Contents

1 Introduction

This manual is intended for software developers who want to combine the reliability of COBOL programs with the flexibility and efficiency of a relational database management system (RDBMS). The manual gives systematic instructions on using the DBMaker COBOL Interface (DCI), a program designed to allow for efficient management and integration of data with COBOL using DBMaker's database engine.

DCI provides a communication channel between COBOL programs and DBMaker. DBMaker COBOL Interface (DCI) allows COBOL programs to efficiently access information stored in the DBMaker relational database. COBOL programs usually store data in standard B-TREE files. Information stored in B-TREE files are traditionally accessed through standard COBOL I/O statements like READ, WRITE and REWRITE.

COBOL programs can also access data stored in the DBMaker RDBMS. Traditionally, COBOL programmers use a technique called embedded SQL to embed SQL statements with COBOL source code. Before compiling the source code, a special pre-compiler translates SQL statements into "calls" to the database engine. These calls execute to access the DBMaker RDBMS during runtime.

Though this technique is a good solution for storing information on a database using COBOL programs, it has some drawbacks. First, it implies COBOL programmers have a good knowledge of the SQL language. Second, a program written in this way is not portable — it cannot work both with B-TREE files and

the DBMaker RDBMS. Furthermore, SQL syntax often varies among databases. This makes COBOL programs embedded with SQL statements for a specific DBMaker RDBMS unable to work with another database. Finally, embedded SQL is difficult to implement with existing programs. In fact, embedded SQL requires significant application re-engineering, including substantial additions to the working storage, data storage, and reworking of each I/O statement's logic.

There is an alternative to embedded SQL. Some suppliers have developed seamless COBOL to database interfaces. These interfaces translate COBOL I/O commands, on the fly, into SQL statements. In this way, COBOL programmers need not be familiar with SQL and COBOL programs can remain portable. However, this solution does present a performance problem.

In fact, SQL has a different purpose than COBOL I/O statements. SQL is intended to be a set-based, ad hoc query language that can find almost any combination of data from a general specification. In contrast, COBOL B-TREE or other data structure calls use well-defined traversal keys or navigation logic or both for direct data access. Therefore, forcing transaction rich, performance sensitive COBOL applications to operate exclusively via SQL-based I/O is often an inappropriate method.

CASEMaker's COBOL interface product, DCI, does not use SQL in this way. DCI provides direct access and traversal to data storage in a manner similar to COBOL's own access to user replaceable COBOL file systems. DCI provides seamless interfaces between COBOL programs and DBMaker file systems. Information exchange between the application and the database are invisible to end users. In cases when full SQL-based file and data storage access is required, like desktop decision support systems (DSS), data warehousing and 4GL applications, DBMaker provides these features along with the reliability and robustness of an RDBMS.

CASEMaker's database and DCI products combine the power of 4GLs and navigational data structures with the ad hoc flexibility of SQL-based database access and reporting while delivering tremendous performance.

1.1 Additional Resources

DBMaker provides a complete set of DBMS manuals including this one. For more information on a particular subject, consult one of the manuals listed below:

- For an introduction to DBMaker's capabilities and functions, refer to the "*DBMaker Tutorial"*
- For more information on designing, administering, and maintaining a DBMaker database, refer to the "*Database Administrator's Guide"*
- For more information on DBMaker management, refer to the *"JServer Manager User's Guide"*
- For more information on DBMaker configurations, refer to the *"JConfiguration Tool Reference"*
- For more information on DBMaker functions, refer to the *"JDBA Tool User's Guide"*
- For more information on the dmSQL interface tool, refer to the "*dmSQL User's Guide*"
- For more information on the SQL language used in dmSQL, refer to the "*SQL Command and Function Reference"*
- For more information on the ESQL/C programming, refer to the "*ESQL/C User's Guide*"
- For more information on the native ODBC API and JDBC API, refer to the "*ODBC Programmer's Guide" and "JDBC Programmer's Guide"*
- For more information on error and warning messages, refer to the "*Error and Message Reference"*

1.2 Technical Support

CASEMaker provides thirty days of complimentary email and phone support during the evaluation period. When software is registered, an additional thirty days of support is included, extending the total software support period to sixty days. However, CASEMaker continues providing email support free of charge for reported bugs after the complimentary support or registered support has expired. For most products, support is available beyond sixty days and may be purchased for twenty percent of the retail price of the product. Please contact sales@casemaker.com for details and pricing.

CASEMaker support contact information, by post mail, phone, or email, for your area is at: www.casemaker.com/support. We recommend searching the most current FAQ database before contacting CASEMaker support staff.

Please have the following information available when phoning support for a troubleshooting enquiry or include this information in your correspondence:

- Product name and version number
- Registration number
- Registered customer name and address
- Supplier/distributor where product was purchased
- Platform and computer system configuration
- Specific action(s) performed before error(s) occurred
- Error message and number, if any
- Any additional helpful information

1.3 Document Conventions

This manual uses a standard set of typographical conventions for clarity and ease of use. The NOTE, Procedure, Example, and Command Line conventions also have a second setting used with indentation.

CONVENTION	DESCRIPTION
Italics	Italics indicate placeholders for information that must be supplied, such as user and table names. The word in italics should not be typed, but is replaced by the actual name. Italics also introduce new words, and are occasionally used for emphasis in text.
Boldface	Boldface indicates filenames, database names, table names, column names, user names, and other database schema objects. It is also used to emphasize menu commands in procedural steps.
KEYWORDS	All keywords used by the SQL language appear in uppercase when used in normal paragraph text.
SMALL CAPS	Small capital letters indicate keys on the keyboard. A plus sign (+) between two key names indicates to hold down the first key while pressing the second. A comma (,) between two key names indicates to release the first key before pressing the second key.
NOTE	Contains important information.
Procedure	Indicates that procedural steps or sequential items will follow. Many tasks are described using this format to provide a logical sequence of steps for the user to follow .
Example	Examples are given to clarify descriptions, and commonly include text, as it will appear on the screen. Other forms of this convention include Prototype and Syntax.
CommandLine	Indicates text, as it should appear on a text-delimited screen. This format is commonly used to show input and output for dmSQL commands or the content in the dmconfig.ini file.

Figure 1-1 Document Conventions Table

2 DCI Basics

This chapter provides essential DCI environment set up and configuration information for DBMaker. Also included is information about using the demonstration program to show you the basic functions of DCI.

The following topics are covered in this chapter:

- Software and hardware requirements
- Step-by-step setup for UNIX and Windows platforms
- Options for configuring DCI for DBMaker
- DCI demonstration program instructions

2.1 DCI Overview

Although traditional COBOL file systems and databases both contain data, they differ significantly. Databases are generally more robust and reliable than traditional file systems. Furthermore, they act as efficient systems for data recovery from software or hardware crashes. In addition, to ensure data integrity, DBMaker RDBMS provides support for referential actions and domain, column and table constraints.

File System and Databases

There are parallels between database data storage and COBOL indexed files. The following table shows each system's data structures and how they correspond.

COBOL Indexed File System Object	Database Object
Directory	Database
File	Table
Record	Row
Field	Column

Figure 2-1 COBOL and Database Object Structures

Indexed file operations are performed on records in COBOL and operations are performed on columns in a database. Logically, a COBOL indexed file represents a database table. Each record in a COBOL file represents a table row in a database and each field represents a table column. Data can have multiple definition types in COBOL while table columns in a database must associate with a particular data type such as integer, character, or date.

➲ **Example**

A COBOL record is defined using the following format:

```
terms-record.
        03      terms-code      PIC 999.
        03      terms-rate      PIC s9v999.
        03      terms-days      PIC 9(2).
        03      terms-descript  PIC x(15).
```

The COBOL record in the above example has the following representation in a database. Notice how each row is an instance of the COBOL 01 level record terms-record.

TERMS_CODE	TERMS_RATE	TERMS_DAYS	TERMS_DESCRIPT
234	1.500	10	net 10
235	1.750	10	net 10
245	2.000	30	net 30
255	1.500	15	net 15
236	2.125	10	net 10
237	2.500	10	net 10
256	2.000	15	net 15

Figure 2-2 COBOL Records Converted to Database Rows

Accessing Data

isCOBOL generic file handler interfaces is JISAM. JISAM is the standard indexed file system supplied with isCOBOL. See isCOBOL manual for more detail about JISAM.

DCI, in combination with data dictionaries, bridges data access in a COBOL-based application program interface (API) and DBMaker's database management system. Users may access data through the API. Furthermore, ad hoc data queries can use either of DBMaker's SQL interfaces: dmSQL or JDBA Tool. The

isCOBOL compiler creates data dictionaries which are discussed in Chapter 3, *Data Dictionaries*.

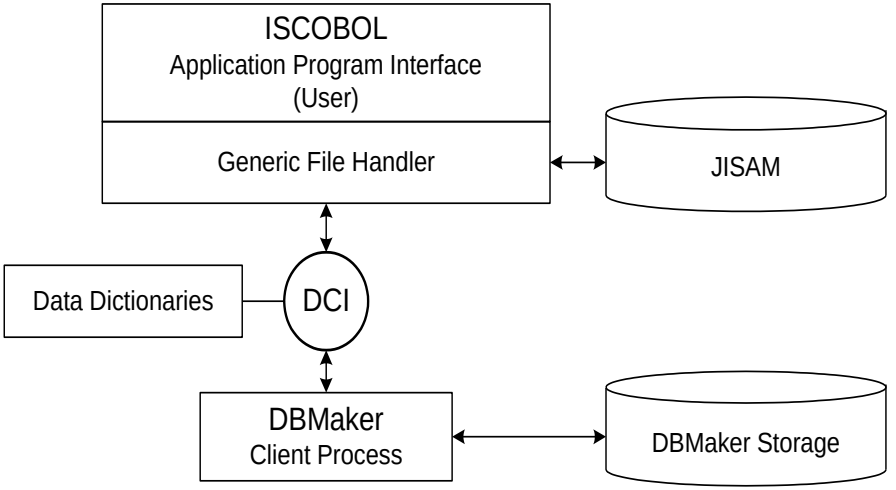


Figure 2-3 Data Flowchart

2.2 System Requirements

DCI for DBMaker is an add-on module that must be linked with the isCOBOL runtime system.

DCI supports the following platforms:

- Windows 32bit and x86_64bit (Windows 2008/7/8/2012/10)
- Linux 32bit (glibc 2.3) and x86_64bit (glibc 2.7)
- Windows 32bit and x86_64bit (Windows 2000/XP/2003/Vista)

DCI requires the following software:

- DBMaker version 5.2 or greater
- isCOBOL Framework
- JDK 1.42 or greater

2.3 Setup Instructions

Linux

First, copy DCI to the directory where isCOBOL is installed as shown in the example below. Next, set the paths for the isCOBOL bin directory and the JDK and JAVAC directories.

➲ **Example: For ISCOBOL 2015 and previous version.**

```
cp libdci.so /app/isCOBOL2013R2/native/lib
```

➲ **Example: For ISCOBOL2016 and above version**

```
cp libdci_2016.so /app/isCOBOL2016R2/native/lib/libdci.so
```

Windows

First, copy DCI to the directory where isCOBOL is installed as shown in the example below. Next, set the OS library path to the DBMaker library directory.

➲ **Example: For ISCOBOL2015 and previous version**

```
copy dci.dll c:/program files/veryant/ISCOBOL2013R2/bin
```

➲ **Example: For ISCOBOL2016 and above version**

```
copy dci_2016.dll c:/program files/veryant/ISCOBOL2016R2/bin/dci.dll
```

2.4 Basic Configuration

The DCI_CONFIG file sets parameters for DCI that determine how data appears in the database and defines certain DBA functions allowing database access. The following configuration variables must be set before using DCI.

- DCI_DATABASE
- DCI_LOGIN
- DCI_PASSWD
- DCI_EFDPATH: first path priority
- DCI_XFDPATH : second path priority

➲ **Example**

Here is a basic DCI_CONFIG file.

```
DCI_LOGIN SYSADM
DCI_PASSWD
DCI_DATABASE DBMaker_Test
DCI_EFDPATH /usr/dbmaker/dictionaries1
DCI_XFDPATH /usr/dbmaker/dictionaries
```

DCI_DATABASE

DCI_DATABASE specifies the database that will transact with DCI. This database must exist. Use DBMaker setup to establish it. Please note, database names are case-sensitive by default, and must be less than or equal to 128 characters.

➲ **Syntax**

The following entry must be included in the configuration file. In this example, we show DBMaker_Test as the database to be used with DCI.

```
DCI-DATABASE DBMaker_Test
```

NOTE Please refer to *Chapter 7, DCI_DATABASE* for more information.

DCI_LOGIN

Provide your COBOL application with a user name to ensure it has permission to access objects in the database. The configuration variable DCI_LOGIN sets the username for all COBOL applications that use DCI. Initially set to SYSADM to ensure full access to databases, this variable is easily changed to another user name as described in *Chapter 7, DCI_LOGIN*.

➲ **Syntax**

The DCI configuration file must include the following entry for DCI to connect with databases via the SYSADM username:

```
DCI_LOGIN SYSADM
```

DCI_PASSWD

Once a username has been specified via the DCI_LOGIN variable, a database account is associated with it. Please note, by default DBMaker sets no password for SYSADM. Your database administrator will know if the account information (LOGIN, PASSWD) is correct. See *Chapter 7, DCI_PASSWD* for more information.

➲ **Syntax**

The configuration file should appear as follows when the database account is set to SYSADM.

```
DCI_PASSWD
```

DCI_EFDPATH

DCI_EFDPATH specifies the directory where data dictionaries are stored. The default value is the current directory. To be compatible with Acucobol, DCI also supports the DCI_XFDPATH.

➲ **Syntax 1**

If data dictionaries are stored in */usr/dbmaker/dictionaries*, include this entry in the configuration file:

```
DCI_EFDPATH /usrdbmaker/dictionaries
```

- **Syntax 2**

  When specifying more than one path, separate by spaces. For example:

```
DCI_EFDPATH /usr/dbmaker/dictionaries /usr/dbmaker/dictionaries1
```

- **Syntax 3**

  Use double-quotes in a WIN-32 environment when including embedded spaces. For example:

```
DCI_EFDPATH c:\tmp\xfdlist "c:\my folder with space\xfdlist"
```

2.5 Invalid Data

Some data is valid for COBOL applications but invalid for DBMaker RDBMS databases. Data types not accepted by RDBMS are listed here along with solutions that DCI implements to solve this problem.

