has occurred.

Action: Refer to your Synergy SDMS documentation.

Format error in 'datafiles'

Cause: The data source file directive `datafiles` is not defined correctly.

Action: Refer to the *GENESISsql Users Guide* for more information.

User does not have drop table permission

Cause: The SQL DROP TABLE statement cannot be performed by this user.

Action: Check the table specified in the SQL DROP TABLE statement.

Table '*name*' not deleted from catalog

Cause: The SQL DROP TABLE statement failed due to an SDMS error.

Action: Notify Trifox support.

Table '*name*' still open by other cursors

Cause: The SQL DROP TABLE statement references a table that is still being accessed by other cursors.

Action: Close all cursors referencing this table first.

'CREATE INDEX' not valid for this table type

Cause: The SQL CREATE INDEX statement is only valid for ISAM type files.

Action: Verify the table specified in the SQL CREATE INDEX statement is an ISAM table.

Table/View '*name*' already in catalog

Cause: The SQL CREATE TABLE/VIEW statement references a table/view that is already defined.

Action: Check the table specified in the SQL CREATE TABLE/VIEW statement.

File '*name*' already exists

Cause: The SQL CREATE TABLE statement references a table that is already created.

Action: Check that the table specified does not already exist.

File '*name*' does not exist

Cause: The SQL statement references a table stored in a file that does not exist. It may have been deleted by another user using a difference catalog.

Action: Notify Trifox support.

Cannot create '*name*': msg

Cause: The specified file cannot be created.

Action: Check *msg* for SDMS error information.

Cannot end transaction: msg

Cause: The SQL COMMIT statement cannot be performed.

Action: Check *msg* for SDMS error information.

Cannot begin transaction: msg

Cause: The SQL COMMIT/ROLLBACK statement cannot be performed.

Action: Check *msg* for SDMS error information.

Cannot delete from *'name*': msg

Cause: The SQL DELETE statement cannot be performed.

Action: Check *msg* for SDMS error information.

Cannot open '*name*' for update: msg

Cause: The specified file cannot be opened for update.

Action: Check *msg* for SDMS error information.

File '*name*' cannot be removed: msg

Cause: The specified file cannot be removed.

Action: Check *msg* for SDMS error information.

Column '*name*' not deleted from catalog: msg

Cause: The specified column cannot be deleted from the catalog.

Action: Check *msg* for SDMS error information.

Index '*name*' not deleted from catalog: msg

Cause: The specified column cannot be deleted from the catalog.

Action: Check *msg* for SDMS error information.

Index column '*name*' not deleted from catalog: msg

Cause: The specified index column cannot be deleted from the catalog.

Action: Check *msg* for SDMS error information.

Cannot insert into '*name*': msg

Cause: The SQL INSERT statement failed.

Action: Check *msg* for SDMS error information.

Cannot update '*name*': msg

Cause: The SQL UPDATE statement failed.

Action: Check *msg* for SDMS error information.

Cannot define DEFAULT_INDEX: msg

Cause: The SQL statement references a table whose file creation failed.

Action: Check *msg* for SDMS error information.

Cannot allocate context: msg

Cause: The SQL statement could not allocated a Synergy SDMS context.

Action: Check *msg* for SDMS error information.

Read error: msg

Cause: The SQL statement caused a read error.

Action: Check *msg* for SDMS error information.

Fetch error: msg

Cause: The SQL statement caused a fetch error.

Action: Check *msg* for SDMS error information.

Buffer overflow: msg

Cause: The SQL UPDATE statement caused a buffer overflow.

Action: Check *msg* for SDMS error information.

Column '*name*', DBL decimal overflow

Cause: An overflow occurred while converting a number to a DBL decimal in column name.

Action: Check GENESIS column definition for this column.

Column '*name*', Unsupported data type: *type*

Cause: GENESIS column has unknown datatype entry.

Action: Check GENESIS column definition for this column.

Column '*name*', Invalid date data: *data*

Cause: The date data is not valid.

Action: Verify the data shown is correct.

SDMS Data Dictionary Utility Messages

******* ERROR: Cannot open files (ddc_init:*data*)**

Cause: An error occurred while opening the Repository files. The `ddc_init` function return code is shown.

Action: Notify your system administrator.

******* ERROR: GENESIS_HOME environment variable not found**

Cause: The `GENESIS_HOME` environment variable is not set.

Action: Set the `GENESIS_HOME` environment variable

******* ERROR: sdms_init failed**

Cause: The `sdms_init` function failed.

Action: Notify your system administrator.

******* ERROR: *tablename* : *SDMS error***

Cause: A SDMS error occurred while accessing the table.

Action: Notify your system administrator.

******* ERROR: Table lookup error: (ddc_fname: *data*)**

Cause: An error occurred while retrieving the list of table definitions. The `ddc_fname` function return code is shown.

Action: Notify your system administrator.

******* ERROR: 8 byte integer not supported on this platform**

Cause: Your platform does not support 8-byte integers.

Action: Remove the 8 byte field definition from your Repository.

******* ERROR: Index *index name*, column *column name* not found**

Cause: The given index references a field that is not defined.

Action: Notify your system administrator.

Column *column name*, unsupported date type: *datemask*

Cause: The column uses an unsupported data format mask.

