Oracle® Migration Workbench

Reference Guide for Informix Dynamic Server 7.3 Migrations

Release 9.2.0 for Microsoft Windows 98/2000 and Microsoft Windows NT

March 2002

Part Number: A97251-01

This reference guide describes how to migrate from Informix Dynamic Server to Oracle9*i* or Oracle8*i* database.

ORACLE®

Oracle Migration Workbench Reference Guide for Informix Dynamic Server 7.3 Migrations, Release 9.2.0 for Microsoft Windows 98/2000 and Microsoft Windows NT

Part Number: A97251-01

Contents

3 Triggers, Packages, and Stored Procedures

4 Distributed Environments

5 The ESQL/C to Oracle Pro*C Converter

6 Disconnected Source Model Loading

A Code Samples

Index

Send Us Your Comments

Oracle Migration Workbench Reference Guide for Informix Dynamic Server 7.3 Migrations, Release 9.2.0 for Microsoft Windows 98/2000 and Microsoft Windows NT

Part Number: A97251-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Email - infomwb_ww@oracle.com
- FAX - +353-1-803-1899
- Postal service:
  Documentation Manager
  Migration Technology Group
  Oracle Corporation
  Eastpoint Business Park
  Dublin 3
  Ireland

Preface

The Oracle Migration Workbench Reference Guide for Informix Dynamic Server 7.3 Migrations provides detailed information about migrating a database from Informix Dynamic Server to Oracle9*i* or Oracle8*i*. It is a useful guide regardless of the conversion tool you are using to perform the migration, but the recommended tool for such migrations is Oracle Migration Workbench (Migration Workbench). This reference guide describes several differences between Informix Dynamic Server and Oracle and outlines how those differences are handled by the Migration Workbench during the conversion process.

This chapter contains the following sections:

Audience

This guide is intended for anyone who is involved in converting an Informix Dynamic Server database to Oracle using the Migration Workbench.

What You Should Already Know

You should be familiar with relational database concepts and with the operating system environments under which you are running Oracle and Informix Dynamic Server.

How This Reference Guide is Organized

This reference guide is organized as follows:

Chapter 1, "Overview"

Introduces the Migration Workbench and outlines features of this tool.

Chapter 2, "Oracle and Informix Dynamic Server Compared"

Contains detailed information about the differences between data types, data storage concepts, and schema objects in Informix Dynamic Server and Oracle.

Chapter 3, "Triggers, Packages, and Stored Procedures"

Introduces triggers and stored procedures, and compares T-SQL and PL/SQL language elements and constructs in Informix Dynamic Server and Oracle.

Chapter 4, "Distributed Environments"

Describes when and why distributed environments are used, and discusses application development tools.

Chapter 5, "The ESQL/C to Oracle Pro*C Converter"

Describes how to use the ESQL/C to Oracle Pro*C Converter and includes an example conversion.

Chapter 6, "Disconnected Source Model Loading"

Describes how to perform a disconnected source model load, using delimited flat files containing schema metadata.

How to Use This Reference Guide

Every reader of this reference guide should read Chapter 1, "Overview" as that chapter provides an introduction to the concept and terminology of the Migration Workbench.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

http://www.oracle.com/accessibility/

Accessibility of Code Examples in Documentation

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Related Documentation

For more information, see these Migration Workbench resources:

- Oracle Migration Workbench Frequently Asked Questions (FAQ)
- Oracle Migration Workbench Release Notes
- Oracle Migration Workbench Online Help

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and you can do it at:

http://otn.oracle.com/membership/index.htm

If you already have a user name and password for OTN, then you can go directly to the Migration Workbench documentation section of the OTN Web site at:

http://otn.oracle.com/tech/migration/workbench

Conventions

This section describes the conventions used in the text and code examples of the this documentation. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold type indicates GUI options. It also indicates terms that are defined in the text or terms that appear in a glossary, or both.	The C datatypes such as **ub4**, **sword**, or **OCINumber** are valid. When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles, emphasis, syntax clauses, or placeholders.	*Reference Guide* Run U*old_release*.SQL where *old_release* refers to the release you installed prior to upgrading.
UPPERCASE monospace (fixed-width font)	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, user names, and roles.	You can specify this clause only for a NUMBER column. You can back up the database using the BACKUP command.
lowercase monospace (fixed-width font)	Lowercase monospace typeface indicates executables and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, user names and roles, program units, and parameter values.	Enter sqlplus to open SQL*Plus. The department_id, department_name, and location_id columns are in the hr.departments table.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
Square Brackets [ ]	Indicates that the enclosed arguments are optional. Do not enter the brackets.	`DECIMAL (digits [ , precision ])`
Curly Braces { }	Indicates that one of the enclosed arguments is required. Do not enter the braces.	`{ENABLE | DISABLE}`
Vertical Line |	Separates alternative items that may be optional or required. Do not type the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
`Ellipses ...`	Indicates that the preceding item can be repeated. You can enter an arbitrary number of similaritems. In code fragments, an ellipsis means that code not relevant to the discussion has been omitted. Do not type the ellipsis	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , coln FROM employees;`
`Italics`	Indicates variables that you must supply particular values.	`CONNECT SYSTEM/`*`system_password`*

`UPPERCASE`	Uppercase text indicates case-insensitive filenames or directory names, commands, command keywords, initializing parameters, data types, table names, or object names. Enter text exactly as spelled; it need not be in uppercase	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase words in example statements indicate words supplied only for the context of the example. For example, lowercase words may indicate the name of a table, column, or file.	`SELECT last_name, employee_id FROM employees;` `sqlplus hr/hr`

1

Overview

This chapter introduces the Oracle Migration Workbench (Migration Workbench) under the following headings:

Introduction

The Migration Workbench is a tool that simplifies the process of migrating data and applications from an Informix Dynamic Server 7.3 environment to an Oracle9*i*, Oracle8*i*, Oracle8*i* Appliance, or Oracle8 destination database. The Migration Workbench allows you to quickly and easily migrate an entire application system, that is the database schema including triggers, views, and stored procedures, in an integrated, visual environment.

Product Description

The Migration Workbench allows you to migrate an Informix Dynamic Server database to an Oracle9*i*, Oracle8*i*, Oracle8*i* Appliance, or Oracle8 database. The Migration Workbench employs an intuitive and informative user interface and a series of wizards to simplify the migration process. To ensure portability, all components of the Migration Workbench are written in Java.

The Migration Workbench uses a repository to store migration information. This allows you to query the initial state of the application before migration. By initially

loading the components of the application system that you can migrate into a repository, you can work independently of the production application.

Furthermore, the Migration Workbench saves useful dependency information about the components you are converting. For example, the Migration Workbench keeps a record of all the tables accessed by a stored procedure. You can then use this information to understand the impact of modifying a given table.

Features

The Migration Workbench is a wizard-driven tool. It is composed of core features and Informix Dynamic Server migration specific features. The Migration Workbench allows you to:

- Migrate a complete Informix Dynamic Server database to Oracle9*i*, Oracle8*i*, Oracle8*i* Appliance, or Oracle8.
- Migrate groups, users, tables, primary keys, foreign keys, unique constraints, indexes, rules, check constraints, views, triggers, stored procedures, and privileges to Oracle.
- Migrate multiple Informix Dynamic Server source databases to a single Oracle database.
- Customize the parser for stored procedures, triggers, or views.
- Generate the Oracle SQL*Loader and Informix Dynamic Server Unload scripts for offline data loading.
- Display a representation of the source database and its Oracle equivalent.
- Generate and view a summary report of the migration.
- Customize users, tables, indexes, and tablespaces.
- Customize the default data type mapping rules.
- Create ANSI-compliant names.
- Automatically resolve conflicts such as Oracle reserved words.
- Remove and rename objects in the Oracle Model.

Glossary

The following terms are used to describe the Migration Workbench:

Application System is the database schema and application files that have been developed for a database environment other than Oracle, for example, Informix Dynamic Server.

Capture Wizard is an intuitive wizard that takes a snapshot of the data dictionary of the source database, loads it into the Source Model, and creates the Oracle Model.

Dependency is used to define a relationship between two migration entities. For example, a database view is dependent upon the table it references.

Destination Database is the Oracle database to which the Migration Workbench migrates the data dictionary of the source database.

Migration Component is part of an application system that you can migrate to an Oracle database. Examples of migration components are tables and stored procedures.

Migration Entity is an instance of a migration component. The table EMP is a migration entity belonging to the table MIGRATION COMPONENT.

Migration Wizard is an intuitive wizard that helps you migrate the source database to Oracle.

Migration Workbench is the graphical tool that allows migration of an application system to an Oracle database environment.

Navigator Pane is the part of the Migration Workbench User Interface that contains the tree views representing the Source Model and the Oracle Model.

Oracle Model is a a series of Oracle tables that is created from the information in the Source Model. It is a visual representation of how the source database looks when generated in an Oracle environment.

Properties Pane is the part of the Migration Workbench User Interface that displays the properties of a migration entity that has been selected in one of the tree views in the Navigator Pane.

Progress Window is the part of the Migration Workbench User Interface that contains informational, error, or warning messages describing the progress of the migration process.

Software Development Kit (SDK) is a set of well-defined application programming interfaces (APIs) that provide services that a software developer can use.

Source Database is the database containing the data dictionary of the application system being migrated by the Migration Workbench. The source database is a database other than Oracle, for example, Informix Dynamic Server.

Source Model is a replica of the data dictionary of the source database. It is stored in the Oracle Migration Workbench Repository and is loaded by the Migration Workbench with the contents of the data dictionary of the source database.

Workbench Repository is the area in an Oracle database used to store the persistent information necessary for the Migration Workbench to migrate an application system.

2

Oracle and Informix Dynamic Server Compared

This chapter contains information comparing the Informix Dynamic Server database and the Oracle database. It includes the following sections:

Database Security

This section includes information on issues of security with Informix Dynamic Server databases and Oracle databases.

Database Authentication

A fundamental difference between Informix Dynamic Server and Oracle is database user authentication. Informix Dynamic Server users are maintained and authenticated by the host operating system, whereas Oracle users are maintained by the database and can use several methods of authentication, usually through the database.

A user can connect to an Informix Dynamic Server database server through the operating system login information, however access to the databases the server supports is restricted by the sysuser table. The sysuser is maintained by each database and the database administrator.

Database as a Logical Partition

Multiple databases on a single Informix Dynamic Server database server are migrated to a single Oracle database. Schemas in different databases are owned by the same user.

Users

Informix Dynamic Server has two special users, `informix` and `root`. A description of these users is as follows:

User name	Description
`informix`	Informix Dynamic Server software owner
`root`	Operating system super user

These users have database administrator access to all the databases supported by the Informix Dynamic Server database server. The two user names do not have to be listed in the sysusers table for any database. The Informix Dynamic Server plug-in creates the two user names in the Oracle database. Another special Informix Dynamic Server database user, which does not have to be an operating system user, is `public`.

You can grant the `public` database user system and object privileges and its database level privileges are entered in the sysusers table. The privileges granted to public are automatically available to every other database user. The Informix Dynamic Server plug-in migrates all the object privileges.

Oracle has the concept of a database group or role, where you can grant privileges. These privileges are made available to all other users in the database. It is also called `PUBLIC`.

The difference is that `public` is not listed as a database user and you cannot grant connect system privilege to `public` to enable any user logged on to the host operating system gain access to the database.

All Informix Dynamic Server object level privileges granted to `public` are migrated to Oracle. None of the three Informix Dynamic Server database privileges granted to public are migrated.

The Informix Dynamic Server plug-in could not detect what operating system users had access to the database. The Informix Dynamic Server plug-in only creates Oracle users for the users listed in sysusers in each of the Informix Dynamic Server databases selected for migration. Therefore, if you rely on granting connect,

resource, or even dba to `public` as a method of allowing operating system users access to the database, then you must explicitly grant each of those users the appropriate database level privilege.

Schema Migration

The schema contains the definitions of the tables, views, indexes, users, constraints, stored procedures, triggers, and other database-specific objects. Most relational databases work with similar objects.

The schema migration topics discussed here include the following:

- Schema Object Similarities
- Schema Object Names
- Informix Dynamic Server Database-level Privileges
- Migrating Multiple Databases
- Table Design Considerations
- Schema Migration Limitations for Informix Dynamic Server

Schema Object Similarities

There are many similarities between schema objects in Oracle and Informix Dynamic Server. However, some schema objects differ between these databases. For specific information about schema objects, see the SQL Statements topic within the Oracle9*i* SQL Reference, Release 1 (9.0.1).

Table 2–1 shows the differences between Oracle and Informix Dynamic Server.

Table 2–1 Schema Objects in Informix Dynamic Server and Oracle

Oracle	Informix Dynamic Server
Database	Database
Schema	Schema
Tablespace	Dbspace
User	User
Role	Role
Table	Table

Table 2–1 Schema Objects in Informix Dynamic Server and Oracle (Cont.)

Oracle	Informix Dynamic Server
Temporary tables	Temporary tables
Index	Cluster Index
Check constraint	Check constraint
Column default	Column default
Unique key	Unique key
Primary key	Primary key
Foreign key	Foreign key
Index	Index
PL / SQL Procedure	SPL Procedure
PL / SQL Function	SPL Function
Packages	N / A
AFTER triggers	Triggers
BEFORE triggers	Triggers
Triggers for each row	Triggers for each row
Synonyms	Synonyms
Sequences	SERIAL datatype for a column
Snapshot	N / A
View	View

Schema Object Names

Reserved words differ between Oracle and Informix Dynamic Server. Many Oracle reserved words are valid object or column names in Informix Dynamic Server. Use of reserved words as schema object names makes it impossible to use the same names across databases. The Migration Workbench appends an underscore (_) to the name of an Informix Dynamic Server object that is an Oracle reserved word.

Neither Oracle nor Informix Dynamic Server is case-sensitive with respect to object names. Object names in Informix Dynamic Server are stored as lower case, while Oracle schema object names are stored as upper case.

Choose a schema object name that is the following:

- unique by case
- by at least one other characteristic

Ensure that the object name is not a reserved word from either database.

For a list of Oracle reserved words, see the Oracle9i SQL Reference, Release 1 (9.0.1).

In non-ANSI-Compliant Informix Dynamic Server databases, schema object names are required to be unique across users. This behavior in Oracle is similar to Informix Dynamic Server ANSI-Compliant mode databases. Different users can create objects with the same name without any conflicts.

Informix Dynamic Server Database-level Privileges

For information on database-level privileges, see the Oracle Migration Workbench Release Notes.

Migrating Multiple Databases

The Migration Workbench supports the migration of multiple Informix Dynamic Server databases if they are on the same Informix Dynamic Server database server.

Table Design Considerations

This section discusses table design issues that you need to consider when converting Informix Dynamic Server databases to Oracle. This section includes the following:

- Data Types
- IMAGE and TEXT Data Types (Binary Large Objects)
- Check Constraints

Data Types

This section outlines conversion considerations for the following data type:

- DATETIME Data Types

DATETIME Data Types The Datetime precision in Informix Dynamic Server is to 5 decimal places, 1/100000th of a second. Oracle9*i* has a new data type TIMESTAMP which has a precision of 1/100000000th of a second. Oracle also has a DATE data type that stores date and time values accurate to one second. The Migration Workbench has a default mapping to the DATE data type.

For applications that require finer date/time precision than seconds, the TIMESTAMP data type should be selected for the datatype mapping of date data types in Informix Dynamic Server. The database stores point-in-time values for DATE and TIME data types.

As an alternative, if an Informix Dynamic Server application uses the DATETIME column to provide unique IDs instead of point-in-time values, you can replace the DATETIME column with a SEQUENCE in the Oracle schema definition.

In the following examples, the original design does not allow the DATETIME precision to exceed seconds in the Oracle table. The examples assume that the DATETIME column is used to provide unique IDs. If millisecond precision is not required, the table design outlined in the following example is sufficient:

Table 2–2 Original Table Design

Informix Dynamic Server	Oracle
CREATE TABLE example_table (datetime_column datetime not null, text_column text null, varchar_column varchar(10) null)	CREATE TABLE example_table (datetime_column date not null, text_column clob null, varchar_column varchar2(10) null)

The design shown in Table 2–3 allows you to insert the value of the sequence into the integer_column. This allows you to order the rows in the table beyond the allowed precision of one second for DATE data type fields in Oracle. If you include this column in the Informix Dynamic Server table, you can keep the same table design for the Oracle database.

Table 2–3 Revised Table Design

Informix Dynamic Server	Oracle
CREATE TABLE example_table (datetime_column datetime not null, integer_column int null, text_column text null, varchar_column varchar(10) null)	CREATE TABLE example_table (datetime_column date not null, integer_column number null, text_column clob null, varchar_column varchar2(10) null)

For Informix Dynamic Server, the value in the integer_column is always NULL. For Oracle, the value for the field integer_column is updated with the next value of the sequence.

Create the sequence by issuing the following command:

```
CREATE SEQUENCE datetime_seq
```

Values generated for this sequence start at 1 and increment by 1.

Many applications do not use DATETIME values as UNIQUE IDs, but still require the date/time precision to be higher than seconds. For example, the timestamp of a scientific application may have to be expressed in milliseconds, microseconds, and nanoseconds. The precision of the Informix Dynamic Server DATETIME data type is 1/100000th of a second; the precision of the Oracle DATE data type is one second. The Oracle TIMESTAMP data type has a precision to 1/100000000th of a second. However, the precision recorded is dependent on the operating system.

IMAGE and TEXT Data Types (Binary Large Objects)

The physical and logical storage methods for BYTE and TEXT data in Informix Dynamic Server is similar to Oracle BLOB data storage. A pointer to the BYTE or TEXT data is stored with the rows in the table while the IMAGE or TEXT data is stored separately. This arrangement allows multiple columns of IMAGE or TEXT data per table. Oracle may store IMAGE data in a BLOB type field and TEXT data may be stored in a CLOB type field. Oracle allows multiple BLOB and CLOB columns per table. BLOBs and CLOBs may or may not be stored in the row

depending on their size. If LONG or LONG RAW appears, only one column is allowed.

If the Informix Dynamic Server TEXT column is such that the data never exceeds 4000 bytes, convert the column to an Oracle VARCHAR2 data type column instead of a CLOB column. An Oracle table can define multiple VARCHAR2 columns. This size of TEXT data is suitable for most applications.

Check Constraints

You can define check constraints in a CREATE TABLE statement or an ALTER TABLE statement in Informix Dynamic Server. You can define multiple check constraints on a table. A table-level check constraint can reference any column in the constrained table. A column can have only one check constraint. A column-level check constraint can reference only the constrained column. These check constraints support complex regular expressions.

Oracle defines check constraints as part of the CREATE TABLE or ALTER TABLE statements. A check constraint is defined at the TABLE level and not at the COLUMN level. Therefore, it can reference any column in the table. Oracle, however, does not support complex regular expressions.

Schema Migration Limitations for Informix Dynamic Server

The schema migration limitations are separated into the following categories:

- Dbspaces
- Mapping for Informix Dynamic Server Database Level Privileges to Oracle System Privileges
- Defaults
- Indexes
- Check Constraints
- Check Constraint Owners

Dbspaces The Migration Workbench captures all dbspaces on the Informix Dynamic Server, even though you may not require all dbspaces. You can delete the dbspaces, as appropriate, from the Source Model. If you delete the root dbspace, the next time you start the Migration Workbench, the sizing information shows up as zero (0).

Mapping for Informix Dynamic Server **Database Level Privileges to Oracle System Privileges** Before migration ensure that the sysmaster database exists for the database server you are migration from. Oracle does not support the migration of Default DATETIME literal.

