Oracle Warehouse Builder 10*g* Release 2

Integrating COBOL Based Legacy Data

March 2007

ORACLE

Note:

This document is for informational purposes. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

Oracle Warehouse Builder 10*g* Release 2

Integrating COBOL File Legacy Data

INTRODUCTION

Many large corporations struggle with the fact that most of their corporate operational data resides in legacy environments. Much of this data is in the form of files that have been created with COBOL programs. These programs may have been written decades ago and the developers may have long since left the organization. Unfortunately, there is often no independent human-oriented documentation for these files. To understand the format of the data files these programs create, the only tactic may be to interpret the COBOL that was used to generate them.

Complicating the task at hand is the fact that COBOL files often contain complex structures that do not readily map into the relational reality of your data warehouse or data mart. As a result, you must not only come to an understanding of the data that resides in your legacy files, but you must also develop a plan for moving that data into a relational model.

Once you understand your data, you can easily use Oracle's Warehouse Builder to integrate this data with other data. You can capitalize on all of the power in Warehouse Builder to load, manage, cleanse and transform this data in the same ways you can data from relational sources.

The paper gives an overview of the flat file support that is available with Warehouse Builder 10*g* Release 2. The paper further examines COBOL data structures and shows a strategy for migrating these structures into a relational model. The paper then demonstrates how to import these structures into Warehouse Builder. The same principles discussed in this paper can be applied for any binary format that has inherent structure.

EXECUTIVE OVERVIEW

Warehouse Builder 10*g*R2 provides data access and extraction capabilities for many files, even those that were created with COBOL programs. These files may have been created in a foreign environment where the data may be in EBCDIC format. The data may be organized in structures that are not supported in relational environments. Additionally, the data may be written in an internal format that needs to be converted to a format supported by the relational database. Warehouse Builder provides the capabilities to extract the data in these files, but goes well beyond the basic extraction by also providing access to the data using SQL based extraction and transformation as if the data was actually stored in relational tables.

Metadata Benefits

Warehouse Builder provides wizards that make it easy to define the physical characteristics of files. The Sample Wizard provides an active interface for defining external format data files. As the file is defined, a display of the data is updated to show the impact of the definition. An external table can also be defined for the file which provides transparent SQL access to the data in the file. The full range of Warehouse Builder metadata services are available for files and external tables, including, but not limited to, lineage and impact reporting.

Data Access and Conversion

Once a file is defined it can be loaded into one or more tables in the warehouse using SQL*Loader. The data is automatically converted from the format defined in the file to the format specified for the table. The data can be filtered and transformed before it is loaded into the database.

You can also choose to access the data in the file as an external table. The external table feature provides SQL select support for the data in the file as if it were a relational table stored in the database.

Cleanse, Transform, Manipulate

Once you have defined the metadata for your files, you can use all the facilities in Warehouse Builder to integrate this data using the SQL based extraction and transformation mechanisms.

UNDERSTANDING COBOL DATA STRUCTURES

COBOL programmers create files by defining the physical files and the logical records that will be used to build those files. The records may be defined within the COBOL program itself, but are usually defined in separate files, called ***copybooks***. These copybooks specify the layout and format of the user data, but do not specify the physical characteristics of the file itself. The physical characteristics of the file identify how the file is organized and accessed. For example, whether records are terminated (eg. with CR or CR/LF) is not part of the user data definition and is therefore not included in the record definition.

Data Hierarchy

COBOL records are defined as a set of data elements and groups. A data element is an atomic data item. A group is a container for data elements. Each item defined in a COBOL record is called a field whether it is a group or an elementary item. Each field definition contains a level number which shows the ***hierarchy*** of the data within the record. Groups can contain groups or elementary items. Items contained in a group are called subordinate elements. Field definitions for elementary items contain complete metadata for the item primarily specified in picture and usage clauses. A group inherits characteristics from its subordinate elements and does not generally contain metadata specification details.

Example 1

```
01  EMPLOYEE-RECORD.
        05 EMP-ID PIC 9(6).
        05 EMP-REGION PIC 9.
        05 EMP-DEPT PIC 999.
        05 EMP-HIRE-DATE.
           10 EMP-HIRE-DATE-MM PIC 99.
           10 EMP-HIRE-DATE-DD PIC 99.
           10 EMP-HIRE-DATE-YYYY PIC 9999.
        05 EMP-SALARY PIC 9(9).
        05 EMP-NAME PIC X(15).
```

The above example shows the definition of the EMPLOYEE-RECORD. It contains 6 fields defined at the 05 Level. All of the fields except EMP-HIRE-DATE are elementary items. The elementary items contain picture clauses that define their data-characteristics. EMP-HIRE-DATE is a group field with subordinate elements at the 10 level. Each 10 level field is an elementary item that contains a picture clause defining its data characteristics. The date can be referenced as a whole by using the EMP-HIRE-DATE group field. This field includes the month, day and year elements. Each subordinate field can also be referenced allowing access to just the year for example.

Data Types

The USAGE and PICTURE clauses are used to define the format and characteristics of data elements. If a USAGE clause is not specified, the data is in DISPLAY format, either external numeric or external character. When considered

together the picture and usage identify the data type. Detailed information concerning COBOL data types and how they are mapped to relational data types are described in the section Data Type Mappings.

Arrays and Varying Arrays

COBOL provides support for both arrays and varying arrays. These complex structures are identified by the use of an OCCURS CLAUSE. Varying arrays are defined with the additional specification of a DEPENDING ON CLAUSE. For arrays, the OCCURS CLAUSE indicates the number of elements in the array. For varying arrays, the occurs specification includes a range of elements FROM n TO n and the DEPENDING ON CLAUSE identifies a field that contains the actual number of elements in the array. An array or a varying array can be defined on an elementary element or on a group.

Example 2: Arrays defined on elementary fields

```
01  EMPLOYEE-RECORD.
        05 EMP-ID PIC 9(6).
        05 EMP-REGION PIC 9.
        05 EMP-DEPT PIC 999.
        05 EMP-HIRE-DATE.
           10 EMP-HIRE-DATE-MM PIC 99.
           10 EMP-HIRE-DATE-DD PIC 99.
           10 EMP-HIRE-DATE-YYYY PIC 9999.
        05 EMP-SALARY PIC 9(9).
        05 EMP-NAME PIC X(15).
        05 EMP-SKILL-LEVEL PIC 99 OCCURS 4 TIMES.
        05 EMP-SKILL-ID PIC 9(4) OCCURS 4 TIMES.
```

Two independent arrays are defined, one on EMP-SKILL_LEVEL and one on EMP-SKILL_ID. In this example, the record contains four occurrences of EMP-SKILL-LEVEL, followed by four occurrences of EMP-SKILL-ID. In the file, each record will be constructed as: EMP_SKILL_LEVEL, EMP_SKILL_LEVEL, EMP_SKILL_LEVEL, EMP_SKILL_LEVEL, EMP_SKILL_ID, EMP_SKILL_ID, EMP_SKILL_ID, EMP_SKILL_ID.

Example 3: Varying Arrays defined on elementary fields

```
01  EMPLOYEE-RECORD.
        05 EMP-ID PIC 9(6).
        05 EMP-REGION PIC 9.
        05 EMP-DEPT PIC 999.
        05 EMP-HIRE-DATE.
           10 EMP-HIRE-DATE-MM PIC 99.
           10 EMP-HIRE-DATE-DD PIC 99.
           10 EMP-HIRE-DATE-YYYY PIC 9999.
        05 EMP-SALARY PIC 9(9).
        05 EMP-NAME PIC X(15).
        05 EMP-SKILL-COUNT PIC 99.
        05 EMP-SKILL-LEVEL PIC 99 OCCURS 1 TO 4 TIMES
                DEPENDING ON EMP-SKILL-COUNT..
        05 EMP-SKILL-ID PIC 9(4) OCCURS 1 to 4 TIMES
                DEPENDING ON EMP-SKILL-COUNT.
```

Two independent arrays are defined, one on EMP-SKILL_LEVEL and one on EMP-SKILL_ID. In this example, the value in EMP-SKILL-COUNT identifies how many occurrences there are in both arrays. In the file, a record with EMP_SKILL_COUNT equal to one is constructed as: EMP_SKILL_LEVEL, EMP_SKILL_ID. If there is a two in EMP-SKILL-COUNT, two occurrences of EMP-SKILL-LEVEL are followed by two occurrences of EMP-SKILL-ID. In the file, a record with EMP_SKILL_COUNT equal to two is constructed as: EMP_SKILL_LEVEL, EMP_SKILL_LEVEL, EMP_SKILL_ID, EMP_SKILL_ID.

Example 4: Arrays defined on a group field

```
01  EMPLOYEE-RECORD.
        05 EMP-ID PIC 9(6).
        05 EMP-REGION PIC 9.
        05 EMP-DEPT PIC 999.
        05 EMP-HIRE-DATE.
           10 EMP-HIRE-DATE-MM PIC 99.
           10 EMP-HIRE-DATE-DD PIC 99.
           10 EMP-HIRE-DATE-YYYY PIC 9999.
        05 EMP-SALARY PIC 9(9).
        05 EMP-NAME PIC X(15).
        05 EMP-SKILLS OCCURS 4 TIMES.
        10 EMP-SKILL-LEVEL PIC 99.
        10 EMP-SKILL-ID PIC 9(4).
```

In this example one array is defined with four elements. Each element contains one occurrence of each field EMP_SKILL_LEVEL and EMP_SKILL_ID. In the file, each record will be constructed as: EMP_SKILL_LEVEL, EMP_SKILL_ID, EMP_SKILL_LEVEL, EMP_SKILL_ID, EMP_SKILL_LEVEL, EMP_SKILL_ID, EMP_SKILL_LEVEL, EMP_SKILL_ID.

Example 5: Varying Array defined on a group

```
01  EMPLOYEE-RECORD.
        05 EMP-ID PIC 9(6).
        05 EMP-REGION PIC 9.
        05 EMP-DEPT PIC 999.
        05 EMP-HIRE-DATE.
           10 EMP-HIRE-DATE-MM PIC 99.
           10 EMP-HIRE-DATE-DD PIC 99.
           10 EMP-HIRE-DATE-YYYY PIC 9999.
        05 EMP-SALARY PIC 9(9).
        05 EMP-NAME PIC X(15).
        05 EMP-SKILL-COUNT PIC 99.
        05 EMP-SKILLS OCCURS 4 TIMES DEPENDING ON EMP-SKILL-COUNT.
           10 EMP-SKILL-LEVEL PIC 99.
           10 EMP-SKILL-ID PIC 9(4).
```

In this example, one array is defined with up to four elements. The value in EMP-SKILL-COUNT identifies how many occurrences there are in the array. Therefore, if there is a one in EMP-SKILL-COUNT, there will be one occurrence of EMP_SKILLS. In the file, a record with EMP_SKILL_COUNT equal to one is constructed as: EMP_SKILL_LEVEL, EMP_SKILL_ID. If there is a two in EMP-SKILL-COUNT, there will be two occurrences of EMP-SKILLS. In the file,

a record with EMP_SKILL_COUNT equal to two is constructed as: EMP_SKILL_LEVEL, EMP_SKILL_ID, EMP_SKILL_LEVEL, EMP_SKILL_ID.

Multiple Definitions

In COBOL, data in a record may have more than one definition. Any one of these definitions can be used to access the data. There are three methods for getting multiple definitions: defining multiple records, defining groups and using redefinition.

Multiple Records

COBOL generated files can contain more than one record type. Whenever there is more than one level 01 item in a file definition, each level 01 provides a separate definition of the data record area. Only one record is read into the data record area at a time, so only one level 01 definition is used at a time. For example, a file may contain two types of records, department records and employee records. Level 01 items are defined for both department and employee records. The hierarchy for each record immediately follows the level 01 item for that record and provides the definitions for the entire record. Each record contains a field that identifies the record type. This record type is at the same position for all record definitions. In the following example, the record type is in the first position.

Example 6: File containing multiple records

```
01   DEPARTMENT-RECORD.
          05 DEPT-RECORD-TYPE PIC X.
          05 DEPT-ID PIC 999.
          05 DEPT-NAME PIC X(30).
          05 DEPT-DESCRIPTION PIX X(160).
01   EMPLOYEE-RECORD.
          05 EMP-RECORD-TYPE PIC X.
          05 EMP-ID PIC 9(6).
          05 EMP-REGION PIC 9.
          05 EMP-DEPT PIC 999.
          05 EMP-HIRE-DATE.
             10 EMP-HIRE-DATE-MM PIC 99.
             10 EMP-HIRE-DATE-DD PIC 99.
             10 EMP-HIRE-DATE-YYYY PIC 9999.
          05 EMP-SALARY PIC 9(9).
          05 EMP-NAME PIC X(15).
```

Group Fields

As we have seen in the section on Data Hierarchy, fields can be organized in groups. These groups actually provide an additional definition of the fields and can be used to access the data.

Redefinition

COBOL provides the ability to redefine a field or a group. The redefinition does not define data at a new location, but instead provides an additional definition of data characters that has been previously defined.

Example 7: Redefinition of a field

```
01  EMPLOYEE-RECORD.
          05 EMP-ID PIC 9(6).
          05 EMP-ID-R REDEFINES EMP-ID.
             10 EMP-ID-GROUP PIC 99.
             10 EMP-ID-NUM PIC 9999.
          05 EMP-REGION PIC 9.
          05 EMP-DEPT PIC 999.
          05 EMP-HIRE-DATE.
             10 EMP-HIRE-DATE-MM PIC 99.
             10 EMP-HIRE-DATE-DD PIC 99.
             10 EMP-HIRE-DATE-YYYY PIC 9999.
          05 EMP-SALARY PIC 9(9).
          05 EMP-NAME PIC X(15).
```

In the above example, the EMP-ID field is defined as a 6 digit numeric field. A redefinition is provided dividing the field into two fields: EMP-ID-GROUP is defined as the first two digits of the EMP-ID field. EMP-ID-NUM is defined as the last four digits of the EMP-ID field. Note that EMP-ID-NUM and EMP-ID-GROUP both begin at position 1 in the record.