Action: Notify your system administrator.

Structure *structure name*: Unknown error

Cause: The ddc_struct function returned DDC_ERR for the given structure.

Action: Notify your system administrator.

Structure *structure name*: Not found

Cause: The ddc_struct function could not find the given structure.

Action: The ddc_struct function did not find any field definitions for the given structure.

Action: None. Information message.

Structure *structure name*: No fields found

Cause: The ddc_struct function did not find any field definitions for the given structure.

Action: None. Information message.

Structure *structure name*: No keys found

Cause: The ddc_struct function did not find any key definitions for the given structure.

Action: None. Information message.

Invisible field *field name* ignored

Cause: The invisible field definition is ignored.

Action: Information message

Group field *field name* ignored

Cause: The group field definition is ignored.

Action: Information message

Null key *index name*, optimization reduced

Cause: (SDMS v7) Null keys can only be used for equality operations.

Action: None. Information message.

Null key '*index name*' ignored

Cause: (SDMS v6) Null keys are ignored.

Action: None. Information message

Index *index name*, column *column name* datatype mismatch

Cause: The datatype of the index segment does not match the one defined in the corresponding column definition

Action: Verify your Repository definition for this index and its segments.

Index *index name*, column *column name*, unsupported segment type *type*

Cause: The segment datatype for the given column in the given index is not supported. Subsequent segment definitions for this index will be ignored.

Action: Verify your Repository definition for this index and its segments.

Index *index name* dropped

Cause: The given index definition was dropped because the first segment has an unsupported segment type.

Action: Verify your Repository definition for this index and its segments.

No indexes defined for table *table name*

Cause: No indexes were successfully defined for the given table.

Action: None. Information message.

Software AG ADABAS C and SQL

SQL statements used with Software AG's ADABAS C system have limitations in creating and dropping tables.

- ***Creating Tables*** — Because there is no programmatic method of creating ADABAS C files, you cannot use the CREATE TABLE or CREATE INDEX statements with ADABAS C.
- ***Dropping Tables*** — The DROP TABLE command removes all GENESIS catalog entries for the table. It does not delete the data from the file.

Software AG ADABAS C-Specific Messages

Invalid database '*name*' or dictionary file '*name*'

Cause: The data source file is either missing the database/dictionary directives or they are set to 0.

Action: Refer to the "***Creating the Dictionary File***" on page 65.

Cannot delete from a subtable

Cause: The SQL DELETE statement is referencing a subtable (PE/MU).

Action: Check your program logic.

Only VIEW creation supported

Cause: The SQL CREATE TABLE statement is not supported.

Action: Check your program logic.

Duplicate View '*name*'

Cause: The SQL CREATE VIEW statement is creating a view that already exists.

Action: Check your program logic.

Cannot drop system tables

Cause: The SQL DROP TABLE statement cannot drop GENESIS system tables.

Action: Verify the table specified in the SQL DROP TALBE statement is not a GENESIS catalog table.

Record not found

Cause: The SQL INSERT/UPDATE statement could not locate the correct MU/PE record.

Action: Check your program logic. The SQL INSERT/UPDATE statement is not on the correct MU/PE record.

Column '*name*', Packed Decimal overflow

Cause: An overflow occurred while converting a number to a Packed Decimal.

Action: Check the GENESIS column definition forthis column name.

Column '*name*', Zoned Decimal overflow

Cause: An overflow occurred while converting a number to a Zoned Decimal.

Action: Check the GENESIS column definition for this column name.

Column '*name*', unsupported data type: *type*

Cause: The ADABAS C FDT datatype in the GENESIS catalog is incorrect.

Action: Check the GENESIS column definitions for column name.

Column '*name*', unsupported data type: *data*

Cause: Internal error.

Action: Notify Trifox support.

FDT Utility Messages

Field 'name' skipped, not in synonym file

Cause: The ADABAS C shortname was not found in the synonym file.

Action: If you want to include the field in the table definition, be sure to put its shortname in the synonym file.

Field 'name' skipped, no preceding PE group field

Cause: The PE field was not preceded by a PE group field defintion.

Action: Add the PE group field definition into the synonym file

Index 'name' skipped, not in synonym file

Cause: The ADABAS C shortname was not found in the synonym file.

Action: If you want to include this index in the table definition, be sure to put its shortname in the synonym file.

Table 'name', index 'name', column 'name' is packed decimal with null suppression, partial key lookups will fail

Cause: Partial key lookups with null suppressed packed decimal fields are not supported in ADABAS C.

Action: You have several options. You can drop the index from the synonym file, or make sure that you only use equality operators with the index.

Table 'name', index 'name', column 'name' (m:n) is a partial field segment and cannot be used

Cause: This means that the superdescriptor contains a segment which consists of only a portion of a field. This is not supported in the SQL syntax and this index will not be processed.

Action: No action required.

Field 'name' is not part of the preceding PE group or is not a MU field. Please modify the synonym file.

Cause: The field is not part of the PE group or is not part of the MU fields being defined for the table.

Action: Modify the synonym file so that only the PE or MU fields along with the foreign key field(s) are listed.

Numerics

A

B

C

D

E

F

G

U

V

W

X

Z