COBOL Value	Where it is considered illegal
LOW-VALUES	In USAGE DISPLAY NUMBERS and text fields
HIGH-VALUES	In USAGE DISPLAY NUMBERS, COMP-2 numbers and COMP-3 numbers
SPACES	In USAGE DISPLAY NUMBERS and COMP-2 numbers
Zero	In DATE fields

Figure 2-4 Illegal COBOL Data

Check the internal storage format of other numeric types to determine which Figure 2-4 category it applies to. BINARY numbers and values in BINARY text fields are always legal.

Certain data types must be converted before DBMaker will accept them. DCI converts these values as follows:

- Illegal LOW-VALUES: stored as the lowest possible value, 0 or –99999, or DCI_MIN_DATE default value
- Illegal HIGH-VALUES: stored as the highest possible value, 99999, or DCI_MAX_DATE default value
- Illegal SPACES: stored as zero (or DCI_MIN_DATE, in the case of a date field)
- Illegal DATE values: stored as DCI_INV_DATE default value
- Illegal TIME: stored as DCI_INV_DATE default value

Null fields sent to DCI from a database convert to COBOL in the following ways:

- Numbers (including dates) convert to zero
- Text (including binary text) converts to spaces

If you want to change these conversion rules, except for the key fields, you can use DCI_NULL_ON_ILLEGAL_DATE that convert to NULL illegal COBOL data.

2.6 Sample Application

A sample application program is included with the DCI files and demonstrates mapping application data to a DBMaker database. This section teaches:

- Setting up the application
- Compiling source code to create application object code
- Application data input
- Data access using dmSQL and JDBA Tool
- How source code conforms to the generated table's schema

Setting up the Application

Located in the DCI directory, the application consists of these files INVD.CBL, INVD.FD, INVD.SL, TOTEM.DEF, ISCOBOL.DEF, ISGUI.DEF, ISCRT.DEF, ISFONTS.DEF, SHOWMSG.DEF, and the DCI.CFG. The steps for compiling the application from the source code, INVD.CBL demonstrated below.

➲ **Running the Application**

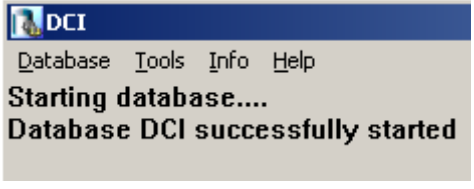
1. Setup a database in DBMaker to accept data from DCI. For example, we created a database called DCI using JServer Manager with all default settings. Specifically, SYSADM is the default login name and no password was set. For information on creating and setting up a database, refer to the *Database Administrator's Reference* or the *JServer Manager User's Guide*.
2. Located in the DCI directory, use a text editor to open the DCI.CFG file. Set the configuration variables to appropriate values. Refer to *Section 2.4, Basic Configuration* for details.

➲ **Example**

```
DCI_DATABASE     DCI
DCI_LOGIN        SYSTEM
DCI_PASSWD
//DCI_LOGFILE
DCI_STORAGE_CONVENTION Dca
//DCI_XFDPATH C:\DCI
```

3. Run the DBMaker Server program: dmserver.exe. When prompted, select the database designated by the DCI.CFG file.

4. The database starts normally and the following window appears. If any problems or error messages occur, refer to the *Error Message Reference* or the *Database Administrator's Guide*.



5. From the command prompt go to the ..*DCI* directory.
6. Define DCI_CONFIG by entering the following at the command prompt.

➲ **Syntax 6a**

```
..\DCI\>SET DCI_CONFIG=C:\..\DCI\DCI.CFG
```

7. Compile the INVD.CBL by entering the following at the command prompt.

➲ **Syntax7a**

```
..\DCI\> iscc -efd INVD.CBL
```

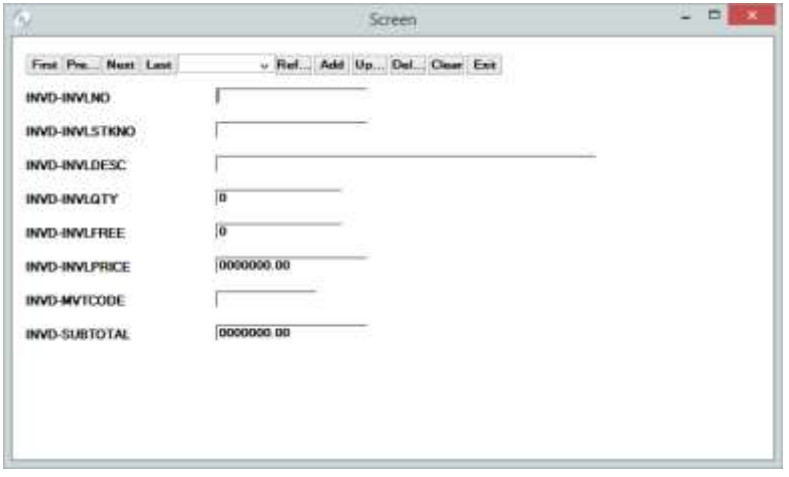
8. The file will be compiled and will create a new object code file INVD.JAVA and data dictionary file INVD.efd. To run the object file follow steps 9 and 10 in "Running the application", below.
9. Run the COBOL program INVD.class using iscrun.

➲ **Syntax 9a**

From the same directory, enter the following command:

```
..\DCI\> iscrun -c CBLCONFIG INVD
```

10. The file CBLCONFIG contains the command line iscobol.file.index=dci and is used to set the default file system. For more information refer to *Chapter 5, Compiler and Runtime Options*.
11. The INVD COBOL application window opens as shown below, allowing data entry.



NOTE *For instructions on adding records, refer to "Adding Records" below*

Adding Records

Once the application has been started (see "Running the application" above) it is a simple matter to add records to the application, and subsequently, to the database. The field INVD-INVLNO is a key field, so a unique value is required for a record to be a valid entry. All other fields may be left blank. When you have finished entering values into the field, click the **Add** button. The values entered into the fields will now be saved in the DBMaker database specified by the DCI.CFG variable, DCI-DATABASE.

➲ **Example**

The file descriptor for the application looks like this:

```
FD INVD
        LABEL RECORDS ARE STANDARD
```

```
01      INVD-R
            05      INVD-INVLNO             PIC X(10).
            05      INVD-INVLSSTKNO         PIC X(10).
            05      INVD-INVLDESC           PIC X(30).
            05      INVD-INVLQTY            PIC 9(8).
            05      INVD-INVLFREE           PIC 9(8).
            05      INVD-INVLPRICE          PIC 9(7)V99.
            05      INVD-MVTCODE            PIC X(6).
            05      INVD-SUBTOTAL           PIC 9(7)V99.
```

Once a record has been added, you may browse through the entries by selecting the **First**, **Previous**, **Next**, or **Last** buttons on the **Screen** window. An individual record may be selected from the drop-down menu at the top of the **Screen** window that displays all the key field values. The section "Accessing Data" describes how to browse data using DBMaker SQL based tools.

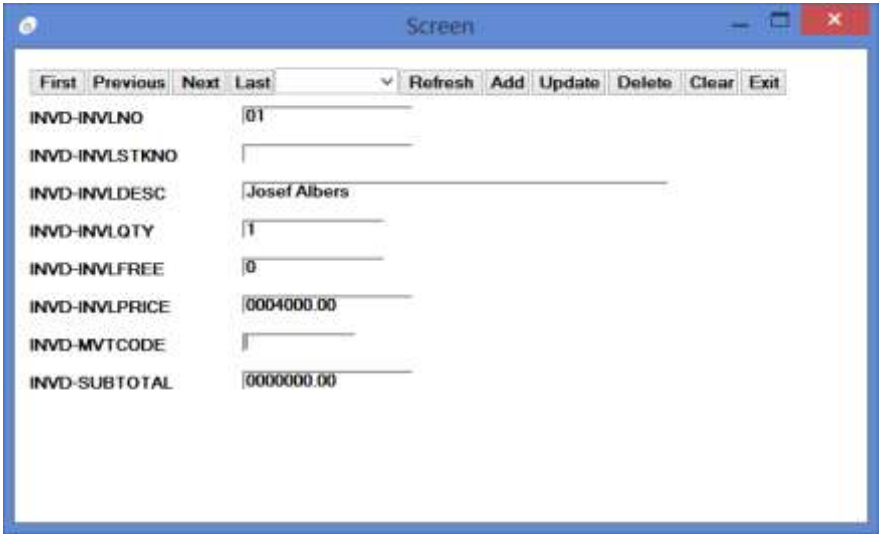


Figure 2-5 INVD-INVLNO Key Field Sample Entry

Accessing Data

To browse and manipulate records created within the sample application is straightforward. First, we recommend that you familiarize yourself with one of

the DBMaker tools: dmSQL, DBA Tool, or JDBA Tool. For information on the use of these tools, refer to the *Database Administrator's Guide*, or the *JDBA Tool User's Guide*. The following example shows how data can be accessed using JDBA Tool.

The INVD application must first be shut down, because it places a lock on the table that has been created within the database. Connect to the database with JDBA. You will be able to see the table by expanding the Tables node within the database tree as shown below.

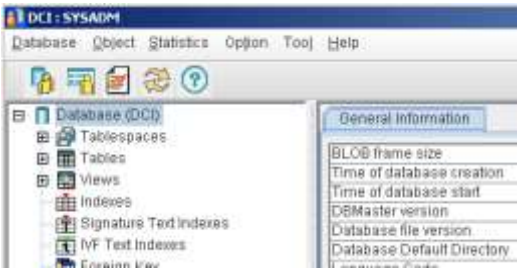


Figure 2-6 INVD Application Tables Tree Node

Double clicking on the **SYSADM.invd** table allows you to view the table's schema. All of the columns and their properties can be viewed here.

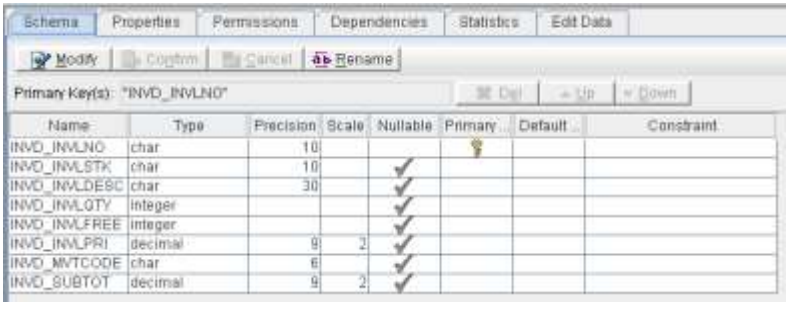
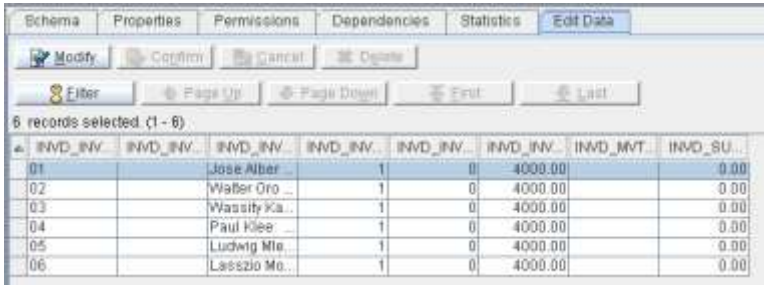


Figure 2-7 SYSADM.invd Table Schema

Selecting the **Edit Data** tab allows you to view the values of each field.



INVD_INV...	INVD_INV...	INVD_INV...	INVD_INV...	INVD_INV...	INVD_INV...	INVD_MVT...	INVD_SU...
01		Jose Alber...	1	0	4000.00		0.00
02		Walter Oro ...	1	0	4000.00		0.00
03		Wassily Ka...	1	0	4000.00		0.00
04		Paul Klee ...	1	0	4000.00		0.00
05		Ludwig Mie...	1	0	4000.00		0.00
06		Lasszlo Mo...	1	0	4000.00		0.00

Figure 2-8 SYSADM.invd Edit Data Tab Field Values

3 Data Dictionaries

Data Dictionaries describe how extended file descriptor (.XML) files are created and accessed. DCI avoids using SQL function calls embedded in COBOL code by using a special feature of isCOBOL. When a COBOL application is compiled using the "–EFD" option data dictionaries are generated. These are known as "extended file descriptors" (EFD files), which are based on COBOL file descriptors. DCI uses the data dictionaries to map data between the fields of a COBOL application and the columns of a DBMaker table. Every DBMaker table used by DCI has at least one corresponding data dictionary file associated with it.

NOTE *Refer to Chapter 5.3 of the isCOBOL User's Guide for more detailed information and rules concerning the creation of EFDs.*

3.1 Assigning Table Names

Database tables correspond to file descriptors in the FILE CONTROL section of the COBOL application. The database tables must have unique names under 128 bytes in length (128 ASCII characters).

It is possible for a DBMaker table to have more columns than a COBOL program's corresponding file descriptor. It is also possible to have different column orders than the COBOL program's corresponding file descriptor.

The number of columns in the database table and the number of fields in the COBOL program that is accessing the table are not required to match. The DBMaker table can have more columns than the COBOL program references; however, the COBOL program may not have more fields than the DBMaker table. Ensure that any extra columns are set correctly when adding new rows to a table.

isCOBOL generates EFD file names by default from the FILE CONTROL section. If the SELECT statement for the file has a variable ASSIGN *name* (ASSIGN TO *filename*), then specify a starting name for the EFD file using a FILE directive (refer to $EFD FILE Directive in chapter 4). If the SELECT statement for the file has a constant ASSIGN *name* (such as ASSIGN TO "EMPLOYEE"), then the constant is used to generate the EFD file name. If the ASSIGN phrase refers to a device and is generic (such as ASSIGN TO "DISK"), then the compiler uses the SELECT name to generate the EFD file name.

File names and usernames are case-insensitive. All file descriptors containing uppercase characters will be converted to lowercase. Users must be aware of this if using a case sensitive operating system.