Example 8: Redefinition of a group

```
01  EMPLOYEE-RECORD.
          05 EMP-ID PIC 9(6).
          05 EMP-ID-R REDEFINES EMP-ID.
             10 EMP-ID-GROUP PIC 99.
             10 EMP-ID-NUM PIC 9999.
          05 EMP-REGION PIC 9.
          05 EMP-DEPT PIC 999.
          05 EMP-HIRE-DATE.
             10 EMP-HIRE-DATE-MM PIC 99.
             10 EMP-HIRE-DATE-DD PIC 99.
             10 EMP-HIRE-DATE-YYYY PIC 9999.
          05 EMP-SALARY PIC 9(9).
          05 EMP-NAME PIC X(15).
          05 EMP-SKILLS OCCURS 4 TIMES.
             10 EMP-SKILL-LEVEL PIC 99.
             10 EMP-SKILL-ID PIC 9(4).
          05 EMP-SKILLS-R REDEFINES EMP-SKILLS.
             10 EMP-SKILL-LEVEL1 PIC 99.
             10 EMP-SKILL-ID1 PIC 9(4).
             10 EMP-SKILL-LEVEL2 PIC 99.
             10 EMP-SKILL-ID2 PIC 9(4).
             10 EMP-SKILL-LEVEL3 PIC 99.
             10 EMP-SKILL-ID3 PIC 9(4).
             10 EMP-SKILL-LEVEL4 PIC 99.
             10 EMP-SKILL-ID4 PIC 9(4).
```

In the above example, the EMP-SKILL array has been redefined so that each element is expanded providing a field for each element.. As in the previous example, the definitions for EMP-SKILLS-R is defining the same data area as EMP-SKILLS. The first occurrence of EMP-SKILL-LEVEL is at the same position in the record as EMP-SKILL-LEVEL1.

COBOL FILE FORMATS

COBOL programs can create files of different organization. These include Line sequential, record sequential, relative and indexed.

Line Sequential Files

Line sequential files are generally known as text files because the primary use of this file type is for display data. The records in these files can only be accessed in the order they were written. Line sequential files contain variable length records. A record delimiter separates each record in the file. The record delimiter that is used is operating system dependent. The record delimiter character is inserted after the last character in each record.

Record Sequential Files

Record sequential files are also accessed in the order they were written. This file organization is more flexible than line sequential. Records can be written as fixed or variable length. The record sequential organization is used for sequential files that contain binary or packed data, or any data that my have other non-printable characters. In fixed length files, every record that is written to the file is the same length. If necessary, the record will be padded with blanks. With variable length records, each record is written based on the actual size of the record. A Record Descriptor Word (RDW) is written at the beginning of each record. The RDW contains the actual length of the record. It is not considered part of the record and is not included in the data definition. In general variable length records are used when there are many small records and few large records. Variable length records may need to be converted before they can be imported.

Relative Files

Relative files can be accessed randomly as well as in the order they are written. Records can be declared as variable, but they are written as fixed. The random access is not by key, but is instead by relative record number. Relative files may need to be converted to sequential before they can be imported.

Indexed Files

Indexed files can be accessed by key field(s) as well as in the order they are written. Records in indexed files can be fixed or variable. Indexed files are actually two physical files, one containing the data and one containing the index. Indexed files will need to be converted to sequential before they can be imported.

RE-INTERPRETING COBOL STRUCTURES RELATIONALLY

Records

As we consider importing COBOL files into a relational database, we must plan how the data will be mapped into the relational database. At the highest level, each record type, level 01 structure, is most naturally considered mapping to a table. This is often a good place to start, but your design may benefit from further analysis.

Records in files are often designed to be independent sources of information. This is a large difference between records and tables. Tables are generally designed to hold information that is closely related. When you consider the records in your file, you should consider if it would be better to define multiple tables for the information. Groups that are used to organize related information are often good candidates for independent tables. For example, you may have a group that is defined for address information, which might naturally fit into a name and address table. Arrays, whether varying or not, are also often good candidates for independent tables.

Arrays

In order to define arrays to your relational database, you can normalize the array by specifying each element in the array independently. This technique can also be used for defining varying arrays when the array is at the end of a record. Records with embedded varying arrays are not necessarily physically stored as variable therefore you may be able to use this technique for loading these records also. Refer to case study 1 which demonstrates the normalization technique being used when loading a varying array.

Data Type Mappings

As discussed earlier, the USAGE and PICTURE clauses are used to define the format and characteristics of data elements. When considered together the picture and usage identify the scalar data type, length, precision and scale. The following table shows data element definitions that apply to COBOL files. It describes the representation of each data type and identifies how that data type is represented to SQL*Loader. The PICTURE represents a mask that describes the data. The values identified within the parentheses are multiplication factors for the preceding picture element. So when n = 5, X(n) indicates that there are 5 characters of type X (alphanumeric data).

COBOL to SQL*Loader Table

COBOL Definition	USAGE	DESCRIPTION
X(n)	DISPLAY	Alphanumeric data. Each X identifies one allowable character from the specified character set. **SQL*Loader Type** CHAR(n) DATE(n) 'mask' when data contains valid date TIMESTAMP INTERVAL
A(n)	DISPLAY	Alphabetic data. Each A identifies any letter of the alphabet or space. **SQL*Loader Type** CHAR(n)
9(n)	DISPLAY	Numeric data. Each 9 identifies one digit. **SQL*Loader Type** INTEGER EXTERNAL(n) DECIMAL EXTERNAL(n) ZONED EXTERNAL(n) DATE(n) 'mask' when data contains valid date
+- mantissa +- exponent	DISPLAY	External floating point data. **SQL*Loader Type** FLOAT EXTERNAL (length)
S9(n)v9(m) SIGN TRAILING	DISPLAY	Numeric data. Each 9 identifies one digit. The v indicates the implied decimal position. The sign is carried in the last byte. **SQL*Loader Type** ZONED(precision, scale) where precision=n+m and scale=m.
9(n)v9(m)	DISPLAY	Numeric data. Each 9 identifies one digit. The v indicates the implied decimal position. **SQL*Loader Type** ZONED(precision,scale) where precision=n+m and scale=m.
9(n)v9(m) S9(n)v9(m)	BINARY COMPUTATIONAL COMP COMPUTATIONAL-4 COMP-4	Internal format data with a radix of 2. The size of the field varies with the value m. n+m = 1-4, length = 2 n+m = 5-9, length = 4 n+m = 10-18, length =8 **SQL*Loader Type** SMALLINT INTEGER (length 2,4,or 8). May use SIGNED|UNSIGNED May require BYTEORDER clause. Scale handled with an expression.
Picture clause not allowed for COMP-1	COMPUTATIONAL-1 COMP-1	Single-precision floating point number, 4 bytes long **SQL*Loader Type** FLOAT. May require BYTEORDER clause.
Picture clause not allowed for COMP-2	COMPUTATIONAL-2 COMP-2	Double-precision floating point number, 8 bytes long **SQL*Loader Type** DOUBLE.

		May require BYTEORDER clause.
9(n)v9(m) S9(n)v9(m)	COMPUTATIONAL-3 COMP-3 PACKED-DECIMAL	Internal format numeric data with a radix of 10. The clause indicates that each digit must use the minimum storage possible. Generally, each byte contains two digits with the last half-byte containing the sign. **SQL*Loader Type** DECIMAL(precision,scale) where precision=n+m and scale=m.
X(n) 9(n)v9(m) S9(n)v9(m)	COMPUTATIONAL-5 COMP-5 COMPUTATIONAL-X COMP-X	(Not commonly used). The internal format of the data is not defined. It is often stored the same as BINARY, however the radix may be reversed.
G(n)	DISPLAY-1	Graphic data that does not contain Shift In and Shift Out characters **SQL*Loader Type** GRAPHIC(n)
05 V. 49 V-LN PIC S9(4) COMP. 05 V-DATA PIC X(n)		Variable length character field **SQL*Loader Type** VARCHAR(max length), can only be loaded correctly between systems where SMALLINT is the same size.
05 V. 49 V-LN PIC S9(4) COMP. 05 V-DATA PIC G(n)		Variable length Graphic data that does not contain Shift In and Shift Out characters **SQL*Loader Type** VARGRAPHIC(max length), can only be loaded correctly between systems where SMALLINT is the same size.

INTEGRATING COBOL DATA

The remainder of this paper will show how COBOL source files are used within Warehouse Builder. Four case studies are demonstrated. The first case study shows loading a table from a file that contains a varying array. The second case study shows loading dimensional data directly into an Oracle OLAP Analytic Workspace using an external table. The third case study shows loading tables from a file containing multiple record types. The fourth case study shows loading data from a file with ebcdic and internal format data.

The process for loading COBOL source files is done in four basic steps:

I. Analyze the copybook.

II. Define and configure the metadata.

III. Create and configure a mapping to the target(s).

IV. Deploy and execute.

Case Study 1: Load Table from a File Containing a Varying Array

This case study demonstrates defining a COBOL varying array structure to Warehouse Builder. It loads the file into a table using SQL*Loader.

I. Analyze the Copybook

The table below illustrates a COBOL copybook definition for an EBCDIC file. The copybook has a group field EMP-HIRE-DATE that contains subordinate fields for month, day and year. You will notice that we have decided to use only the group field and are mapping it to a DATE field. The subordinate fields for month, day and year are ignored. Additionally, there is an 'occurs depending on' data structure. The table illustrates an approach to map this varying array to Oracle SQL*Loader, each element in the occurs is mapped to a field. We know there is a maximum number of occurrences in this array of 4, so we can create fields for skill level and skill id for each element in the potential array. The SQL*Loader 'trailing nullcols' and external table's 'value null when blank' will be utilized in order to load this file.

COBOL Copybook for Employee Record

```
01  EMPLOYEE-RECORD.
    05  EMP-RECORD-LENGTH PIC 9(4).
    05  EMP-ID PIC 9(6).
    05  EMP-REGION PIC 9.
    05  EMP-DEPT PIC 999.
    05  EMP-HIRE-DATE.
        10 EMP-HIRE-MM PIC 99.
        10 EMP-HIRE-DD PIC 99.
        10 EMP-HIRE-YYY PIC 9999.
    05  EMP-SALARY PIC 9(9).
    05  EMP-NAME PIC X(15).
    05  EMP-SKILL-COUNT PIC 99.
```

```
05  EMP-SKILL OCCURS 4 DEPENDING ON EMP-SKILL-COUNT.
    10 EMP-SKILL-LEVEL PIC 99.
    10 EMP-SKILL-ID PIC 9999.
```

COBOL to SQL*Loader Table

COBOL Copybook	**SQL*Loader Field name**	**Start: End**	**SQL*Loader Type**
01 EMPLOYEE-RECORD.			
05 EMP-RECORD-LENGTH PIC 9(4).			
05 EMP-ID PIC 9(6).	EMP_ID	5:10	INTEGER EXTERNAL(6)
05 EMP-REGION PIC 9.	EMP_REGION	11:11	INTEGER EXTERNAL(1)
05 EMP-DEPT PIC 999.	EMP_DEPT	12:14	INTEGER EXTERNAL(3)
05 EMP-HIRE-DATE.	EMP_HIRE_DATE	15:22	DATE(8) MMDDYYYY DEFAULTIF EMP_HIRE_DATE= BLANKS
10 EMP-HIRE-MM PIC 99.			
10 EMP-HIRE-DD PIC 99.			
10 EMP-HIRE-YYYY PIC 9999.			
05 EMP-SALARY PIC 9(9).	EMP_SALARY	23:31	INTEGER EXTERNAL(9)
05 EMP-NAME PIC X(15).	EMP_NAME	32:46	CHAR(15) NULLIF EMP_NAME= BLANKS
05 EMP-SKILL-COUNT PIC 99.	EMP_SKILL_COUNT	47:48	INTEGER EXTERNAL(2)
05 EMP-SKILLS OCCURS 4 TIMES DEPENDING ON EMP-SKILL-COUNT.			
10 EMP-SKILL-LEVEL PIC 99.	SKILL_LEVEL1	49:50	INTEGER EXTERNAL(2)
10 EMP-SKILL-ID PIC 9999.	SKILL_ID1	51:54	INTEGER EXTERNAL(4)
	SKILL_LEVEL2	49:50	INTEGER EXTERNAL(2)
	SKILL_ID2	57:60	INTEGER EXTERNAL(4)
	SKILL_LEVEL3	61:62	INTEGER EXTERNAL(2)
	SKILL_ID3	63:66	INTEGER EXTERNAL(4)
	SKILL_LEVEL4	67:68	INTEGER EXTERNAL(2)
	SKILL_ID4	69:70	INTEGER EXTERNAL(4)

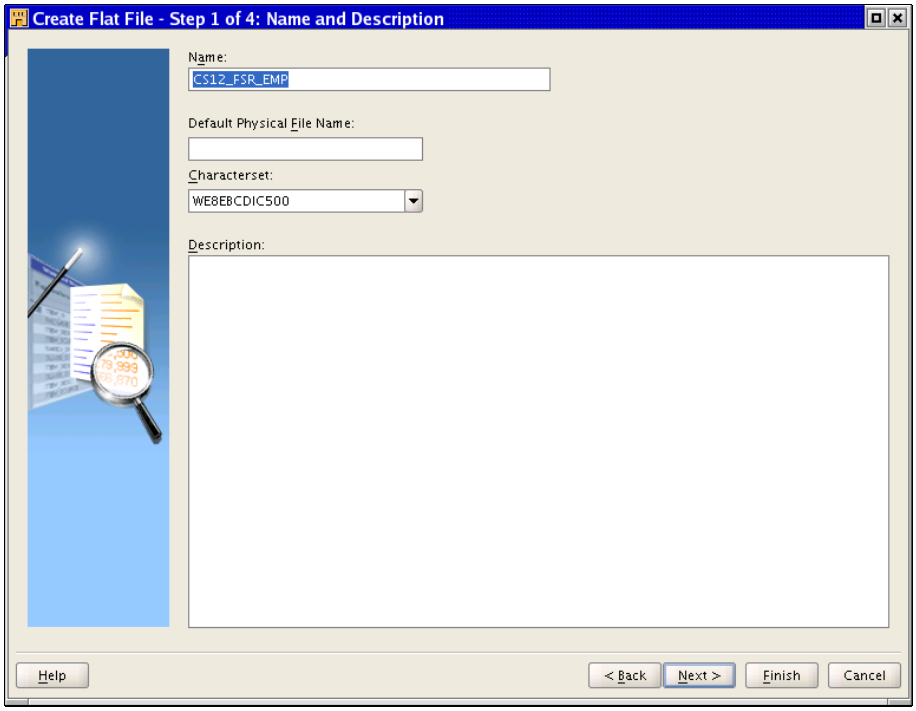
II. Define and Configure the Metadata

1. Define the flat file using Create Flat File Wizard.

Step 1: Define the file and character set

In most cases a file definition is sampled and the files are used to create the metadata definition in Warehouse Builder. In this case, we will create the file definition directly without sampling.