The Informix Dynamic Server CONNECT privilege maps to the following Oracle system privileges:

- Create Session
- Alter Session
- Create View
- Create Any View
- Create Synonym
- Create Any Synonym
- Create Public Synonym
- Drop Public Synonym
- Create Cluster
- Create Database Link
- Create Sequence
- Unlimited Tablespace
- Create Table
- Create Procedure
- Create Trigger

The Informix Dynamic Server RESOURCE privilege maps to the following Oracle system privileges:

- Create Session Operator
- Alter Session
- Create Any View
- Create View
- Create Synonym
- Create Any Synonym

- Create Public Synonym
- Drop Public Synonym
- Create Cluster
- Create Database Link
- Create Sequence
- Unlimited Tablespace
- Create Table
- Create Procedure
- Create_trigger
- Create_type
- Create_indextype
- Create_operator

The Informix Dynamic Server DBA privilege maps to the Oracle system All Privileges privilege.

Note: nformix database users granted the CONNECT database level privilege do not have the privilege to create tables, procedures or triggers. However, CONNECT users may be the owner of these object types, created for them by more privileged users.

The Migration Workbench creates schema objects connected to the Oracle database as the owner of the object. Any attempt to create an object without the appropriate privilege generates an error. Therefore, the Informix Dynamic Server plug-in maps users, that have the Informix Dynamic Server CONNECT privilege, to Oracle with the system privileges to create tables, procedures and triggers. To revoke the privileges from the users after migration execute the following:

```
revoke create table, create procedure, create
trigger from <user>;
```

You cannot migrate Informix Dynamic Server DBA users with the WITH ADMIN OPTION for any of the system privileges. The Informix Dynamic Server DBA cannot grant the privileges to other users.

Defaults The following limitations apply to the Defaults schema object:

- An Informix Dynamic Server user name can be up to 8 characters long. Oracle users can be up to 30 characters long. The Informix Dynamic Server `USER` system function maps to the Oracle `USER` system function. However, if you use the Oracle `USER` function as the default, the addition of the default fails because the column definition, such as `CHAR(8)`, in Informix Dynamic Server is too small for Oracle `USER` names. Change the length of the column in the Oracle Model to `CHAR(30)` before migrating.
- You cannot delete defaults in the Source Model although it appears this is possible within the Migration Workbench. For more information, see Bug 1642519 in the Oracle Bug Database.
- Defaults for `INTERVAL` columns that are not a number and are migrated to CHAR(30) fail during migration.
- Defaults for `DATETIME` columns that you do not specify in the YYYY-MM-DD HH:MI:SS format fail to migrate properly.

Indexes Migrate indexes, then migrate unique constraints and primary key constraints. If you do not migrate the schema objects in this order, a system generated index is created for unique constraints and primary keys. This causes the `CREATE INDEX` statement to fail.

Check Constraints Check constraints within Informix Dynamic Server are not parsed to Oracle syntax. The user should ensure that they can successfully execute all check constraints listed in the Oracle Model. For more information, see Bug 1644309 in the Oracle Bug Database.

Check Constraint Owners If a user creates a check constraint on another users' table, the check constraint is created in the Oracle Model and the check constraint is owned by the owner of that table.

Data Types

This section provides descriptions of the differences in data types used by Informix Dynamic Server and Oracle databases. This section contains the following information:

- A table showing the base Informix Dynamic Server data types available and how they are mapped to Oracle data types
- Recommendations based on the information listed in the table

Table 2–4 Data Types Summary Table

Informix Dynamic Server	Description	Oracle	Comments
INTEGER INT	Four-byte integer, 31 bits, and a sign. Stores whole numbers in the range -2,147,483,647 to +2,147,483,647	NUMBER(10)	You may wish to place a check constraint on columns of this type to enforce values between 2^31 and2^31.
SMALLINT	Two-byte integer, 15 bits, and a sign. Stores whole numbers in the range -32,767 to +32,767.	NUMBER(5)	You may wish to place a check constraint on columns of this type to enforce values between -2^15 and 2^15.
SERIAL	Stores a sequential INTEGER assigned automatically by the database server when a row is inserted.	NUMBER(10)	A Sequence and Trigger is created automatically to update the column that was originally SERIAL.
DECIMAL DECIMAL(p) floating point DEC DEC(p)	A floating point number with p digits of precision. If you omit p, p defaults to 16.	NUMBER	A floating point number with 38 digits of precision.
DECIMAL(p,s) fixed-point DEC (p,s)	A fixed point number with precision p and scale s.	NUMBER(p,s)	A fixed point number with precision p and scale s.
SMALLFLOAT REAL	Stores single-precision floating-point numbers corresponding to the float datatype in C.	FLOAT(63)	
FLOAT(p) DOUBLE PRECISION	Stores double-precision floating-point numbers corresponding to the double datatype in C. p specifies a precision, 1.14, however it is ignored.	FLOAT(126)	You may want to add a check constraint to constrain range of values. Also, you get different answers when performing operations on this type due to the fact that the Oracle NUMBER type is much more precise and portable than FLOAT.

Table 2–4 Data Types Summary Table (Cont.)

Informix Dynamic Server	Description	Oracle	Comments
CHAR(n) CHARACTER(n)	Fixed-length string of exactly n 8-bit characters, blank padded. 0< *n* < 32768	CHAR(n) if n <= 2000 VARCHAR2(n) if 2000 < n <= 4000 CLOB or LONG if n > 4000	Oracle CHAR can only hold up to 2000 bytes of data. Oracle VARCHAR2 can hold up to 4000 bytes of data. Oracle LONG can hold up to 2G of data, but there are many restrictions on LONG columns. Oracle CLOB can hold up to 4G.
VARCHAR(n)	Varying-length character string. 0 < *n* < 256.	VARCHAR2(n)	
TEXT	Stores any kind of text data, up to 2G. A table can contain more than one TEXT column.	CLOB LONG	The CLOB datatype can hold up to 4G of character data. A table can have more than one CLOB column. LONG has a limit of 2G but there are several restrictions on LONG columns.
BYTE	Stores any kind of binary data, up to 2G. A table can contain more than one BYTE column.	BLOB LONG RAW	The BLOB datatype can hold up to 4G of binary data. A table can have more than one BYPE column. LONG RAW can store binary data. has a limit of 2G but there are several restrictions on LONG columns.

Table 2–4 Data Types Summary Table (Cont.)

Informix Dynamic Server	Description	Oracle	Comments
DATETIME	Stores and instance in time expressed as a calendar date and time of day. It can be defined with qualifiers to specify the precision. For example: `DATETIME` largest_qualifier TO smallest_qualifier	DATE	The Informix Dynamic Server `DATETIME` data type has higher precision, YEAR to Fraction of Second, than the Oracle DATE data type, Year to Second. The fractional second information, if specified in the Informix Dynamic Server column definition, is lost in the migration.
	Qualifier Values YEAR MONTH DAY HOUR MINUTE SECOND FRACTION a decimal fraction of a second with up to five digits of precision.		The Oracle TIMESTAMP data type can also be used . It has a precision of 1/10000000th of a second.
DATE	DATE is stored internally as an integer equal to the number of days since December 31,1899.	DATE	Store a date and time, the time defaults to 12:00AM midnight.
INTERVAL		NUMBER(p)	Where p is the precision of the largest qualifier value.
MONEY(p,s)		NUMBER(p,s)	

Recommendations

You can map data types from Informix Dynamic Server to Oracle with the equivalent data types listed in Table 2–4. You can define how the base type is mapped to an Oracle type in the Data Type Mappings page in the Options dialog.

BYTE

The Informix Dynamic Server BYTE datatype stores any type of binary data and has a maximum limit of 2^31 bytes (2G). The comparable Oracle datatypes are LONG RAW and BLOB.

Oracle LONG RAW stores variable-length raw binary data field used for binary data up to 2G in length. Although you can insert RAW data as a literal in an INSERT statement (a hexidecimal character represents the bit pattern for every four bits of RAW data, 'CB' = 11001011), there are several restrictions on LONG and LONG RAW columns, for example, only one LONG columns is allowed per table and LONG columns can not be indexed. The LONG RAW datatype is provided for backward compatibility with existing applications. For new applications, Oracle recommends the use of BLOB and BFILE datatypes for large amounts for binary data.

Oracle9*i* and Oracle8*i* BLOB, and other Oracle9*i* and Oracle8*i* LOB types, CLOB, NCLOB, and BFILE, have a much greater storage capacity than LONG RAW, storing up to 4G of data and Oracle9*i* and Oracle8*i* tables can have multiple LOB columns.

Oracle LONGs support on sequential access, while Oracle9*i* and Oracle8*i* LOBs support random piece wise access. Although Oracle9*i* and Oracle8*i* SQL cannot directly manipulate LOBs, you can access LOBs from SQL through the Oracle9*i* and Oracle8*i* supplied DBMS_LOB PL/SQL package. The DBMS_LOB package provides many functions and procedures to append the contents of one LOB to another, compares contents or parts of contents, copies contents, reads from and writes to LOBs, and also returns part of a LOB from a given offset and length.

For example, with Informix Dynamic Server you can select any part of a BYTE column by using subscripts:

```
select cat_picture[1,75] from catalog where catalog_num = 10001;
```

A similar request in Oracle9*i* and Oracle8*i* follows:

```
blob_loc BLOB;
binchunk RAW;

SELECT cat_picture INTO blob_loc FROM catalog WHERE catalog_num = 10001;
binchunk := dbms_lob.substr(blob_loc, 75, 1);
```

CHAR(n)

The Informix Dynamic Server CHAR datatype stores any sequence of letters, numbers, and symbols. It can store single byte and multibyte characters. A character column has a maximum length of *n* bytes, where 1<=*n*<=32767. If *n* is not specified, 1 is the default. If a character string is less than *n* bytes, then the string is extended with spaces to make up the length. If the string value is longer than *n* bytes, the string is truncated without raising an error.

The comparable Oracle datatypes are:

- CHAR(n), fixed-length field, up to 2000 bytes in length
- VARCHAR2(n), variable-length character data, up to 4000 bytes
- LONG, variable-length character data up to 2G in length
- CLOB, character large object up to 4G in length

Informix Dynamic Server CHAR(n) datatypes can be up to 32767 bytes in length. Columns defined as CHAR with a length <= 2000 can be migrated to the Oracle CHAR datatype and functionality contains nearly the same functionality. Both are fixed-length character strings and if you insert a shorter string, the value is blank-padded to the fixed length. If, however, a string is longer, Oracle returns an error.

Note: Oracle compares CHAR values using blank-padded comparison semantics. For more information, see the Comparison Sematics topic.

Oracle VARCHAR2 can hold data up to 4000 bytes in length. Oracle Corporation recommends you us a migration to VARCHAR2 when you are migrating Informix Dynamic Server CHAR columns that store more than 2000 bytes of data but less than or equal to 4000. VARCHAR2 is a variable length datatype and uses non-padded comparison semantics.

If Informix Dynamic Server tables have CHAR(n) columns defined with n > 4000 then the only option is to migrate to Oracle LONG or CLOB.

The LONG datatype can store variable-length character data up to 2G in length. LONG columns can be used in SELECT lists, SET clauses of UPDATE statements, and VALUES clause of INSERT statements. The LONG datatype is provided for backward compatibility and CLOB should be used for storing large amounts of character data. There are several restrictions on LONG datatypes, such as the following:

- One LONG columns is allowed per table
- LONG columns can not be indexed

The CLOB datatype is just one of the LOB datatypes supported by Oracle. LOB datatypes differ form LONG datatypes in several ways:

- A table may contain multiple LOB columns but only one LONG column
- A table containing one or more LOB columns can be partitioned, but a table containing a LONG columns can not be partitioned
- Maximum size of a LOB is 4G, maximum size of LONG is 2G
- LOBs support random access to data, but LONGs only support sequential access

LOB datatyes can be stored in-line within a table, or out-of-line within a tablespace, using a LOB locator, or in an external file -- a BFILE datatype. It is not currently supported by the Migration Workbench.

Using PL/SQL to manipulate LOBs, VARCHAR2s in PL/SQL can store up to 32767 bytes of data, so handling large Informix Dynamic Server CHAR datatypes should be reasonably efficient when stored in Oracle as CLOBs.

```
...
clob_loc CLOB;
some_text VARCHAR2(32767);
text_len INTEGER;
...
INSERT INTO page_info (page_num, page_text) VALUES (101, empty_clob);
SELECT page_text INTO clob_loc FROM page_info where page_num = 101;
text_len := LENGTH(some_text);
DBMS_LOB.WRITE(clob_loc, text_len,1, some_text);
```

You can use subscripts on BYTE columns. Subscripts can also be used on CHAR, VARCHAR2, NCHAR, NVARCHAR, and TEXT columns. The subscripts indicate the starting and ending character positions that define each column substring. With the DBMS_LOB package functions, you can choose to receive all or part of the CLOB, using READ and SUBSTR.

Collation Order

Data stored in CHAR columns is sorted based on the order of the code-set for the character set of the database, irrespective of the current location. Sorting in Oracle is based on the Oracle NLS settings. If the Oracle NLS specify a different sort order than the character code set, a sort-by code set can be enabled to ensure that the expected result remains the same. For more information, see the NCHAR(n) topic.

Multibyte Character Sets

Just as is the case for Informix Dynamic Server CHAR and VARCHAR datatypes, the length of Oracle CHAR and VARCHAR2 datatypes is specified in bytes. If the database character set is multibyte, make sure to calculate the appropriate space requirements to allow for the maximum possible number of bytes for a given number of characters.

Comparison Sematics

Informix Dynamic Server comparison semantics for the CHAR datatype and the Oracle CHAR datatype are the same. If you are migrating CHAR(n) columns that have a length where *n* such that 2000< *n* <= 4000 to then see the VARCHAR section for more details on comparison semantics for VARCHAR2.

Empty Strings

Informix Dynamic Server CHAR (and VARCHAR) columns can store an empty string, i.e. no data with a length zero. Even though a CHAR column may appear blank-padded, its length is 0. The empty string is not the same as NULL, which indicates that the value is undefined and of unknown length. However, Oracle does not have the concept of an empty string. Therefore, Oracle inserts empty strings as NULL.

You should check the application code and logic for unexpected behavior, such as empty strings migrating to NULL.

CHARACTER(n)

CHARACTER is a synonym for CHAR.

NCHAR(n)

Informix Dynamic Server CHAR and NCHAR datatypes both store the same type of data, a sequence of single-byte or multibyte letters, numbers, and symbols. The main difference between the datatypes is the order the Informix Dynamic Server database server sorts the data. CHAR columns are sorted on code set, the numeric values of the characters defined by the character encoding scheme. NCHAR columns are sorted based on the locale-specific localized order.

Oracle can sort CHAR data based on both the code set and the local-specific order.

The Migration Workbench for Informix Dynamic Server migrates both the Informix Dynamic Server CHAR and NCHAR datatypes to the Oracle CHAR datatype.

Collation Order

The Oracle NLS settings, either by default or as configured by the DBA, define the exact behavior of the sorting order. The NLS settings can be made at the database / init.ora, environment, and session levels.

A locale-specific sort is as a known as a linguistic sort in Oracle. You can use a linguistic sort by setting one of the Oracle collation parameters, NLS_SORT. The following is an example of a linguistic sort:

```
NLS_SORT = French
```

A code set sort is known as a binary sort in Oracle. If an application, for some reason, needs to sort data in a CHAR column based only on the code set, then the application can set the NLS_SORT to be a binary sort. The following is an example of a binary sort:

```
NLS_SORT = BINARY
```

Aside

Oracle does have a built-in NCHAR datatype, as well as NVARCHAR2 and NCLOB datatypes. These three datatypes can be used to store fixed-width and variable-width multibyte character set data as specified by the NATIONAL CHARACTER SET setting in the CREATE DATABASE command.

The National Character Set is an alternative character set to the Database Character Set. It is particularly useful in databases with a variable-width multibyte database character set because NCHAR, NVARCHAR2, and NCLOB can store fixed-width multibyte characters. Storing fixed-width multibyte characters enhances performance by allowing optimized string processing on these columns. An Oracle database can not be created with a fixed-width multibyte character set but the National Language datatypes allow storage of fixed-width multibyte character set data. The properties of a fixed-width character set may be more desirable for extensive processing operations or for facilitating programming.

VARCHAR(m,r)

The Informix Dynamic Server VARCHAR datatype stores varying length single-byte and multibyte character strings of letters, numbers. and symbols. The maximum size of this column is *m*, which can range from 1 to 255. The minimum reserved space is *r*. This is optional and defaults to 0 if not specified. The minimum reserved space can range from 0 to 255.

The comparable Oracle datatype is VARCHAR2(n) that also stores variable-length character strings. An Oracle VARCHAR2(n), however, can have a maximum string length of between 1 and 4000 specified for *n*.

Specifying a minimum reserved space is useful if the data in a row is initially small but is expected to grow at a later date. If this is the case then, when migrating Informix Dynamic Server tables that contain VARCHAR columns consider increasing the PCTFREE value in the storage clause for these tables in the Oracle database. If this column is used in an index, then the PCTFREE values for the corresponding index storage should also be considered. For indexes based on VARCHAR columns, Informix Dynamic Server allocates the maximum storage.

Comparison Semantics

Informix Dynamic Server VARCHAR values are compared to other VARCHAR values and to character values in the same way that character values are compared. The shorter values are blank-padded until the values have equal lengths, then they are compared for the full length.

Oracle VARCHAR2 comparisons are made using non-padded comparison semantics. Trailing blanks are important and are included in the comparison. Two values are only equal if they have the same characters and are of equal length.

Oracle CHAR comparison uses blank-padded comparison semantics, similar to the way Informix Dynamic Server compares CHAR and VARCHAR data. If two values have different lengths, Oracle adds blanks at the end of the shorter value, until the two values are the same length. Oracle then compares the values, character by character, up to the first character that differs. So two values that are different only in the number of trailing blanks are considered equal.

This is important behavior for the migration of the applications. It is possible for some comparisons on Informix Dynamic Server VARCHAR columns may fail when migrated to Oracle VARCHAR2 columns where trailing blanks are involved. To offset this, you may need to use RTRIM() on all columns in a comparison to strip off the trailing blanks.

Collating VARCHAR

The main difference between the NVARCHAR and VARCHAR datatype is the difference in collation sequencing. NVARCHAR character collation order depends on the database server locale, while the collation of VARCHAR characters depends on the code set. For more information on how these collation methods are implemented in Oracle and the impact on Informix Dynamic Server, see the NCHAR(n) topic.

Aside

Oracle has a built-in VARCHAR datatype that is currently synonymous with the VARCHAR2 datatype. However, VARCHAR is reserved for future use. In a later version of Oracle, the definition of VARCHAR may change and since VARCHAR2 is fully supported, the VARCHAR2 datatype is used to store variable-length character strings to avoid any possible changes from the current behavior.

CHARACTER VARYING(m,r)

The Informix Dynamic Server CHARACTER VARYING datatype is the ANSI-compliant format for character data of varying length. The Informix Dynamic Server VARCHAR datatype supports the same functionality and is treated as one in the database server. This datatype is treated the same as the VARCHAR datatype and migrates to the Oracle VARCHAR2 datatype. For more information, see the VARCHAR(m,r) topic.

NVARCHAR(m,r)

The Informix Dynamic Server NVARCHAR(m,r) datatype stores data of varying length, similar to VARCHAR, except that it compares data in the order that the locale specifies.

The Informix Dynamic Server NVARCHAR(m,r) datatype is migrated to the Oracle VARCHAR2(n) datatype.

For more information on migration issues, see the VARCHAR(m,r) and NCHAR(n) topics.

DATE

The DATE datatype stores the calendar date and the default display format is *mm/dd/yyyy* where *mm* is the month (01-12), *dd* is the day of the month (01-31) and *yyyy* is the year (0001-9999).