➲ **Example 1**

If the FILE CONTROL section contains the following line of text:

```
SELECT FILENAME ASSIGN TO "Customer"
```

➲ **Example 2**

DCI, based on dictionary information read in "customer.xml", will make a DBMaker table called "*username*.customer". The isCOBOL compiler always

creates a file name in lowercase. The “username” default is determined by the DCI_LOGIN value in the DCI_CONFIG file, or can be changed with the DCI_USER_PATH configuration variable.

```
SELECT FILENAME ASSIGN TO "CUSTOMER"
```

➲ **Example 3**

If the file has a file extension, DCI replaces “.” characters with “_”. DCI will open a DBMaker table named “username.customer_dat”.

```
SELECT FILENAME ASSIGN TO "customer.dat"
```

➲ **Example 4**

DCI_MAPPING can be used to make the dictionary customer.efd available. Since DCI uses the base name to look for the EFD dictionary, in this case it looks for an XML file named “customer_dat.xml”. The following setting is based on an XML file named “customer.xml”.

```
DCI_MAPPING customer*=customer
```

COBOL applications may use the same base file name in different directories. For example a COBOL application opens a file named “customer” in different directories such as “/usr/file/customer” and “/usr1/file/customer”. To make the file names unique we would include directory paths in the file names. A way to do this is to change the DCI_CONFIG variable DCI_USEDIR_LEVEL to “2”. DCI will then open a table as follows:

COBOL	RDBMS	EFD FILENAME
/usr/file/customer	Usrfilecustomer	usrfilecustomer.xml
/usr1/file/customer	usr1filecustomer	usr1filecustomer.xml

Figure 3-1 Sample DCI_USEDIR_LEVEL to “2”Table

NOTE *Please remember there is a limit to the maximum length of DBMaker table names and that DCI_MAPPING must be used to map .efd file dictionary definitions.*

COBOL Code	Resulting File Name	Resulting Table Name
ASSIGN TO "usr/hr/employees.dat"	employees_dat.efd	employees_dat
SELECT DATAFILE, ASSIGN TO DISK	datafile.xml	datafile
ASSIGN TO "-D SYS$LIB:EMP"	emp.xml	Emp
ASSIGN TO FILENAME	(user specified)	(user specified)

Figure 3-2 Example Table Names Formed From Different COBOL Statements

➲ **Example**

Table names are, in turn, generated from the XML file name. Another way to specify the table name is to use the $EFD FILE directive.

```
*(( EFD FILE = PURCHASE-FILE2 ))
FD PURCHASE-FILE.
01 PURCHASE-RECORD.
05 DATE-PURCHASED.
   10 YYYY            PIC 9(04).
   10 MM              PIC 9(02).
   10 DD              PIC 9(02).
05 PAY-METHOD         PIC X(05).
```

The final name is formed as follows:

- The compiler converts extensions and includes them with the starting name by replacing the "." with an underscore "_".
- It constructs a universal base name from the file name and directory information as specified by the DCI_CONFIG variable DCI_USEDIR_LEVEL. It reduces the base name to 32 characters and converts it to lowercase depending of DCI_CASE value.

3.2 Mapping Columns and Records

The table that is created is based on the largest record in the COBOL file. It contains all of the fields from that record and any key fields. Key fields are specified in the FILE CONTROL section using the KEY IS phrase. Key fields correspond to primary keys in the database table and are discussed in detail in the next section. Note that DCI will create column names for the database that are case-sensitive, unlike table names.

➲ **Example 1**

The following illustrates how data is transferred.

```
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
      SELECT HR-FILE
             ORGANIZATION   IS INDEXED
             RECORD KEY     IS EMP-ID
             ACCESS MODE    IS DYNAMIC.
 DATA DIVISION.
 FILE SECTION.
 FD  HR-FILE
     LABEL RECORDS ARE STANDARD.
 01  EMPLOYEE-RECORD.
     05 EMP-ID              PIC 9(06).
     05 EMP-NAME            PIC X(17).
     05 EMP-PHONE           PIC X(10).
 WORKING-STORAGE SECTION.
 01  HR-NUMBER-FIELD        PIC 9(05).
 PROCEDURE DIVISION.
 PROGRAM-BEGIN.
     OPEN I-O HR-FILE.
     PERFORM GET-NEW-EMPLOYEE-ID.
     PERFORM ADD-RECORDS UNTIL EMP-ID = ZEROS.
     CLOSE HR-FLE.
PROGRAM-DONE.
     STOP RUN.
GET-NEW-EMPLOYEE-ID.
```

```
    PERFORM INIT-EMPLOYEE-RECORD.
    PERFORM ENTER-EMPLOYEE-ID.
INIT-EMPLOYEE-ID.
     MOVE SPACES TO EMPLOYEE-RECORD.
     MOVE ZEROS  TO EMP-ID.
ENTER-EMPLOYEE-ID.
     DISPLAY "ENTER EMPLOYEE ID NUMBER (1-99999),"
     DISPLAY "ENTER 0 TO STOP ENTRY".
     ACCEPT   HR-NUMBER-FIELD.
     MOVE     HR-NUMBER-FIELD TO EMP-ID.
ADD-RECORDS.
     ACCEPT  EMP-NAME.
     ACCEPT  EMP-PHONE.
     WRITE   EMPLOYEE-RECORD.
     PERFORM GET-NEW-EMPLOYE-NUMBER.
```

➲ **Example 2**

The preceding program normally would write all fields sequentially to file. The output would appear as follows:

```
ENTER EMPLOYEE ID NUMBER (1-99999), ENTER 0 TO STOP ENTRY
51100
LAVERNE HENDERSON
2221212999
ENTER EMPLOYEE ID NUMBER (1-99999), ENTER 0 TO STOP ENTRY
52231
MATTHEW LEWIS
2225551212
ENTER EMPLOYEE ID NUMBER (1-99999), ENTER 0 TO STOP ENTRY
```

In a traditional COBOL file system, records will be stored sequentially. Every time a write command is executed, the data is sent to the file. When DCI is used, the data dictionary will create a map for the data to be stored in the database. In this case, the record (EMPLOYEE-RECORD) is the only record in the file.

➲ **Example 3**

The database will create a distinct column for each field in the file descriptor. The table name will be HR-FILE in accordance with the SELECT statement in the FILE-CONTROL section. The database records in the example would therefore have the following structure:

EMP_ID (INT(6))	EMP_NAME (CHAR(17))	EMP_PHONE (DEC(10))
51100	LAVERNE HENDERSON	2221212999
52231	MATTHEW LEWIS	2225551212

Figure 3-3 Table EMPLOYEE-RECORD

In this table, the column EMP-ID is the primary key as defined by the KEY IS statement of the input-output section. The data dictionary creates a "mapping" that allows it to retrieve records and place them in the correct fields. A COBOL application that stores information in this way can take advantage of the backup and recovery features of the database, as well as take advantage of the capabilities of SQL.

Identical Field Names

In COBOL, fields with identical names are distinguished by qualifying them with a group item. DBMaker does not allow duplicate column names on a table. If fields have the same name, DCI will not generate columns for those fields. Adding a NAME directive that associates an alternate name with one or both of the conflicting fields is a method for handling this situation. Please refer to $EFD NAME Directive in Chapter 4 for additional details.

➲ **Example**

In the following example, you would reference PERSONNEL and PAYROLL in your program:

```
FD HR-FILE
        LABEL RECORDS ARE STANDARD.
01      EMPLOYEE-RECORD.
        03      PERSONNEL.
                05      EMP-ID          PIC 9(6).
                05      EMP-NAME        PIC X(17).
                05      EMP PHONE       PIC 9(10).
        03      PAYROLL.
                05      EMP-ID          PIC 9(6).
                05      EMP-NAME        PIC X(17).
```

```
                05      EMP PHONE       PIC 9(10).
```

Long Field Names

DBMaker supports table names up to 128 characters. DCI truncates field names longer than 128 characters. In the case of the OCCURS clause, described below, the truncation is to the original name, not the appended index numbers. The final name, however, including the index number, is limited to the 32 characters. For example, if the field name is Employee-statistics-01 it truncates to Employee_statis_01. It is important to ensure that field names are unique and meaningful within the first 18 characters.

NAME can be used to rename a field with a long name, but within the COBOL application you must continue using the original name. The NAME directive affects only the corresponding column name in the database.

3.3 Using Multiple Record Formats

The example in the previous section showed how fields are used to create a database table. However, the example only shows the case of an application with one record.

A multiple record format is stored differently from a single record format. COBOL programs with multiple records map all records from the "master" (largest) record in the file and any key fields in the file. Smaller records map to the database table by the EFD file but do not appear as discrete, defined columns in the table; they occupy new records in the existing columns of the database.

➲ **Example 1**

Take the previous example but modify the file descriptor to include several records.

```
DATA DIVISION
FILE SECTION
FD HR-FILE
  LABEL RECORDS ARE STANDARD.
01 EMPLOYEE-RECORD.
   05 EMP-ID          PIC 9(6).
   05 EMP-NAME        PIC X(17).
   05 EMP PHONE       PIC 9(10).
01 PAYROLL-RECORD.
   05 EMP-SALARY      PIC 9(10).
   05 DD              PIC 9(2).
   05 MM              PIC 9(2).
   05 YY              PIC 9(2).
```

In this case, the data dictionary is created from the largest file. The record EMPLOYEE-RECORD contains 33 characters, while the record PAYROLL-RECORD contains only 16. In this case, records are sequentially entered into the database. The record EMPLOYEE-RECORD is used to create the schema for the table column size and data type.

EMP_ID (INT(6))	EMP_NAME (CHAR(17))	EMP_PHONE (DEC(10))

Figure 3-4 Preceding Example Table

Fields from the following record are written into the columns according to the character positions of the fields. The result is that no discrete columns exist for the smaller records. The data can be retrieved from the database by the COBOL application because the EFD file contains a map for the fields, but there are no columns in the table representing those fields.
In the previous example, when adding the first record to the database, there is a correlation between the columns and the COBOL fields. When adding the second record, there is no such correlation. The data occupies its corresponding character position according to the field. So the first five characters of EMP_SALARY occupy the EMP_ID column, the last five characters of EMP_SALARY occupy the EMP_NAME column. The fields DD and MM and YY are also located within the EMP_NAME column.

➲ **Example 2**

The following example illustrates this. Given the following input to the COBOL application:

```
ENTER EMPLOYEE ID NUMBER (1-99999), ENTER 0 TO STOP ENTRY
51100
LAVERNE HENDERSON
2221212999
5000000000
01
04
00
```

The fields have been merged and split according to the character positions of the fields relative to the table's schema. Furthermore, the data type of the column EMP_NAME is CHAR. Because DCI has access to the data dictionary, all fields will be mapped back to the COBOL application in the correct positions.
This is a very important fact; by default, the fields of the largest record are used to create the schema of the table, therefore table schema must be carefully considered when creating file descriptors. To take advantage of the flexibility of SQL, data types are consistent between fields for different records that will occupy the same character positions. If a PIC X field is written to a DECIMAL type database column, the database will return an error to the application.

➲ **Example 3**

An SQL select on the first record of all columns in EMP_NAME would display the following:

```
51100, LAVERNE HENDERSON, 2221212999
```

➲ **Example 4**

An SQL select on the second record of all columns in EMP_NAME would display the following:

```
500000, 0000010400
```

3.4 Using EFD File Defaults

In many cases, directives can override the default behavior of DCI. Please refer to EFD Directives for more information.

The compiler uses special methods to deal with the following COBOL elements:

- REDEFINES Clause
- KEY IS phrase
- FILLER data items
- OCCURS Clauses

REDEFINES Clause

A REDEFINES clause creates multiple definitions for the same field. DBMaker supports a single data definition per column. Therefore, a redefined field occupies the same position in the table as the original field. By default, the data dictionary uses the field definition of the subordinate field to define the column data type. Multiple record definitions are essentially redefines of the entire record area. Please refer to the previous section for details on multiple record definitions. Group items are not included in the data dictionary's definition of the resultant table's schema. Instead, the individual fields within the group item are used to generate the schema. Grouped fields may be combined using the USE GROUP directive.

KEY IS Phrase

The KEY IS phrase in the input-output section of a COBOL program defines a field or group of fields as a unique index for all records. The data dictionary maps fields included in the KEY IS phrase to primary keys in the database. If the field named in the KEY IS phrase is a group item, the subordinate fields of the group item become the primary key columns of the table. Use the USE GROUP directive to collect all subordinate fields into one field. Please see $EFD USE GROUP Directive in Chapter 7 for more details.

FILLER Data Items

FILLER data items are placeholders in a COBOL file descriptor. They do not have unique names and cannot be uniquely referenced. The data dictionary maps all other named fields as if the fillers existed in terms of character position, but does not create a distinct field for the FILLER data item.
If a FILLER must be included in the table schema it can be combined with other fields using the USE GROUP directive (see $EFD USE GROUP Directive in Chapter 7) or the NAME directive (see $EFD NAME Directive in Chapter 7).

OCCURS Clauses

The OCCURS clause allows a field to be defined as many times as necessary. DCI must assign a unique name for each database column, but multiple fields defined with an OCCURS clause will all have identical names. To avoid this problem, a sequential index number is appended to the field specified in the OCCURS clause.

➲ **Example 1**

The following file descriptor part…

```
   03 EMPLOYEE-RECORD OCCURS 20 TIMES.
       05 CUST-ID                        PIC 9(5).
```

➲ **Example 2**

…generates the following column names for the database:

```
EMP_ID_1
EMP_ID_2
.
.
.
EMP_ID_5
EMP_ID_6
.
.
.
EMP_ID_19
EMP_ID_20
```

3.5 Mapping Multiple Files

It is possible at runtime to use a single EFD file for multiple files with different names. If the record definitions are identical then it is unnecessary to create a separate EFD for each file.

The runtime configuration variable DCI_MAPPING determines file mapping to an EFD.

Suppose a COBOL application has a SELECT with a variable ASSIGN name, such as EMPLOYEE-RECORD. This variable assumes different values (such as EMP0001 and EMP0002) during program execution. To provide a base name for the EFD, use the FILE directive (see (EFD DATE, USE GROUP)).

➲ **Example**

If "EMP" is the base, then the compiler generates an EFD named "Emp.xml". The asterisk ("*") in the following example is a wildcard character that replaces any number of characters in the file name. The file extension ".xml" is not included in the map. This statement would cause the EFD "emp.xml' to be used for all files with names that begin with "EMP". Add the following entry in the runtime configuration file to ensure that all employee files, each having a unique but related name, use the same EFD:

```
DCI_MAPPING EMP* = EMP
```

The DCI_MAPPING variable is read during the open file stage. The "*" and "?" wildcards can be used within the pattern as follows:

* matches any number of characters

? matches a single occurrence of any character

EMP????? matches EMP00001 and EMPLOYEE, but does not match EMP001 or EMP0001

EMP* matches all of the above

EMP*1 matches EMP001, EMP0001, and EMP00001, but does nor match EMPLOYEE.

*OYEE matches EMPLOYEE

does not match EMP0001 or EMP00001

➲ **Syntax**

Where <*pattern*> consists of any valid filename characters and may include "*" or "?". The DCI_MAPPING variable has the following syntax:

```
DCI_MAPPING [<pattern> = base-efd-name],
```

3.6 Using Views

DCI allows the use of DBMaker views instead of a table. In this case, DCI users must manually create a view and be aware of the following limitations:

- Open a view and do all DML operations when the view is a single table view and the projection column on the original table is without an expression, aggregate or UDF.
- For other kinds of views, open the view as an OPEN INPUT and perform a READ operation only.