A name is given to the file definition and the character set is identified as WE8EBCDIC500. This allows SQL*Loader to recognize the data in the file as EBCDIC data. A default physical file name can be provided at this time also.



Step 2: Define the file properties.

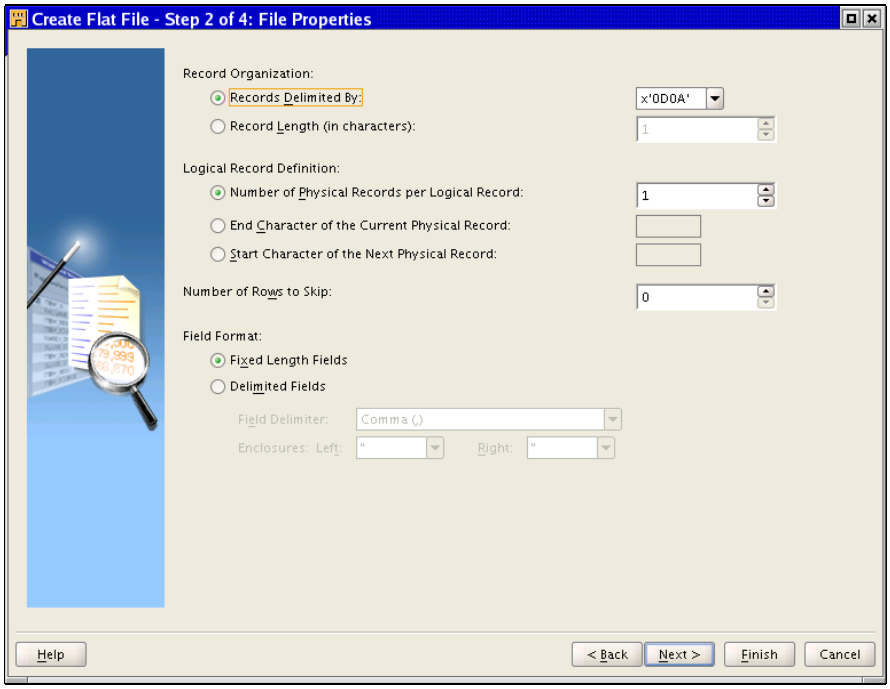
The physical file properties are not specified in the copybook. Instead, they are determined by the file specification in the COBOL program (described earlier under COBOL File Format), the environment the file was created in and the ftp options that were used if the file was transferred.

Record Delimiter

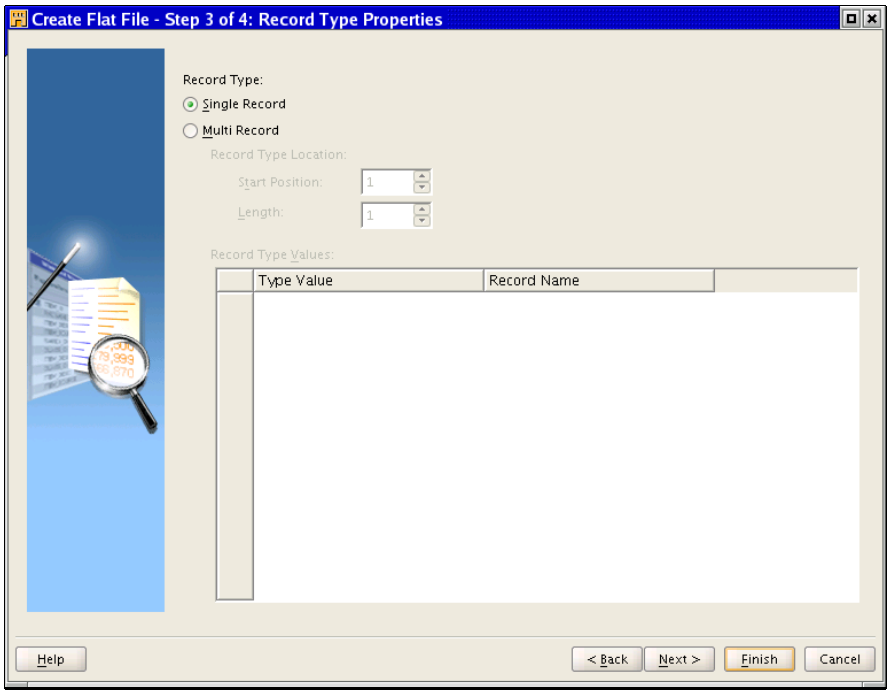
The record delimiter indicates the end of the physical record in the file. The record delimiter in this file is the binary value 0D0A. It is specified as x'0D0A'. Any text, character or hexadecimal, that is not in the drop down list of values can simply be typed directly in the Delimiter box.

Field Format

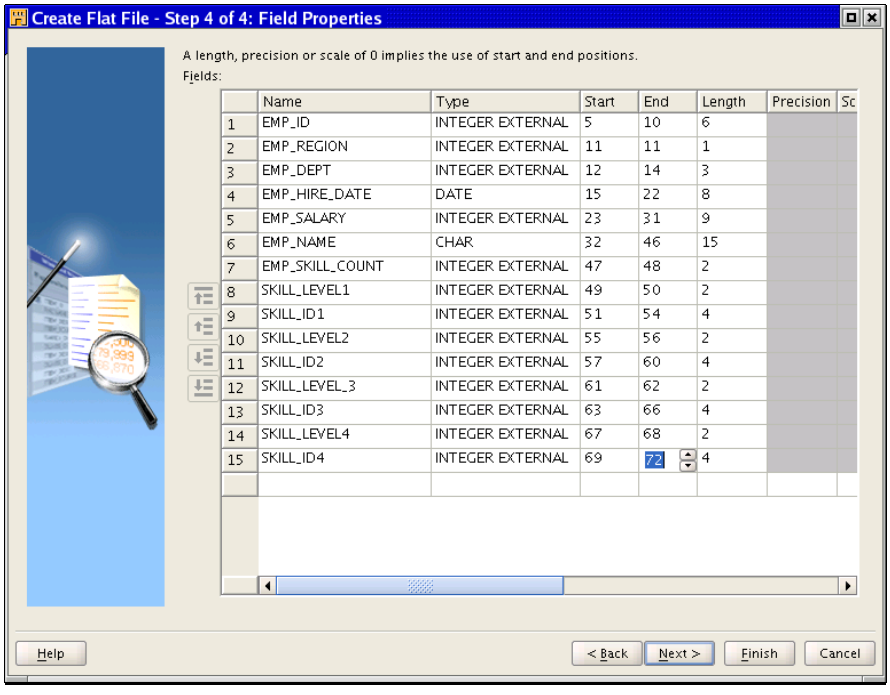
The fields in the record are not delimited or enclosed. Instead each field is at a constant position in the records and the file is therefore defined as containing fixed length fields.



Step 3: Identify the file as containing a single record type.



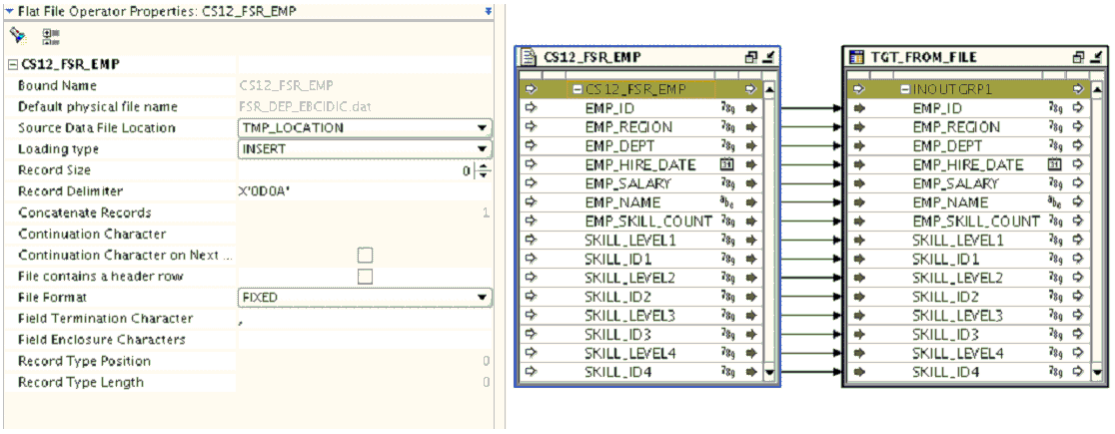
Step 4: Define the fields in the record using the COBOL to SQL*Loader Table.



II. Create and Configure a Mapping to the Target.

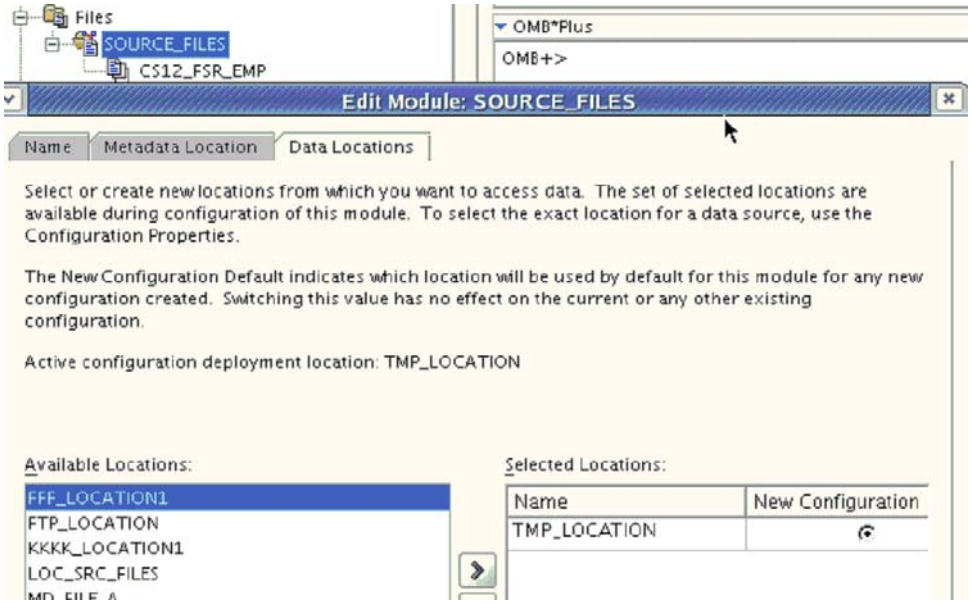
1. Create a SQL*Loader mapping to the target table.

The map below has a file operator as a source loading directly into a target table that has the same structure as the file. The properties should not require any changes and are just shown for reference.