The DATE values are stored as integers thus DATE can be used in arithmetic expressions. For example, subtracting a DATE value from another DATE value returns the number of days that have elapsed between the two dates.

Subtracting two Oracle DATE datatypes from each other returns the number of days between the two dates. If only calendar dates are stored in Oracle, then the default time of 12:00:00AM (midnight) is also stored, so any subtraction results in a whole number indicating the number of days between the two dates. For the month, Informix Dynamic Server accepts a number value of either 1 or 01 for January, and

so on. Similarly, for the day, Informix Dynamic Server accepts either 1 or 01 for the first day of the month. This is also true in Oracle.

DATETIME

The Informix Dynamic Server DATETIME datatype stores an instant in time expressed as a calendar date and time of day. The precision that a DATETIME value is stored can be chosen, with the precision ranging from a year to a fraction of a second. DATETIME in effect is a family of 28 datatypes.

The Oracle DATE datatype matches just one of the 28 datetime types, DATETIME YEAR TO SECOND.

The Informix Dynamic Server plug-in stores DATETIME values as Oracle DATE values, losing the FRACTION part of DATETIME. If you need to keep the fraction part of DATETIME, before migration, add a new column to the table and store the fraction part as a DECIMAL with the appropriate precision migrated to the appropriate NUMBER datatype.

Any DATETIME table columns that do not store a particular precision use the Oracle defaults. The defaults for Oracle DATE are the first day of the current month and 12:00:00AM (midnight).

For applications that need to manipulate various DATETIME precisians, the SQL code needs to be changed. For example, if a column is defined as MONTH TO DAY, and contains two values, *date1: March 10* and *date2: February 18* shown as (mm/dd) 03/10 and 02/18. If these values are stored in Oracle DATE (if the current year is 1999, the values would be *date1: 1999/03/10 12:00:00* and *date2:1999/02/18 12:00:00* respectively. The expression *date1 - date2* UNITS DAY returns 20 days. However, if the year was 2000, then the expression *date1 - date 2* UNITS DAY returns 21 days.

Using a combination of TO_DATE and TO_CHAR and appropriate date format masks, the year the DATE was stored can be replaced with the current year for use in the expression. The following is an example of this combination:

```
SQL> SELECT TO_CHAR(TO_DATE(TO_CHAR(TO_DATE('01-01-1997',
'MM-DD-YYYY'), 'MM-DD'), 'MM-DD'), 'MM-DD-YYYY') from dual;
TO_CHAR(TO
----------
01-01-2000
```

Oracle DATE Arithmetic

Subtraction of DATE returns days. Because each date contains a time component, most results of date operations include a fraction. The fraction indicates a portion of one day. For example, 1.5 days is 36 hours.

The MONTHS_BETWEEN function returns the number of months between two dates. The fractional portion of the result represents that portion of a 31 day month.

You cannot add dates, but another Oracle function available for date arithmetic is ADD_MONTHS(date,n). To add days to a date, add a number constant to the date.

Note: Evaluate the logic of the addition or subtraction. Remember you can have months that are 28, 29, 30, or 31 days and you can have years that are 365 or 366 days.

INTERVAL

Currently, there is no corresponding Oracle datatype for the Informix Dynamic Server INTERVAL datatype.

The Informix Dynamic Server INTERVAL datatype can be defined as one of 18 different precisions, YEAR TO YEAR, YEAR TO MONTH, MONTH TO MONTH, DAY TO DAY and so on right down to FRACTION TO FRACTION(f). These are divided into two classes, YEAR TO MONTH, and DAY TO FRACTION.

Value	INTERVAL
YEAR TO YEAR	NUMBER(4)
	Informix Dynamic Server default precision for YEAR
DAY(3) TO DAY	NUMBER(3)
SECOND(6) TO SECOND	NUMBER(6)

If the largest qualifier value and the smallest qualifier value are not the same, then the INTERVAL column is migrated to CHAR(30).

Manipulating Oracle DATE with Informix Dynamic Server INTERVAL Values

Numeric constants can be added or subtracted from the Oracle DATE datatype -- to that Informix Dynamic Server DATE and DATETIME datatypes are mapped -- and are treated in terms of days. Therefore any operations involving the second class of

Informix Dynamic Server INTERVAL, DAY TO FRACTION must be expressed as a fraction in terms of days. For example,

```
CURRENT + INTERVAL (10 12) DAY TO HOUR
```

should be expressed as

```
SYSDATE + 10.5
```

To handle addition and subtraction of the first class of INTERVAL, YEAR TO MONTH, the INTERVAL needs to be expressed in terms of months and passed as a parameter, along with the date.

The Oracle function ADD_MONTHS(date,n) can be used for arithmetic:

```
TODAY + INTERVAL (2) YEAR TO YEAR
```

should be expressed as

```
ADD_MONTHS(SYSDATE, 24)
```

You do have the option to migrate all INTERVAL columns as CHARACTER(30) preserving all details, including subsecond information. However, the application must manipulate this data appropriately, using TO_DATE() and others.

DECIMAL

Informix Dynamic Server DECIMAL datatype can take two forms:

- DECIMAL(p) floating-point
- DECIMAL(p,s) fixed point

DECIMAL(p) floating point

DECIMAL(p) floating point stores decimal floating point numbers up to a maximum of 32 significant digits.

The total number of significant digits is p. This is optional, DECIMAL is treated as DECIMAL(16). DECIMAL(p) has an absolute values range of between 10-130 and 10124.

In an ANSI-compliant Informix Dynamic Server database, DECIMAL(p) defaults to DECIMAL(p,0). If only p is specified, s is actually stored as 255 in the catalog tables.

In Oracle, Informix Dynamic Server DECIMAL(p) floating point values are always stored as NUMBER. It has 38 significant digits since it is not possible to restrict the

total number of significant digits for storing a floating-point number. Oracle can store negative and positive values in the range 1.0x10-130 and 9.9...9x10125, which is 38 nines followed by 88 zeros. NUMBERS are stored in scientific notation. Leading and trailing zeros are not stored.

Since Oracle can store floating-point numbers with a greater precision than Informix Dynamic Server, there should be no loss of precision after the migration to Oracle.

DECIMAL(p,s) fixed-point

The precision is p with a range 1 to 32. The number of digits to the right of the decimal place is s. Numbers < 0.5x10-s have the value 0.

In Oracle, DECIMAL(p,s) maps to NUMBER(p,s).

MONEY(p,s)

The MONEY datatype is always a fixed-point number with a maximum 32 significant digits.

MONEY(p) = DECIMAL(P,2)

MONEY = DECIMAL(16,2)

The Informix Dynamic Server MONEY datatype is represented as DECIMAL. The Informix Dynamic Server MONEY(p,s) datatype maps to Oracle NUMBER(p,s).

INTEGER

The Informix Dynamic Server `INTEGER` datatype is mapped to NUMBER(10).

Range Boundaries

The Informix Dynamic Server `INTEGER` datatype can store values in the range -2,147,483,647 to 2,147,483,647. If a value to be inserted is outside this range, the Informix Dynamic Server database server does not store the value and returns an error. A column defined as NUMBER(10) in an Oracle database allows values in the range -9,999,999,999 to 9,999,999,999 to be inserted without raising an error. If mapped, `INTEGER` columns should enforce the original range, then a check constraint can be added to the columns to ensure that values entered into these columns are within the range -2,147,483,647 to 2,147,483,647.

Storage

Informix Dynamic Server stores INTEGER as a signed binary integer and requires 4 bytes per value.

Oracle stores numeric data in variable length format, in scientific notation. The smallest storage space Oracle uses to represent an INTEGER is 2 bytes, 12 bytes is the maximum storage space required. The storage space for the value depends on the number of significant digits.

Inserting Fractions

If you insert 7.2 and 7.8 into Informix Dynamic Server INTEGER datatype, fractional parts are truncated, therefore the values 7 and 7 are stored.

If you insert 7.2 and 7.8 into Oracle NUMBER(10), fractional parts are rounded, therefore the values are stored as 7 and 8.

It may be necessary to check application code and logic to ensure there is no unexpected behavior. This is because it is assumed that fractional parts of any number are automatically truncated when inserted into the Informix Dynamic Server database.

INT

The Informix Dynamic Server INT datatype is a synonym for INTEGER

SMALLINT

The Informix Dynamic Server SMALLINT datatype is mapped to NUMBER(5).

Range Boundaries

The Informix Dynamic Server INTEGER datatype can store values in the range -32,767 to 32767. If a value is outside this range, the Informix Dynamic Server database server does not store the value and returns an error. A column defined as NUMBER(10) in an Oracle database allows values in the range -99,999 to 99,999 to be inserted without raising an error. If mapped, INTEGER columns should enforce the original range then a check constraint can be added to the columns to ensure that values entered into these columns are within the range -32,767 to 32767.

Storage

Informix Dynamic Server SMALLINT datatype values take up 2 bytes per value.

Oracle stores a values in an NUMBER(5) datatype with a minimum of 2 bytes and a maximum of 4 bytes

Inserting Fractions

For information on differences in behavior for Informix Dynamic Server INTEGER and Oracle NUMBER, see the Inserting Fractions topic.

SERIAL

The Informix Dynamic Server SERIAL datatype creates a column in a table that auto-increments an INTEGER value every time a row is inserted into the table. By default, if the column is simply defined as SERIAL, the column begins inserting with the value 1. Other starting values can be set by defining the column. For example, SERIAL(1000)creates a column that begins inserting with the value 1000. The starting number cannot be 0 and the maximum value SERIAL can reach, or be initially set to, is 2,147,483,647. After reaching the maximum value, the SERIAL column resets to 1. Only one SERIAL column may be defined for an Informix Dynamic Server table. The SERIAL datatype is not automatically a unique column, a unique index must be created for this column to prevent duplicate serial numbers.

The Migration Workbench for Informix Dynamic Server maps the Informix Dynamic Server SERIAL datatype to an Oracle NUMBER(10) datatype and flags the column as an auto-increment column. The Migration Workbench also creates a NOT NULL CONSTRAINT on that column, as is the case with Informix Dynamic Server SERIAL columns.

The Migration Workbench creates an Oracle sequence and an Oracle trigger on the table that contained the SERIAL column. The trigger fires every time a row is inserted into the table. It gets the next value in the sequence and inserts it into the field.

For example, the following JOBS table was migrated to Oracle and the JOB_ID column was originally defined as an Informix Dynamic Server SERIAL datatype:

```
CREATE TABLE clerk.JOBS(JOB_ID NUMBER (10) NOT NULL,
              JOB_DESC VARCHAR2 (50) NOT NULL,
              MIN_LVL NUMBER (5),
              MAX_LVL NUMBER (5))
TABLESPACE PUBS;
REM
REM Message : Created Sequence: clerk.SEQ_11_1
REM User : system
```

```
CREATE SEQUENCE clerk.SEQ_11_1 START WITH 1
/
REM
REM Message : Created Sequence Trigger: clerk.TR_SEQ_11_1
REM User : system
CREATE TRIGGER clerk.TR_SEQ_11_1
BEFORE INSERT ON clerk.JOBS FOR EACH ROW
BEGIN
  SELECT clerk.SEQ_11_1.nextval INTO :new.JOB_ID FROM dual; END;
/
```

The Oracle trigger and sequence is created after a table with a SERIAL column is migrated. The sequence is created using the option START WITH 1. If the data for this table is not moved automatically by the Migration Workbench, the sequence starts inserting with 1.

If the table data is selected to be moved automatically by the Migration Workbench while database table objects are created, the Migration Workbench creates the trigger and sequence after the data has been moved. Before the sequence is created the Migration Workbench selects the maximum value from the SERIAL column (for example, 1231) and add 1 to this value and use it as the START WITH value in the CREATE SEQUENCE statement; as follows:

```
CREATE SEQUENCE clerk.SEQ_11_1 START WITH 1232;
```

In the resulting Oracle database inserts to the table continue to auto-increment by one a value for the old serial column every time a row is inserted into the table.

Additional Oracle Sequence Options for Informix Dynamic Server SERIAL Migrations

The Migration Workbench uses the following command to create the sequence:

```
CREATE SEQUENCE sequence_name START WITH integer;
```

The Oracle CREATE SEQUENCE command has several options, the only option that is used is the START WITH option.

Many of these options have defaults that are what would be required to replicate the Informix Dynamic Server SERIAL datatype. However, there are a couple of settings that can be altered on the SEQUENCE to make it behave even more closely to the Informix Dynamic Server SERIAL datatype.

Option	Description
START WITH *integer*	Specify the start sequence value. For more information, see Resetting the Start Value.
INCREMENT BY *integer*	Specify the interval between sequence numbers. If this value is negative, then the sequence descends. For Informix Dynamic Server, use the Oracle default of 1.
MAXVALUE *integer*	NOMAXVALUE is the default setting. For Informix Dynamic Server, set this to 2147483647 to override the Oracle default value.
NOMAXVALUE	Specify NOMAXVALUE to indicate a maximum value of (10^27)-1, twenty eight 9's in a row, for an ascending sequence or -1 for a descending sequence. This is the default.
MINVALUE *integer*	Specify the sequence minimum value. For Informix Dynamic Server, indicate 1 as the value so that if the sequence ever restarts, it restarts with this value.
NOMINVALUE	Specify NOMINVALUE to indicate a minimum value of 1 for an ascending sequence or -(10^26) for a descending sequence. This is the default. For Informix Dynamic Server, use the default because the default INCREMENT BY value of 1, we get a default minimum value of 1.
CYCLE	Specify CYCLE to indicate that the sequence continues to generate values after reaching either maximum or minimum value. After an ascending sequence reaches maximum value, it generates minimum value.
SERIAL	For Informix Dynamic Server, the column resets to 1 after reaching 2147483647.
NOCYCLE	Specify NOCYCLE to indicate that the sequence cannot generate more values after reaching maximum or minimum value. This is the default. For Informix Dynamic Server, override this default Oracle behavior to CYCLE.
CACHE integer	Specify how many values of the sequence Oracle preallocates and keeps in memory for faster access.

Option	Description
NOCACHE	Specify NOCACHE to indicate that values of the sequence are not preallocated. If both CACHE and NOCACHE are omitted, Oracle caches 20 sequence numbers by default. For Informix Dynamic Server, since MINVALUE is 1, MAXVALUES is at least 2,147,483,647 and INCREMENT is 1, then there should be no problems with the default. If the table is a target of high activity then, the CACHE values may have to be reviewed along with FREELISTS, INITTRANS, MAXTRANS, and others.)

In Oracle, all the options that were used to create a sequence can be altered except for START WITH.

Resetting the Start Value

To restart an Oracle sequence at a different number, you must drop and re-create it.

The following Table 2–5 shows that Informix Dynamic Server changes the next value to be used in a SERIAL column, provided 1000 is not less than the current maximum for the column.

Table 2–5 Serial Column Comparison

Informix Dynamic Server	Oracle
`ALTER TABLE clerk.jobs MODIFY ( job_id SERIAL(1000) );`	`DROP SEQUENCE seq_11_1;` `CREATE SEQUENCE seq_11_1 STARTWITH 1000 MAXVALUE 2147483647 CYCLE;`

Some Exceptional Cases

Occasionally the migrated SERIAL column does not behave as it would in Informix Dynamic Server.

Example 1

If the last number values inserted into a table are deleted from the table, the migrated tables sequence begin before the next.

If the table was created as follows:

```
CREATE TABLE table_with_serial_col ( col1 SERIAL, col2 CHAR(5) )
```

and after several inserts on the table, as follows:

```
INSERT INTO table_with_serial_col VALUES ("XXX");    [col1 = 1]
INSERT INTO table_with_serial_col VALUES ("XXX");    [col1 = 2]
INSERT INTO table_with_serial_col VALUES ("XXX");    [col1 = 3]
```

the definition is changed, as follows:

```
ALTER TABLE table_with_serial_col MODIFY ( col1 SERIAL(1000) )
```

If the database is migrated at this point, execute the following command on Informix Dynamic Server:

```
INSERT INTO table_with_serial_col VALUES ("XXX");
```

results in the new row with a value of 1000 for `col1`. If you execute the same command in the migrated Oracle environment, the new row would have a value of 4 for `col1`.

Example 2

Another possibility follows:

```
INSERT INTO table_with_serial_col VALUES ("XXX");    [col1 = 1]
INSERT INTO table_with_serial_col VALUES ("XXX");    [col1 = 2]
INSERT INTO table_with_serial_col VALUES ("XXX");    [col1 = 3]

DELETE FROM table_with_serial_col WHERE col1 = 3;
```

If the database is migrated at this point, execute the following command on Informix Dynamic Server:

```
INSERT INTO table_with_serial_col VALUES ("XXX");
```

results in the new row with a value of 4 for `col1`. If you execute the same command in the migrated Oracle environment, the new row would have a value of 3 for `col1`.

It is possible that this would have no effect on the execution of the application or the integrity of the data. The only dependency is that this value is unique and auto-incremental, but it may be useful to check the application logic if situations similar to the examples could occur.

How to examine current Informix Dynamic Server SERIAL values

For Informix Dynamic Server, set the `SERIAL` value to the values of `sysmaster:systabinfo(ti_serialv)` where `sysmaster:systabinfo(ti_partnum)` is the `partnum` of the table with the `serial` column.

```
select c.dbsname, a.owner, a.tabname, d.ti_serialv
```

```
  from systables a, syscolumns b, sysmaster:informix.systabnames c,
sysmaster:informix.systabinfo d
 where (b.coltype = 6  OR b.coltype = 262)
   and a.tabid = b.tabid
   and a.tabid > 99
   and a.owner = c.owner
   and a.tabname = c.tabname
   and c.dbsname = "<DATABASENAME>"
   and c.partnum = d.ti_partnum;
```

Replace <*DATABASENAME*> as appropriate.

In the Oracle environment, use the following SQL statements to get the next sequence number to be generated for each sequence:

```
SQL> SELECT sequence_name FROM USER_SEQUENCES;
SQL> SELECT (<sequence_name>.CURRVAL+1) FROM DUAL;
```

Replace <*sequence_name*> as appropriate.

Data Storage Concepts

This chapter provide a description of the conceptual differences (and in many cases, similarities) in data storage for Informix Dynamic Server and Oracle9*i* and Oracle8*i* databases.

Recommendations

The following are recommendations:

1. The conceptual differences in the storage structures do not affect the conversion process directly.
2. Both Oracle and Informix Dynamic Server have a way to control the physical placement of database objects:
   - IN dbspace for Informix Dynamic Server
   - TABLESPACE for Oracle.
3. Storage information can be preserved when converting to Oracle. The decisions made when defining the storage of the database objects for Informix Dynamic Server should also apply for Oracle. Especially important are the initial object and physical object placement.

An Oracle database server consists of a shared memory area, several processes that access the database and maintain data integrity and consistency, the Oracle Instance, and a database that stores the data.

An Informix Dynamic Server database server also consists of a shared memory area, several process to access the data and maintain data integrity and consistency, however a single Informix Dynamic Server database server can support several separate databases.

A Oracle database consists of one or more tablespaces. Tablespaces provide logical storage space that link a database to the physical disks that hold the data. A tablespace is created from one or more data files. Data files are files in the file system or an area of disk space specified by a raw device. A tablespace can be enlarged by adding more data files.

An Oracle database consists of a least a SYSTEM tablespace, where the Oracle tables are stored. It can also consist of user defined tablespaces. A tablespace is the logical storage location for database objects. For example, you can specify where a particular table or index gets created in the tablespace.

The size of a tablespace is determined by the amount of disk space allocated to it. Each tablespace is made up of one or more data files.