➲ **Example**

This example illustrates creating and opening a view. It assumes there are table, named idx_table and the view named idx_view as follows:

```
create table idx_table (idx_1_key char(10) not null primary key, idx_1_data
char(10));
create view idx_view as select idx_1_data, idx_1_key from idx_table;
```

Additionally, these tables contain some data.

```
identification division.

       FILE-CONTROL.
           SELECT IDX-1-FILE
           ASSIGN TO "idx_view"
           ORGANIZATION IS INDEXED
           ACCESS IS DYNAMIC
           RECORD KEY IS IDX-1-KEY.

       DATA DIVISION.
       FILE SECTION.
       FD  IDX-1-FILE.
       01  IDX-1-RECORD.
           03  IDX-1-DATA  PIC X(10).
           03  IDX-1-KEY   PIC X(10).
       WORKING-STORAGE SECTION.
       procedure division.
       main.
```

```
set environment "file.index" to "dci"

open input idx-1-file
 perform until 1=2
  read idx-1-file next
     at end exit perform
   end-read
   display idx-1-record
end-perform

close idx-1-file.
```

3.7 Using Synonyms

DCI allows the use of DBMaker synonyms instead of a table or view. Users can create the synonym on a table, view or remote database's table or view. If the synonym for the view is not a single table view, the user can only OPEN INPUT with that synonym.

3.8 Open Tables in Remote Databases

Users can access the remote database's table or view by adding a special token "@" in the COBOL SELECT statement. For example:

```
SELECT tb1 ASSIGN TO RANDOM, "lnk1@tbl"
```

To use a different user name and password, set DD_DDBMD=1 in the dmconfig.ini and create a remote database link.

➲ **Example**

Connecting to database **dci_db1** and accessing a table in database **dci_db2**.

1. Set DD_DDBMD=1 in dmconfig.ini.
2. Create the table in dci_db2

NOTE*Use dmSQL tool to create the tables in the dci_db2 database*

3. Use a COBOL program to connect to dci_db1 and then open the table in dci_db2.

```
dmconfig.ini
[DCI_DB1]
DB_SVADR = 127.0.0.1
DB_PTNUM = 22999
DD_DDBMD = 1

[DCI_DB2]
DB_SVADR = 127.0.0.1
DB_PTNUM = 23000
DD_DDBMD = 1

Use dmSQL tool to create the table
connect to DCI_DB2 SYSADM;
create table tb1 (c1 int not null, c2 int, c3 char(10), primary key c1);
commit;
disconnect;

COBOL program
identification division.
program-id.RemoteTable.
```

```
date-written.
remarks.
environment division.
input-output section.
file-control.
      SELECT tb1 ASSIGN TO RANDOM, "dci_db2@tb1"
             ORGANIZATION IS INDEXED
             ACCESS IS DYNAMIC
             FILE STATUS IS I-O-STATUS
             RECORD KEY  IS C1.

data division.
file section.
FD  tb1.
01  tb1-record.
        03 C1             PIC 9(8) COMP-5.
        03 C2             PIC 9(8) COMP-5.
        03 C3             PIC X(10).
working-storage section.
77   I-O-STATUS pic xx.
procedure division.
main.
      set environment "file.index" to "dci"
      call "DCI_SETENV" using "DCI_DATABASE" "DCI_DB1"
      call "DCI_SETENV" using "DCI_LOGIN" "SYSADM"

      open i-o tb1
      move 100 TO C1.
      move 200 TO C2.
      move "AAAAAAAAAA" TO C3.
      write tb1-record.
      initialize tb1-record.
      read tb1 next.
          display C1, C2, " ", C3.
      close tb1.
      accept omitted.
      stop run.
```

4 EFD Directives

Directives are comments placed in COBOL file descriptors that alter how the database table is built. Directives change the way data is defined in the database and they assign names to database fields. Directives can also assign names to .EFD files, assign data to binary large object (BLOB) fields, and add comments.

4.1 Using Directive Syntax

Each directive occupies the entire line located immediately before the related line of COBOL code. All directives have the prefix $EFD; a $ symbol in the 7th column followed immediately by EFD.

➲ **Syntax 1**

The following command provides a unique database name for an undefined COBOL variable. Locate the directive directly above the line it affects; in this case the second instance of the COBOL defined variable *qty*.

```
. . .
    03 QTY        PIC 9(03).
 01 CAP.
$EFD NAME=CAPQTY
    03 QTY        PIC 9(03).
```

➲ **Syntax 2**

Directives may also be specified using this ANSI-compliant syntax:

```
*(( EFD NAME=CAPQTY ))
```

➲ **Syntax 3**

More than one directive may be combined together. Directives may share the same line when preceded by the prefix $EFD and separated by a space or comma.

```
$EFD NAME=CAPQTY, ALPHA
```

➲ **Syntax 4**

Alternatively, the following can be used.

```
*(( EFD NAME=CAPQTY, ALPHA))
```

4.2 Using EFD Directives

Directives are used when a COBOL file descriptor is mapped to a database field. The $EFD prefix tells the compiler the proceeding command will be used during generation of the data dictionary.

$EFD ALPHA Directive

To store non-numeric data like, LOW-VALUES or special codes, in numeric keys, this directive allows a data item that has been defined as numeric in the COBOL program to be treated as alphanumeric text (CHAR (n) n 1-max column length) in the database.

➲ **Syntax 1**

```
$EFD ALPHA
```

➲ **Syntax 2**

```
*(( EFD ALPHA ))
```

Moving a non-numeric value like “A234” to the key without using the $EFD ALPHA directive would be rejected by the database.

➲ **Example 1**

Assume the KEY IS code-key has been specified and we have the following record definition. CODE-NUM is a numeric value and since group items are disregarded in the database, it is the key field,.

```
01 EMPLOYEE-RECORD.
    05 EMP-KEY.
       10 EMP-NUM          PIC 9(5).
```

➲ **Example 2**

Using the $EFD ALPHA directive changes a non-numeric value like “A234” saves the record from rejection by the database, since “A234” is an alphanumeric value and CODE-NUM is a numeric value.

```
01 EMPLOYEE-RECORD.
    05 EMP-KEY.
$EFD ALPHA
```

```
      10 EMP-NUM           PIC 9(5)
```

➲ **Example 3**

Now, the following operation can be used without worrying about any rejection .

```
MOVE "C0531" TO CODE-KEY.
WRITE CODE-RECORD.
```

$EFD BINARY Directive

To allow for data in a field to be alphanumeric data of any type, for example, LOW-VALUES, use the BINARY directive. With LOW-VALUES, for example, COBOL allows both LOW and HIGH-VALUES in a numeric field but DBMaker does not.

BINARY directives transform the COBOL fields into DBMaker BINARY data types.

➲ **Syntax 1**

```
$EFD BINARY
```

➲ **Syntax 2**

```
*(( EFD BINARY ))
```

➲ **Example**

This allows moving of LOW-VALUES to CODE-NUM.

```
01 EMPLOYEE-RECORD.
    05 EMP-KEY.
       10 EMP-TYPE      PIC X.
$(( EFD BINARY ))
       10 EMP-NUM       PIC 9(05).
       10 EMP-SUFFIX    PIC X(03).
```

$EFD COMMENT DCI SERIAL n Directive

This directive defines a serial data field and an optional starting number "n". Trigger DBMaker to generate a serial number by inserting a record and supplying a 0 value for the serial field. DBMaker will not generate a serial number when inserting a new row but supplying an integer value instead of a 0 value. If the supplied integer value is greater than the last serial number generated, DBMaker

resets the sequence of generated serial numbers to start with the supplied integer value.

➲ **Syntax 1**

```
$EFD COMMENT DCI SERIAL 1000
```

➲ **Syntax 2**

```
*(( EFD COMMENT DCI SERIAL 1000 ))
```

➲ **Example**

```
01 EMPLOYEE-RECORD.
    05 EMP-KEY.
       10 EMP-TYPE      PIC X.
$(( EFD COMMENT DCI SERIAL 250 ))
       10 EMP-COUNT     PIC 9(05).
```

$EFD COMMENT Directive

This directive identifies comments in an EFD file. In this way, information can be embedded in an EFD file so that other applications can access the data dictionary. Embedded information in the form of a comment using this directive does not interfere with processing by DCI interfaces. Each comment will be recognizable in the EFD file as having the "#"symbol in column 1.

➲ **Syntax 1**

```
$EFD COMMENT text
```

➲ **Syntax 2**

```
*(( EFD COMMENT text ))
```

$EFD DATE Directive

DATE type data is a special data format supported by DBMaker that is not supported by COBOL. To take advantage of the properties of this data type, convert fields from numeric type data. The DATE directive's purpose is storage of dates in fields in the database. This directive differentiates dates from other numbers, so that they enjoy the properties associated with dates in the RDBMS.

➲ **Syntax 1**

```
$(( EFD DATE=date-format-string ))
```

➲ **Syntax 2**

```
*((EFD DATE= ))
```

If no date-format-string is specified, then six-digit (or six-character) fields are retrieved as YYMMDD from the database. Eight-digit fields are retrieved as YYYYMMDD.

The *date-format-string* is a description of the desired date format, composed of characters.

CHARACTER	DESCRIPTION
M	Month (01-12)
Y	Year (2 or 4 digit)
D	Day of month (01-31)
J	Julian day (00000000-99999999)
E	Day of year (001-366)
H	Hour (00-23)
N	Minute (00-59)
S	Second (00-59)
T	Hundredths of a second

Figure 4-1 date-format-string Characters

Each character in a date format string is a placeholder representing the type of information stored at that location. The characters also determine how many digits will be used for each type of data.

For example, although you would typically represent the month with two digits, if you specify MMM as part of your date format, the resulting date will use three digits for the month, with a left-zero filling the value. If the month is given as M, the resulting date will use a single digit, and will truncate on the left.

JULIAN DATES

The definition of Julian dates varies, so the DATE directive allows for a flexible representation of Julian dates. Many sources define a Julian day as the day of the year, with January 1st being 001, January 2nd being 002, etc. To use this definition, use FEE (day of year) in the date formats.

Other references define a Julian day as the number of days since a specific base date. This definition is represented in the DATE directive by the letter J. For example, a six-digit date field would be preceded with the directive $EFD DATE=JJJJJJ. The default base date for this form of Julian date is *01/01/0001AD*. You may define your own base date for Julian date calculations by setting the configuration variable DCI_JULIAN_BASE_DATE.

DCI considers dates in the following range to be valid:

01/01/0001 to 12/31/9999

If a COBOL program attempts to write a record containing a date that DCI knows is invalid, DCI inserts a date value that depends on the setting specified by the DCI_INV_DATE, DCI_MIN_DATE and DCI_MAX_DATE configuration variables into the date field and writes the record.

If a COBOL program attempts to insert into a record from a table with a NULL date field, zeroes are inserted into that field in the COBOL record.

If a date field has two-digit years, then years 0 through 19 are inserted as 2000 through 2019, and years 20 through 99 are inserted as 1920 through 1999. You can change this behavior by changing the value of the variable DCI_DATE_CUTOFF. Also, refer to the configuration variables DCI_MAX_DATE and DCI_MIN_DATE for information regarding invalid dates when the date is in a key.

NOTE *If a field is used as part of a key, the field cannot be a NULL value.*

USING GROUP ITEMS

You may place the DATE directive in front of a group item, so long as you also use the USE GROUP directive.

➲ **Example 1**

```
$EFD DATE
    05 DATE-PURCHASED      PIC 9(08).
    05 PAY-METHOD          PIC X(05).
```

The date-purchased column will have eight digits and will be type DATE in the database, with a format of YYYYMMDD.

➲ **Example 2**

```
$(( EFD DATE, USE GROUP ))
    05 DATE-PURCHASED.
       10 YYYY            PIC 9(04).
       10 MM              PIC 9(02).
       10 DD              PIC 9(02).
    05 PAY-METHOD         PIC X(05).
```

$EFD FILE Directive

The FILE directive names the data dictionary with the file extension .EFD. This directive is required when creating a different .EFD name from that specified in the SELECT COBOL statement. Another case that requires this kind of directive is when the COBOL file name is not specific.

➲ **Syntax 1**

```
$EFD FILE=filename
```

➲ **Syntax 2**

```
*(( EFD FILE=filename ))
```

➲ **Example**

In this case, the isCOBOL compiler makes an EFD file name called CUSTOMER.xml.

```
ENVIRONMENT DIVISION.
 FILE-CONTROL.
 SELECT FILENAME ASSIGN TO VARIABLE-OF-WORKING.
 . . .
 DATA DIVISION.
 FILE SECTION.
$EFD FILE=CUSTOMER
 FD FILENAME
 . . .
```

$EFD NAME Directive

The NAME directive assigns a DBMaker RDBMS column name to the field defined on the next line. In DBMaker, all column names are unique and must be less than 128 characters. This directive can be used to avoid problems created by columns with incompatible or duplicate names.

➲ **Syntax 1**

```
$EFD NAME=columnname
```

➲ **Syntax 2**

```
*(( EFD NAME=columnname ))
```

➲ **Example**

In DBMaker RDBMS, the COBOL field cus-cod will map to a RDBMS field named customercode.

```
$EFD NAME=customercode
    05 cus-cod        PIC 9(05).
```

$EFD NUMERIC Directive

The NUMERIC directive causes the subsequent field to be treated as an unsigned integer if it is declared as alphanumeric.

➲ **Syntax 1**

```
$EFD NUMERIC
```

➲ **Syntax 2**

```
*(( EFD NUMERIC ))
```

➲ **Example**

The field customer-code will be stored as INTEGER type data in the DBMaker table.

```
$efd numeric
        03      customer-code            PIC x(7).
```

$EFD USE GROUP Directive

The USE GROUP directive assigns a group of items to a single column in the DBMaker table. The default data type for the resultant dataset in the database column is alphanumeric (CHAR (n), where n = 1- max column length). The directive may be combined with other directives if the data is stored as a different type (BINARY, DATE, NUMERIC). Combining fields into groups improves processing speed on the database, so effort is made to determine which fields can be combined.