2. Configure the File Module's Location

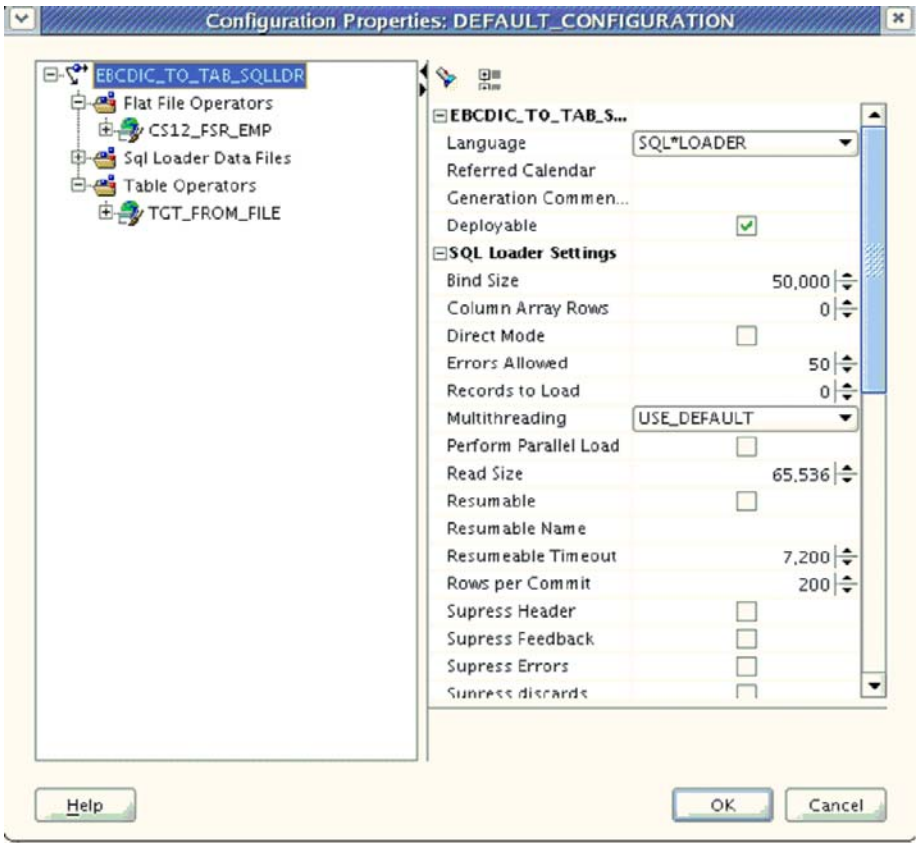
Ensure the file module is configured correctly for the desired data location – where the flat files will be found when the SQL*Loader map/external table is used. This consists of two steps, one setting the data location and the second setting the configuration for the File Module. To set the data location edit the File Module itself. To configure the File Module, right mouse click on the File Module and choose configure.



3. Configure the mapping

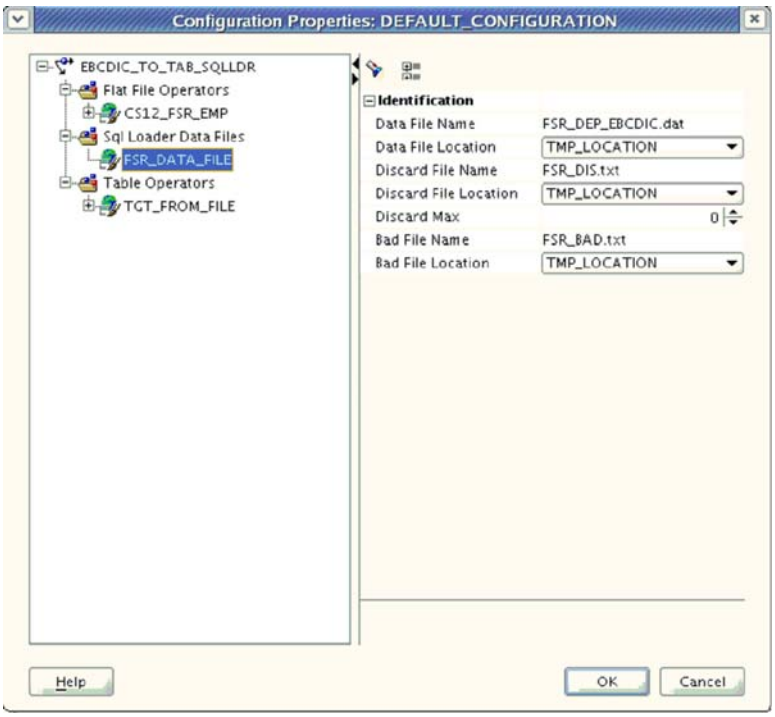
SQL*Loader tuning properties

Tune any SQL*Loader parameters such as parallelism and thread counts. To find these properties right mouse click on the mapping. This will open the configuration window as shown below where the following steps are done.



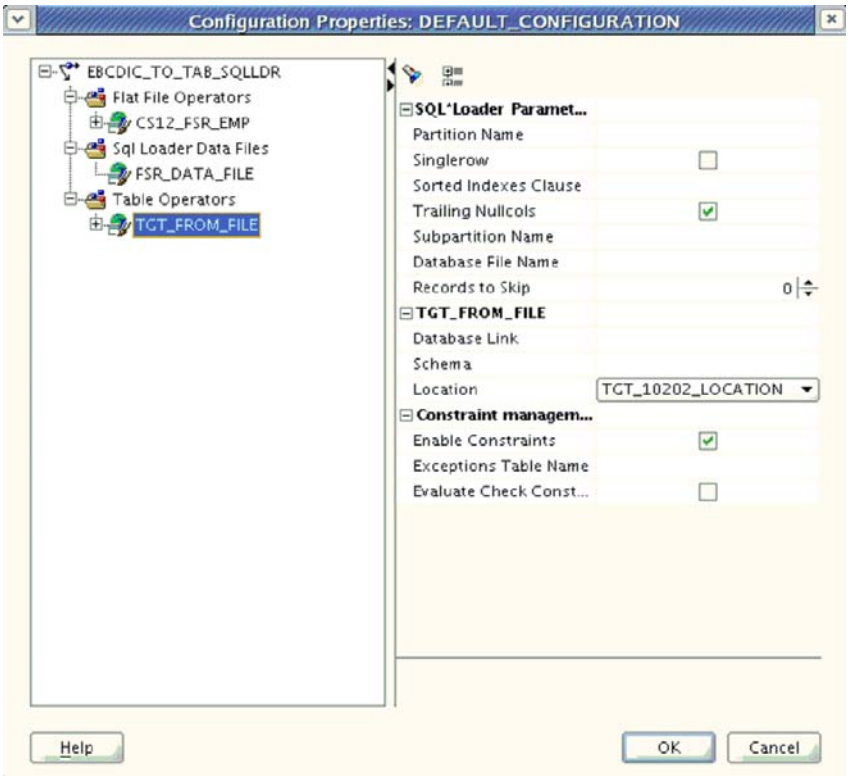
*Add a Data File for Sql*Loader Data Files*

Create a data file by clicking on 'Sql Loader Data Files' and adding a data file. Then define the data file name, and discard/bad file names if desired. It is good practice to do so as it simplifies debugging of the mapping.



Set the 'Trailing Nullcols' property

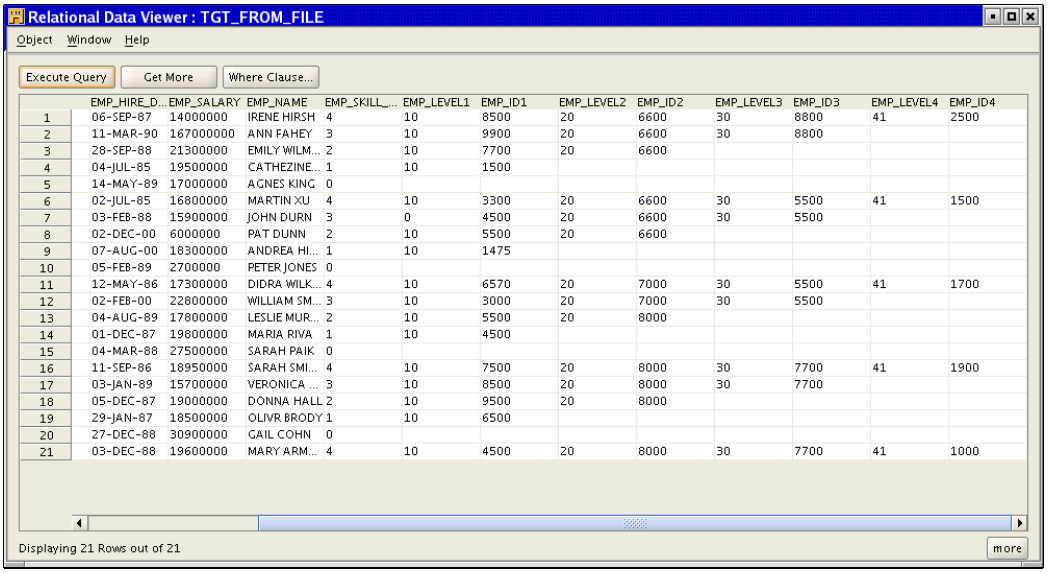
The SQL*Loader 'TRAILING NULLSCOLS' property can be found on the target table being loaded. Select the table in the tree and select the 'Trailing Nullscols' property in the property inspector allowing the varying array.



IV. Deploy and Execute.

Deploy the tables and mapping to create them in the target database. When the mapping is executed, the TGT_FROM_FILE table is loaded with the converted data. SKILL_LEVELx and SKILL_IDx columns are partially filled since the file structure uses OCCURS 4 TIMES depending on ITEM_COUNT.

TGT_FROM_FILE Table Data



	EMP_HIRE_D...	EMP_SALARY	EMP_NAME	EMP_SKILL_...	EMP_LEVEL1	EMP_ID1	EMP_LEVEL2	EMP_ID2	EMP_LEVEL3	EMP_ID3	EMP_LEVEL4	EMP_ID4
1	06-SEP-87	14000000	IRENE HIRSH	4	10	8500	20	6600	30	8800	41	2500
2	11-MAR-90	167000000	ANN FAHEY	3	10	9900	20	6600	30	8800		
3	28-SEP-88	21300000	EMILY WILM...	2	10	7700	20	6600				
4	04-JUL-85	19500000	CATHEZINE...	1	10	1500						
5	14-MAY-89	17000000	AGNES KING	0								
6	02-JUL-85	16800000	MARTIN XU	4	10	3300	20	6600	30	5500	41	1500
7	03-FEB-88	15900000	JOHN DURN	3	0	4500	20	6600	30	5500		
8	02-DEC-00	6000000	PAT DUNN	2	10	5500	20	6600				
9	07-AUG-00	18300000	ANDREA HI...	1	10	1475						
10	05-FEB-89	2700000	PETER JONES	0								
11	12-MAY-86	17300000	DIDRA WILK...	4	10	6570	20	7000	30	5500	41	1700
12	02-FEB-00	22800000	WILLIAM SM...	3	10	3000	20	7000	30	5500		
13	04-AUG-89	17800000	LESLIE MUR...	2	10	5500	20	8000				
14	01-DEC-87	19800000	MARIA RIVA	1	10	4500						
15	04-MAR-88	27500000	SARAH PAIK	0								
16	11-SEP-86	18950000	SARAH SMI...	4	10	7500	20	8000	30	7700	41	1900
17	03-JAN-89	15700000	VERONICA ...	3	10	8500	20	8000	30	7700		
18	05-DEC-87	19000000	DONNA HALL	2	10	9500	20	8000				
19	29-JAN-87	18500000	OLIVR BRODY	1	10	6500						
20	27-DEC-88	30900000	GAIL COHN	0								
21	03-DEC-88	19600000	MARY ARM...	4	10	4500	20	8000	30	7700	41	1000

Case Study 2: Load Dimensional Data Using External Table

In this case study we create an external table and use it to load Dimensional Data into an Analytic Workspace. You can easily create an external table from a file definition. Defining an external table for the file allows you to select from a record as if it were a relational table and provides greater capability during the load.

To demonstrate some additional capabilities for Warehouse Builder, lookup tables called EDEPT and EREGION) for dept name and region name are also used. This case study allows us to illustrate loading directly to Oracle OLAP from an EBCDIC legacy data file. This is a unique capability of Warehouse Builder.

A. Analyze the Copybook.

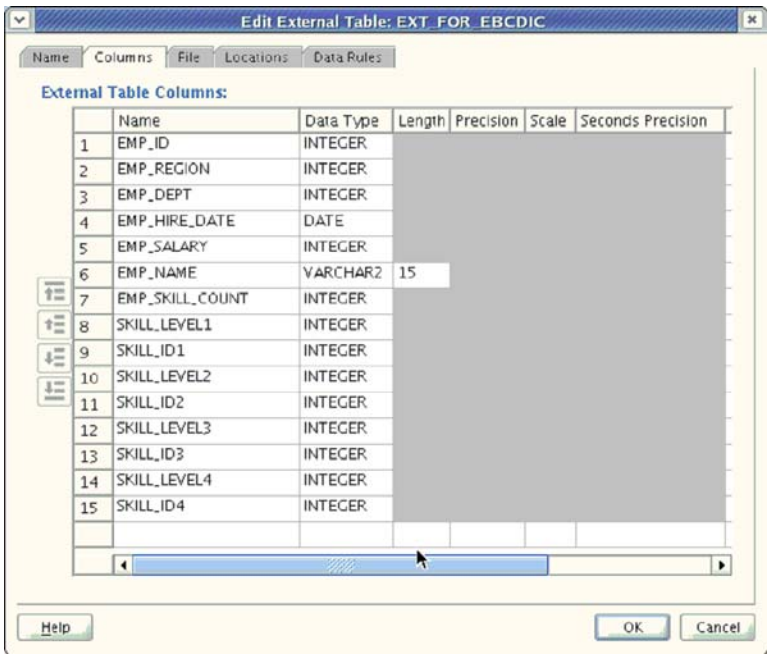
This case study uses the copybook and file definition from Case Study 3.