Data Storage Concepts Table

Table 2–6 Data Storage Concepts in Informix Dynamic Server and Oracle

Informix Dynamic Server	Oracle
Chunks	**Data Files**
Physical disk space is allocated in terms of chunks. A chunk can be from a file system or raw disk space. The root dbspace is mapped to a chunk specified by a raw device name or the full path name of a file in a file system, through the initialization file, on_config.	One or more data files are created for each tablespace to physically store the data of all the logical structures in a tablespace.

Table 2–6 Data Storage Concepts in Informix Dynamic Server and Oracle (Cont.)

Informix Dynamic Server	Oracle
Page and Blobpage Page: A chunk has its space divided into pages, each with a specified number of bytes. Any I/O must be performed in page units. Blobpage: A blobpage stores BYTE and a TEXT data within a blobspace. The size of blobpage is a unit of disk allocation selected by the user who creates the blobspace, and can vary from blobspace to blobspace.	**Data Block** One data block corresponds to a specific number of bytes of physical space on disk. The database block size can be specified when creating the database.
Extent Extent is the allocation of disk space to a database object in units of physically contiguous pages and cannot span chunk boundaries. All database objects have space allocated in increment of one extent. For a single table, extents can be located in different chunks of the same dbspace. Within an extent, all data pertains to a single tblspace.	**Extent** An extent is a specific number of contiguous data blocks, obtained in a single allocation.
Tblspace The total diskspace allocated to a table includes pages allocated to: ■ data ■ indexes ■ storage of blob data (BYTE or TEST) in the dbspace (excluding pages storing blob data in separate blobspace) ■ tracking page usage within the table extents	**Segments** A segment is a set of extents allocated for a certain logical structure. The extents of a segment may or may not be contiguous on disk, and may or may not span datafiles.

Table 2–6 Data Storage Concepts in Informix Dynamic Server and Oracle (Cont.)

Informix Dynamic Server	Oracle
Physical Log A unit of contiguous disk pages containing "before images" of data that has been modified during processing. **Logical Log** Logical log file is the name of each of these additions of space. This log records logical operations during on-line processing. All transaction information is stored in the logical files as a database is created with the transaction log.	**Redo Log Files** Each database has a set of two or more redo log files. All changes made to the database are recorded in the redo log. Redo log files are critical in protecting a database against failures.
Root dbspace The root dbspace stores information about all databases created.	**System Tablespace** Oracle Control Files Each database has a control file. This file records the physical structure of the database, such as the database name, name and location of the database data files and redo logs.
Dbspace and Blobspace Dbspaces: Database objects are stored in a dbspace, which is a minimum of one piece of physical disk or chunk. BYTE and TEXT (Binary Large Objects, or BLOBs) data can be stored in a dbspace, but performance may suffer if the BLOBs are larger than two dbspace pages. The ROOT dbspace is the name of the first dbspace created. Specific pages and internal tables in the ROOT dbspace describe and track all other dbspaces, blobspaces, and tblspaces. Blobspaces: A blobspace provides a storage area for TEXT and BYTE data using a larger and more efficient space allocation mechanism more suited to large objects as opposed to storing them in a dbspace with more traditional data types.	**Tablespace** A database is divided into logical storage units called tablespaces. A tablespace is used to group related logical structures together. A database typically has one system tablespace an one or more user tabletops.

3

Triggers, Packages, and Stored Procedures

This chapter includes the following sections:

Introduction

Informix Dynamic Server stores triggers and stored procedures with the server. Oracle stores triggers and stored subprograms with the server. Oracle has three different kinds of stored subprograms: functions, stored procedures, and packages. For detailed discussion on all these objects, see the PL/SQL User's Guide and Reference, Release 1(9.0.1).

Triggers

Triggers provide a way of executing PL/SQL code on the occurrence of specific database events. For example, you can maintain an audit log by setting triggers to fire when insert or update operations are carried out on a table. The insert and update triggers add an entry to an audit table whenever the table is altered.

The actions that Informix Dynamic Server triggers perform are constrained to multiple insert, update, delete, and execute procedure clauses; whereas, Oracle allows triggers to execute arbitrary PL/SQL code. Oracle triggers are similar to stored procedures in that they can contain declarative, execution, and exception handling code blocks.

Additionally, Oracle enables triggers to be invoked by many events other than table insert, update and delete operations. However, there are restrictions.

For more information on trigger restrictions, see the Oracle9i Application Developer's Guide - Fundamentals, Release 1 (9.0.1).

Mapping Triggers

All Informix Dynamic Server trigger types have an equivalent Oracle trigger type. The converter takes the optional `WHEN` clause in Informix Dynamic Server and converts it to an `IF` clause. This is shown in the following example:

Informix Dynamic Server SPL

```
create trigger t_traffic
update of comments
on msg_traffic
referencing new as new
for each row
when (new.msg_id>10000)
 (update msg_traffic set msg_traffic.update_dt = CURRENT year to fraction(3)
  where (((msg_id = new.msg_id ) AND (msg_source = new.msg_source ) )
  AND (sub_msg_id = new.sub_msg_id ) ) );
```

Oracle PL/SQL

```
CREATE OR REPLACE TRIGGER t_traffic
BEFORE UPDATE OF comments ON msg_traffic
REFERENCING NEW as new_ FOR EACH ROW
BEGIN
DECLARE
ItoO_selcnt         NUMBER;
ItoO_rowcnt         NUMBER;
BEGIN
 IF :new_.msg_id > 10000 THEN
 UPDATE msg_traffic
 SET msg_traffic.update_dt  =  SYSDATE
 WHERE ( ( ( msg_id = :new_.msg_id )
 AND ( msg_source = :new_.msg_source ) )
 AND ( sub_msg_id = :new_.sub_msg_id ) );
 END IF;
  END;
END;
```

Informix Dynamic Server declares triggers on a per table basis with BEFORE and AFTER triggers held together in a single trigger declaration. In Oracle, the BEFORE and AFTER triggers are declared separately. Therefore, the convertor creates multiple Oracle triggers when parsing Informix Dynamic Server per table trigger code.

In the initial release, the Oracle triggers display one after the other in the same text area. The Oracle triggers require manual intervention to build on the Oracle destination database.

Mutating Tables

When you are using Oracle, the trigger or function may cause a mutating table error. This causes you to receive the following error message while executing the trigger:

```
ORA-04091: table SCOTT.Emp_tab is mutating, trigger/function may not see it.
```

If you receive this error, you need to manually alter the trigger so that the per row information is stored in an interim PL/SQL table. It is then copied into the destination table after the per row triggers have been fired. For more information, see the Mutating: Containing Tables topic at the following Web site:

http://otn.oracle.com/tech/migration/workbench/htdocs/mutating.htm

Packages

Packages are PL/SQL constructs that enable the grouping of related PL/SQL objects, such as procedures, variables, cursors, functions, constants, and type declarations. Informix Dynamic Server does not support the package construct.

A package can have two parts: a specification and a body. The specification defines a list of all objects that are publicly available to the users of the package. The body defines the code that is used to implement these objects, such as, the code behind the procedures and functions used within the package.

The general PL/SQL syntax for creating a package specification is:

```
CREATE [OR REPLACE] PACKAGE package_name {IS | AS}
  procedure_specification
..function_specification
..variable_declaration
..type_definition
..exception_declaration
..cursor_declaration
END [package_name];
```

The general PL/SQL syntax for creating a package body is:

```
CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
..procedure_definition
..function_definition
..private_variable_declaration
..private_type_definition
..cursor_definition
[BEGIN
  executable_statements
[EXCEPTION
..exception_handlers]]
END [package_name];
```

The package body is optional. If the package contains only variable, cursor and type definitions then the package body is not required.

As the package specification is accessible to users of the package, it can be used to define global variable definitions within PL/SQL.

The Migration Workbench automatically creates packages during the conversion process for the following reasons:

- The Utilities package, which is used to emulate built-in Informix Dynamic Server functions, is not available in Oracle.

- Packages have to be created to emulate Informix Dynamic Server `GLOBAL` variable definitions.

For more information on package creation, see the following sections:

- Converting TRACE Statements
- GLOBAL Variable Declarations
- Converting RETURN WITH RESUME Statements
- Returning Section

For more information on package creation and use, see the PL/SQL User's Guide and Reference, Release 1 (9.0.1).

Stored Procedures

Stored procedures provide a powerful way to code application logic that can be stored on the server. Informix Dynamic Server and Oracle both use stored procedures. Oracle also uses an additional type of subprogram called a function.

The language used to code stored procedures is a database-specific procedural extension of SQL. In Oracle it is PL/SQL and in Informix Dynamic Server it is Informix Dynamic Server Stored Procedure Language (SPL). These languages differ considerably. However, most of the individual SQL statements and the procedural constructs, such as `if-then-else`, are similar in both languages.

Note: The PL/SQL procedure examples included in the document are the actual output of the Migration Workbench. They are longer than the source Informix Dynamic Server SPL procedures because they are converted to emulate SPL functionality. When the PL/SQL procedures are written for equivalent Oracle functionality, the Output code is shorter.

The PL/SQL procedures, which the Migration Workbench generates, add appropriate comments to indicate the manual conversion required. In general, the Migration Workbench deals with the Informix Dynamic Server constructs in one of the following ways:

- Converts ANSI-standard SQL statements to PL/SQL because Oracle supports ANSI-standard SQL.

- Converts into PL/SQL constructs if the equivalent constructs are available in PL/SQL.
- Ignores some constructs and incorporates appropriate comments in the output file.
- Wraps constructs that require manual conversion around proper comments in the output file.
- Displays an appropriate error message, including the line number, for those contructs resulting in syntax errors.

The following sections provide a comparison of Informix Dynamic Server and Oracle:

- NULL as an Executable Statement
- Parameter Passing
- Individual SPL Statements
- Error Handling within Stored Procedures
- DDL Statements in SPL Code
- Using Keywords as Identifiers
- Issues with Converting SPL Statements

NULL as an Executable Statement

In some cases within stored procedure code, it may be necessary to indicate that no action should be taken. To accomplish this in Oracle, the `NULL` statement is used. Unlike Informix Dynamic Server, Oracle treats the `NULL` statement as executable within a PL/SQL code block. In Oracle the `NULL` statement does not perform an action. Instead, it forms a syntactically legal statement that serves as a placeholder.

Oracle places a NULL statement into PL/SQL code in the following situations:

- When converting a `CONTINUE` statement within a `FOR`, `FOREACH`, or `WHILE LOOP` construct is encountered.
- When encountering an unsupported SPL statement.

For information on how the converter uses NULL statements, see the following sections:

- Converting CONTINUE Statements
- Issues with Converting SPL Statements

Parameter Passing

An Informix Dynamic Server stored procedure contains the following logical parts:

1. Procedure name
2. Parameters area
3. Returning section
4. Statement block
5. Document section
6. With listing directive

Parts two and three define how data is passed to and from a stored procedure. Part two ties data values that are passed by the client to variable names.

Part three is optional. It defines a listing of the data types that the stored procedure returns to the client or calling environment.

The following example demonstrates parts one, two and three: the Informix Dynamic Server stored procedure code for the procedure name, parameters area, and the returning section.

Informix Dynamic Server SPL

```
/* Procedure name */
CREATE PROCEDURE bal_enquiry(
/* The Parameters area */
cust_id   NUMBER,
account_num NUMBER)
/* The Returning section */
RETURNING NUMBER;
```

Unlike Informix Dynamic Server, Oracle does not require the use of a Returning section. Instead, Oracle passes values to the stored procedure and from the stored procedure by using `IN`, `OUT` or `IN OUT` parameter modes.

In a similar way to Informix Dynamic Server, PL/SQL parameters within Oracle can have default values assigned to them.

Oracle Parameter Passing Modes

The modes for Oracle formal parameters are `IN`, `OUT`, or `IN OUT`. If a mode is not specified for a parameter, it defaults to the `IN` mode. Table 3–1 describes parameter modes within Oracle.

Table 3–1 Parameter Passing Modes in Oracle

Mode	Description
IN	The value of the parameter is passed into the procedure when the procedure is invoked. It is similar to read-only
OUT	Any value the parameter has when it is called is ignored. When the procedure finishes, any value assigned to the parameter during its execution is returned to the calling environment. It is similar to write-only
IN OUT	This mode is a combination of both IN and OUT. The value of the parameter can be passed into the procedure when the procedure is invoked. It is then manipulated within the procedure and returned to the calling environment. It is similar to read-write

Input Parameters

Informix Dynamic Server uses all parameters defined within the parameters area to pass values into the stored procedure. These parameters cannot pass data back to the client. If a default value is included for each variable, clients that execute the procedure do not have to send data to the procedure. Each parameter within the parameters area can, therefore, be converted to a functionally equivalent Oracle IN parameter. An example of an Informix Dynamic Server SPL procedure definition and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
CREATE PROCEDURE informix.update_bal(
cust_id INT,
amount  INT DEFAULT 1)
```

Oracle PL/SQL

```
CREATE OR REPLACE PROCEDURE "INFORMIX".update_bal(
cust_id_IN        NUMBER,
amount_IN         NUMBER DEFAULT 1) AS
BEGIN
cust_id           NUMBER := cust_id_IN;
amount            NUMBER := amount_IN;
```

Output Parameters

You use the Informix Dynamic Server returning section to define a list of data types to be returned to the client. If you use a returning section, the type and number of data values listed after the RETURN statement must match what was declared in the returning clause. The RETURN statement only sends one set of results back to the calling environment. If multiple contiguous sets of results need to be returned then you can add the WITH RESUME keywords.

If you use the WITH RESUME keywords, after the RETURN statement executes, the next invocation of the procedure starts at the statement that directly follows the RETURN statement.

If a procedure is defined using a WITH RESUME clause, a FOREACH loop within the calling procedure or program must call the procedure. In Informix Dynamic Server, a procedure returning more than one row or set of values is called a cursory procedure.

In effect, Informix Dynamic Server stored procedures have to be invoked repeatedly should multiple values need to be passed back to the calling environment. So *n* invocations returns *n* sets of contiguous singleton results.

If the Informix Dynamic Server stored procedure does not contain a WITH RESUME clause, it has been designed to be invoked only once and, optionally, send singleton values back to the calling environment.

In this case, all returning section parameters are converted to be OUT parameters within the generated Oracle PL/SQL code.

If a WITH RESUME statement is present within the Informix Dynamic Server stored procedure, then the Migration Workbench uses each returning clause parameter to build a global temporary table to store the procedures interim results. The Migration Workbench then uses this temporary tableto build and return a populated cursor to the calling environment.

For more information on the strategy the Migration Workbench employs to convert the Informix Dynamic Server returning section to PL/SQL, see the following sections:

- Returning Section
- Converting RETURN WITH RESUME Statements

Individual SPL Statements

Both Informix Dynamic Server and Oracle use a database-specific procedural extension of SQL as their procedural language. However, the languages are not common so it is necessary that Migration Workbench emulates Informix Dynamic Server functionality that is not found in Oracle within the converted stored procedure PL/SQL code.

The following statements or constructs have to be, to a varying degree of complexity, emulated within the generated Oracle PL/SQL code:

- Returning Section
- DOCUMENT Clause
- GLOBAL Variable Declarations
- LIKE and MATCHES Comparison Conditions
- FOR LOOP Constructs
- FOREACH LOOP Constructs
- Compound LET Statements
- Converting CONTINUE Statements
- Converting RETURN WITH RESUME Statements
- Built-in Functions
- Converting the SYSTEM Statement
- Converting TRACE Statements
- Set Up Tasks for the DEBUG Procedure
- SELECT Statements as Conditions
- Exception Blocks
- `RAISE EXCEPTION` Statements

Returning Section

The Informix Dynamic Server returning section is used to define the list of data types being returned to the client. The way the Migration Workbench converts the Returning section is determined by whether the `RETURN WITH RESUME` statement resides within the Informix Dynamic Server stored procedure. The Migration Workbench converts the returning section using one of the following methods:

- Informix Dynamic Server Procedures Containing no WITH RESUME Clause
- Informix Dynamic Server Procedures Containing a WITH RESUME Clause

Informix Dynamic Server Procedures Containing no `WITH RESUME` Clause If only one parameter is specified in the Informix Dynamic Server returning section and the procedure contains no `WITH RESUME` clause, then Migration Workbench converts the procedure to an Oracle `FUNCTION`. An example of a procedure returning one value in Informix Dynamic Server and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
CREATE PROCEDURE "informix".add_category(
name like Recipecategory.category_name,
desc like recipeCategory.category_desc)
RETURNING integer;
```

Oracle PL/SQL

```
CREATE OR REPLACE FUNCTION informix.add_category(
name_IN                Recipecategory.category_name%TYPE,
desc__IN               recipeCategory.category_desc%TYPE)
RETURN NUMBER AS
```

If multiple returning parameters are defined within the Informix Dynamic Server returning section and the procedure contains no `WITH RESUME` clause, Migration Workbench converts each returning parameter to an Oracle `OUT` parameter. An example of a procedure returning multiple singleton values and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
CREATE PROCEDURE "root".ocsa_list_total(sp_order_id INT)
    RETURNING DECIMAL(9,4), DECIMAL(9,4),
              DECIMAL(9,4), DECIMAL(10,4);
/* Other statements, one of which is of type
   RETURN <decimal>, <decimal>, <decimal>, <decimal>; */
```

Oracle PL/SQL

```
CREATE OR REPLACE PROCEDURE root.ocsa_list_total(
sp_order_id_IN         NUMBER,
/* SPCONV-MSG:(RETURNING) Informix RETURNING clause parameters converted to
Oracle OUT parameters. */
OMWB_outParameter1  OUT NUMBER,
```

```
OMWB_outParameter2  OUT NUMBER,
OMWB_outParameter3  OUT NUMBER,
OMWB_outParameter4  OUT NUMBER) AS
```

Informix Dynamic Server Procedures Containing a WITH RESUME Clause The method used to pass sets of results back to the client in Oracle differs considerably from the one used in Informix Dynamic Server.

Oracle stored procedures are only ever invoked once in order to return multiple sets of results and therefore PL/SQL does not contain any such `WITH RESUME` construct

Multiple sets of data are returned to the calling environment through the use of `OUT` or `IN OUT` parameters of type `REF CURSOR`. This cursor variable is similar to the user-defined record type and array type. The cursor stored in the cursor variable is like any other cursor. It is a reference to a work area associated with a multi-row query. It denotes both the set of rows and a current row in that set. The cursor referred to in the cursor variable can be opened, fetched from, and closed just like any other cursor. Since it is a PL/SQL variable, it can be passed into and out of procedures like any other PL/SQL variable.

If the Informix Dynamic Server stored procedure contains a `WITH RESUME` clause, the procedure is classed as a cursory procedure, which is a procedure that returns a result set. Each parameter defined within the procedures returning section is then used to construct a global temporary table uniquely associated with the procedure. This global temporary table is then used to store the procedures interim results.

The following Informix Dynamic Server code causes the converter to create a temporary table named `get_slistTable`. This table is then used to store the interim results of the procedure.

Informix Dynamic Server SPL

```
CREATE PROCEDURE "root".get_slist(
v_uid like PHPUser.user_id,
v_listid like ShoppingList.list_id)
returning integer, char(75), char(255);

/* Other stored procedure statements one of which is of type
  RETURN <integer>, <char>, <char> WITH RESUME
*/

END PROCEDURE;
```

Oracle PL/SQL temp table Definition

```
CREATE GLOBAL TEMPORARY TABLE get_slistTable(
/* The first column 'col00' is used to create an ordered
    SELECT statement when populating the REF CURSOR
    OUT parameter to the procedure */
col00  NUMBER,
col01  NUMBER,
col02  CHAR(75),
col03  CHAR(255))
ON COMMIT DELETE ROWS;
```

The converter then adds an OUT parameter whose type is derived from a packaged WEAK REF CURSOR type to the PL/SQL stored procedure parameter list. For example:

```
CREATE OR REPLACE PROCEDURE root.get_slist(
v_uid_IN               informix.PHPUser.user_id%TYPE,
v_listid_IN
informix.ShoppingList.list_id%TYPE,
/* The following cursor is added to the procedure by the converter */
OMWB_ret_cv            OUT
AS
```

Using a cursor variable in this way in PL/SQL emulates the Informix Dynamic Server cursory procedure. The main difference from Informix Dynamic Server SPL is that the PL/SQL procedure is invoked only once and it returns a cursor variable containing the complete set of results.

For more information, see the following:

- Converting RETURN WITH RESUME Statements
- FOREACH LOOP Constructs

DOCUMENT Clause

The DOCUMENT clause enables a synopsis or description of the Informix Dynamic Server stored procedure to be detailed. The text contained after the DOCUMENT keyword is inserted into the Informix Dynamic Server sysprocbody system catalogue during the procedures compilation. This text can then be queried by the users of the stored procedure. Oracle PL/SQL has no such DOCUMENT clause.

The Migration Workbench converts the Informix Dynamic Server DOCUMENT clause to a multi-line comment within the PL/SQL stored procedure. This is demonstrated by the following example:

Informix Dynamic Server SPL

```
create procedure "informix".min_two(first integer, scd integer)
returning integer;
 if (first < scd) then
  return first;
 else
  return scd;
 end if;
end procedure
DOCUMENT 'The following procedure accepts two INTEGER values and returns the
smallest of the two.';
```

Oracle PL/SQL

```
CREATE OR REPLACE FUNCTION informix.min_two(
 first_IN               NUMBER,
 scd_IN                 NUMBER) RETURN NUMBER AS

/*
'The following procedure accepts two INTEGER values and returns the smallest of
the two.'
*/

first               NUMBER(10) := first_IN;
scd                 NUMBER(10) := scd_IN;
ItoO_selcnt         NUMBER;
ItoO_rowcnt         NUMBER;

BEGIN
 IF ( first < scd ) THEN
  RETURN first;
 ELSE
  RETURN scd;
 END IF;
END min_two;
```

GLOBAL Variable Declarations

Informix Dynamic Server enables the definition of GLOBAL variables by using the GLOBAL keyword within the variable declaration. For example:

Informix Dynamic Server SPL

```
DEFINE GLOBAL gl_var INT;
```

This specifies that the GLOBAL variable gl_var is available to other procedures running within the same session. The first declaration of the GLOBAL variable establishes it within the Informix Dynamic Server global environment. Subsequent definitions of the same GLOBAL variable, within other procedures, are ignored.

The first procedure to define the GLOBAL variable can also set its initial value through the use of the DEFAULT clause. For example:

Informix Dynamic Server SPL

```
DEFINE GLOBAL gl_var INT DEFAULT 20;
```

If another stored procedure has already defined the GLOBAL variable within the global environment, the DEFAULT clause is ignored.

Therefore, if two procedures define the same GLOBAL variable with different DEFAULT values, the procedure executed first within the current session is the one that sets the GLOBAL variable's initial value.

Informix Dynamic Server GLOBAL variables can be emulated in Oracle by defining the variables within a package.

Variables defined within a package specification are available to the users of the package. The package specification emulates the per-session Informix Dynamic Server global environment.

Two Informix Dynamic Server procedures and the converted equivalent in Oracle are as follows.

Informix Dynamic Server SPL

```
CREATE PROCEDURE proc01()
        DEFINE GLOBAL gl_var INT DEFAULT 10;
        LET gl_var = gl_var + 1;
END PROCEDURE;

CREATE PROCEDURE proc02()
        DEFINE GLOBAL gl_var INT DEFAULT 20;
        LET gl_var = gl_var - 1;
END PROCEDURE;
```

Oracle PL/SQL Package

```
CREATE OR REPLACE PACKAGE informix.globalPkg AS
  gl_var  NUMBER;
END globalPkg;
```

Oracle PL/SQL

```
CREATE OR REPLACE PROCEDURE informix.proc01 AS
BEGIN
  IF(globalPkg.gl_var IS NULL) THEN
    globalPkg.gl_var := 10; /* Only set default if value is NULL */
  ENDIF;
  globalPkg.gl_var := globalPkg.gl_var +1;
END proc01;

CREATE OR REPLACE PROCEDURE informix.proc02 AS
BEGIN
  IF(globalPkg.gl_var IS NULL) THEN
    globalPkg.gl_var := 20; /* Only set default if value is NULL */
  ENDIF;
  globalPkg.gl_var := globalPkg.gl_var -5;
END proc02;
```

In the previous example, if `proc01` is executed first, the procedure checks if the value of the `globalPkg.gl_out` packaged variable is `NULL`. As this is the first time the package has been initialized, the variable contains a `NULL` value, therefore `proc01` sets the value of the `globalPkg.gl_var` variable to 10 before adding 1 to the value within the statement block. If `proc02` is then executed, the procedure again checks to see if the `globalPkg.gl_var` packaged variable has a `NULL` value. As `proc01` has previously set this variable (initially to 10 and then to 11), the boolean `IF` statement condition within `proc02` `IF(globalPkg.gl_var IS NULL)` does not return true and the value of 20 is not set. `proc02` then subtracts 5 from the current value of the variable, setting its final value to 6.

If `proc02` is executed first, it checks if the value of the `globalPkg.gl_out` variable is `NULL`. As this is the first time the package has been initialized, the variable contains a `NULL` value, therefore `proc02` sets the value of the `globalPkg.gl_out` variable to 20 before subtracting 5 from the value within the statement block. If `proc01` is then executed, the procedure again checks to see if the `globalPkg.gl_out` variable has a `NULL` value. As `proc02` has previously set this variable (initially to 20 and then to 15), the boolean `IF` statement condition `IF(INFORMIX.gl_var IS NULL)` returns false, therefore, the value of 10 is not set. proc01 then adds 1 to the current value of the variable, setting its final value to 16.

Both the converted procedures reflect the same functionality found within the original Informix Dynamic Server procedures.

LIKE and MATCHES Comparison Conditions

Informix Dynamic Server uses the LIKE and MATCHES comparison conditions to test for matching character strings. Oracle has only one of these pattern-matching constructs, the LIKE clause. The Informix Dynamic Server and Oracle LIKE clauses are functionally identical and so no conversion of the original pattern is required.

The Informix Dynamic Server specific MATCHES clause works in a similar way to the LIKE clause. The only difference between the two types of clause is in the range of pattern-matching wildcard characters available for use. A comparison of the Informix Dynamic Server MATCHES and Oracle LIKE wildcard operators are displayed in tables Table 3–2 and Table 3–3.

Table 3–2 **Informix Dynamic Server SPL MATCHES Clause Wildcards**

Wildcard	Description
*	Matches 0 or more characters
?	Matches any single character.
\	Removes the special significance of the next character used.
[..]	Matches any of the enclosed characters.
^	When used within the [..] wildcard operator, it matches any character not specified within the [..] character range

Table 3–3 Oracle PL/SQL LIKE Clause Wildcards

Wildcard	Description
%	Matches 0 or more characters.
_	Matches any single character.

If the [. .] pattern matching operator is not used within the original pattern, the Migration Workbench takes one of the following actions when it encounters a MATCHES clause:

- The MATCHES keyword is converted to the Oracle LIKE keyword.
- All ? characters within the original pattern are converted to functionally equivalent _ characters.
- All * characters within the original pattern are converted to functionally equivalent % characters.

If the [..] pattern matching operator is used within the original pattern and a character range is specified, the Migration Workbench converts each MATCHES clause that it encounters to a BETWEEN clause.

If the [..] pattern matching operator is used within the original pattern and no character range has been specified, the Migration Workbench converts each MATCHES clause it encounters to an Oracle IN clause.

The following table presents example Informix Dynamic Server MATCHES clauses and the converted Oracle equivalent:

MATCHES Statements	Conversion Results
MATCHES '[A-Z]'	BETWEEN 'A' AND 'Z'
MATCHES '[abcdefg]''	IN ('a','b','c','d','e','f','g')
MATCHES '*tennis*'	LIKE '%tennis%'
MATCHES '?ennifer*'	LIKE '_ennifer%'
MATCHES '[^qwerty]'	NOT IN ('q','w','e','r','t','y')
MATCHES '[^a-z]	NOT BETWEEN 'a' AND 'z'

If the Migration Workbench can not fully convert an Informix Dynamic Server MATCHES clause, it takes the following actions:

1. Generates a warning within the converted PL/SQL stored procedure code.
2. Converts the Informix Dynamic Server MATCHES keyword to the PL/SQL LIKE keyword.
3. The original pattern remains unchanged.

It is therefore necessary for you to manually convert any search pattern not handled by the Migration Workbench.

FOR LOOP Constructs

Informix Dynamic Server allows a number of FOR LOOP constructs that Oracle does not support. The most difficult of these to convert to Oracle is a FOR LOOP that mixes RANGE and EXPRESSION LISTs within the same iteration definition. In PL/SQL, it is necessary to split each defined iteration range into its own unique FOR LOOP or functionally equivalent PL/SQL code block.

In the following example, the converter splits the original Informix Dynamic Server FOR LOOP construct into four functionally equivalent PL/SQL code blocks. One

PL/SQL code block for each iteration range defined within the Informix Dynamic Server FOR LOOP construct. An example of an Informix Dynamic Server FOR LOOP construct and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
CREATE PROCEDURE forloop_example()
        DEFINE iterator_var, j INT;
        LET j = 10;
        FOR iterator_var IN (
            /* A range definition */
            1 TO 20 STEP 2,
             /* a SELECT statement */
            (SELECT aval from atable where avalid = j),
            /* An expression range definition */
            j+10 TO j-20,
            /* A singleton value */
            1000)
              INSERT INTO testtable VALUES(iterator_var);
        END FOR;
    END PROCEDURE;
```

Oracle PL/SQL

```
CREATE OR REPLACE PROCEDURE forloop_example AS
iterator_var        NUMBER(10);
j                   NUMBER(10);
ItoO_selcnt         NUMBER;
ItoO_rowcnt         NUMBER;

CURSOR cursor1 IS
    SELECT aval
    FROM atable
    WHERE avalid = j;
BEGIN
    j := 10;
    /* A range definition */
    iterator_var := 1;
LOOP
    INSERT INTO testtable
    VALUES(iterator_var);
    iterator_var := iterator_var + 2;
    EXIT WHEN (iterator_var >= 20);
END LOOP;
/* A SELECT statement */
FOR cursor1Record IN cursor1 LOOP
```

```
    iterator_var := cursor1Record.aval;
    INSERT INTO testtable
    VALUES(iterator_var);
END LOOP;
/* An expression range definition */
FOR iterator_var IN j + 10 .. j - 20 LOOP
    INSERT INTO testtable
    VALUES(iterator_var);
END LOOP;
/* A singleton value */
iterator_var := 1000;
INSERT INTO testtable
VALUES(iterator_var);
END forloop_example;
```

FOREACH LOOP Constructs

An Informix Dynamic Server FOREACH LOOP is the equivalent of a PL/SQL cursor. When an Informix Dynamic Server FOREACH statement executes, the database server:

1. Declares and implicitly opens a cursor.
2. Obtains the first row from the query contained within the FOREACH LOOP or it obtains the first set of values returned by the procedure.
3. Assigns each variable in the variable list the value of the corresponding value from the active set that the SELECT statement or called cursory procedure returns.
4. Executes the statement block.
5. Fetches the next row from the SELECT statement or procedure on each iteration and repeats steps 3, 4, and 5.
6. Terminates the loop when it finds no more rows that satisfy the SELECT statement or when no more data is returned from the procedure. The implicit cursor is closed when the loop terminates.

Within Informix Dynamic Server, FOREACH statements can be one of following types:

- FOREACH .. SELECT .. INTO Statement
- FOREACH CURSOR Statement
- FOREACH Execute Procedure Statement

FOREACH .. SELECT .. INTO Statement The Migration Workbench emulates FOREACH .. SELECT .. INTO statement in PL/SQL by converting the Informix Dynamic Server FOR EACH SELECT statement into a cursor definition. Then it iterates over the cursor contents, assigning the values within the current cursor row to the original list of variables defined within the SELECT INTO statement. Migration Workbench repeats this process until no more data is found. An example of a FOREACH..SELECT..INTO statement and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
DECLARE name VARCHAR(30);
DECLARE address VARCHAR(255);
FOREACH SELECT ename, eaddress INTO name, address FROM emp
  INSERT INTO mailing_list VALUES(name, address);
END FOREACH;
```

Oracle PL/SQL

```
/* Declare original variables */
name VARCHAR(30);
address VARCHAR(255);

/* Declare a cursor using the original SELECT statement
   Notice how the converter has now named the cursor within
   PL/SQL */
CURSOR cursor1 IS
SELECT ename, eaddress
FROM emp;
BEGIN
/* Open the previously declared (now) named cursor */
OPEN cursor1;
/* Iterate over the cursor contents */
LOOP
    /* Fetch the values of the cursor's current row
       into the original variables */
    FETCH cursor1
    INTO name,
         address;
    /* Exit the LOOP when no more data found */
    EXIT WHEN cursor1%NOTFOUND;
    /* The original statement block */
    INSERT INTO mailing_list
    VALUES(name,
           address);
```

```
END LOOP;
/* Close the cursor */
CLOSE cursor1;
END;
```

FOREACH CURSOR Statement An Informix Dynamic Server FOREACH statement can contain an explicitly named cursor. This enables the use of the WHERE CURRENT OF clause within the statement block contained within the FOREACH construct. The Informix Dynamic Server FOREACH cursor statement is converted to PL/SQL in a similar way to the FOREACH.. SELECT .. INTO statement. The named cursor is defined within the PL/SQL procedure, opened, and the contents iterated over until no more data is found. A FOREACH CURSOR statement and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
CREATE PROCEDURE "informix".update_list
DECLARE name VARCHAR(30);
DECLARE address VARCHAR(255);
FOREACH namedCursor FOR
  SELECT ename, eaddress INTO name, address FROM emp
  INSERT INTO mailing_list VALUES(name, address);
  IF(ename="McAllister") THEN
    UPDATE emp SET sal = sal + 2000 WHERE CURRENT OF namedCursor;
    CONTINUE FOREACH;
  END IF
  UPDATE emp SET sal = sal + 1000 WHERE CURRENT OF namedCursor;
END FOREACH
END PROCEDURE
```

Oracle PL/SQL

```
CREATE OR REPLACE PROCEDURE "informix".update_list AS

name                VARCHAR2(30);
address             VARCHAR2(255);
ItoO_selcnt         NUMBER;
ItoO_rowcnt         NUMBER;

CURSOR namedCursor IS
    SELECT ename,
           eaddress
    FROM emp FOR UPDATE;
BEGIN
```

```
    OPEN namedCursor;
    LOOP
        FETCH namedCursor
        INTO name,
            address;
        EXIT WHEN namedCursor%NOTFOUND;
        INSERT INTO mailing_list
        VALUES(name,
               address);
        IF ( ename = 'McAllister' ) THEN
            UPDATE emp
            SET sal  =  sal + 2000
            WHERE CURRENT OF namedCursor;
            /* SPCONV-MSG:(CONTINUE FOREACH) Statement emulated using GOTO
               statement and LABEL definition. */
            GOTO Continue_ForEach1;
        END IF;
    UPDATE emp
    SET sal  =  sal + 1000
    WHERE CURRENT OF namedCursor;
    <<Continue_ForEach1>>
    NULL;
END LOOP;
CLOSE namedCursor;
END update_list;
```

For more information on the translation of Informix Dynamic Server CONTINUE statements, see Converting CONTINUE Statements.

FOREACH Execute Procedure Statement If a FOREACH execute statement is encountered by the convertor, it assumes the procedure being called is a cursory procedure. As cursory procedures are automatically converted to utilize PL/SQL REF CURSORS, the procedure being called always return a REF CURSOR as it's last parameter. This cursor variable contains the full set of results returned by the called stored procedures.

The Informix Dynamic Server FOREACH EXECUTE statement can be emulated by iterating over the contents of the cursor variable returned by the converted cursory procedure.

The following shows an example of the Informix Dynamic Server FOREACH EXECUTE statement repeatedly executing a cursory procedure bar() until no more results are returned and the converted equivalent in Oracle:

Informix Dynamic Server SPL

```
FOREACH EXECUTE PROCEDURE bar(100,200) INTO i
  INSERT INTO tab2 VALUES(i);
END FOREACH
```

Oracle PL/SQL

```
/* DEFINE a cursor variable of the correct type */
OMWB_cv1 OMWB_emulation.globalPkg.RCT1;

/* Cursor variable added to the call to procedure bar() */
bar(100,200,OMWB_cv1);
/* Iterate over the cursor contents */
LOOP
  /* FETCH the contents into the original variable */
  FETCH OMWB_cv1
  INTO i;
  /* EXIT the LOOP when no more data found */
  EXIT WHEN OMWB_cv1%NOTFOUND;
  /* execute statement block */
  INSERT INTO tab2 VALUES(i);
END LOOP;
```

Compound LET Statements

Informix Dynamic Server uses the LET statement to assign values to variables. PL/SQL only allows simple assignments, which assign a single value to a single variable. Informix Dynamic Server SPL allows compound assignments, which assign values to two or more variables within the same statement.

In order to convert compound LET statements into functionally equivalent PL/SQL code, the converter splits the Informix Dynamic Server compound assignment statement into logically equivalent simple assignment statements.

An example of both Informix Dynamic Server simple assignments and compound assignments and the converted equivalent in Oracle are as follows:

Informix Dynamic Server SPL

```
/* Simple assignment */
LET a = 10;
/* Compound assignment */
LET b,c = 20,30;
```

Oracle PL/SQL

```
/* Simple assignment conversion*/
a := 10;
```

```
/* Compound assignment conversion*/
b := 20;
c := 30;
```

The two original Informix Dynamic Server LET statements have been converted into three logically equivalent PL / SQL statements. One PL / SQL statement for every variable used within both Informix Dynamic Server LET statements. Informix Dynamic Server also enables SELECT statements and PROCEDURE calls to assign values to variables within a LET statement.

Using SELECT Statements in LET Assignment Statements Informix Dynamic Server enables the use of a SELECT statement as part of the LET statement assignment list.

The following shows an example of an Informix Dynamic Server SELECT statement as part of a LET statement assignment list and the converted equivalent in PL / SQL:

Informix Dynamic Server SPL

```
LET enum = (SELECT empnum FROM emp WHERE empname = "McAllister");
```

Oracle PL/SQL

```
SELECT empnum INTO enum FROM emp WHERE empname = 'McAllister';
```

Calling Procedures in LET Assignment Statements Informix Dynamic Server enables the use of a procedure call within a LET statement. The procedure may return more than one value into a list of variables.

An example of an Informix Dynamic Server procedure call that returns three values into the variables a, b, and c and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
LET a,b,c = someProc(100,200);
```

Oracle PL/SQL

```
someProc(100, 200, OMWB_outparameter1 => a, OMWB_outparameter2 => b, OMWB_
outparameter3 => c);
```

The someProc procedure is converted to pass these values back as Oracle OUT parameters. These OUT parameters are explicitly named:

```
OMWB_outparamater<number>
```

Thus, if the original Informix Dynamic Server stored procedure returned *n* values, the converter adds *n* OUT parameters to the converted stored procedure, sequentially named OMWB_outparameter1 .. OMWB_outparameter*n*.

An example of an Informix Dynamic Server LET statement which assigns a value to only one variable and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
LET a = anotherProc(200);
```

In the above example, the converter assumes that the procedure being called has been converted to a function within PL/SQL and convert the statement to read:

Oracle PL/SQL

```
a := anotherProc(200);
```

For more information on named and positional parameter passing notation, see the following:

- *PL/SQL User's Guide and Reference Release 1 (9.0.1)*
- Parameter Passing

Converting CONTINUE Statements

An Informix Dynamic Server CONTINUE statement is used to start the next iteration of the innermost FOR, FOREACH or WHILE loop. When a CONTINUE statement is encountered, the rest of the statements contained within the innermost LOOP of the innermost TYPE are skipped and execution continues at the next iteration of the LOOP.