➲ **Syntax 1**

```
$EFD USE GROUP
```

➲ **Syntax 2**

```
*(( EFD USE GROUP ))
```

➲ **Example 1**

By adding the USE GROUP directive, the data is stored as a single numeric field where the column name is *code-key*.

```
01 CODE-RECORD.
$EFD USE GROUP
    05 CODE-KEY.
       10 AREA-CODE-NUM    PIC 9(03).
       10 CODE-NUM         PIC 9(07).
```

➲ **Example 2**

The USE GROUP directive can be combined with other directives. The fields are mapped into a single DATE type data column in the database.

```
$(( EFD DATE, USE GROUP ))
    05 DATE-PURCHASED.
       10 YYYY             PIC 9(04).
       10 MM               PIC 9(02).
       10 DD               PIC 9(02).
```

$EFD VAR-LENGTH Directive

VAR-LENGTH directives force DBMaker to use a BLOB field to save COBOL fields. This is useful if the COBOL field is over or near the maximum allowable

column size for regular data types. Please refer to *Chapter 8, COBOL Conversions.*
Since BLOB fields cannot be used in any key field and are slower to retrieve then normal data type fields such as CHAR, we suggest you use this directive only when needed.

➲ **Syntax 1**

```
$EFD USE VAR-LENGTH
```

➲ **Syntax 2**

```
*(( EFD USE VAR-LENGTH ))
```

➲ **Example**

```
$EFD USE VAR-LENGTH
    05 LARGE-FIELD    PIC X(10000).
```

$EFD WHEN Directive for File Names

The WHEN directive is used to build certain columns in DBMaker that wouldn't normally be built by default. By specifying a WHEN directive in the code, the field (and subordinate fields in the case of a group item) immediately following this directive will appear as an explicit column, or columns, in the database tables. The database stores and retrieves all fields regardless of whether they are explicit or not. Furthermore, key fields and fields from the largest record automatically become explicit columns in the database table. The WHEN directive is only used to guarantee that additional fields will become explicit columns when you want to include multiple record definitions or REDEFINES in a database table.
One condition for how the columns are to be used is specified in the WHEN directive. Additional fields you want to become explicit columns in a database table must not be FILLER or occupy the same area as key fields.

➲ **Syntax 1**
Equal to

```
$EFD WHEN field = value
```

➲ **Syntax 2**
Less than or equal to)

```
$EFD WHEN field <= value
```

➲ **Syntax 3**

Less than

```
$EFD WHEN field < value
```

➲ **Syntax 4**

Greater than or equal to

```
$EFD WHEN field >= value
```

➲ **Syntax 5**

Greater than

```
$EFD WHEN field > value
```

➲ **Syntax 6**

Not equal to

```
$EFD WHEN field != value
```

➲ **Syntax 7**

OTHER can only be used with the symbol "=". In this case, the field or fields after OTHER must be used only if the WHEN condition or conditions listed at the same level are not met. OTHER can be used before one record definition, and, within each record definition, once at each level. It is necessary to use a WHEN directive with OTHER in the eventuality that the data in a field doesn't meet the explicit conditions specified in the other WHEN directives. Otherwise, the results will be undefined.

```
$EFD WHEN field = OTHER
```

➲ **Syntax 8**

Value is an explicit data value used in quotes, and field is a previously defined COBOL field.

```
*(( EFD WHEN field(operator)value ))
```

➲ **Example**

Explicit data values in quotes ("") are permitted.

```
     05 AR-CODE-TYPE        PIC X.
$EFD WHEN AR-CODE-TYPE="S"
     05 SHIP-CODE-RECORD    PIC X(04).
```

```
$EFD WHEN AR-CODE-TYPE=”B”
     05 BACKORDER-CODE-RECORD REDEFINES SHIP-CODE-RECORD.
$EFD WHEN AR-CODE-TYPE=OTHER
     05 OBSOLETE-CODE-RECORD  REFEFINES SHIP-CODE-RECORD.
```

TABLENAME Option

The WHEN directive has the TABLENAME option to change the table name according to the value of the WHEN directive during runtime.
When using the TABLENAME option in a WHEN statement, be aware of the DCI_DEFAULT_RULES and filename_RULES DCI configuration variables.

➲ **Example 1**

A COBOL FD structure using the “When” directive with two table names.

```
FILE SECTION.
$EFD FILE=INV
 FD  INVOICE.
$EFD WHEN INV-TYPE = "A" TABLENAME=INV-TOP
 01  INV-RECORD-TOP.
     03 INV-KEY.
        05 INV-TYPE         PIC X.
        05 INV-NUMBER       PIC 9(5).
        05 INV-ID           PIC 999.
     03 INV-CUSTOMER        PIC X(30).
$EFD WHEN INV-TYPE = "B" TABLENAME=INV-DETAILS
 01  INV-RECORD-DETAILS.
     03 INV-KEY-D.
        05 INV-TYPE-D       PIC X.
        05 INV-NUMBER-D     PIC 9(5).
        05 INV-ID-B         PIC 999.
     03 INV-ARTICLES        PIC X(30).
     03 INV-QTA             PIC 9(5).
     03 INV-PRICE           PIC 9(17).
```

➲ **Example 2**

The DCI interface makes two tables named “inv-top” and “inv-details” based on the value of the inv-type fields in example 1. DCI checks the value of the inv-type field to know where to fill the record.

```
*MAKE TOP ROW
```

```
   MOVE "A" TO INV-TYPE
   MOVE 1 TO INV-NUMBER
   MOVE 0 TO INV-ID
   MOVE "acme company" TO INV-CUSTOMER
   WRITE INV-RECORD-TOP
*MAKE DETAIL ROWS
   MOVE "B" TO INV TYPE
   MOVE 1 TO INV-NUMBER
   MOVE 0 TO INV-ID
   MOVE "floppy disk" TO INV-ARTICLES
   MOVE 10 TO INV-QTA
   MOVE 123 TO INV-PRICE
   WRITE INV-RECORD-DETAILS
```

Running the preceding code, DCI fills the "TOP-ROW" record in the "INV-TOP" table and "DETAIL-ROW" in the "INV-DETAILS" table. When DCI reads the above record, it can use sequential reading, or use the key to access filled records. If you plan to use sequential reading through record types, you must set DCI_DEFAULT_RULES = POST or = COBOL. Alternately, if you plan to use sequential reading inside record types you must set DCI_DEFAULT_RULES=BEFORE or = DBMS.

There are advantages and disadvantages to using this rule. To have a 100% COBOL ANSI reading behavior, you should use the "POST" or "COBOL" method, but this method can degrade performance (more records are read and all involved tables are open at the same time).

If you use the "BEFORE" or "DBMS" method, the involved table is opened when the $WHEN condition matches at the read record level.

➲ **Example 3**

In other words, if you use the previous records, and code the following statements

```
      OPEN INPUT INVOICE.
*  to see the customer invoice
      READ INVOICE NEXT.
      DISPLAY "Customer: " INV-CUSTOMER
      DISPLAY "Invoice number: " INV-NUMBER
*  to see the invoice details
      READ INVOICE NEXT.
      DISPLAY INV-ARTICLES.
```

If the method is “POST” or “COBOL”, the “open input” opens both tables and “read next”, reads thru different tables.

➲ **Example 4**

The matched table is opened at the “start” statement level.
If the method is “BEFORE” or “DBMS” the code is changed as follows.

```
        open input invoice.
*     to see the customer invoice
        move "A" to  inv-type
        move 1   to  inv-number
        move 0   to  inv-id
        start invoice key is = inv-key.
        read invoice next
        display "Customer  " inv-customer
display "Invoice number "inv-number
*     to see the  invoice details
        move "B" to  inv-type
        move 1   to  inv-number
        move 0   to  inv-id
start invoice key is = inv-key.
        read invoice next
        display inv-articles
```

$EFD COMMENT DCI SPLIT

The DCI SPLIT directive is used to define one or more table splitting points starting where the DCI interface makes a new DBMS table.

➲ **Example 1**

A COBOL FD structure using DCI SPLIT directive.
In this example three DBMaker tables named INVOICE, INVOICE_A, and INVOICE_B are created with fields between the split points.

```
FILE SECTION.
FD  INVOICE.
01  INV-RECORD-TOP.
     03 INV-KEY.
        05 INV-TYPE         PIC X.
```

```
       05 INV-NUMBER       PIC 9(5).
       05 INV-ID           PIC 999.
    03 INV-CUSTOMER        PIC X(30).
$EFD DCI SPLIT
    03 INV-KEY-D.
       05 INV-TYPE-D       PIC X.
       05 INV-NUMBER-D     PIC 9(5).
       05 INV-ID-B         PIC 999.
$EFD DCI SPLIT
    03 INV-ARTICLES        PIC X(30).
    03 INV-QTA             PIC 9(5).
    03 INV-PRICE           PIC 9(17).
```

5 Compiler and Runtime Options

This section describes the configuration settings needed for isCOBOL to specify desired file systems.

5.1 Using isCOBOL Default File System

Existing files opened with a COBOL application are associated with their respective file systems as defined in isCOBOL's configuration file. When new files are created by a COBOL application, you must specify what file system to be used. The isCOBOL configuration file can be configured so that new files will use the file system of your choice.

For new files, the file.index setting tells isCOBOL which file system to use when no file system is specified. If no value is assigned to this variable, isCOBOL defaults to the JISAM file system. Additionally, with the file.index.filename setting a file system can be specified for a specific file. The name of the file should replace *filename* in the setting.

The following variables, found in the isCOBOL configuration file, allow the file system of your choice to be used.

➲ **Syntax 1**

```
file.index (*)
```

➲ **Syntax 2**

```
File.index (*)
```

5.2 Using DCI Default File System

To take advantage of DBMaker's reliability and features such as replication, backup and integrity constraints, we suggest using iscobol.file.index=dci to ensure isCOBOL does not use the JISAM file system. If no file system is specified, the JISAM file system is the default.

➲ **Syntax 1**

In this case, all new files will be DBMaker files, unless the new files have been designated as a different file system.

```
iscobol.file.index=dci
```

➲ **Syntax 2**

In this case, all new files will be JISAM files, unless otherwise specified.

```
iscobol.file.index=jisam
```

5.3 Using Multiple File Systems

isCOBOL uses file.index.FILENAME to associate new files to a particular file system. In this way, files that use a different file system from the default can still be used.

➲ **Example**

In this case, file1 and file2 will use DBMaker, while the other files will use the JISAM file system.

```
iscobol.file.index=jisam
iscobol.file.index.file1=dci
iscobol.file.index.file2=dci
```

5.4 Using the Environment Variable

To allow the file system to be setup during execution of a program, specify the following in the COBOL code. The (*) is only used for isCOBOL runtime. Please note, specifying a file system usually happens in the runtime configuration file and is not changed by a COBOL program.

NOTE *Refer to the isCOBOL User's Manual for detailed instructions on using the isCOBOL compiler and runtime.*

➲ **Syntax 1**

```
SET ENVIRONMENT "file.index.FILENAME" TO "filesystem" (*)
```

➲ **Syntax 2**

```
SET ENVIRONMENT "file.index" TO "filesystem" (*)
```

6 Configuration File Variables

This section lists the acceptable ranges of data for DCI, as well as tables specifying how COBOL data types are mapped to DBMaker data types. Configuration file variables are used to modify the standard behavior of DCI and are stored in a file called DCI_CONFIG.

6.1 Setting DCI_CONFIG Variables

It is possible to give a configuration file a different address by setting a value to an environment variable called DCI_CONFIG. The value assignable to this environment variable can be either a full pathname or simply the directory where the configuration file resides. In this case, DCI will look for a file called DCI_CONFIG stored in the directory specified in the environmental variable. If the file specified in the configuration variable doesn't exist, DCI doesn't display an error and assumes that no configuration variables have been assigned. This variable is set in the COBOL runtime configuration file.

➲ **Syntax 1**

In UNIX, DCI will look for the file DCI_CONFIG. This environment variable is used to establish the path and name of the DCI configuration file. Working with the Bourne shell, the following command can be used.

```
DCI_CONFIG=/usr/marc/config;export DCI_CONFIG
```

➲ **Syntax 2**

In DOS, DCI reads the configuration file called DCI_CONFIG in the directory c:\etc\test.

```
set DCI_CONFIG=c:\etc\test
```

➲ **Syntax 3**

In UNIX, DCI utilizes the file called "DCI" in the directory /home/test.

```
DCI_CONFIG=/home/test/dci; export DCI_CONFIG
```

DCI_CASE

File names in COBOL are case insensitive but table names are case sensitive. The DCI_CASE configuration variable determines how file names are translated into table names. Setting it to *lower* translates file names into table names with all lowercase characters. Setting it to *upper* translates file names into table names with all uppercase characters. Setting it to *ignore* means no file names are translated into table names with all lowercase or uppercase characters. The default

setting for DCI_CASE is *lower*. If your file names are DBCS words, set DCI_CASE to *ignore*.

➲ **Example:**

```
DCI_CASE IGNORE
```

DCI_COMMIT_COUNT

The DCI_COMMIT_COUNT configuration variable indicates the conditions under which a COMMIT WORK operation is issued. There are two possible values, 0 and <*n*>.

`DCI_COMMIT_COUNT = 0`
No automatic commit is done (default value).

`DCI_COMMIT_COUNT = <N>`
Under this condition DCI waits until the number of WRITE, REWRITE, AND DELETE operations are equal to the value <*n*> before issuing a COMMIT WORK statement. This rule applies only when opening the file in "output" or "exclusive" mode.

DCI_DATABASE

DCI_DATABASE is used to specify the name of the database established during the setup of DBMaker.

➲ **Example 1**
The following entry has to be included in the configuration file if the database used is named DBMaker_Test.

```
DCI_DATABASE DBMaker_Test
```

➲ **Example 2**
Sometimes, the database name is not known in advance, and for this reason it is necessary to set it dynamically during runtime. In cases like this, it is possible to write special code in the COBOL program similar to the one listed below. The

following code has to be executed before the first OPEN statement has been executed.

```
CALL "DCI_SETENV" USING "DCI_DATABASE" , "DBMaker_Test"
```

➲ **Example 3**

You can use DCI_DATABASE to connect to more than one database and dynamically switch between databases to access a table on a different database.

```
* connect to DBNAME to access idx-1-file
CALL "DCI_SETENV" USING "DCI_DATABASE" "DBNAME"
....
open output idx-1-file
....
* connect to DCIDB to access idx-2-file CALL "DCI_SETENV" USING "DCI_DATABASE"
"DCIDB"
....
open output idx-2-file

* to switch dynamically to DBNAME connection
CALL "DCI_SETENV" USING "DCI_DATABASE" "DBNAME"
close idx-1-file
...
```

DCI_DATE_CUTOFF

This variable uses a two-digit value and establishes the two-digit years that will be interpreted by the program as being in the 20th Century and the two-digit years that will be interpreted by the program as being in the 21st Century.
The default value for the DCI_DATE_CUTOFF is 20. In this case, 2000 will be added to the two-digit years that are smaller than "20" (or whatever value you give to this variable), and will therefore make them part of the 21st Century. 1900 will be added to the two-digit years that are larger than "20" (or whatever value you give to this variable), making them part of the 20th Century. A COBOL date like 99/10/10 will be translated into 1999/10/10. A COBOL date like 00/02/12 will be translated into 2000/02/12.