B. Define the Metadata.

1. Define the Flat File as specified in Case Study 3.

2. Define the External Table using Create External Table Wizard.

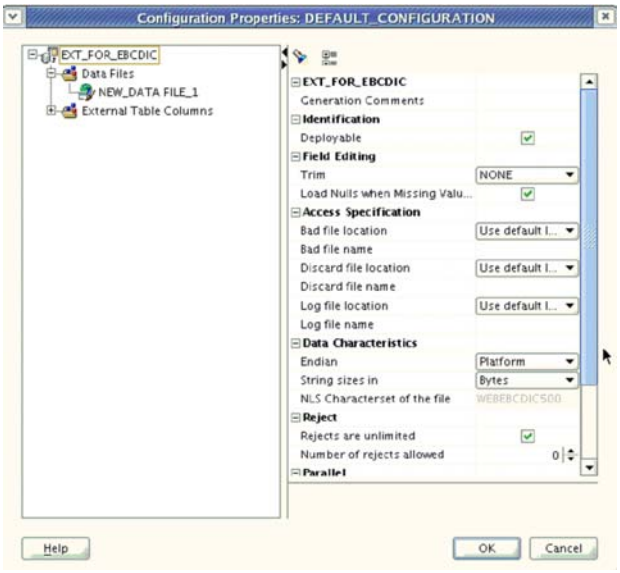
Create an external table, select the file we defined earlier in the create new external table wizard. The columns are automatically derived from the file definition.



3. Configure the External Table

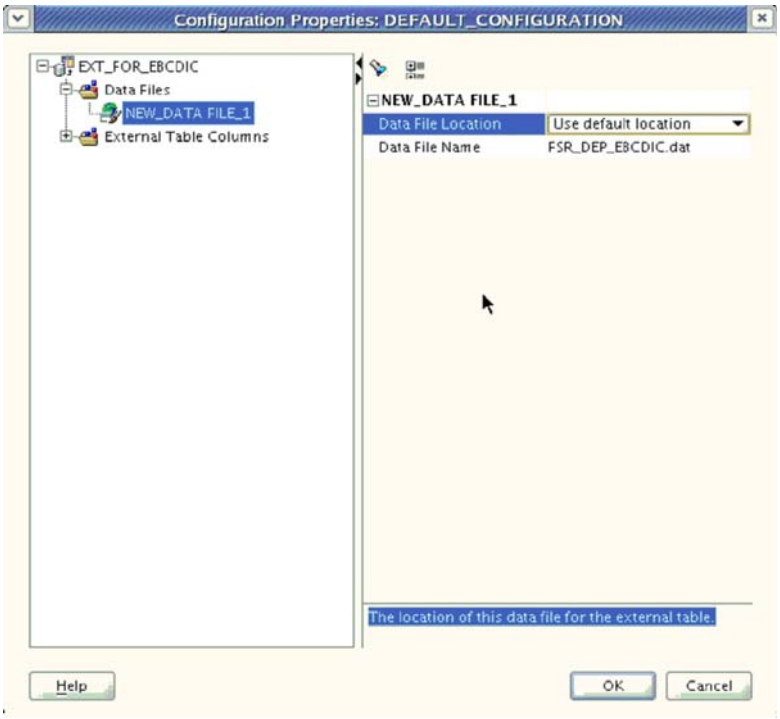
Set the "Load Nulls When Missing Property"

Select the 'Load Nulls when Missing Values' property. Make sure to do this as this allows for the varying array.



Create a Data File and define data file name

Right click 'Data Files' node in the tree and add a file, enter the data file name and location (if different from default).



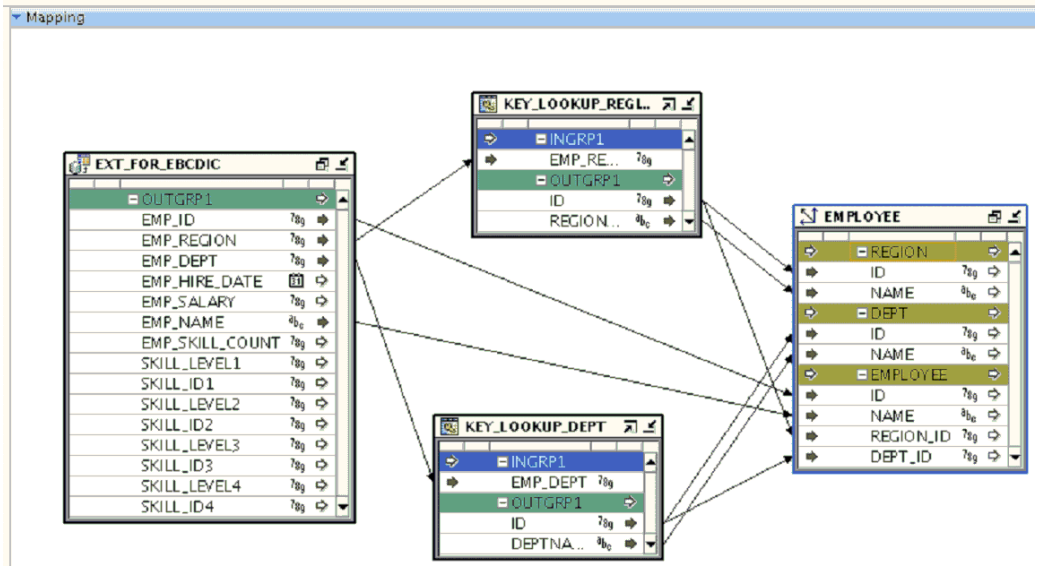
4. Define and load EDEPT and EREGION Tables

Simple lookup tables, EDEPT and EREGION are used to provide information for the employee dept and region columns that are in the source data file. After deploying EDEPT and EREGION, execute the SQL below on the target schema).

```
insert into eregion values (1,'USWEST');
insert into eregion values (2,'EMEA');
insert into eregion values (4,'APAC');
insert into eregion values (3,'USEAST');
insert into edept values (153,'Customer Support');
insert into edept values (650,'Platform Engineering');
insert into edept values (138,'R and D');
insert into edept values (144,'Finance');
insert into edept values (117,'Operations');
insert into edept values (161,'Tools Development');
insert into edept values (288,'Applications Development');
insert into edept values (265,'Documentation');
insert into edept values (165,'Curriculum Devt');
insert into edept values (127,'Sales');
commit;
```

III. Create and Configure a Mapping to the Target.

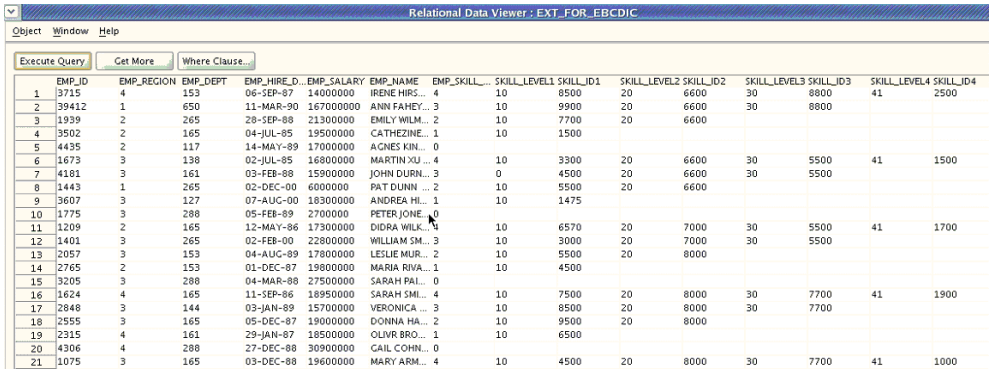
Here we create a mapping that uses the external table, and performs a lookup on the dept and region tables to get the names and then loads the data into the dimension operator for loading the AW dimension (the dimension has 2 hierarchies Region <- Employee and Dept <- Employee).



IV. Deploy and Execute

EXT_FOR_EBCDIC Table Data

The source data projected through the external table looks like the following;

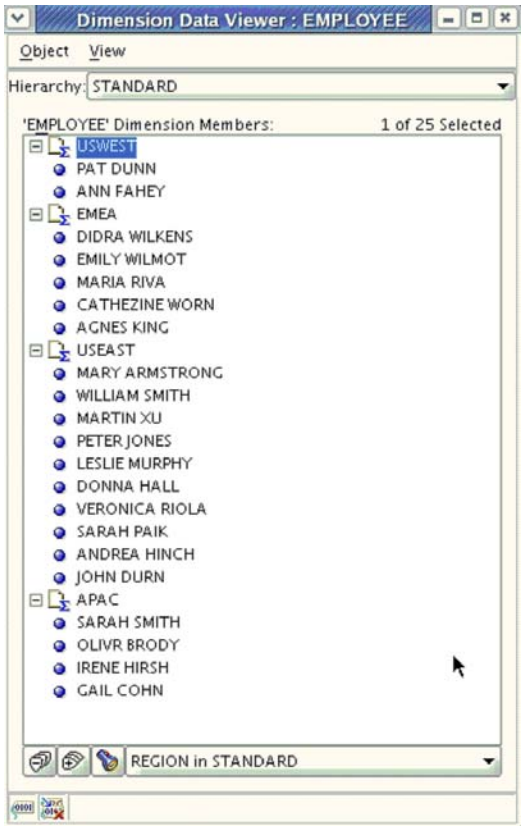


	EMP_ID	EMP_REGION	EMP_DEPT	EMP_HIRE_D...	EMP_SALARY	EMP_NAME	EMP_SKILL_...	SKILL_LEVEL1	SKILL_ID1	SKILL_LEVEL2	SKILL_ID2	SKILL_LEVEL3	SKILL_ID3	SKILL_LEVEL4	SKILL_ID4
1	3715	4	153	06-SEP-87	14000000	IRENE HIRS...	4	10	8500	20	6600	30	8800	41	2500
2	39412	1	650	11-MAR-90	167000000	ANN FAHEY...	3	10	9900	20	6600	30	8800		
3	1939	2	265	28-SEP-88	21300000	EMILY WILM...	2	10	7700	20	6600				
4	3502	2	165	04-JUL-85	19500000	CATHEZINE...	1	10	1500						
5	4435	2	117	14-MAY-89	17000000	AGNES KIN...	0								
6	1673	3	138	02-JUL-85	16800000	MARTIN XU ...	4	10	3300	20	6600	30	5500	41	1500
7	4181	3	161	03-FEB-88	15900000	JOHN DURN...	3	0	4500	20	6600	30	5500		
8	1443	1	265	02-DEC-00	6000000	PAT DUNN ...	2	10	5500	20	6600				
9	3607	3	127	07-AUG-00	18300000	ANDREA HI...	1	10	1475						
10	1775	3	288	05-FEB-89	2700000	PETER JONE...	0								
11	1209	2	165	12-MAY-86	17300000	DIDRA WILK...	4	10	6570	20	7000	30	5500	41	1700
12	1401	3	265	02-FEB-00	22800000	WILLIAM SM...	3	10	3000	20	7000	30	5500		
13	2057	3	153	04-AUG-89	17800000	LESLIE MUR...	2	10	5500	20	8000				
14	2765	2	153	01-DEC-87	19800000	MARIA RIVA...	1	10	4500						
15	3205	3	288	04-MAR-88	27500000	SARAH PAI...	0								
16	1624	4	165	11-SEP-86	18950000	SARAH SMI...	4	10	7500	20	8000	30	7700	41	1900
17	2848	3	144	03-JAN-89	15700000	VERONICA ...	3	10	8500	20	8000	30	7700		
18	2555	3	165	05-DEC-87	19000000	DONNA HA...	2	10	9500	20	8000				
19	2315	4	161	29-JAN-87	18500000	OLIVR BRO...	1	10	6500						
20	4306	4	288	27-DEC-88	30900000	GAIL COHN...	0								
21	1075	3	165	03-DEC-88	19600000	MARY ARM...	4	10	4500	20	8000	30	7700	41	1000

You can see the SKILL_LEVELx and SKILL_IDx columns are partially filled (since the file structure uses OCCURS 4 TIMES depending on ITEM_COUNT).

EMPLOYEE Dimensional Data Loaded in AW

After executing the dimension map we see the dimensional data loaded in the Oracle OLAP Analytic Workspace.



Case Study 3: Load Tables from a File Containing Multiple Record Types

This case study demonstrates defining a multiple record file to Warehouse Builder. It loads the file into three separate tables in one pass using SQL*Loader.

I. Analyze the Copybook

Examine the copybook and identify the COBOL to SQL*Loader Mappings. In this example the copybook defines two records, Employee and Payroll. The REC_TYPE field in each record is the first position. Records with the letter "E" as the first character are employee records while records with the letter "P" as the first character are pay records. Note that there is nothing special about the definition of the field containing the record type except that it is at the same position in both records.

COBOL Copybook for Employee and Payroll Records

```
01  EMPLOYEE-RECORD.
    05  EMP-RECORD-TYPE PIC X.
    05  EMP-ID PIC 9(6).
    05  EMP-REGION PIC 9.
    05  EMP-DEPT PIC 999.
    06  EMP-HIRE-DATE.
        10 EMP-HIRE-MM PIC 99.
        10 EMP-HIRE-DD PIC 99.
        10 EMP-HIRE-YYY PIC 9999.
    05  EMP-SALARY PIC 9(9).
    05  EMP-NAME PIC X(15).

01  PAYROLL-RECORD.
    05  PAY-REC-TYPE PIC X.
    05  PAY-ID PIC 9(6).
    05  PAY-PAY-END-DATE PIC 9(8).
    05  PAY-CHECK-DATE PIC 9(8).
    05  PAY-CHECK-NUM PIC 9(5).
    05  PAY-GROSS PIC 9(9).
    05  PAY-YTD-GROSS PIC 9(9).
    05  PAY-DEDUCTIONS PIC 9(5) OCCURS 7.
```

The table below identifies the SQL*Loader fields that will be defined.