Oracle PL/SQL does not contain a CONTINUE statement so Migration Workbench emulates the statement by using a PL/SQL LABEL definition and a code branching GOTO statement. This label is defined as the penultimate statement within the converted looping constructs statement block. As PL/SQL requires the statement directly following a label definition to be executable, Migration Workbench adds a NULL statement directly after the inserted label definition. The END LOOP PL/SQL statement is declarative, not executable, whereas, the NULL statement within PL/SQL is executable.

An example of an Informix Dynamic Server CONTINUE statement and its converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
CREATE PROCEDURE continue_test()
```

```
indx  INT;
FOR indx IN 1 TO 10 LOOP
  IF(indx = 5) THEN
    CONTINUE FOR;
  END IF
  INSERT INTO tab VALUES(indx) ;
END FOR
```

Oracle PL/SQL

```
CREATE OR REPLACE PROCEDURE continue_test AS
indx INTEGER;
BEGIN
  FOR indx IN 1 .. 10 LOOP
    IF(indx = 5) THEN
      /* The original Informix CONTINUE statement has been
          replaced by a PL/SQL GOTO statement*/
      GOTO FOR_LABEL1;
    END IF
    /* Original statement block */
    INSERT INTO tab VALUES(indx) ;
    /* The following label definition are placed at the end of the
        LOOP constructs statement block*/
    <<FOR_LABEL1>>
    /* Label definitions have to be followed by an executable
       statement. As PL/SQL treats the END LOOP statement as
       being declarative, a NULL statement is placed after
       the label definition. NULL statements within PL/SQL are
       classed as being executable */
    NULL;
  END LOOP;
END;
```

Converting RETURN WITH RESUME Statements

Informix Dynamic Server enables procedures to return multiple sets of results by the inclusion of the WITH RESUME keywords after the RETURN statement. An Informix Dynamic Server procedure of this type is called a cursory procedure.

The result set returned by an Informix Dynamic Server cursory procedure is emulated within Oracle by adding a REF CURSOR variable to the parameter list of the converted PL/SQL procedure.

This cursor variable stores the complete set of results returned from the stored procedure.

An Oracle temporary table is used to return an identical set of results in an identical order within the PL/SQL procedure as would have been returned in the original Informix Dynamic Server procedure. This temporary table stores the interim results in an ordered sequence.

In the following Informix Dynamic Server example, the procedure returns every continuous integer value between 1 and 100, except the values between 49 and 61, in ascending order to the parent procedure or calling environment.

In order to successfully emulate the order in which these results are returned in Informix Dynamic Server, the Migration Workbench creates a GLOBAL TEMPORARY TABLE specifically to store the interim procedure results. The Migration Workbench then converts the Informix Dynamic Server RETURN WITH RESUME statement to INSERT results into this temporary table. The Migration Workbench then uses the temporary table to populate the cursor returned to the calling environment.

An example of a RETURN WITH RESUME statement and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
CREATE PROCEDURE resume_test() RETURNING NUMBER;
indx INT;
FOR indx = 1 to 100 LOOP
  IF(indx > 49 and indx < 61) THEN
    CONTINUE FOR;
  END IF
  RETURN indx WITH RESUME;
END FOR;
END resume_test;
```

Oracle PL/SQL temporary table DDL statement

```
CREATE GLOBAL TEMPORARY TABLE resume_testTable(
/* The first column 'col00' is used to create an ordered
    SELECT statement when populating the REF CURSOR
    OUT parameter to the procedure */
col00  NUMBER,
col01  NUMBER)
ON COMMIT DELETE ROWS;
```

Oracle PL/SQL Converted Procedure

```
CREATE OR REPLACE PROCEDURE resume_test(
/* Define the cursor used to pass back the complete list
   of results to the calling environment as an OUT
   parameter  */
```

```
OMWB_ret_cv OUT OMWB_emulation.globalPkg.RCT1)
AS
indx INTEGER;
/* A counter is automatically added by the converter.
   This is used to INSERT a sequential set of results
   into the GLOBAL TEMPORARY TABLE resume_testTable. */
OMWB_resume_testSeq INTEGER := 0;
BEGIN
/* Clear the temporary table of old results at the start
   of the procedure */
DELETE FROM resume_testTable;
FOR indx IN 1 .. 100 LOOP
  IF(indx > 49 and indx < 61) THEN
    /* CONTINUE statement emulated by using a GOTO
       statement and LABEL definition */
    GOTO FOR_LABEL1;
  END IF;
  /* Return with resume statement converted to INSERT the
     return data into this procedures GLOBAL TEMPORARY
     TABLE.
     The OMWB_resume_testSeq variable is used in order to
     create a continuous sequence of values when ordering
     the results for insertion into the return cursor
     OMWB_ret_cv */
  INSERT INTO resume_testTable
  VALUES(OMWB_resume_testSeq,
         indx);
  /* Now we increment the sequence variable ready for the
     next converted RETURN WITH RESUME statement */
  OMWB_resume_testSeq := OMWB_resume_testSeq + 1;
  /* Label definition used by the GOTO statement above */
  <<FOR_LABEL1>>
  NULL;
END LOOP;
/* The temporary table is then used to populate the
   REF CURSOR we return to the calling environment.
   The first column is used to return the results from
   the select statement in an ordered fashion and is
   never made part of the return data */
  OPEN OMWB_ret_cv FOR
  SELECT col01
  FROM resume_testTable
  ORDER BY col00;
END resume_test;
```

When the PL/SQL procedure in this example is called, it deletes past results from the associated temporary table of the procedure using the DELETE FROM syntax. For example:

Oracle PL/SQL

```
DELETE FROM resume_testTable;
```

The table is now void of results and ready for use within the procedure. The Informix Dynamic Server RETURN WITH RESUME statement is then converted to INSERT results into this temporary table. An INTEGER variable called:

```
OMWB_<procedure name>Seq
```

This is automatically added to the variable declaration section within the stored procedure. This variable is used to insert an ordered sequence number into the first column of the resume_testTable table.

To populate the cursor variable designed to return the results to the calling environment, the converter then adds an OPEN CURSOR .. FOR .. SELECT statement as the last executable line of the procedure. At this stage of the procedures execution, the temporary table is populated with a full set of results.

The first column of the temporary table is used within the ORDER BY section of the last SELECT statement to populate the cursor rows with the ordered temporary table data. The procedure completes execution and the populated cursor is returned to the calling environment.

Built-in Functions

Some built-in functions within Informix Dynamic Server are not available in Oracle. These functions are emulated within Oracle using the utilities package. Migration Workbench automatically creates this package within the destination Oracle database. It contains a suite of PL/SQL stored functions and procedures that mimic the functionality of the following Informix Dynamic Server built-in procedures:

- HEX
- DAY
- MONTH
- WEEKDAY
- YEAR

- MDY
- TRACE

The Migration Workbench creates a new user within the destination Oracle database. The user name is OMWB_emulation and the password is oracle. This OMWB_emulation users schema stores the utilities package. To enable access to this package to all database users, the Migration Workbench executes the following statement:

Oracle PL/SQL

```
GRANT EXECUTE ON OMWB_emulation.utilities TO PUBLIC;
```

Every time the stored procedure converter encounters a reference to one of the unsupported built-in functions within the Informix Dynamic Server SPL code, it generates a reference to the equivalent emulation function within the OMWB_emulation users utilities package. An example of a SPL statement converted to reference the OMWB_emulation.utilities.HEX emulation function within Oracle is as follows:

Informix Dynamic Server SPL

```
LET a = HEX(255);
```

Oracle PL/SQL

```
a := OMWB_emulation.utilities.HEX(255);
```

With the exception of the Informix Dynamic Server TRACE function, all emulation functions have the same names as their Informix Dynamic Server counterpart. The TRACE statement is converted to reference a procedure named DEBUG within the OMWB_emulation.utilities package.

> **Caution:** It is imperative that you test the utilities package and all functions and procedures within before implementation in a production environment.

Converting the SYSTEM Statement

The SYSTEM statement enables operating system commands to be executed from within an Informix Dynamic Server stored procedure. For example:

Informix Dynamic Server SPL

```
SYSTEM ‘ ls -al /tmp/salary_upgrades > /tmp/salary_upgrades/totals.out’;
```

Oracle does not have any such SYSTEM statement so it is necessary to emulate the Informix Dynamic Server SYSTEM functionality by using an Oracle external procedure. This external procedure is written in C and compiled into an executable. A stored procedure named SHELL is then associated with a call to the executable.

In essence, a call to the associated PL/SQL stored procedure actually invokes the compiled executable resident on the file system. This binary executable then performs the operating system command passed into the SHELL stored procedure. You need to manually compile this executable before emulation of the Informix Dynamic Server SYSTEM command can commence.

The Migration Workbench creates a placeholder PL/SQL stored procedure named SHELL within the OMWB_Emulation users schema. It then converts each Informix Dynamic Server SYSTEM statement to reference this placeholder procedure. For example, the previous SYSTEM statement is converted into the following PL/SQL code:

Oracle PL/SQL

```
OMWB_Emulation.SHELL(' ls -al /tmp/salary_upgrades >
/tmp/salary_upgrades/totals.out');
```

This placeholder procedure currently contains no executable code, it is a stub created within the destination database so that any procedure containing references to it does not fail compilation.

Oracle invalidates a stored procedure if any other stored procedure it references is itself invalid. Therefore, the stub procedure is required until the set-up tasks have been performed. If the stub procedure is invoked prior to the set-up tasks being performed, the string containing the operating system command is not executed.

Set-Up Tasks for Configuring the SHELL Procedure In order to configure the SHELL procedure so that it executes the operating system command, you should first perform the following set-up tasks on the destination server:

Note: The following set-up tasks are specific to Windows NT.

---

1. Download and install Borland's free C++ compiler from the Web site at:

http://www.borland.com

2. Create the file shell.c:

```
==============begin shell.c=================
```

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

void __declspec(dllexport) sh(char *);
void sh(char *cmd)
{
system(cmd);
}

============end shell.c======================
```

3. Create a test program shell_run.c:

```
=============begin shell_run.c===============

void __declspec(dllimport)ch (char*);

int main(int argc, char *argv[])
{
sh(argv[1]);
return 0;
}

============end shell_run.c==================
```

4. Create and run `shell_compile.bat` that compiles and link shell.c and shell_run_c:

```
============begin shell_compile.bat =========

bcc32 -WD shell.c
implib shell.lib shell.dll
bcc32 shell_run.c shell.lib

============end shell_compile.bat ===========
```

5. Test shell.dll by issuing the following command on the DOS prompt:

```
C:\> shell_run "any operating system command"
```

6. Configure the destination databases `listener.ora` and `tnsnames.ora` files for external procedures.

   For the configuration of external procedures, you need to define a `tnsnames.ora` entry: `extproc_connection_data`.

When the server references an external-procedure, it looks into the `tnsnames.ora` file to find the listener address. The alias used is the hard-coded `extproc_connection_data` value. This alias contains the address of the listener process and the SID for the extproc agent. With this info, the server contacts the listener and the listener spawns the new extproc-process.

Add the following entry to the `tnsnames.ora` file:

```
EXTPROC_CONNECTION_DATA =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC))
    )
    (CONNECT_DATA =
      (SID = PLSExtProc_817)
      (PRESENTATION = RO)
    )
  )
```

Configure the listener.ora file, add an `SID_DESC` entry similar to the following:

```
SID_LIST_LISTENER =
  (SID_LIST =
    (SID_DESC =
      (SID_NAME = PLSExtProc_817)
      (ORACLE_HOME = <ORACLE_HOME>)
      (PROGRAM = extproc)
    )
   )
```

7. Create the external library and replace the stub `OMWB_Emulation.SHELL` wrapper procedure using SQL*Plus:

```
SQL> create library shell_lib is 'shell.dll';
SQL> create or replace procedure OMWB_emulation.SHELL (
cmd IN varchar2)
as external
library shell_lib
name "_sh"
language C
parameters (
cmd string);
/
```

8. Test the external library from the SQL*Plus command line:

```
SQL> exec shell('any operating system command');
```

The external procedure will execute all operating system commands using Oracle permissions. For example, the following statement creates the hello.txt file within the /home/myname directory:

```
OMWB_emulation.SHELL('echo "Hello" > /home/myname/hello.txt');
```

The hello.txt file is owned by Oracle. To reassign the file to another user, you should alter the call to the SHELL procedure. For example:

```
OMWB_emulation.SHELL('echo "Hello" > /home/myname/hello.txt; chown myname
hello.txt');
```

Converting TRACE Statements

The Informix Dynamic Server TRACE statement is used to control the generation of debugging output. The TRACE statement sends output to the file specified by the SET DEBUG FILE statement. Tracing within Informix Dynamic Server prints the current values of the following items:

- Variables
- Procedure arguments
- Return values
- SQL error codes
- ISAM error codes

The Informix Dynamic Server TRACE statement can also be used to print expressions to the debug file using the syntax: TRACE expression. For example:

Informix Dynamic Server SPL

```
TRACE "This is a trace statement and is written out to the debug log";
```

All statements are traced within Informix Dynamic Server by the issue of the TRACE ON command. This implies that all statements and procedure calls are traced, such as the value of all variables before they are used and the return values of procedure calls. The Informix Dynamic Server statement TRACE OFF is used in order to turn tracing off. The TRACE *<expression>* statement can still be used even if the TRACE OFF statement has been issued.

The Migration Workbench only supports the conversion of the Informix Dynamic Server TRACE *<expression>* statement. All other TRACE statements cause the converter to flag a warning and output the original TRACE statement within the PL/SQL code as a single line comment along with an accompanying executable NULL statement. An example of an unsupported TRACE statement and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
TRACE PROCEDURE;
```

Oracle PL/SQL

```
/* SPCONV-WRN:(TRACE PROCEDURE) Statement not converted. Manual conversion
required. */
--TRACE PROCEDURE;
NULL;
```

The TRACE *<expression>* statement is emulated using the DEBUG stored procedure resident within the utilities package. The DEBUG stored procedure is generated automatically by the Migration Workbench.

The DEBUG stored procedure enables the logging of debug messages to the console window using the DBMS_OUTPUT package, a table within the database or, using the UTL_FILE package, a flat file stored locally on the file system. The supplied DEBUG stored procedure logs messages to a table called debug_table by default.

The Migration Workbench converts all Informix Dynamic Server TRACE <expression> statements to reference the DEBUG stored procedure. For example:

Informix Dynamic Server SPL

```
TRACE "This is a trace statement and is written out to the debug log";
```

Oracle PL/SQL

```
OMWB_emulation.utilities.DEBUG('This is a trace statement and is written out to
the debug log');
```

Informix Dynamic Server TRACE *<expression>* statements are used to build a log of systematic debug information. Because of this, converted TRACE expression statements can become a powerful quality assurance monitor. You can compare the logs produced by the original Informix Dynamic Server TRACE statements against the logs built by the converted statements within the destination Oracle database. This may aid in the unit testing of each converted stored procedure.

For a code listing of the complete utilities package, refer to Appendix1.

Set Up Tasks for the DEBUG Procedure

The DEBUG procedure is designed by default to log messages to the debug_table resident under the OMWB_emulation user's schema. The following shows the DDL statement that the Migration Workbench uses to construct the debug_table:

Oracle PL/SQL

```
CREATE TABLE debug_table(
log_date       DATE,
log_user       VARCHAR(100),
log_message    VARCHAR(4000))
```

The Migration Workbench automatically creates and executes the appropriate database grants on this table. Therefore, in order to use the OMWB_emulation.utilities.DEBUG procedure, immediate set-up tasks are not necessary.

If you want to log all DEBUG messages to a flat file, you should first create a UTL_FILE_DIR entry within the init.ora initialization file of the destination Oracle database.

This init.ora parameter defines a list of directories into which the UTL_FILE package can write. The directories specified have to reside on the database servers local file system.

In the init.ora file, each accessible directory is stipulated by a line such as

```
utl_file_dir = D:\Oracle\Migration\Debug
```

The previous line enables the UTL_FILE package to write to files present within the D:\Oracle\Migration\Debug directory. Access to files within subdirectories is forbidden. You must explicitly define each directory within the init.ora file.

Using the DEBUG Procedure After have added the UTL_FILE_DIR entries to the init.ora initialization file, you need to configure the DEBUG procedure. To do this, you alter the value of the following utilities package variables:

- utilities.DebugOut
- utilities.DebugFile:
- utilities.DebugDir

The utilities.DebugOut variable is an integer value that indicates whether to log trace messages to a flat file, the console windrow, or a table within the database. You can set this variable programmatically within stored procedures by including the following line of PL / SQL code:

Oracle PL/SQL

```
OMWB_Emulation.utilities.DebugOut := <variable value>;
```

The variable value can be one of the following:

- A value of 1 instructs the DEBUG procedure to log all converted trace messages to a file. The filename used is specified by the value of the utilities.DebugFile variable. The value of the utilities.DebugDir variable specifies the directory where this file is located.
- A value of 2 instructs the DEBUG procedure to log all converted trace messages to the console window.
- Any other value instructs the DEBUG procedure to log messages to a table named debug_table resident under the OMWB_Emulation users schema.

If the DEBUG procedure has been configured to log trace messages to a file, the value of the utilities.DebugFile variable determines the filename. You can set this variable programmatically within stored procedures by including the following:

```
OMWB_Emulation.utilities.DebugFile := <variable value>;
```

The value for this variable has to be a string expression that evaluates to a legal operating system filename. For more information on this variable, see the SET DEBUG FILE Statement topic.

If the procedure has been configured to log trace messages to a file, the variable value of the utilities.DebugDir variable determines the directory where the file is created. You can set this variable programmatically within stored procedures by including the following:

```
OMWB_Emulation.utilities.DebugDir := <variable value>;
```

The value for this variable has to be a string expression that evaluates to a legal operating system file path. The file path has to exist at runtime or an error is raised. Additionally, this file path must have a matching UTL_FILE_DIR entry.

For example, in order to configure a stored procedure to log converted trace messages to a file named `procA.out` within the `D:\logs` directory, include the following lines within the stored procedure code:

```
utilities.DebugOut := 1;
utilities.DebugFile := 'procA.out';
utilities.DebugDir := 'D:\logs\';
```

Alternatively, in order to log messages to the console window, include the following:

```
utilities.DebugOut := 2;
```

In order to log converted trace messages to the `debug_table`, set the `utilities.DebugOut` variable to any value except 1 or 2. Therefore, any one of the following three values is legal:

```
utilities.DebugOut := 3;
utilities.DebugOut := 300000;
utilities.DebugOut := NULL;
```

`SET DEBUG FILE` Statement Informix Dynamic Server uses the `SET DEBUG FILE` statement to indicate the file where `TRACE` messages are logged. The Migration Workbench emulates the Informix Dynamic Server `TRACE` statement by using the `utilities.DEBUG` procedure. This PL/SQL stored procedure offers an option that enables you to log debug messages to a flat file stored locally on the file system.

If the `DEBUG` procedure has been configured to log messages to a file then the converted `SET DEBUG FILE` statement determines the name of the file within the destination Oracle database.