DCI_DEFAULT_RULES

Default management methods for the WHEN directive located in multi-definition files. The BEFORE statement indicates that a table be open when the $WHEN condition is matched. The POST statement indicates that all related tables be open when the COBOL application opens multi-definition files.
The possible values are:
POST or COBOL
BEFORE or DBMS

DCI_DEFAULT_TABLESPACE

This variable is used to set the default tablespace where new tables are to be stored. The tablespace specified must already exist in the database. If no tablespace is specified by this variable, then new tables will be created in the default user tablespace.

DCI_DUPLICATE_CONNECTION

DCI_DUPLICATE_CONNECTION is used to acquire a lock when opening the same table and locking the same record two times in the same COBOL application by opening the same table using the same COBOL process but with a different database connection.
The default value is off (0).

➲ **Example**
To allow a COBOL application to acquire a lock on a table using different database connections:

```
DCI_DUPLICATION_CONNECTION 1
```

DCI_GET_EDGE_DATES

DCI_SET_EDGE_DATE is used to specify the value to be displayed if a user enters a low/high value in the DATE field. When a user inputs low/high value for a DATE field in a COBOL program, for example, by entering 00010101/99991231, the date will be displayed using COBOL's low/high value

00000000/99999999. When this variable is used, the low/high value of the DATE field will be displayed using the database's low/high value 00010101/99991231. This rule is also applied when the DATE field is a part of a key. The default value is off.

➲ **Syntax:**

The following line must be added in the dci.cfg file:

```
DCI_GET_EDGE_DATES 1
```

DCI_INV_DATE

This variable is used to establish an invalid date (like 2000/02/31) in order to avoid problems that can occur when an incorrect date format has been written to the database. The default for this variable is 99991230 (December 30th, 9999).

DCI_LOGFILE

This variable specifies the pathname of the DCI log file used to write all of the I/O operations executed by the interface. The *dci_trace.log* log file, stored in the */tmp* directory is used for debugging purposes. The use of a log file slows down the performance of DCI. For this reason it is recommended not add this variable in the configuration file unless deemed absolutely necessary.

➲ **Example**

A sample log file entry in the Config.ini file:

```
DCI_LOGFILE /tmp/dci_trace.log
```

DCI_LOGIN

The variable DCI_LOGIN allows specification of a username for connecting to the database system. It has no default value. Therefore, if no username is specified, no login will be used.

The username specified by the DCI_LOGIN variable should have RESOURCE authority or higher with the database. Additionally, the user should have permission with existing data tables. New users may be created using the JDBA Tool or dmSQL.

NOTE *For more detailed information on creating new users, refer to the JDBA Tool User's Guide or the Database Administrator's Guide.*

➲ **Example**

A sample username entry, JOHNDOE, made in the Config.cfg file:

```
DCI_LOGIN JOHNDOE
```

DCI_JULIAN_BASE_DATE

This variable, used with the DATE directive, sets the base date for Julian date calculations. It utilizes the format YYYYMMDD. The default value for this variable is January 1st, 1 AD.

One usage of this variable could be a COBOL program that uses dates from 1850 onwards. These dates can be stored in a database by setting the DATE directive to $XFD DATE = JJJJJJ (the date field must have the same number of characters) and setting the DCI configuration variable DCI_JULIAN_BASE_DATE to 18500101.

DCI_LOGTRACE

This variable sets different levels for the trace log.

- 0: no trace
- 1: connect trace
- 2: record i/o trace
- 3: full trace
- 4: internal debug trace

DCI_MAPPING

This variable is used to associate particular filenames with a specific XFD in the DCI system. In this way, one XFD can be used in conjunction with multiple files. A "*pattern*" can be made up of any valid filename characters. It may include the wildcard "*" symbol, which stands for any number of characters, or the question mark "?", which stands for a single occurrence of any one character and can be used multiple times.

➲ **Syntax**

```
DCI_MAPPING [pattern = base-xfd-name]  ...
```

➲ **Example 1**

The pattern "CUST*1" and base-XFD-name "CUSTOMER" will cause filenames such as "CUST01", "CUST001", "CUST0001" and "CUST00001" to be associated with the XFD "customer.XFD".

```
DCI_MAPPING CUST*1=CUSTOMER ORD*=ORDER "ord cli*=ordcli"
```

➲ **Example 2**

The pattern "CUST????" and base-XFD-name "CUST" will cause filenames such as "CUSTOMER" and "CUST0001" to be associated with the XFD "cust.XFD".

```
DCI_MAPPING  CUST????=CUST
```

DCI_MAX_ATTRS_PER_TABLE

A DBMaker table may have up to 2000 columns. A COBOL file with more than 2000 fields will not be able to map all fields to columns in the table. DCI provides the DCI_MAX_ATTRS_PER_TABLE configuration variable to define the number of fields at which the table will split into two or more distinct tables. The resulting tables must have unique names, so DCI appends the table name with an underscore (_) character followed by letters in consecutive order (A, B, C, …).

➲ **Example 1**

A COBOL file has 300 fields, and the following statement:

```
SELECT FILENAME ASSIGN TO "customer"
```

➲ **Syntax**

The following line must be added in the **dci.cfg** file:

```
DCI_MAX_ATTRS_PER_TABLE = 100.
```

➲ **Example 2**

Three tables will be created with the following names:

```
customer_a
customer_b
customer_c
```

DCI_MAX_BUFFER_LENGTH

DCI_MAX_BUFFER_LENGTH is used to split a COBOL data record into multiple database tables, similar to the function performed by DCI_MAX_ATTRS_PER_ TABLE. However, the cutoff value used to determine where a table will be split is determined by buffer length. The default value is 32640.

➲ **Example 1**

A COBOL record size contains 9000 bytes of data, and the following statement:

```
SELECT FILENAME ASSIGN TO "customer"
```

➲ **Syntax:**

The following line must be added in the dci.cfg file:

```
DCI_MAX_BUFFER_LENGTH 3000
```

➲ **Example 2**

Three tables will be created with the following names:

```
customer a
customer b
customer_c
```

DCI_MAX_DATE

This variable is used to establish a high-value date in order to avoid problems in cases where invalid dates have been incorrectly written to the database. The default for this variable is 99991231 (December 31st, 9999).

DCI_MIN_DATE

This variable is used to establish a low-value, 0 or space date in order to avoid problems that can occur when invalid dates have been incorrectly written to the database. The default for this variable is 00010101 (January 1st, 1AD).

DCI_NULL_ON_ILLEGAL_DATE

DCI_NULL_ON_ILLEGAL_DATE determines how COBOL data that is considered illegal by the database will be converted before it is stored. The value

1 causes all illegal data (except key fields) to be converted to null before it is stored. The value 0 (default value) causes the following conversions to occur:

- Illegal LOW-VALUES: stored as the lowest possible value (0 or - 99999...) or DCI_MIN_DATE default value.
- Illegal HIGH-VALUES: stored as the highest possible value (99999...) or DCI_MAX_DATE default value.
- Illegal SPACES: stored as zero (or DCI_MIN_DATE, in the case of a date field).
- Illegal DATE values: stored as DCI_INV_DATE default value.
- Illegal TIME: stored as DCI_INV_DATE default value.
- Illegal data in key fields is always converted, regardless of the value of this configuration variable.

DCI_PASSWD

Once a username has been specified via the DCI_LOGIN variable, a database account is associated with it. A password needs to be designated to this database account. This can be done using the variable DCI_PASSWD.

➲ **Example 1**

If the password you want to designate to the database account is SUPERVISOR, the following must be specified in the configuration file:

```
DCI_PASSWD SUPERVISOR
```

➲ **Example 2**

A password can also be accepted from a user upon execution of the program. This allows for greater reliability. To do this, the DCI_PASSWD variable must be set according to the response:

```
ACCEPT RESPONSE NO-ECHO.
CALL "DCI_SETENV" USING  "DCI_PASSWD" , RESPONSE.
```

In this case, however, you should furnish a native API to call in order to read and write environment variables,

➲ **Syntax 1**

This statement can be used in the COBOL program to write or update the environment variable.

```
CALL "DCI_SETENV" USING "environment variable", value.
```

➲ **Syntax 2**

This statement can be used in the COBOL program to read the environment variable.

```
CALL "DCI_GETENV" USING "environment variable", value.
```

DCI_STORAGE_CONVENTION

This variable sets the COBOL storage convention. There are four value types currently supported by DBMaker.

DCI

Selects the IBM storage convention. It is compatible with IBM COBOL, as well as with several other COBOL versions including RM/COBOL-85. It is also compatible with the X/Open COBOL standard.

DCM

Selects the Micro Focus storage convention. It is compatible with Micro Focus COBOL when the Micro Focus "ASCII" sign-storage option is used (this is the Micro Focus default).

DCN

Causes a different numeric format to be used. The format is the same as the one used when the "-DCI" option is used, except that positive COMP-3 items use "x0B" as the positive sign value instead of "x0C". This option is compatible with NCR COBOL.

DCA

Selects the isCOBOL storage convention. It is the default setting. This convention is also compatible with data produced by RM/COBOL (not RM/COBOL-85) and previous versions of isCOBOL.

DCI_USEDIR_LEVEL

If this variable is set > 0, use the directory in addition to the name of the table.

➲ **Example 1**

The following line is equal to; /usr/test/01/clients 01clients

```
DCI_USEDIR_LEVEL 1
```

➲ **Example 2**

The following line is equal to; /usr/test/01/clients test01clients

```
DCI_USEDIR_LEVEL 2
```

➲ **Example 3**

The following line is equal to; /usr/test/01/clients usrtest01clients

```
DCI_USEDIR_LEVEL 3
```

DCI_USER_PATH

When DCI looks for a file or files, the variable DCI_USER_PATH allows for specification of a username, or names. The user argument can be a period (.) with regard to the files, or the name of a user on the system.

➲ **Syntax**

```
DCI_USER_PATH user1 [user2] [user3] .
```

The type of OPEN statement issued for a file will determine the results of this setting.

OPEN STATEMENT	DCI_USER_PATH	DCI SEARCH SEQUENCE	RESULT
OPEN INPUT or OPEN I/O	Yes	1-list of users in USER_PATH 2-the current user	The first valid file will be opened.
OPEN INPUT or OPEN I/O	No	The user associated with DCI_LOGIN.	The first file with a valid user/file- name will be opened.
OPEN OUTPUT	Yes or no	Doesn't search for a user.	A new table will be made for the name associated with DCI_LOGIN.

Figure 6-1 Types of OPEN Statements

DCI_EFDPATH

DCI_EFDPATH is used to specify the name of the directory where data dictionaries are stored. The default value is the current directory.

➲ **Example 1**

Include the following entry in the configuration file in order to store data dictionaries in the directory */usr/dbmaker/dictionaries*.

```
DCI_EFDPATH /usr/dbmaker/dictionaries
```

➲ **Example 2**

If it is necessary to specify more than one path, different directories have to be separated by spaces.

```
DCI_EFDPATH /usr/dbmaker/dictionaries /usr/dbmaker/dictionaries1
```

➲ **Example 3**

In a WIN-32 environment, "embedded spaces" can be specified using double-quotes.

```
DCI_EFDPATH c:\tmp\efdlist "c:\my folder with space\efdlist"
```

*<filename>*_RULES

Default management for a multi-definition file. The actual file name replaces <*filename*>.

➲ **Example**

All of the files will use the POST rule except for the CLIENT file when the following commands are used.

```
DCI_DEFAULT_RULES          POST
CLIENT_RULES               BEFORE
```

DCI TABLE CACHE Variables

By default, DCI pre-reads data into the client data buffer to reduce client/server network traffic. The default maximum pre-read buffer is the smaller of 8kb ÷ (record size) or 5 records.

It is possible that user's application will read a small table and only read a few records which are less than 8kb ÷(record size). For example, for a table with an average record size of 20 bytes and a total of 1,000 records, DBMaker will be able to read about 400 records (8kb ÷20) but the user's application may only read 4 or 5 records then call the START statement again. In this case, set the following variable to reduce the cache size and improve performance. Consider the application and data's behavior carefully when using these variables or it may increase network traffic and cause reductions in performance.

The following are the three DCI_CACHE variables to set in the DCI_CONFIG file:

- DCI_DEFAULT_CACHE_START – sets the first read records to cache for START or READ. The default is the maximum of 8kb ÷(record size) or 5 records.
- DCI_DEFAULT_CACHE_NEXT – sets the next read records after the first cached record for START or READ have been read or discarded. The default is the maximum of 8kb ÷(record size) or 5 records.
- DCI_DEFAULT_CACHE_PREV – sets the read records for caching the previous records after the first cache record for START or READ have been read or discarded.

The default is DCI_DEFAULT_CACHE_NEXT/2.
Setting these variables in the DCI_CONFIG will affect all the tables in the user's application.

➲ **Example:**

```
DCI_DEFAULT_CACHE_START        10
DCI_DEFAULT_CACHE_NEXT         10
DCI_DEFAULT_CACHE_PREV         5
```

DCI_TABLESPACE

This allows you to define in which tablespace to create a table. It also works with wildcards. It is important only when a table is first created. Once the table exists, DCI does not monitor the value of this variable.

➲ **Example 1:**
You want to create the customer table in tablespace tbs1:

```
DCI_TABLESPACE customer=tbs1
```

➲ **Example 2:**
You want to create all tables that begin with cust in tablespace tbs1.

```
DCI_TABLESPACE cust*=tbs1
```

DCI_AUTOMATIC_SCHEMA_ADJUST

This variable directs DCI to alter the table schema definition when the EFD differs from the table schema. This variable is incompatible with the split tables (those with a number of columns > 2000, and those who's record size is greater than 32 KB -exclude the BLOB field).
The possible values of this variable are:
0 Default, does nothing
1 Add the new fields to the table, and drop the ones who are not in the EFD
2 Add the new fields to the table, but do not drop the ones who are not in the EFD

DCI_INCLUDE

This variable permits the inclusion an additional DCI_CONFIG file. It works as the COBOL COPY statement, and allows you to define more complex configurations.

➲ **Example:**

```
DCI_INCLUDE /etc/generic_dci_config
```

DCI_IGNORE_MAX_BUFFER_LENGTH

This variable is used to ignore the setting of DCI_MAX_BUFFER_LENGTH value. It will not split the table when the record length > 32k. The default is off.