COBOL to SQL*Loader Table

COBOL Copybook	SQL*Loader Field name	Length	SQL*Loader Type
01 EMPLOYEE-RECORD.			
05 EMP-REC-TYPE PIC X.	TYPE	1	CHAR(1)
05 EMP-ID PIC 9(6).	ID	6	INTEGER EXTERNAL(6)
05 EMP-REGION PIC 9.	REGION	1	INTEGER EXTERNAL(1)
05 EMP-DEPT PIC 999.	DEPT	3	INTEGER EXTERNAL(3)
05 EMP-HIRE-DATE.	HIRE_DATE	8	DATE(8) MMDDYYYY DEFAULTIF EMP_HIRE_DATE= BLANKS
10 EMP-HIRE-MM PIC 99.			
10 EMP-HIRE-DD PIC 99.			

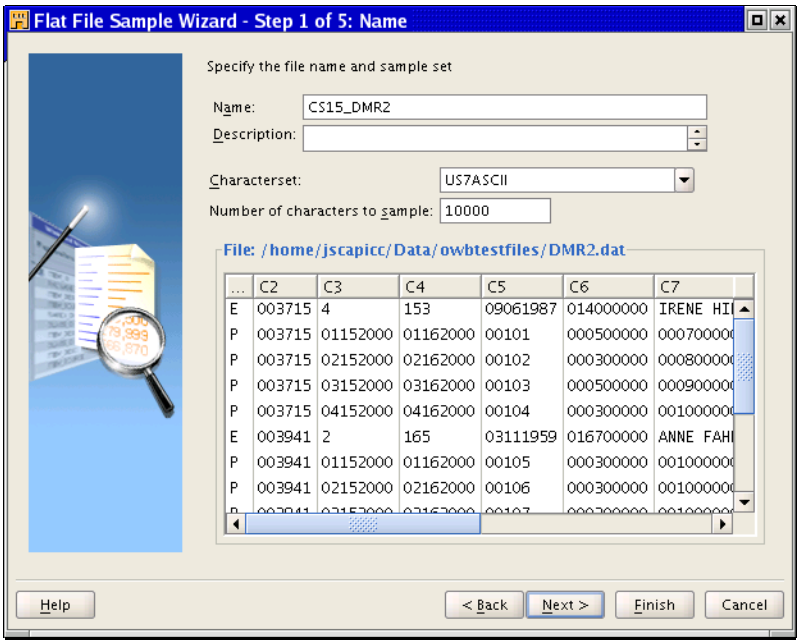
10 EMP-HIRE-YYYY PIC 9999.			
05 EMP-SALARY PIC 9(9).	SALARY	9	INTEGER EXTERNAL(9)
05 EMP-NAME PIC X(15).	NAME	15	CHAR(15) NULLIF EMP_NAME= BLANKS
01 PAY-RECORD.			
05 PAY-REC-TYPE PIC X.	TYPE	1	CHAR(1)
05 PAY-ID PIC 9(6).	ID	6	INTEGER EXTERNAL(6)
05 PAY-END-DATE PIC 9(8).	END_DATE	8	DATE
05 PAY-CHECK-DATE PIC 9(8).	CHECK_DATE	8	DATE
05 PAY-CHECK-NUM PIC 9(5).	CHECK_NUM	5	INTEGER EXTERNAL(5)
05 PAY-GROSS PIC 9(9).	GROSS_AMT	9	INTEGER EXTERNAL(9)
05 PAY-YTD-GROSS PIC 9(9).	YTD_AMT	9	INTEGER EXTERNAL(9)
05 DEDUCTIONS PIC 9(5) OCCURS 7.	DED1	5	INTEGER EXTERNAL(5)
	DED2	5	INTEGER EXTERNAL(5)
	DED3	5	INTEGER EXTERNAL(5)
	DED4	5	INTEGER EXTERNAL(5)
	DED5	5	INTEGER EXTERNAL(5)
	DED6	5	INTEGER EXTERNAL(5)
	DED7	5	INTEGER EXTERNAL(5)

II. Define and Configure the Metadata

1. Define the flat file using Flat File Sample Wizard.

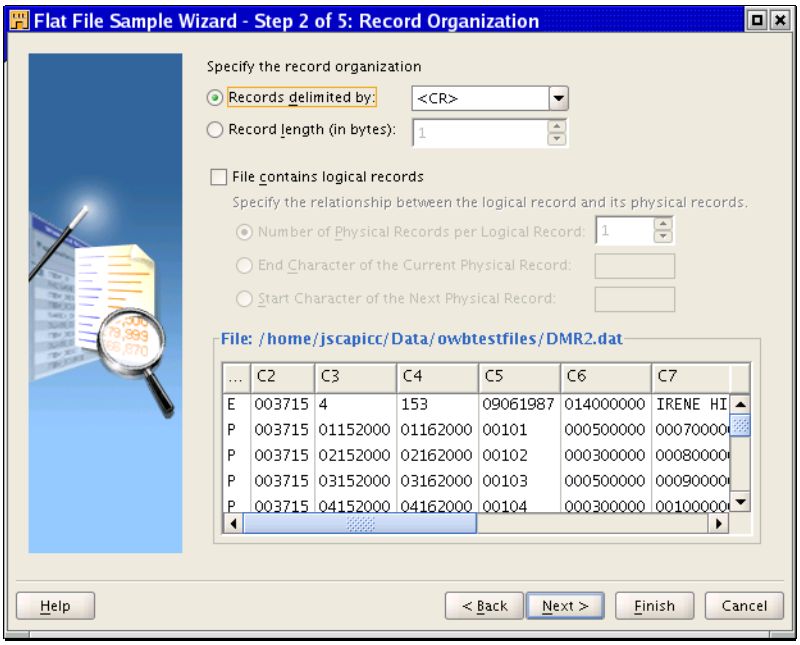
Step 1: Define the file and character set

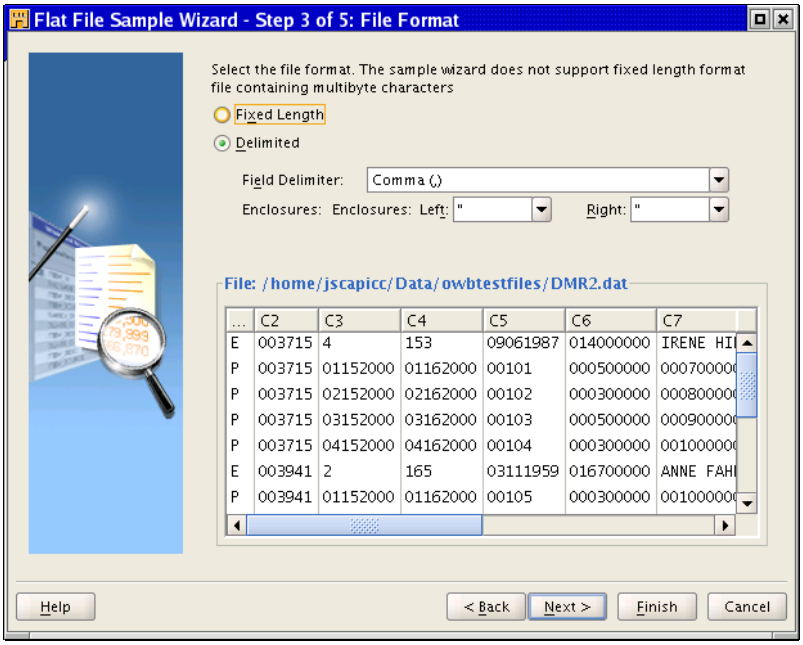
You will notice that the sample wizard recognizes the file as a CSV file and displays the fields formatted in columns to help simplify the file definition.



Steps 2 & 3: Identify the Record Organization and File Format

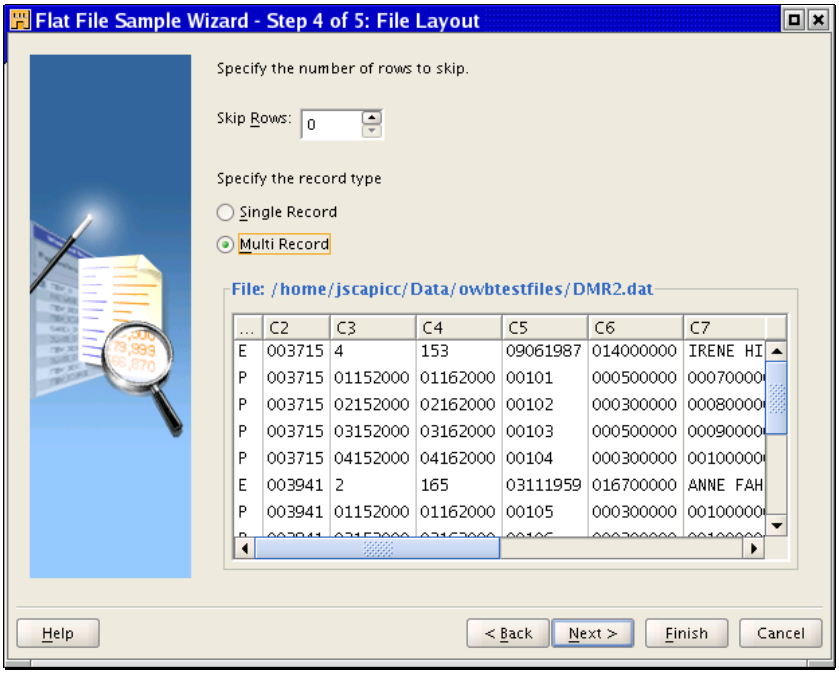
The records in the file are terminated with a CR, so the default record organization can be used. The fields are delimited with a comma, so the default file format can also be used.

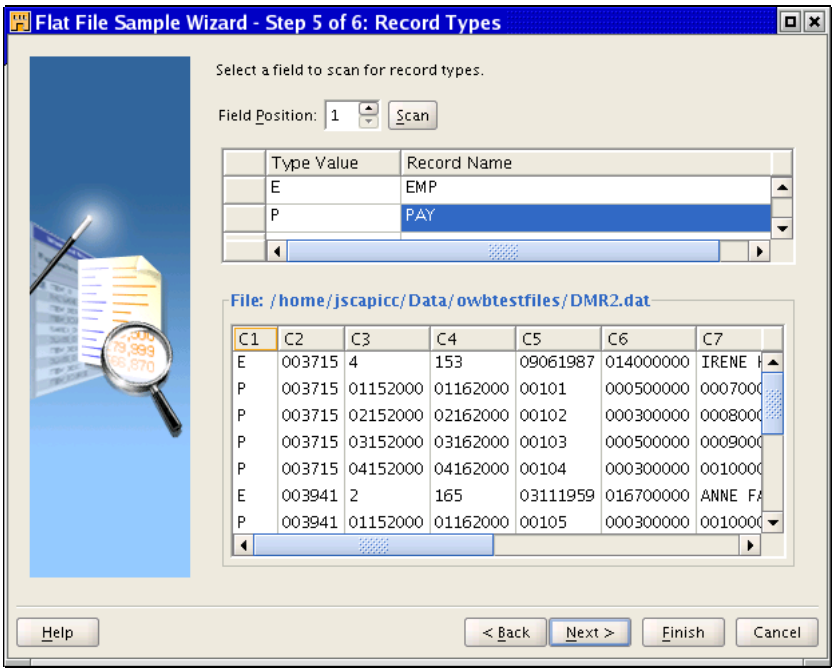




Steps 5 & 6: Identify the File Layout and Record Types

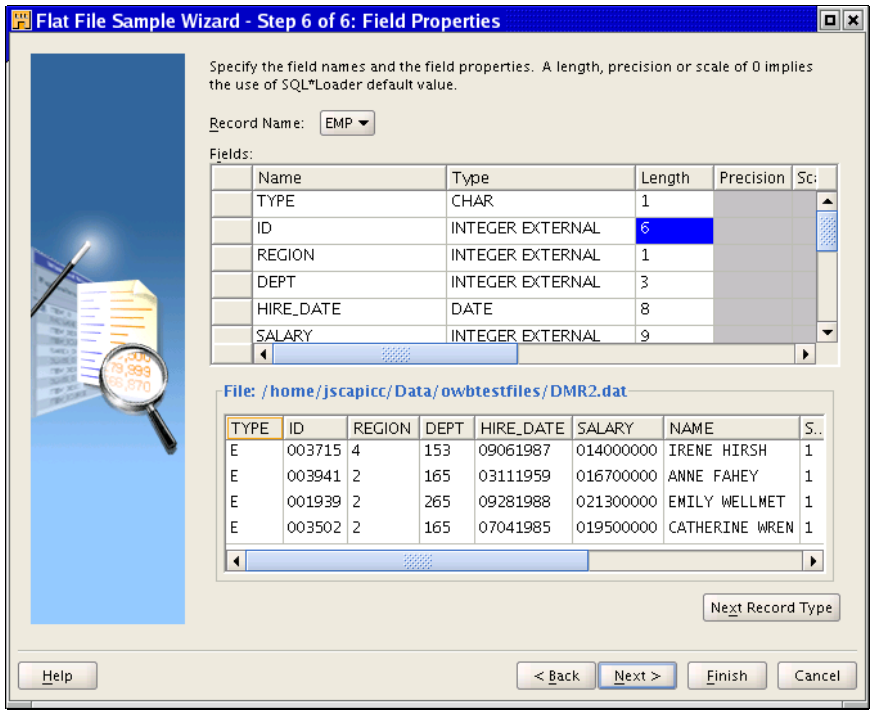
The file is identified as a multi-record file. Scanning the first position for record types, produces a list of record types in the file.