The following shows an example of an Informix Dynamic Server `SET DEBUG FILE` statement:

Informix Dynamic Server SPL

```
SET DEBUG FILE TO 'errorlog.out';
```

The Migration Workbench converts this statement by setting the name of the file written to by the `utilities.DEBUG` procedure to `errorlog.out`. The converted SET DEBUG FILE statement sets the value of a variable named `DebugFile` defined within the `utilities` package. The following shows the converted PL/SQL code:

Oracle PL/SQL

```
OMWB_Emulation.utilities.DebugFile := 'errorlog.out';
```

The filename stipulated within the Informix Dynamic Server SET DEBUG FILE statement may also contain a file path, for example

Informix Dynamic Server SPL

```
SET DEBUG FILE TO 'D:\informix\audit\errorlog.out';
```

If this is the case, the converter extracts the file path and use it to set the value of a variable named utilities.DebugDir also defined within the utilities package. For example, the preceding SET DEBUG FILE statement is converted into the following lines:

Oracle PL/SQL

```
OMWB_Emulation.utilities.DebugFile := 'errorlog.out';
OMWB_Emulation.utilities.DebugDir := 'D:\informix\audit\';
```

For further information on the use of the DEBUG package, see the Converting TRACE Statements topic. A code listing of the utilities package can be viewed in Appendix 1.

BEGIN WORK Statement Informix Dynamic Server uses the BEGIN WORK statement to start a transaction. The Migration Workbench converts this statement into a named PL/SQL savepoint. The BEGIN WORK statement and its equivalent in Oracle are as follows:

Informix Dynamic Server SPL

```
BEGIN WORK;
```

Oracle PL/SQL

```
SAVEPOINT SP1;
```

Savepoints within Oracle are used to mark a place within a transaction. Once the savepoint is defined, it is possible to rollback to it using the ROLLBACK WORK statement.

The Migration Workbench automatically generates a savepoint name of the form SP<*integer*>. The integer value starts at 1 and increments each time a new BEGIN WORK statement is converted. Using savepoints in this way enables finer transaction control within the Oracle stored procedure. It is recommended that you manually convert the generated stored procedure to take full advantage of the nested savepoint capabilities within Oracle. For more information on Oracle savepoints, see the PL/SQL User's Guide and Reference Release 1 (9.0.1).

ROLLBACK WORK Statement Informix Dynamic Server uses the ROLLBACK WORK statement to undo any of the changes made since the beginning of the transaction. The Oracle ROLLBACK statement acts in an identical way. However, only part of the transaction need be undone. To achieve this Oracle SAVEPOINT definitions within the PL/SQL stored procedure code are used.

The Migration Workbench automatically converts Informix Dynamic Server BEGIN WORK statements into Oracle SAVEPOINTs. These savepoints are then integrated into the conversion of the original Informix Dynamic Server ROLLBACK WORK statement. An example of the ROLLBACK WORK and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
BEGIN WORK
INSERT INTO student VALUES(300, 'Tara', 'Finn');
INSERT INTO major VALUES(300, 1237);
ROLLBACK WORK;
```

Oracle PL/SQL

```
SAVEPOINT SP1;
INSERT INTO student
VALUES(300,
       'Tara',
       'Finn');
INSERT INTO major
VALUES(300,
       1237);
ROLLBACK TO SAVEPOINT SP1;
```

SELECT Statements as Conditions

Informix Dynamic Server allows you to use SELECT statements within an IF statement condition. Oracle does not enable you to use SELECT queries as conditions in this way. In order to emulate this Informix Dynamic Server statement, the Migration Workbench automatically generates a Boolean variable within the PL/SQL code. It then sets the value of this Boolean variable within a SELECT.. FROM DUAL statement that incorporates the original SELECT statement within the WHERE clause.

DUAL is a table automatically created by Oracle along with the data dictionary. DUAL is in the schema of the user SYS, but is accessible by the name DUAL to all users. It has one column, DUMMY, defined to be VARCHAR2(1), and contains one row with

a value 'X'. Selecting from the DUAL table is useful for computing a constant expression with the SELECT statement. Because DUAL has only one row, the constant is returned only once.

An Informix Dynamic Server example of a SELECT statement used as a condition and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
IF EXISTS (SELECT content_id
           FROM slistcontent
           WHERE list_id = sp_list_id
           AND thing_id = new_thing)
THEN
  /* statement block */
END IF;
```

Oracle PL/SQL

```
/* SPCONV-MSG:(SUBQUERY) Subquery within IF statement emulated by using Boolean
variable. */
OMWB_tempBoolVar1 := FALSE;
SELECT TRUE INTO OMWB_tempBoolVar2 FROM DUAL
WHERE EXISTS (SELECT content_id
FROM informix.slistcontent
WHERE list_id = sp_list_id
AND thing_id = new_thing);
IF(OMWB_tempBoolVar1) THEN
/* statement block */
END IF;
```

The Migration Workbench automatically adds the Boolean variable OMWB_tempBoolVar1 to the generated PL/SQL code. The value of this variable is then set by the SELECT .. FROM DUAL statement, which itself contains the original Informix Dynamic Server SELECT statement as part of the WHERE clause. The Boolean variable added by the converter is then used within the IF condition.

Exception Blocks

Informix Dynamic Server exception blocks are declared prior to the statement block they encapsulate. Oracle exception blocks are declared at the end of the statement block they encapsulate. This causes the Migration Workbench to transfer the converted exception handling code to the bottom of the statement block within the generated PL/SQL code.

If the exception block have been defined with the keywords `WITH RESUME`, the following warning is also output within the generated PL/SQL code:

Informix Dynamic Server SPL

```
/* SPCONV-WRN:(WITH RESUME) Oracle has no such construct. Manual conversion
required. */
```

The converter automatically maps the following Informix Dynamic Server error numbers to Oracle predefined exceptions. When the convertor encounters any Informix Dynamic Server error number not presented within the following table, it outputs the error number as a comment within the generated PL/SQL stored procedure and indicate that manual conversion of the exception block is required.

Informix Dynamic Server Error Number	Oracle Predefined Exception
-239	DUP_VAL_ON_INDEX
100	NO_DATA_FOUND
-259	INVALID_CURSOR
-415	VALUE_ERROR
-1213	INVALID_NUMBER
-1214	VALUE_ERROR
-1215	VALUE_ERROR
-1348	ZERO_DIVIDE
-248	TOO_MANY_ROWS

The following shows an example of an Informix Dynamic Server stored procedure that defines one exception block to catch multiple errors and it's converted equivalent in Oracle PL/SQL:

Informix Dynamic Server SPL

```
CREATE PROCEDURE "root".add_slist_thing(
v_uid like PHPUser.user_id,
v_lstid like ShoppingList.list_id,
v_thgid like Thing.thing_id,
v_cntdesc like SListContent.content_desc)
RETURNING smallint;
BEGIN
on exception in (-239, -310)
```

```
    return -2;
end exception;

insert into listcontent
values (v_lstid, v_uid, v_thgid, v_cntdesc);
let returnCode = upd_slist_date(v_lstid, v_uid);
return returncode;
END
END PROCEDURE;
```

Oracle PL/SQL

```
CREATE OR REPLACE FUNCTION root.add_slist_thing(
v_uid_IN               informix.PHPUser.user_id%TYPE,
v_lstid_IN             informix.ShoppingList.list_id%TYPE,
v_thgid_IN             informix.Thing.thing_id%TYPE,
v_cntdesc_IN           informix.SListContent.content_desc%TYPE) RETURN NUMBER AS

v_uid               informix.PHPUser.user_id%TYPE := v_uid_IN;
v_lstid             informix.ShoppingList.list_id%TYPE := v_lstid_IN;
v_thgid             informix.Thing.thing_id%TYPE := v_thgid_IN;
v_cntdesc           informix.SListContent.content_desc%TYPE := v_cntdesc_IN;
ItoO_selcnt         NUMBER;
ItoO_rowcnt         NUMBER;

BEGIN
BEGIN
INSERT INTO listcontent
VALUES(v_lstid,
       v_uid,
       v_thgid,
       v_cntdesc);
returnCode := upd_slist_date ( v_lstid , v_uid );
RETURN returncode;
EXCEPTION
    /* SPCONV-WRN:(EXCEPTION) Could not convert 1 Informix error number to a
    predefined Oracle exception. Manual conversion required. */
    WHEN DUP_VAL_ON_INDEX THEN  /* Not Converted : -310 */
        RETURN - 2;
END;
END add_slist_thing;
```

RAISE EXCEPTION Statements

The Informix Dynamic Server RAISE EXCEPTION statement is used to simulate the generation of an error message. It passes program control to the execution handler that is designed to explicitly catch the raised exception. The execution of the stored procedure can then continue.

If the RAISE EXCEPTION statement is encountered within the Informix Dynamic Server stored procedure, it is converted into a call to the built-in Oracle RAISE_APPLICATION_ERROR function. This function enables the raising of errors containing user defined messages. The following shows an example of the RAISE EXCEPTION statement and its conversion to an Oracle PL/SQL RAISE_APPLICATION_ERROR function call:

Informix Dynamic Server SPL

```
RAISE EXCEPTION -208, 0, 'Cannot insert. Required datafile ' || datafilename ||
' missing. insert_seq_proc procedure';
```

Oracle PL/SQL

```
RAISE_APPLICATION_ERROR(-299999, /* Informix error number : -208, 0 */ "Cannot
insert. Required datafile ' || datafilename || ' missing. insert_seq_proc
procedure");
```

The following is an abbreviated syntax of the Oracle RAISE_APPLICATION_ERROR function:

Oracle PL/SQL Syntax

```
RAISE_APPLICATION_ERROR(error number, error message);
```

Where the error number is a number between -20000 and -20999 and error message the text associated with this error. An additional keep errors parameter is also available. For more information on this parameter, see the PL/SQL User's Guide and Reference Release 1 (9.0.1).

The original error number used within the Informix Dynamic Server RAISE EXCEPTION statement is output as a comment within the call to RAISE_APPLICATION_ERROR.

The Informix Dynamic Server RAISE EXCEPTION statement is always used in conjunction with an ON EXCEPTION statement. The RAISE EXCEPTION statement simulates an error condition and program control passes to the ON EXCEPTION exception handler designed to catch the error condition raised.

An example of an exception block and the converted equivalent in Oracle is as follows:

Informix Dynamic Server SPL

```
ON EXCEPTION IN (-208)
  DELETE FROM students;
END EXCEPTION;

IF(clear_table=1)THEN
  RAISE EXCEPTION -208, 0, 'No datafile';
END IF;
```

Oracle PL/SQL

```
BEGIN
  IF(clear_table=1)THEN
    RAISE_APPLICATION_ERROR(-29999,
                            /* Informix error number : -208, 0 */
                           "No datafile"
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    DELETE FROM students;
END
```

The converted Informix Dynamic Server exception block is still resident within the PL/SQL code but the converted `RAISE EXCEPTION` statement now calls the built-in `RAISE_APPLICATION_ERROR` function instead of calling the embedded exception block originally defined for it. The Oracle `RAISE_APPLICATION_ERROR` statement also terminates the execution of the stored procedure and returns to the calling routine. However, the execution of the Informix Dynamic Server stored procedure continues.

Using the `RAISE_APPLICATION_ERROR` function in this way changes the execution flow and error handling functionality of the converted PL/SQL stored procedure. Therefore, manual conversion of the procedure is usually required.

For more information, see the following sections:

- Exception Blocks
- Error Handling within Stored Procedures

Error Handling within Stored Procedures

Oracle PL/SQL checks each SQL statement for errors before proceeding to the next statement. If an error occurs, control immediately jumps to an exception handler. This prevents you from having to check the status of every SQL statement. For example, if a `SELECT` statement does not find any rows in the database, an exception is raised and the code to deal with this error is executed.

Informix Dynamic Server has similar error handling capabilities to Oracle. Blocks of exception handler code resident within the SPL stored procedure catch any errors raised by the database server during execution of the stored procedure code.

Informix Dynamic Server error handlers, unlike Oracle error handlers, can continue execution of the stored procedure after the error occurs. This fundamental difference has immediate implications for the conversion process.

While Informix Dynamic Server SPL exception blocks can be translated into syntactically correct PL/SQL, the execution flow of the PL/SQL stored procedure differs to a considerable extent should an error occur. The Oracle server terminates execution of the stored procedure, while the Informix Dynamic Server server resumes execution of the stored procedure.

In order to successfully convert Informix Dynamic Server SPL exception blocks to functionally equivalent PL/SQL, you must manually convert the generated PL/SQL code.

If you have to maintain control within the executable commands section of the PL/SQL stored procedure, you should use `IF` statements to check for possible errors before they occur.

After conversion, it is recommended that you re-write large or complex stored procedures in a more modular way so that each stored procedure performs one task and contains all the DML statements required to perform that task. Placing task related DML statements into logical units enables greater control over both the transaction model and the error model. This leads to the production of a more re-usable, maintainable, and stable PL/SQL code base.

For more information on the strategy employed by the Migration Workbench in the conversion of Informix Dynamic Server exception blocks to PL/SQL, see the Exception Blocks topic.

DDL Statements in SPL Code

Informix Dynamic Server enables certain DDL statements to reside within stored procedure code. Oracle does not support the direct inclusion of DDL statements

within PL/SQL code. Oracle offers two ways to dynamically execute DDL statements: an internal DBMS package named DBMS_SQL (available since Oracle 7.1) and Native Dynamic SQL (available since Oracle 8i).

As the DBMS_SQL package does not support new Oracle8 data types, the Oracle Migration Workbench uses Native Dynamic SQL to execute any DDL statement present within the original Informix Dynamic Server SPL code. This is accomplished by offering a DDL_Manager stored procedure. The Migration Workbench automatically creates this stored procedure in the destination Oracle database under the OMWB emulation users schema.

When the converter encounters a DDL statement within the Informix Dynamic Server stored procedure, the resulting PL/SQL code uses the DDL_Manager procedure to dynamically execute the DDL statement. For example, the following Informix Dynamic Server DDL statement is converted into a call to the DDL_Manager PL/SQL stored procedure:

Informix Dynamic Server SPL

```
alter table pushprefs modify (preferences_value char(100));
```

Oracle PL/SQL

```
/* SPCONV-MSG:(ALTER TABLE) OMWB_Emulation.DDL_MANAGER procedure used
to execute DDL statement. */
 OMWB_Emulation.DDL_Manager('ALTER TABLE informix.pushprefs MODIFY (
preferences_value  CHAR(100) )');
```

The DDL_Manager procedure is created with `invokers_rights` permissions. This means that any person who executes the procedure executes any DDL statement within their own schema and not the schema that the `DDL_Manager` procedure resides within, in this case, the `OMWB_Emulation` user's schema. For more information on the invokers rights model, see the PL/SQL User's Guide and Reference Release 1 (9.0.1).

A code listing of the DDL_Manager procedure is as follows:

Oracle PL/SQL

```
CREATE OR REPLACE PROCEDURE DDL_Manager(
ddl_statement varchar)
AUTHID CURRENT_USER IS
BEGIN
  EXECUTE IMMEDIATE ddl_statement;
EXCEPTION
  WHEN OTHERS THEN
```

```
  RAISE;
END DDL_Manager;
```

It is recommended that you check all DDL statement strings passed to the DDL_Manager procedure for errors before the creation of the encapsulating procedure in the destination Oracle database.

Informix Dynamic Server DDL statements that are not dispatched to the DDL_Manager procedure for execution are explained in the following sections:

- Creating Temporary Tables
- DROP TABLE Statements

Creating Temporary Tables

The Migration Workbench converts temporary tables to Oracle global temporary tables. Unlike Informix Dynamic Server temporary tables, Oracle temporary table structures are persistent across sessions, therefore the converted CREATE TEMP TABLE statement is only ever executed once within the Oracle database.

When the converter encounters an Informix Dynamic Server CREATE TEMPORARY TABLE *<table name>* statement, it generates the DDL to create an equivalent Oracle global temporary table. It then inserts a PL/SQL DELETE FROM *<table name>* statement into the converted stored procedure. This ensures that the table is void of data before it is used within the PL/SQL code. The CREATE GLOBAL TEMPORARY TABLE DDL statement generated by the converter is executed before the stored procedure is created in the destination Oracle database. This ensures that referential integrity constraints are met during the creation of the stored procedure within the destination Oracle database.

An example of an Informix Dynamic Server CREATE TABLE statement and the generated Oracle DDL statement that is executed before the stored procedure is created within the destination Oracle database is as follows:

Informix Dynamic Server SPL

```
CREATE TEMP TABLE temp_table AS
  SELECT emp_num, emp_name
  FROM emp;
```

Oracle PL/SQL

```
CREATE GLOBAL TEMP TABLE temp_table AS
   SELECT emp_num,
          emp_name
```

```
   FROM emp
ON COMMIT PRESERVE ROWS;
```

Additionally, the following DELETE FROM statement appears within the converted PL/SQL code.

Oracle PL/SQL

```
DELETE FROM temp_table;
```

The previous statement that appears within the converted PL/SQL code clears the temp table of all data. This leaves the Oracle table in a state consistent with the original Informix Dynamic Server table at this point within the procedures execution.

DROP TABLE Statements

When the Migration Workbench converts Informix Dynamic Server temporary tables to Oracle temporary tables, any DROP TABLE statement within an Informix Dynamic Server stored procedure becomes redundant within the converted PL/SQL code. Oracle temporary tables are created once. The definition is persistent across sessions although the data held within the tables is not persistent.

The following actions occurs when a DROP TABLE statement is encountered by the stored procedure converter.

- A warning message outputs to the log window. If you selected the Display parser warnings option from the Parser Options tab within the Migration Workbench, a warning message is placed into the converted PL/SQL code.
- The original Informix Dynamic Server DROP TABLE statement is displayed within the converted PL/SQL code as a single line comment.
- An executable NULL statement is also added to the PL/SQL code.

The following shows the DROP TABLE statement and the converted equivalent in Oracle:

Informix Dynamic Server SPL

```
DROP TABLE temp_table;
```

Oracle PL/SQL

```
/* SPCONV-WRN:(DROP TABLE) Statements never passed to the DDL_Manager procedure.
*/
--DROP TABLE temp_table;
NULL
```

Using Keywords as Identifiers

Informix Dynamic Server SPL allows keywords to be used as identifiers. This can cause ambiguous SQL statements and unreadable SPL code. An example of a keyword used as an identifier is as follows:

Informix Dynamic Server SPL

```
SELECT ordid INTO order FROM table1;
```

The keyword `order` is used in this context as a variable name.

Oracle does not enable keywords to be used as identifiers. All keywords within Oracle are reserved. This eradicates ambiguous PL/SQL code. The preceding Informix Dynamic Server `SELECT` statement is not syntactically valid within PL/SQL and produces a compilation error within the destination Oracle database.

In order to convert Informix Dynamic Server SPL into syntactically correct PL-SQL, the stored procedure parser needs to recognize keywords used in the context of an identifier in an Informix Dynamic Server SPL statement. The Migration Workbench parser handles this by adding a trailing underscore character to the identifier name. The following table illustrates how the Migration Workbench appends an underscore to the Informix Dynamic Server SPL reserved `word order`:

Informix Dynamic Server SPL	`SELECT ordid INTO order FROM table1;`
Oracle PL/SQL	`SELECT ordid INTO order_ FROM table1;`

The Migration Workbench stored procedure converter does not support any of the following list of Informix Dynamic Server keywords as identifiers:

- `INTO`
- `WHERE`
- `HAVING`
- `FROM`
- `END: * NEW *`
- `LET`
- `IF`

- ELSE
- TRUNC
- WITH
- RESUME
- RETURN
- INSERT
- TRIM
- UPPER
- LENGTH
- GLOBAL
- LIKE
- NULL
- OUTER
- DBINFO
- WEEKDAY
- SELECT
- FOREACH
- CALL
- UPDATE
- DELETE
- CASE

If the converter encounters an unsupported keyword when an identifier is expected, one of the following actions occurs:

- Parsing process fails. This causes an error message to be generated within the Log window. An example error message is shown as follows:

  ```
  SPCONV-ERR[23]:(UPDATE) Encounterd the word UPDATE when expecting one of the following.
  ```

- Produces syntactically incorrect PL / SQL code. This causes the PL / SQL stored procedure to fail compilation within the destination Oracle database.