DCI_NULL_DATE

When DCI writes a date field with this value it will write NULL, and when DCI reads a date with a NULL value, it will return DCI_NULL_DATE to a COBOL program.

DCI_NULL_ON_MIN_DATE

With this variable set to 1 the following action occurs. When a COBOL program writes a value of 0 to a DATE field, the value is stored in the database as NULL. Likewise, when a NULL value is read from the database the COBOL FD will be 0.

DCI_VARCHAR

With this variable set to 1 the following action occurs: When a COBOL program creates a new table (trough OPEN OUTPUT verb) all fields that were created as CHAR will become VARCHAR.

7 DCI Functions

This section lists the DCI functions that could be called in the COBOL program. To enable these functions, the user must add these functions in the sub85.c and rebuild the DCI runtime.

7.1 Calling DCI functions

Add this to your COBOL program to call these DCI functions:

```
CALL "dci_function_name" USING variable [, variable, ...]
```

DCI_SETENV

This function writes or updates the environment variable.

➲ **Syntax:**

```
 CALL "DCI_SETENV" USING "environment variable", value
```

➲ **Example**

```
call "DCI_SETENV" using  "DCI_DATABASE" , "DBNAME"
```

DCI_GETENV

This function reads the environment variable.

➲ **Syntax**

```
CALL "DCI_GETENV" USING "environment variable", variable
```

➲ **Example**

```
CALL "DCI_GETENV" USING "DCI_DATABASE", ws_dci_database
```

DCI_GET_TABLE_NAME

This function gets the table name of the passed COBOL name (It's not always so immediate to know the effective table name, because there can be some manipulation in these cases: XFD WHEN ... TABLENAME.).

```
CALL "DCI_GET_TABLE_NAME" USING ws-filename, ws-dci-file-name
```

DCI_SET_WHERE_CONSTRAINT

This function specifies an additional WHERE condition for a succeeding START operation. The maxiumn length for the where_constraint string is 4096 bytes.

```
move "IDX_1_KEY = '2'" to dci_where_constraint.
```

```
CALL "DCI_SET_WHERE_CONSTRAINT" USING DCI_WHERE_CONSTRAINT.
move spaces to idx-1-key.
start IDX-1-FILE key is not less idx-1-key..
READ IDX-1-FILE NEXT
```

DCI_SET_TABLE_CACHE

This function dynamically changes the cache for tables set these variables before START or READ statements.

➲ **Example:**

```
…
WORKING-STORAGE SECTION.
       01 CACHE-START PIC 9(5) VALUE 10.
       01 CACHE-NEXT  PIC 9(5) VALUE 20.
       01 CACHE-PREV  PIC 9(5) VALUE 30.
…
PROCEDURE DIVISION.
      OPEN INPUT IDX-1-FILE
           MOVE SPACES TO IDX-1-KEY
           CALL "DCI SET TABLE CACHE" USING CACHE-START
                                            CACHE-NEXT
                                            CACHE-PREV
           START IDX-1-FILE KEY IS NOT LESS IDX-1-KEY.
           PERFORM VARYING IND FROM 1 BY 1 UNTIL IND = 10000
              READ IDX-1-FILE NEXT AT END EXIT PERFORM END-READ
              DISPLAY IND AT 0101
           END-PERFORM
           CLOSE IDX-1-FILE
```

DCI_BLOB_ERROR

This function gets the error after calling DCI_BLOB_GET or DCI_BLOB_PUT.

➲ **Example:**

```
      working-storage section.
       77   BLOB-ERROR-ERRNO pic S9(4) COMP-5.
       77   BLOB-ERROR-INT-ERRNO pic S9(4) COMP-5.
```

```
 PROCEDURE DIVISION.
CALL "DCI_BLOB_ERROR" USING BLOB-ERROR-ERRNO
                                  BLOB-ERROR-INT-ERRNO
       DISPLAY "BLOB-ERROR-ERRNO=" BLOB-ERROR-ERRNO.
       DISPLAY "BLOB-ERROR-INT-ERRNO=" BLOB-ERROR-INT-ERRNO.
```

DCI_BLOB_GET

This function enables users to more effectively use BLOB data in a COBOL program. By using the DCI_BLOB_GET command you can quickly and efficiently access BLOB data using COBOL. When using the DCI_BLOB_GET command you must follow the rules listed below:

- The user's table must have a BLOB (long varchar/long varbinary) data type
- Users cannot set the field with BLOB type in the COBOL FD
- Users can only use the DCI_BLOB_GET command after the READ, READ NEXT or READ PREVIOUS command

➲ **Example**

A user creates a table by:

```
CREATE TABLE BLOBTB (
         SB_CODCLI  char(8),
         SB_PROG    SERIAL,
         IL_BLOB    LONG VARBINARY,
         PRIMARY KEY ("sb_codcli")) LOCK MODE ROW NOCACHE;
```

The following gives a practical application of the use of the DCI_BLOB_GET in the COBOL program.

```
      identification division.
      program-id. blobtb.
      date-written.
      remarks.
      environment division.
      input-output section.
      file-control.
          SELECT BLOBTB ASSIGN TO RANDOM, "BLOBTB"
                      ORGANIZATION IS INDEXED
                      ACCESS IS DYNAMIC
                      FILE STATUS IS I-O-STATUS
```

```
                    RECORD KEY  IS SB-CODCLI.
     *================================================================*
      data division.
      file section.
      FD  BLOBTB.
      01  SB-RECORD.
          03  SB-CODCLI         PIC X(8).
          03  SB-PROG           PIC S9(9) COMP-5.
     *================================================================*
      working-storage section.
      77   I-O-STATUS pic xx.
      77   BLOB-ERROR-ERRNO pic S9(4) COMP-5.
77   BLOB-ERROR-INT-ERRNO pic S9(4) COMP-5.

procedure division.
main.
             open i-o blobtb
             initialize sb-record.
             READ blobtb next.
             CALL "DCI_BLOB_GET" USING "il_blob" "laecopy.bmp" 1.
             CALL "DCI_BLOB_ERROR" USIG BLOB-ERROR-ERRNO
                                        BLOB-ERROR-INT-ERRNO
             DISPLAY "BLOB-ERROR-ERRNO=" BLOB-ERROR-ERRNO.
             DISPLAY "BLOB-ERROR-INT-ERRNO=" BLOB-ERROR-INT-ERRNO.
             DISPLAY "SB-CODCLI=" SB-CODCLI.
             DISPLAY "SB-PROG=" SB-PROG.
             close blobtb.
             ACCEPT OMITTED.
             stop run.
```

DCI_BLOB_PUT

This function enables users to more effectively use BLOB data in a COBOL program. Using the DCI_BLOB_PUT command you can insert data into a BLOB. When using the DCI_BLOB_PUT command you must follow the rules listed below:

- The user's table must have a BLOB (long varchar/long varbinary) data type.
- Users cannot set the field with BLOB type in the COBOL FD.

- Users can only call the DCI_BLOB_PUT command before a WRITE or REWRITE command.
  - If user does not call DCI_BLOB_PUT before a WRITE statement, the default value will be inserted in the blob column.
  - If user does not call DCI_BLOB_PUT before REWRITE statement, the blob column will not be updated.

➲ **Example:**

The following gives a practical application of the use of the DCI_BLOB_PUT in the COBOL program.

First the user creates a table:

```
CREATE TABLE BLOBTB (
          SB_CODCLI  char(8),
          SB_PROG    SERIAL,
          IL_BLOB    LONG VARBINARY,
          PRIMARY KEY ("sb_codcli")) LOCK MODE ROW NOCACHE;
```

Once the table is created the user continues with the following.

```
identification division.
       program-id. blobtb.
       date-written.
       remarks.
       environment division.
       input-output section.
       file-control.
           SELECT BLOBTB ASSIGN TO RANDOM, "BLOBTB"
                       ORGANIZATION IS INDEXED
                       ACCESS IS DYNAMIC
                       FILE STATUS IS I-O-STATUS
                       RECORD KEY  IS SB-CODCLI.
      *================================================================*
       data division.
       file section.
       FD  BLOBTB.
       01  SB-RECORD.
           03  SB-CODCLI         PIC X(8).
           03  SB-PROG           PIC S9(9) COMP-5.
      *================================================================*
       working-storage section.
```

```
      77   I-O-STATUS pic xx.
      77   BLOB-ERROR-ERRNO pic S9(4) COMP-5.
77   BLOB-ERROR-INT-ERRNO pic S9(4) COMP-5.

procedure division.
main.
             open i-o blobtb
             move "AAAAAAAA" TO SB-CODCLI.
             move 0 TO SB-PROG.
             CALL "DCI_BLOB_PUT" USING "il_blob" "laetitia.bmp".
             WRITE SB-RECORD.
             close blobtb.
             ACCEPT OMITTED.
             stop run.
```

DCI_GET_TABLE_SERIAL_VALUE

This function gets the serial value after a WRITE statement.

➲ **Example:**

```
      FD  SERIALTB.
       01  SB-RECORD.
           03  SB-CODCLI         PIC X(8).
      $XFD COMMENT dci serial
           03  SB-PROG           PIC S9(9) COMP-5.
       working-storage section.
       77   I-O-STATUS pic xx.
       77   SERIAL-NUM pic S9(9) COMP-5.

       procedure division.
       main.
              open i-o serialtb
              move "AAAAAAAA" TO SB-CODCLI.
              move 0 TO SB-PROG.
              WRITE SB-RECORD.
              CALL "DCI_GET_TABLE_SERIAL_VALUE" USING SERIAL-NUM.
              DISPLAY "SERIAL-NUM=" SERIAL-NUM.
```

8 COBOL Conversions

Transactions are enforced in DCI during conversions. All I/O operations are done using transactions. DCI sets AUTOCOMMIT off and manages DBMaker transactions to make record changes for users available. DCI fully supports COBOL transaction statements like START TRANSACTION, COMMIT/ROLLBACK TRANSACTION.
DCI doesn't support record encryption, record compression, or the alternate collating sequence. If these options are included in code, they will be disregarded. DCI also doesn't support the "P" PICture edit function in the EFD data definition and all file names are converted to lowercase.

DBMaker Database Settings	Range Limit
Indexed key size.	4000
Number of columns per key.	32
Length for a CHAR field.	32640
Simultaneous RDBMS connections.	4800
Character for column names.	128
Database tables simultaneously open by a single process.	256

Figure 8-1 DBMaker Database Settings Range Limits table

8.1 Using Special Directives

DBMaker can use the same sort or retrieval sequence as the Vision file system, but it requires that a BINARY directive be placed before each key field containing signed numeric data. High and low values can create complications in key fields.

The DBMaker OID, VARCHAR(size), and FILE data types are not currently supported with special directives.

DBMaker Data Type	Directive
DATE	Using EFD DATE
TIME	Using EFD DATE
TIMESTAMP	Using EFD DATE
LONGVARCHAR	Using EFD VAR-LENGH
LONGVARBINARY	Using EFD VAR-LENGH*
BINARY	Using EFD BINARY
SERIAL	Using EFD COMMENT DCI SERIAL

Figure 8-2 DBMaker Data Types Supported using Special Directives

8.2 Mapping COBOL Data Types

DCI establishes what it considers to be the best match for COBOL data types in the creation of all columns in a DBMaker database table. Any data the COBOL date type can contain can also be contained in the database column. The EFD directives that have been specified will be checked first.

COBOL	DBMAKER	COBOL	DBMAKER
9(1-4)	SMALLINT	9(5-9) comp-4	INTEGER
9(5-9)	INTEGER	9(10-18) comp-4	DECIMAL(10-18)
9(10-18)	DECIMAL(10-18)	9(1-4) comp-5	SMALLINT
s9(1-4)	SMALLINT	9(5-10) comp-5	DECIMAL(10)
s9(5-9)	INTEGER	s9(1-4) comp-5	SMALLINT
s9(10-18	DECIMAL(10-18)	s9(5-10) comp-5	DECIMAL(10)
9(n) comp-1 n (1-17)	INTEGER	9(1-4) comp-6	SMALLINT
s9(n) comp-1 n (1-17)	INTEGER	9(5-9) comp-6	INTEGER
9(1-4) comp-2	SMALLINT	9(10-18) comp-6	DECIMAL(10-18)
9(5-9) comp-2	INTEGER	s9(1-4) comp-6	SMALLINT
9(10-18) comp-2	DECIMAL(10-18)	s9(5-9) comp-6	INTEGER
s9(1-4) comp-2	SMALLINT	s9(10-18) comp-6	DECIMAL(10-18)
s9(5-9) comp-2	INTEGER	signed-short	SMALLINT
s9(10-18) comp-2	DECIMAL(10-18)	unsigned-short	SMALLINT
9(1-4) comp-3	SMALLINT	signed-int	CHAR(10)
9(5-9) comp-3	INTEGER	unsigned-int	CHAR(10)
9(10-18) comp-3	DECIMAL(10-18)	signed-long	CHAR(18)
s9(1-4) comp-3	SMALLINT	unsigned-long	CHAR(18)
s9(5-9) comp-3	INTEGER	float	FLOAT
s9(10-18) comp-3	DECIMAL(10-18)	Double	DOUBLE
9(1-4) comp-4	SMALLINT	PIC x(n)	CHAR(n) n 1-max column length

Figure 8-3 COBOL to DBMaker Data Type Conversion Chart

8.3 Mapping DBMaker Data Types

DCI reads data from the database by doing a COBOL-like MOVE from the native data types to the COBOL data types (most of which have a CHAR representation so you can display them by using dmSQL).

It is not necessary to worry about exactly matching the database data types to COBOL data types. PIC X(nn) can be used for each column with regards to database types having a CHAR representation. PIC 9(9) is a closer COBOL match for databases that have INTEGER types. The more you know about a database type, the more flexible you can be in finding a matching COBOL type. For example, if a column in a DBMaker database only contains values between zero and 99 (0-99), PIC 99 would be a sufficient COBOL date match.

Choosing COMP-types can be left to the discretion of the programmer since it has little effect on the COBOL data used. BINARY data types will usually be re-written without change, because they are foreign to COBOL. However, a closer analysis of BINARY columns might allow you to find a different solution. The DECIMAL, NUMERIC, DATE and TIMESTAMP types have no exact COBOL matches. They are returned from the database in character form, so the best COBOL data type equivalent would be USAGE DISPLAY.