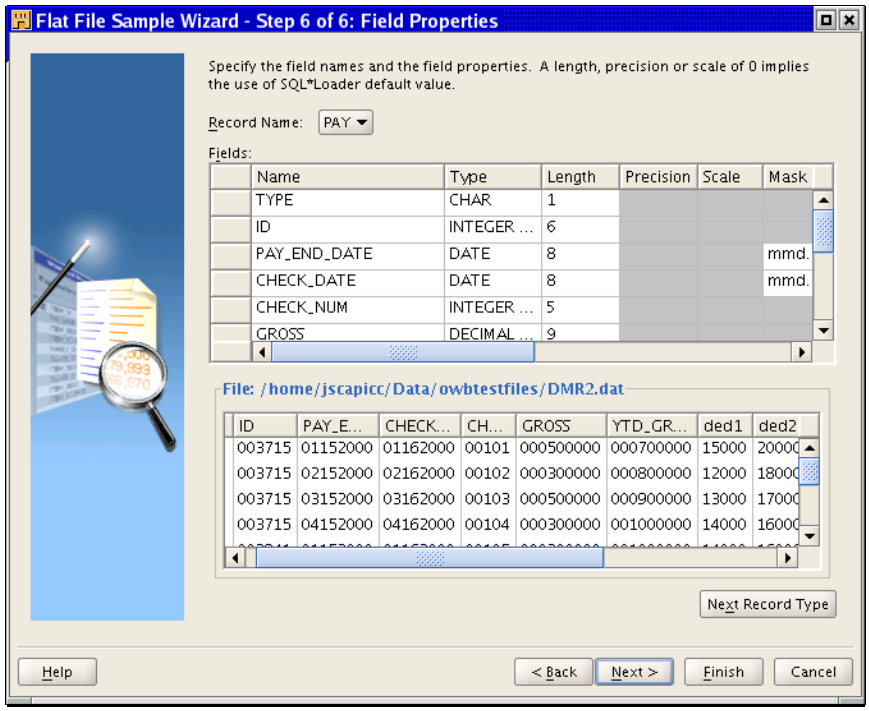




Steps 5 & 6: Identify the Field Properties for both records

Using the COBOL to SQL*Loader Table enter the names, datatypes, lengths and date masks for the fields in both records.

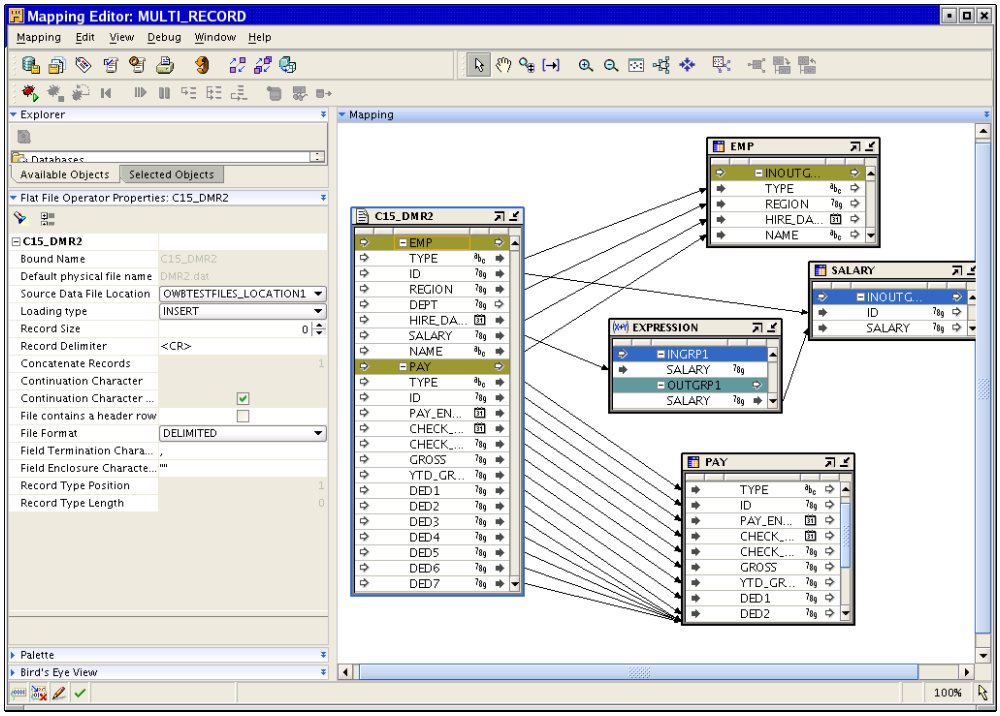




II. Create and Configure a Mapping to the Target Tables.

1. Create a SQL*Loader mapping to the target tables

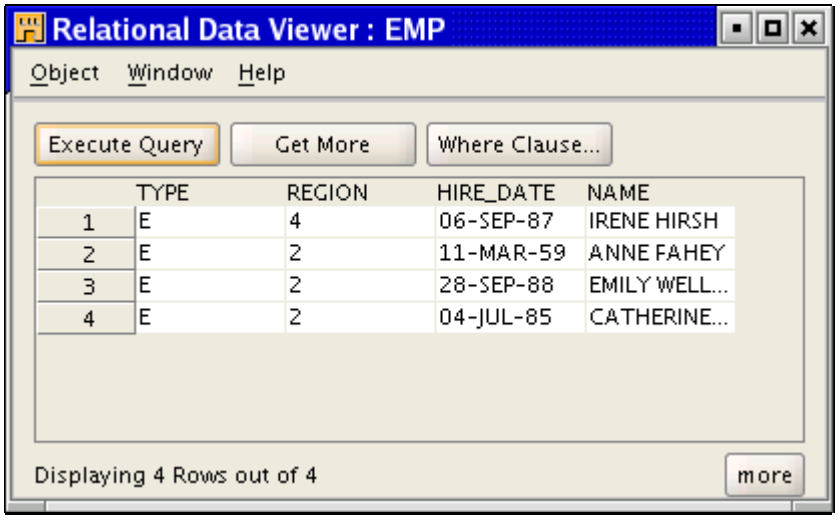
The map below has a file operator as a source loading into three target tables. The file operator automatically groups the two records defined in the file into the two mapping operator groups shown below. Based on this division on groups, the generated code for SQL*Loader will identify the records using the data in the first field. In this mapping, the EMP record type is loaded into two different tables. When loading the file into a table, the data can be transformed using expressions and functions as is done here with the SALARY field to adjust the decimal positions.



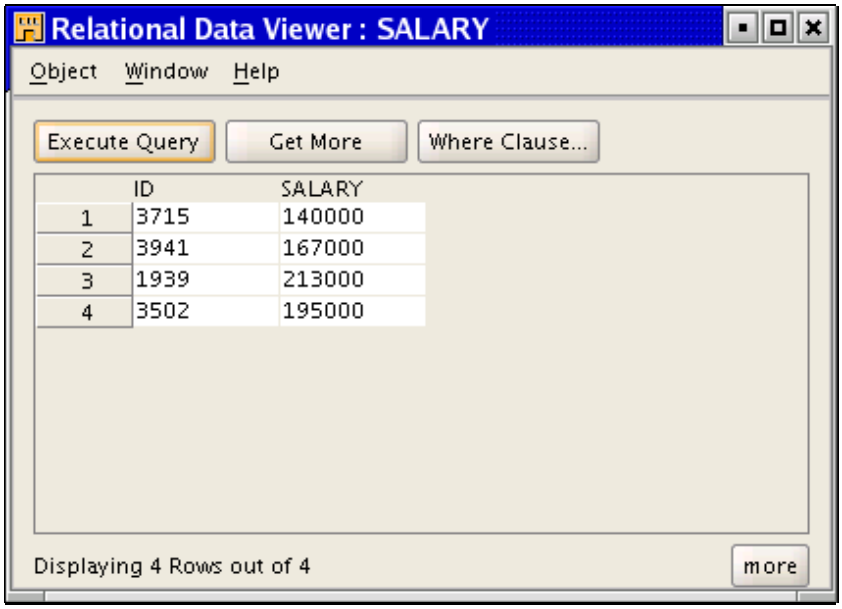
IV. Deploy and Execute.

Deploy the tables and mapping to create them in the target database. When the mapping is executed, the EMP, SALARY and PAY tables are loaded with the converted data.

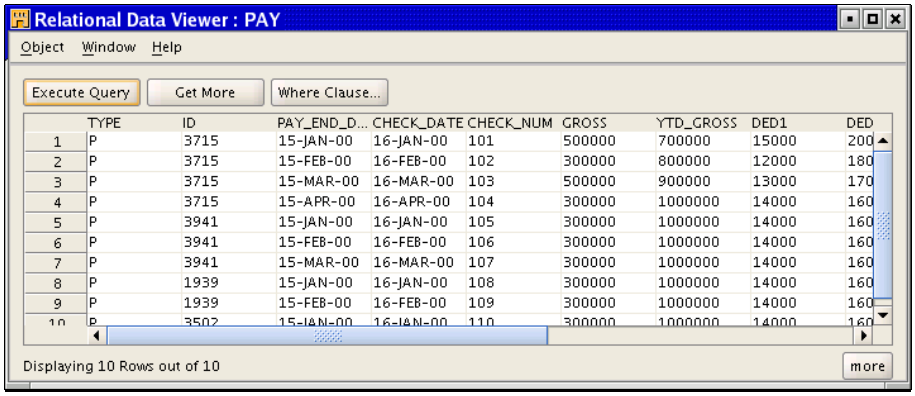
EMP Table Data



SALARY Table Data



PAY Table Data



Case Study 4: Load Table from a File Containing Internal Format Data

This case study demonstrates defining a file that contains binary and packed data as well as ebcdic external data. It loads the file into a table using SQL*Loader.

I. Analyze the Copybook

The table below illustrates a COBOL copybook definition that contains EBCDIC character data, binary data and packed decimal data. The copybook has a group field EMP-HIRE-DATE that contains subordinate fields for month, day and year. You will notice that we have decided to use only the group field and are mapping it to a DATE field. The subordinate fields for month, day and year are ignored. We will use SQL*Loader features to cleanse the data by specifying DEFAULTIF and NULLIF processing. For example, the NULLIF clause on EMP-NAME indicates that if EMP-NAMES contains blanks, NAME will be loaded with a NULL value instead of blanks.

COBOL to SQL*Loader Table

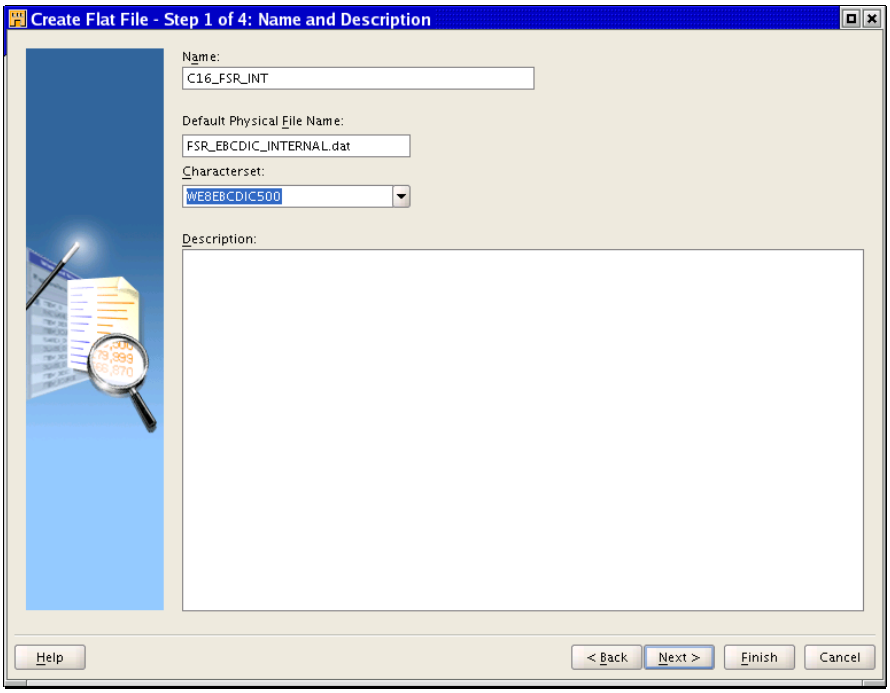
COBOL Copybook	**SQL*Loader Field name**	**Length**	**SQL*Loader Type**
01 EMPLOYEE-RECORD.			
05 EMP-ID PIC 9(5) COMP.	ID	2	INTEGER(2)
05 EMP-REGION PIC 9.	REGION	1	INTEGER EXTERNAL(1)
05 EMP-DEPT PIC 999.	DEPT	3	INTEGER EXTERNAL(3)
05 EMP-HIRE-DATE.	HIRE_DATE	8	DATE(8) MMDDYYYY DEFAULTIF EMP_HIRE_DATE= BLANKS
10 EMP-HIRE-MM PIC 99.			
10 EMP-HIRE-DD PIC 99.			
10 EMP-HIRE-YYYY PIC 9999.			
05 EMP-SALARY PIC 9(7)V99 COMP-3.	SALARY	9	DECIMAL(9,2)
05 EMP-NAME PIC X(15).	NAME	15	CHAR(15) NULLIF EMP_NAME= BLANKS

II. Define and Configure the Metadata

1. Define the flat file using the Create Flat File Wizard.

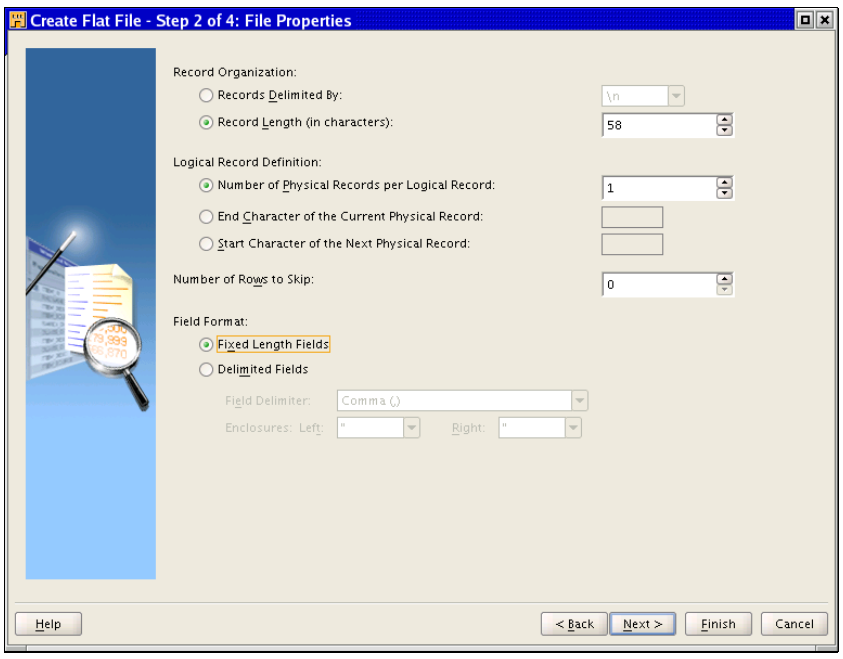
Step 1: Define the file and character set

A name is given to the file definition and the character set is identified as WE8EBCDIC500. This allows SQL*Loader to recognize the data in the file as EBCDIC data. A default physical file name is provided at this time also.