The following table illustrates the best matches for database data types and COBOL data types:

DBMAKER	COBOL	DBMAKER	COBOL
SMALLINT	9(1-4)	INTEGER	9(5-9) comp-4
INTEGER	9(5-9)	DECIMAL(10-18)	9(10-18) comp-4
DECIMAL(10-18)	9(10-18)	SMALLINT	9(1-4) comp-5
SMALLINT	s9(1-4)	DECIMAL(10)	9(5-10) comp-5
INTEGER	s9(5-9)	SMALLINT	s9(1-4) comp-5
DECIMAL(10-18)	s9(10-18	DECIMAL(10)	s9(5-10) comp-5
INTEGER	9(n) comp-1 n (1-17)	SMALLINT	9(1-4) comp-6
INTEGER	s9(n) comp-1 n (1-17)	INTEGER	9(5-9) comp-6
SMALLINT	9(1-4) comp-2	DECIMAL(10-18)	9(10-18) comp-6
INTEGER	9(5-9) comp-2	SMALLINT	s9(1-4) comp-6
DECIMAL(10-18)	9(10-18) comp-2	INTEGER	s9(5-9) comp-6
SMALLINT	s9(1-4) comp-2	DECIMAL(10-18)	s9(10-18) comp-6
INTEGER	s9(5-9) comp-2	SMALLINT	signed-short
DECIMAL(10-18)	s9(10-18) comp-2	SMALLINT	unsigned-short
SMALLINT	9(1-4) comp-3	CHAR(10)	signed-int
INTEGER	9(5-9) comp-3	CHAR(10)	unsigned-int
DECIMAL(10-18)	9(10-18) comp-3	CHAR(18)	signed-long
SMALLINT	s9(1-4) comp-3	CHAR(18)	unsigned-long
INTEGER	s9(5-9) comp-3	FLOAT	float
DECIMAL(10-18)	s9(10-18) comp-3	DOUBLE	Double
SMALLINT	9(1-4) comp-4	CHAR(n) n 1-max column length	PIC x(n)

Figure 8-4 DBMaker to COBOL Data Type Conversion Chart

8.4 Troubleshooting Runtime Errors

Runtime errors have the format “9D, xx”, where “9D” indicates a file system error (reported in the FILE STATUS variable) and “xx” indicates a secondary error code.

Error	Definition	Interpretation	Solution
9D,01	There is a read error on the dictionary file.	An error occurred while reading the EFD file. The EFD file is corrupt.	Recompile with -EFD to re-create the dictionary file.
9D,02	There is a corrupt dictionary file. The dictionary file cannot be read.	The dictionary file for a COBOL file is corrupt.	Recompile with -EFD to re-create the dictionary file.
9D,03	A dictionary file (.xml) has not been found.	The dictionary file for a COBOL file cannot be found.	Specify a correct directory in the DCI_EFDPATH configuration file variable (it may be necessary to recompile using –EFD).
9D,04	There are too many fields in the key.	There are more than16 fields in a key.	Check key definitions, re-structure illegal key, recompile with –EFD.
9D,12	There is an unexpected error on a DBMaker library function.	A DBMaker library function returned an unexpected error.	
9D,13	The size of the “xxx” variable is illegal.	An elementary data item in an FD is larger than 255 bytes.	
9D,13	The type of data for the “xxx” variable is illegal.	There is no DBMaker type that matches the data type used.	
9D,14	There is more than one table with the same name.	More than one table had the same name when they were listed.	

Figure 8-5 DCI Secondary Errors Chart

8.5 Troubleshooting Native SQL Errors

Some native SQL errors may be generated by a database while using DCI for DBMaker. The exact error number and wording may vary from database to database.

NUMBER	DEFINITION	INTERPRETATION	SOLUTION
9D, 6523,6018	Invalid column name or reserved word.	A column was named using a word that has been reserved for the database.	Compare a file trace of CREATE TABLE to the list of database reserved words. Apply the NAME directive to the FD field of an invalid column and recompile to create a new EFD file.
9D, 1310	Journal full, command rolled back to internal savepoint		Add "start transaction/commit/roll back" code in the COBOL program. Or set DCI_COMMIT_COUNT in the DCI configuration file.
9D, 5503	invalid key name	The table does not have the index	Create the index with correct index name and columns
9D, 5504	Cannot use host variable		User cannot use host variable in the sql command
9D, 5508	do not have INSERT/UPDATE/DELETE privilege	User cannot open I-O for table that they does not have the insert/delete/update privilege	OPEN INPUT with that table
9D, 5512	cannot issue select query		User cannot issue select statement in the sql command
9D,5513	client-server version mismatch when dci connect	User’s DCI runtime is newer than the dmserver	User should upgrade their dmserver before running new DCI runtime

9D, 5514	invalid column number	COBOL FD column number > table column number	User need to check the FD column number and table column number
9D, 5515	invalid EFD column name or data type and length does not match	COBOL FD column name or column type does not match with table definition	Compare the FD and table definition. Fix this problem by either change the COBOL FD or alter table.
9D, 5518	DCI blob data is null	When user get blob from a column and the data is null	
9D, 5519	DCI blob file does not exist		User should ensure the blob file has existed.

Figure 8-6 Native SQL Errors Char

8.6 Converting COBOL Files

For migrating COBOL files to DBMaker, please reference isCOBOL User's Guide -> Utilities -> ISMIGRATE.

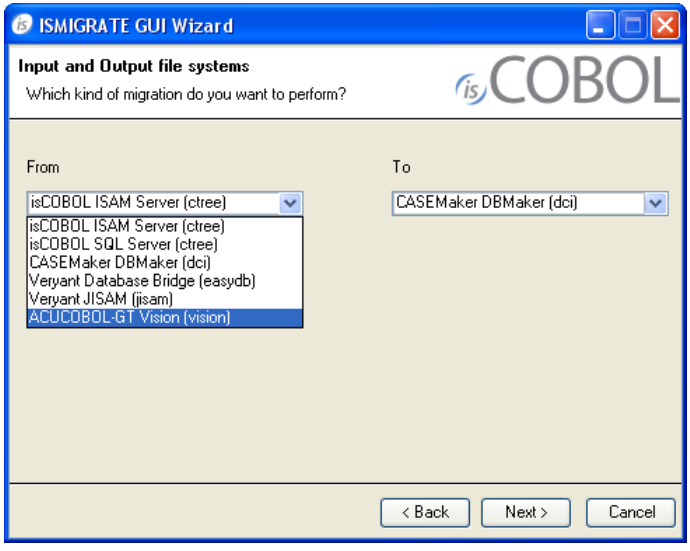
The conversion of disk indexed files to DBMaker is performed through the utility ISMIGRATE, installed along with isCOBOL.

Launch the command is migrated from the isCOBOL bin directory to start the utility. On Windows you also find a link in the start menu, under the isCOBOL program group.

➲ **The utility consists in a graphical wizard procedure made of the following steps.**

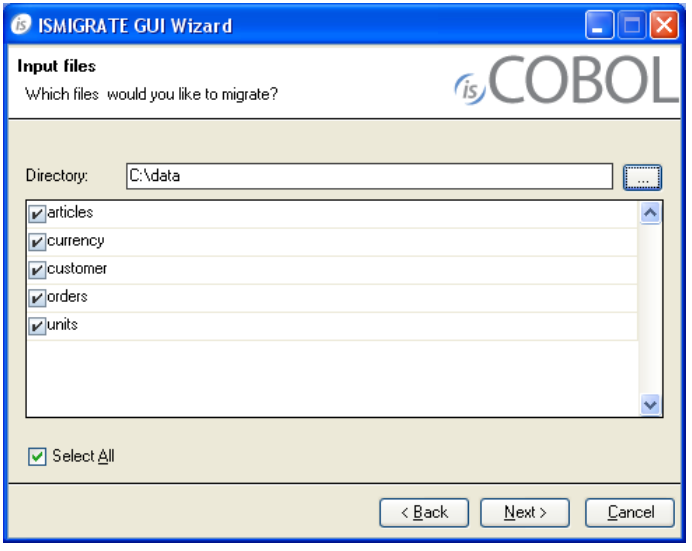
1. Open the **Welcome to the ISMIGRATE Graphical Wizard** window. The wizard procedure will guide you through the setup of the data migration process.

2. Click the **Next** button. The **input and output file systems** window appear. You can select the source and destination file systems. Choose between Jisam, Vision or other file systems supported by isCOBOL as input. Choose DCI as output.

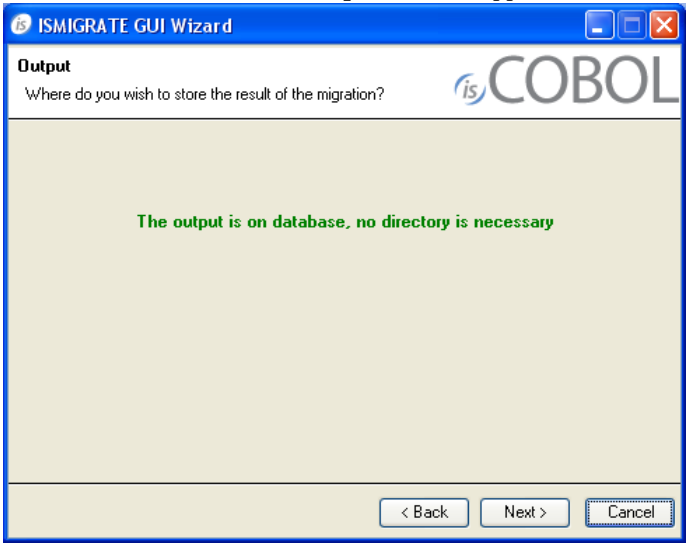


NOTE *ISMIGRATE uses file system libraries in order to manage files and tables. If these libraries are not found in PATH and CLASSPATH, a red warning message appears below the selected file system.*

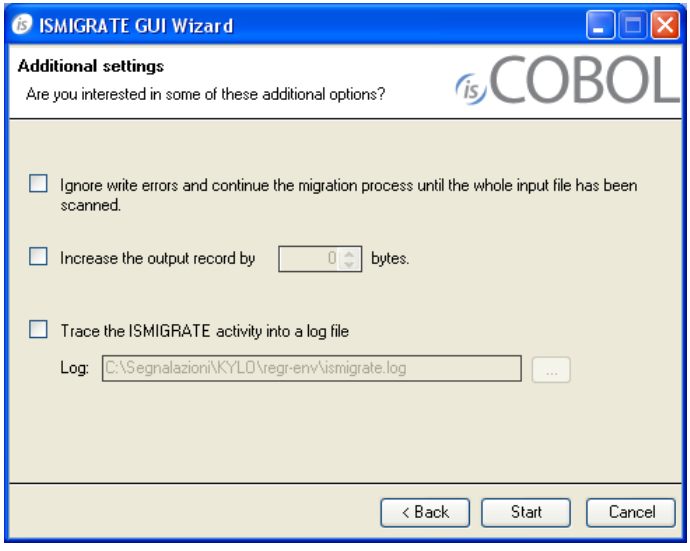
3. Click the **Next** button if you selected the properly source and destination file systems. The **Input files** window appears. Choose the directory with the indexed files you want to migrate, and check the desired files in the list.



4. Click the **Next** button. The **Output** window appears.

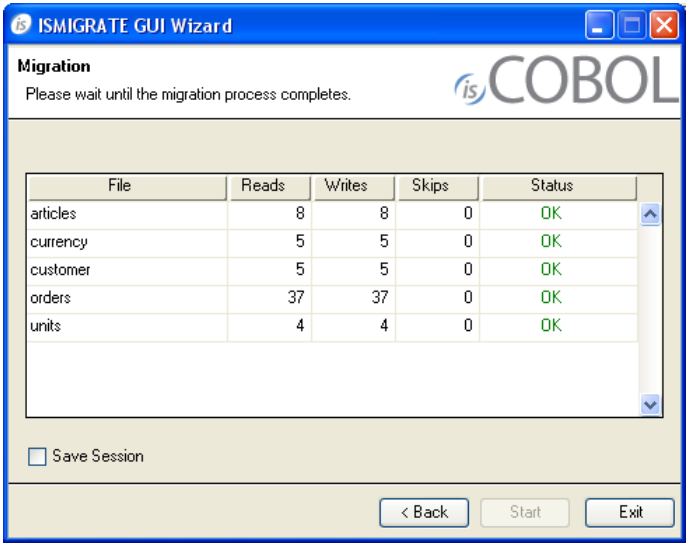


5. Click the **Next** button to proceed. The **Additional settings** window appears. Users can set some additional optional configurations.



NOTE *ISMIGRATE doesn't allow configure DCI. It assumes that you have a valid DCI_CONFIG in the environment.*

6. Click the **Next** button, the **Migration** window appears. If everything is fine, click **Exit** to leave. Otherwise, leave the mouse pointer on the word "Failed" to obtain a tool-tip with error description. Please refer to the isCOBOL documentation for more information.



Glossary

API
Application Programming Interface: The API is an interface between an application and an operating system.

Binary Large Object (BLOB)
A large block of data stored in the database that is not stored as distinct records in a table. A BLOB cannot be accessed through the database in the same way as ordinary records. The database can only access the name and location of a BLOB; typically, another application is used to read the data.

Buffer
A buffer is an internal memory space (zone) where data is temporarily stored during input or output operations.

Client
A computer that can access and manipulate data that is stored on a central server computer.

Column
A set of data in a database table defined as multiple records consisting of the same data type.

Data dictionaries
Also known as extended file descriptors; they serve as maps (links) between database schema and the file descriptors in a COBOL application.

Directive
An optional comment placed in the COBOL code that sets the proceeding field or fields to a data type other than the default DCI setting.

Field
Part of a COBOL file descriptor roughly corresponding to a database column. It is a discrete data item contained in a COBOL record.

File Descriptor
A file descriptor is an integer that identifies a file that is operated on by a process. Operations that read, write, or close a file use the file descriptor as an input parameter.

Indexed file
File containing a list of keys that uniquely identify all records.

Key
A unique value used to identify a record in a database. (See ***Primary Key*** for more details).

Primary key
A primary key consists of a column of unique (or key) values, which can be used to identify individual records contained in a table.

Query
In DBMaker, SQL commands used to execute data query requests made by a user to obtain specific information.

Record
In COBOL, a group of related fields defined in the Data Division. In DBMaker, a record is also referred to as a row, and defines a set of related data items in table columns.

Relational Database
A relational database is a database system where internal database tables on different databases may be related to one another by the use of keys or unique indexes.

Schema
A database table's structure as defined by its columns. Data type, size, number of columns, keys, and constraints all define a table's schema.

Server
A server is a central computer that stores and handles network configuration files, which also can consist of a database management system to store data (database) and distribute data to clients via a network connection.

SQL
Structured Query Language is the language DBMaker and other ODBC compliant programs use to access and manipulate data.

Table
A logical storage unit in a database that consists of columns and rows used to store records.

EFD file
An acronym for extended file descriptor or data dictionary. It also forms the file extension for the data dictionary.

Index