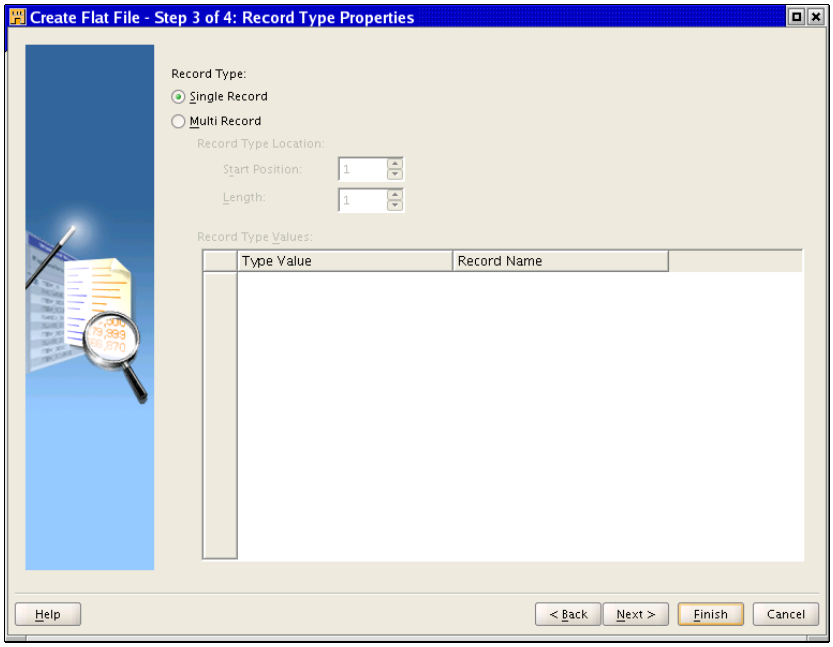
Step 2: File Properties

Define the file with a fixed record length and fixed length fields.



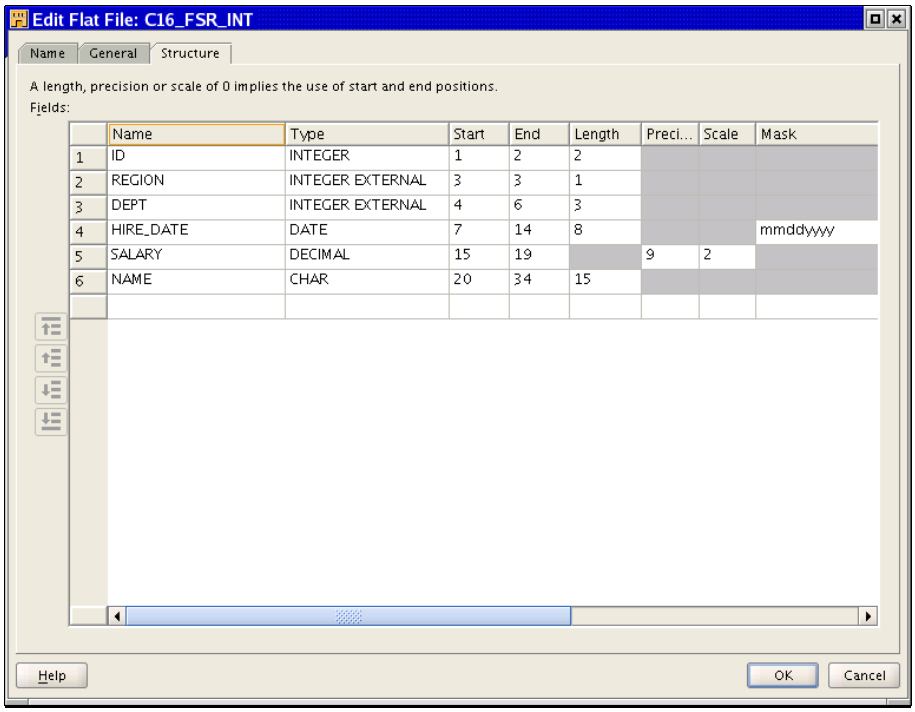
Step 3: Record Type Properties

Indicate that the file contains only one record type.



Steps 4: Identify the Field Properties for the record

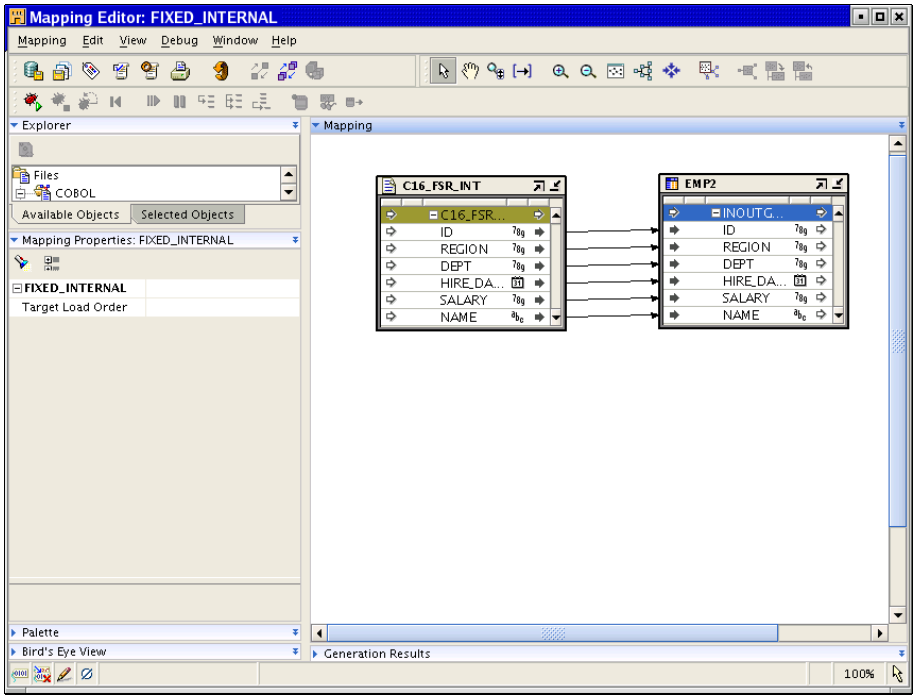
Using the COBOL to SQL*Loader Table enter the names, datatypes, lengths and date masks for the fields. Note that the ID field has been defined as INTEGER and the SALARY field is defined as decimal both internal data formats.



III. Create and Configure a Mapping to the Target.

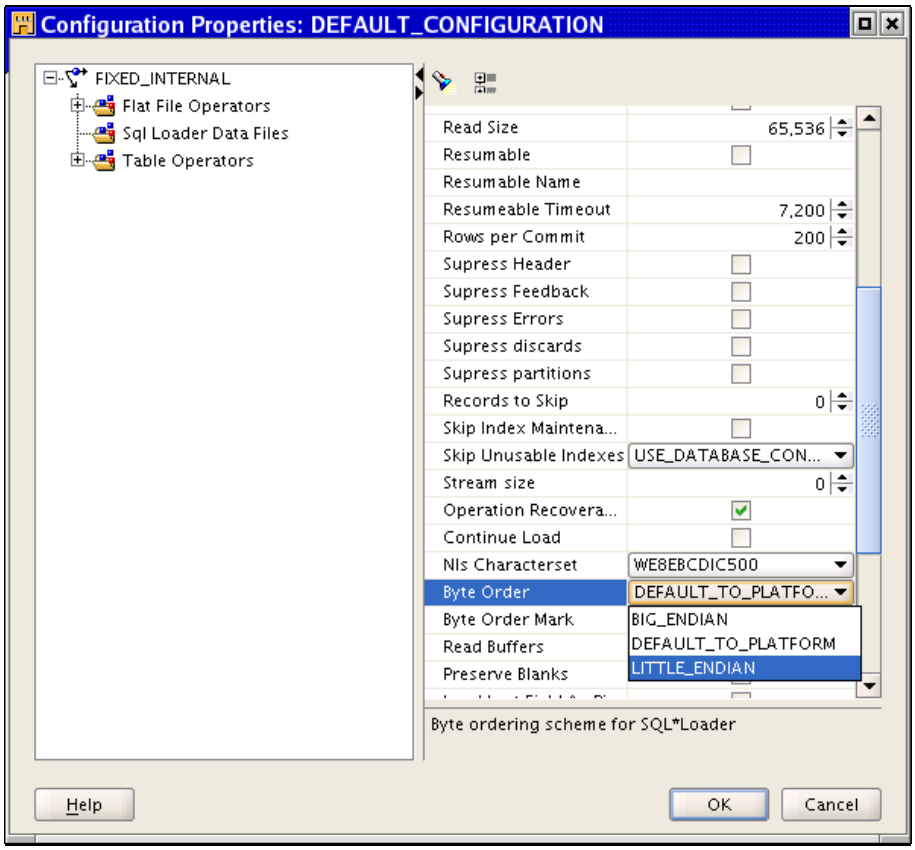
1. Create a SQL*Loader mapping to the target table

We have created a simple map to load the data from the file into a single table using SQL*Loader. SQL*Loader handles the character set and data conversion automatically.



2. Configure the mapping

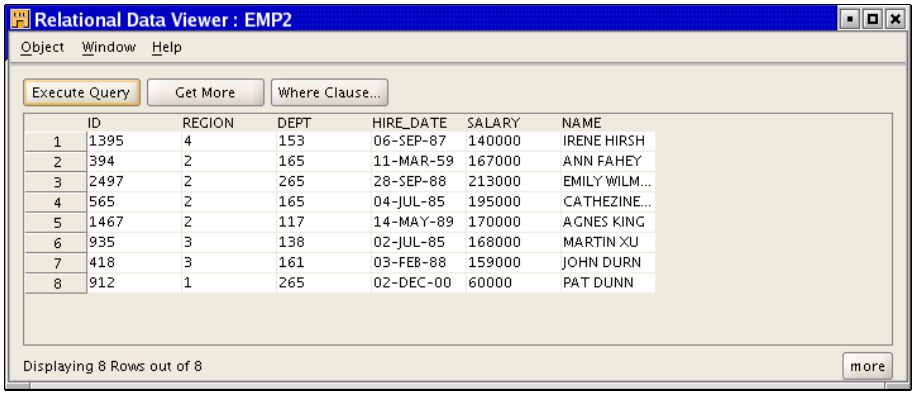
Configure the mapping if desired to identify file locations, data options or tuning options. When loading binary data, you may need to set the byte order if the file was created on a platform that has an internal representation for binary data different from the loading platform. Byte order options can be set by configuring the mapping as shown below.



IV. Deploy and Execute.

Deploy the table and mapping to define them to the target database. When the mapping is executed, the table is loaded with the converted data.

EMP2 Table Data



	ID	REGION	DEPT	HIRE_DATE	SALARY	NAME
1	1395	4	153	06-SEP-87	140000	IRENE HIRSH
2	394	2	165	11-MAR-59	167000	ANN FAHEY
3	2497	2	265	28-SEP-88	213000	EMILY WILM...
4	565	2	165	04-JUL-85	195000	CATHEZINE...
5	1467	2	117	14-MAY-89	170000	AGNES KING
6	935	3	138	02-JUL-85	168000	MARTIN XU
7	418	3	161	03-FEB-88	159000	JOHN DURN
8	912	1	265	02-DEC-00	60000	PAT DUNN

SUMMARY

Warehouse Builder 10*g* Release 2 delivers a set of tools for integrating data files including COBOL legacy data into your Oracle platform. The examples provided show easy loading of data using COBOL copybooks.

The data loading and external table access features provide powerful and easy to use facilities for accessing data files. By leveraging the power of SQL*Loader for extraction, transformation and loading into the Oracle database, bulk extraction is fast and straightforward. By using external tables, the full capability of SQL access and transformation is provided for your files.

When migrating legacy data or loading legacy data as part of other data integration programs, Warehouse Builder should be your obvious choice for the task at hand.

Oracle Warehouse Builder 10*g*R2 – Integrating COBOL based legacy data
March 2007
Author: Joyce Scapicchio

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com