Oracle recommends that keyword / identifier issues are removed from the original Informix Dynamic Server stored procedure code before you initiate the conversion process. You can manually edit the stored procedure text within the Informix Dynamic Server Source Model of the Migration Workbench.

Issues with Converting SPL Statements

The Migration Workbench parser may not convert some SPL statements to PL / SQL code. Generally, this happens when the statement functionality cannot be replicated in PL / SQL, if the statement is unnecessary within the PL / SQL code, or if the statement requires manual conversion by the DBA. The following list of statements are currently not supported:

- `DBINFO('sqlca.sqlerrd1')`
- `DBINFO(DBSPACE, number)`
- All `SET` statements with the exception of `SET DEBUG FILE`

When the parser encounters any unsupported statement, it takes the following actions:

1. A parser warning (SPCONV-WRN) is produced within the Log window.
2. If you have selected the Display parser Warnings parser option for the current procedure, the converter places a warning message within the PL / SQL stored procedure text in the form of a comment.
3. The original Informix Dynamic Server statement is added to the PL / SQL text as a comment.
4. An executable `NULL;` statement is added to the PL / SQL text.

An example of and unsupported `SET` statement and the converted equivalent is as follows:

Informix Dynamic Server SPL

```
SET ISOLATION TO DIRTY READ
```

Oracle PL/SQL

```
/* SPCONV-ERR:(SET) Statement ignored. Manual conversion may be required. */
--SET ISOLATION TO DIRTY READ
NULL;
```

4

Distributed Environments

This chapter includes the following sections:

Distributed Environments

A distributed environment is chosen for various applications where:

- The data is generated at various geographical locations and needs to be available locally most of the time.
- The data and software processing is distributed to reduce the impact of any particular site or hardware failure.

Accessing Remote Databases in a Distributed Environment

When a relational database management system (RDBMS) allows data to be distributed while providing the user with a single logical view of data, it supports location transparency. Location transparency eliminates the need to know the actual physical location of the data. Location transparency thus helps make the development of the application easier. Depending on the needs of the application, the database administrator (DBA) can hide the location of the relevant data.

To access a remote object, the local server must establish a connection with the remote server. Each server requires unique names for the remote objects. The methods used to establish the connection with the remote server, and the naming conventions for the remote objects, differ from database to database.

Oracle and Remote Objects

Oracle allows remote objects (such as tables, views, and procedures) throughout a distributed database to be referenced in SQL statements using global object names. In Oracle, the global name of a schema object comprises the name of the schema that contains the object, the object name, followed by an at sign (@), and a database name. For example, the following query selects information from the table named scott.emp in the SALES database that resides on a remote server:

```
SELECT * FROM
scott.emp@sales.division3.acme.com
```

A distributed database system can be configured so that each database within the system has a unique database name, thereby providing effective global object names.

Furthermore, by defining synonyms for remote object names, you can eliminate references to the name of the remote database. The synonym is an object in the local database that refers to a remote database object. Synonyms shift the responsibility of distributing data from the application developer to the DBA. Synonyms allow the DBA to move the objects as desired without impacting the application.

The synonym can be defined as follows:

```
CREATE PUBLIC SYNONYM emp FOR
scott.emp@sales.division3.acme.com;
```

Using this synonym, the SQL statement outlined above can be changed to the following:

```
SELECT * FROM emp;
```

Informix Dynamic Server and Remote Objects

Informix Dynamic Server requires schema objects throughout a distributed database to be referenced in SQL statements by fully qualifying the object names. The complete name of a schema object has the following format:

```
database_name@server_name:object_owner_name.object_name
```

The `server_name` is the name of a remote server. The `database_name` is the name of a remote database on the remote server.

Informix Dynamic Server does not allow you to create a database link, but does allow you to create a synonym. So a remote object can be referred to by a synonym. A remote object is specified by the database name followed by the at sign (@), the remote server name, and then the name of the schema and the object. For example, the following synonym (`myemp`) is created for the object `scott.emp` that is in the `sales` database on the `boston` server:

```
CREATE SYNOYM myemp FOR sales@boston:scott:emp
```

You can use this synonym to reference a remote object:

```
SELECT * FROM myemp
```

You can make a common name refer to remote objects that can work for both Oracle databases and Informix Dynamic Server databases.

Application Development Tools

Several application development tools that are currently available use specific features of one of the various database servers; you may have to invest significant effort to port these products to other database servers. With critical applications, it is sometimes best to develop and maintain a different set of application

development tools that work best with the underlying database, as ODBC support is not adequate in such cases.

The majority of applications are written in Informix Dynamic Server 4GL.

If a Visual Basic application is written with ODBC as the connection protocol to access Informix Dynamic Server, it is possible to modify and fix the Visual Basic application to work with an Oracle back-end.

The Informix 4GL application development environment does not provide connectivity to other databases, including Oracle. To overcome this limitation in Informix 4GL, you must convert or migrate your code. Oracle recommends one of our partners in converting or migrating your Informix 4GL code:

- ArtInSoft at the Web site:

  http://www.artinsoft.com

- QueriX at the Web site:

  http://www.querix.com

- 4Js at the Web site:

  http://www.4js.com

- Freesoft at the Web site:

  http://www.freesoft.hu/index2.html

5

The ESQL/C to Oracle Pro*C Converter

This chapter describes the E/SQL to Pro*C Converter, its scope and some of its limitations in this initial release. The chapter includes the following sections:

Introduction to E/SQL and Pro*C

Oracle and Informix Dynamic Server have similar methods of embedding their SQL statements into a third generation language, in this case C (or C++). You run a precompiler that converts the C containing embedded SQL into pure C. High-level embedded SQL directives are replaced by vendor-specific C code. You can use a standard C compiler to compile and link with the vendor libraries to produce an executable.

Oracle Pro*C is easier to write and maintain than a pure C. One reason for this is that the C and embedded SQL are separated in the source code.

The SQL used in Oracle Pro*C files complies with ANSI standards, as are some of the embedded SQL commands and techniques. There are some differences, however, that are resolved manually or by using a tool such as the ESQL/C to Oracle Pro*C Converter, or a combination of both methods.

Using the ESQL/C to Oracle Pro*C Converter

This section provides an example conversion using the ESQL/C to Oracle Pro*C Converter, and describes some common conversions handled automatically by the tool.

Example Capture of ESQL/C Source Files

The following example shows the use of the tool. You capture this code by choosing **Action->Capture ESQL/C Source Files** in the Migration Workbench.

For details beyond the core code, refer to "Conversion Errors and Warnings" on page 5-14 and "Using Demonstration Code" on page 5-24.

Example Code:

```
 /*
 *  esqlproc1.ec
 *
 *  This program connects to the database, declares and opens a cursor,
 *  fetches the names, salaries, and commissions of all
 *  salespeople, displays the results, then closes the cursor.
 */

#include <stdio.h>
#include <string.h>
#include <sqlca.h>
#include <stdlib.h>

#define UNAME_LEN      20
#define PWD_LEN        11

EXEC SQL BEGIN DECLARE SECTION;
    char username[20]="informix";
    char password[20]="inform9";
    char       emp_name[11];
    float      salary;
    float      commission;
EXEC SQL END DECLARE SECTION;

void sql_error(msg)
    char *msg;
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
```

```
    printf("\n%s\n", msg);

    EXEC SQL ROLLBACK;
    exit(EXIT_FAILURE);
}

void main()
{

/* Connect to the database. */

    EXEC SQL WHENEVER SQLERROR GO TO connecterror;

    EXEC SQL connect to 'turloch@mtg1_tcp' user :username using :password;
    printf("\nConnected to the database as user: %s\n", username);

    EXEC SQL WHENEVER SQLERROR GO TO declareerror;
/* Declare the cursor. All static SQL explicit cursors
 * contain SELECT commands. 'salespeople' is a SQL identifier,
 * not a (C) host variable.
 */
    EXEC SQL DECLARE salespeople CURSOR FOR
        SELECT ENAME, SAL, COMM INTO
          :emp_name,
          :salary,
          :commission
            FROM EMP
            WHERE JOB LIKE 'SALES%';

    EXEC SQL WHENEVER SQLERROR GO TO openerror;
/* Open the cursor. */
    EXEC SQL OPEN salespeople;

/* Get ready to print results. */
    printf("\n\nThe company's salespeople are--\n\n");
    printf("Salesperson   Salary   Commission\n");
    printf("-----------   ------   ----------\n");

/* Loop, fetching all salesperson's statistics.
 * Cause the program to break the loop when no more
 * data can be retrieved on the cursor.
 */
    EXEC SQL WHENEVER SQLERROR GO TO fetcherror;
    EXEC SQL WHENEVER NOT FOUND go to breakout;
```

```
    for (;;)
    {
        EXEC SQL FETCH salespeople ;
        printf("%s %9.2f %12.2f\n", emp_name,
                salary, commission);
    }

breakout:
/* Close the cursor. */
    EXEC SQL WHENEVER SQLERROR GO TO closeerror;
    EXEC SQL CLOSE salespeople;

    EXEC SQL WHENEVER SQLERROR GO TO freeerror;
    EXEC SQL FREE salespeople;

    printf("\nArrivederci.\n\n");

    EXEC SQL WHENEVER SQLERROR GO TO disconnecterror;
    EXEC SQL disconnect current;
    exit(EXIT_SUCCESS);
 freeerror:
    sql_error("free error--");
 connecterror:
    sql_error("connect error--");
 declareerror:
    sql_error("declare error--");
 openerror:
    sql_error("open error--");
 fetcherror:
    sql_error("fetch error--");
 closeerror:
    sql_error("close error--");
 disconnecterror:
    sql_error("disconnect error--");
}
```

Oracle Pro*C Conversion

To pass the code through the converter choose **Action->Convert E/SQL to Pro*C**. To generate the files created by this action, and save them on to the file system, choose **Action->Generate PRO*C Source Files**.

The following example Oracle Pro*C file is generated:

```
/*
 *  esqlproc1.ec
 *
 *  This program connects to the database, declares and opens a cursor,
 *  fetches the names, salaries, and commissions of all
 *  salespeople, displays the results, then closes the cursor.
 */

#include <stdio.h>
#include <string.h>
#include <sqlca.h>
#include <stdlib.h>

#define UNAME_LEN      20
#define PWD_LEN        11

EXEC SQL BEGIN DECLARE SECTION;
    char username[20]="informix";
    char password[20]="inform9";
    char       emp_name[11];
    float      salary;
    float      commission;
EXEC SQL END DECLARE SECTION;

void sql_error(msg)
    char *msg;
{
    /* ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("\n%s\n", msg);
    /* ESQL-CONV-MSG:(ROLLBACK) Statement automatically closes all cursors
referenced in a CURRENT OF clause when compilation mode = ORACLE. */
    /* ESQL-CONV-MSG:(ROLLBACK) Extra ROLLBACK option available in Oracle.
Please see the Oracle 8i users guide for more information. */
    EXEC SQL ROLLBACK WORK;
    exit(EXIT_FAILURE);
}

void main()
{

/* Connect to the database. */

    /* ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */
```

```
    EXEC SQL WHENEVER SQLERROR GOTO connecterror;

    /* ESQL-CONV-ERR:(CONNECT) Manual conversion of the username, password and
database required. */
    {
      char oracleid = '/';
      EXEC SQL CONNECT :oracleid;
    }
    printf("\nConnected to the database as user: %s\n", username);

    /* ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */
    EXEC SQL WHENEVER SQLERROR GOTO declareerror;
/* Declare the cursor. All static SQL explicit cursors
 * contain SELECT commands. 'salespeople' is a SQL identifier,
 * not a (C) host variable.
 */
    /* ESQL-CONV-MSG:(INTO) Clause removed from cursor definition and integrated
into FETCH statement. */
    EXEC SQL
        DECLARE salespeople CURSOR FOR
         SELECT ENAME,
                SAL,
                COMM
         FROM EMP
         WHERE JOB LIKE 'SALES%';
    /* ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */
    EXEC SQL WHENEVER SQLERROR GOTO openerror;
/* Open the cursor. */
    EXEC SQL
         OPEN salespeople;

/* Get ready to print results. */
    printf("\n\nThe company's salespeople are--\n\n");
    printf("Salesperson   Salary   Commission\n");
    printf("-----------   ------   ----------\n");

/* Loop, fetching all salesperson's statistics.
 * Cause the program to break the loop when no more
 * data can be retrieved on the cursor.
 */
    /* ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */
    EXEC SQL WHENEVER SQLERROR GOTO fetcherror;
    /* ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */
    EXEC SQL WHENEVER NOT FOUND GOTO breakout;
```

```
    for (;;)
    {
        /* ESQL-CONV-MSG:(INTO) Clause originally declared within cursor
declaration. */
        EXEC SQL
             FETCH salespeople
             INTO :emp_name,
                  :salary,
                  :commission;
        printf("%s %9.2f %12.2f\n", emp_name,
                salary, commission);
    }

breakout:
/* Close the cursor. */
    /* ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */
    EXEC SQL WHENEVER SQLERROR GOTO closeerror;
    /* ESQL-CONV-MSG:(CLOSE) Statement not required when compilation mode =
ORACLE. */
    /* EXEC SQL CLOSE salespeople */

    /* ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */
    EXEC SQL WHENEVER SQLERROR GOTO freeerror;
    /* ESQL-CONV-MSG:(FREE) Statement not required in ORACLE. */
    /* EXEC SQL FREE salespeople; */

    printf("\nArrivederci.\n\n");
    /* ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */
    EXEC SQL WHENEVER SQLERROR GOTO disconnecterror;
    EXEC SQL COMMIT WORK RELEASE;
    exit(EXIT_SUCCESS);
 freeerror:
    sql_error("free error--");
 connecterror:
    sql_error("connect error--");
 declareerror:
    sql_error("declare error--");
 openerror:
    sql_error("open error--");
 fetcherror:
    sql_error("fetch error--");
 closeerror:
    sql_error("close error--");
 disconnecterror:
    sql_error("disconnect error--");
```

```
}
```

Manual Changes to the Oracle Pro*C File

In the "Oracle Pro*C Conversion" example on page 5 -4 you must add the CONNECT details, as shown in the following table:

Code Generated by ESQL/C to Oracle Pro*C Converter	`{`
	`  char oracleid = '/';`
	`  EXEC SQL CONNECT :oracleid;`
	`}`

Oracle Pro*C Code	`{`
	`EXEC SQL BEGIN DECLARE SECTION;`
	`  char *oracleid = "examp/examp";`
	`EXEC SQL END DECLARE SECTION;`
	`  EXEC SQL CONNECT :oracleid;`
	`}`

The executable relies on a database populated by data. Refer to "Using Demonstration Code" on page 5 -24 for an example of how to populate the database. The following example shows how to produce an executable:

1. Precompile the code from Oracle Pro*C to a C file using the Oracle Pro*C executable `proc esqlproc1.pc`.
2. Compile the C code using a suitable development environment. The Oracle Pro*C example shipped by with Oracle contains project files for the Visual C++ development environment. The project contains details of the libraries and include files required to build a small executable based on Oracle Pro*C. The project file used is `%ORACLE_HOME%\precomp\demo\proc\sample.dsp`. You have to add the C file `esqlproc1.c` to the project.

For more information on Oracle Pro*C / C++ refer to the *Pro*C/C++ Precompiler Programmer's Guide.*

The following shows sample output from the executable.

The following is sample output from the executable:

```
The company's salespeople are--
Salesperson   Salary   Commission
-----------   ------   ----------
ALLEN        1600.00       300.00
WARD         1250.00       500.00
```

```
Arrivederci.
```

Syntactical Conversion Issues

This section provides information about the Informix Dynamic Server ESQL/C constructs and the equivalent Oracle constructs generated by the Migration Workbench. Examples of how to resolve syntactical conversion issues are provided where relevant. The following constructs are described in detail:

- EXEC SQL Statement
- INCLUDE Files
- UPDATE Statement
- ANSI Compliance
- Double equal sign in WHERE Clause
- OUTER JOIN Syntax
- FETCH Clause
- Header Files SQLNOTFOUND
- CURSOR Declaration
- DECLARE CURSOR Statement
- FOR UPDATE Option

EXEC SQL Statement

In all programs you replace the dollar ($) sign preceding the SQL sign with EXEC SQL and replace all the dollar signs before host variables with a colon (:). The following table compares the dollar sign in Informix Dynamic Server and the EXEC SQL Statement in Oracle:

Database Language	Example

Informix Dynamic Server ESQL/C	`$`
	`SELECT login_no`
	`INTO $login_no;`

Database Language	Example

Oracle Pro*C	`EXEC SQL`
	`SELECT login_no`
	`INTO :login_no;`

INCLUDE Files

The INCLUDE files for Informix Dynamic Server ESQL/C and Oracle are different., You must replace Informix Dynamic Server ESQL/C INCLUDE files with Oracle INCLUDE files.

UPDATE Statement

The UPDATE statement works similarly in Informix Dynamic Server and Oracle, but they have different syntax. The Oracle syntax is clearer, and the resulting code is easier to maintain. The following table compares the UPDATE statement in Informix Dynamic Server and Oracle:

Database Language	Example

Informix Dynamic Server ESQL/C	`EXEC SQL UPDATE employees`
	`SET (emp_no, emp_name) =`
	`(:emp_no, :emp_name)`
	`WHERE emp_no == :old_emp_no;`

Oracle Pro*C	`EXEC SQL UPDATE employees`
	`SET emp_no = :emp_no,`
	`emp_name = :emp_name`
	`WHERE emp_no = :old_emp_no;`

ANSI Compliance

The Oracle precompiler can generate the C code in either ANSI or non-ANSI compliant code. The default is non-ANSI.

Double equal sign in WHERE Clause

Check the WHERE clause in the SELECT, UPDATE, and DELETE statements in Informix Dynamic Server ESQL/C for double equal signs (==). An equal operator can be a single or a double equal sign. Oracle supports the ANSI standard single equal sign. The following table compares the WHERE clause equal operator in the SELECT, UPDATE, and DELETE statements in Informix Dynamic Server and Oracle:

Database Language	Example

Informix Dynamic Server ESQL/C	`EXEC SQL SELECT login_no`
	`INTO :login_no`
	`FROM users`
	`WHERE user_name == 'PAM';`

Oracle Pro*C	`EXEC SQL SELECT login_no`
	`INTO :login_no`
	`FROM users`
	`WHERE user_name='PAM';`

OUTER JOIN Syntax

The Informix Dynamic Server ESQL/C OUTER JOIN syntax is different from Oracle Pro*C. The following table compares the OUTER JOIN syntax in Informix Dynamic Server and Oracle:

Database Language	Example

Informix Dynamic Server ESQL/C	`EXEC SQL SELECT login_no`
	`INTO :login_no`
	`FROM users a , OUTER logins b`
	`WHERE a.user_name = b.user_name;`

Oracle Pro*C	`EXEC SQL`
	`SELECT login_no`
	`INTO :login_no`
	`FROM users a, logins b`
	`WHERE a.user_name =`
	`b.user_name (+);`

Note: In release 9.2.0.1.0 of the Migration Workbench, you must manually add the plus sign.

FETCH Clause

The Informix Dynamic Server ESQL/C FETCH clause allows the NEXT keyword in the statement. The following table compares the FETCH clause in Informix Dynamic Server and Oracle:

Database Language	Example
Informix Dynamic Server ESQL/C	`EXEC SQL FETCH NEXT cur` `INTO :emp_no;`

Oracle Pro*C	`EXEC SQL`
	`FETCH cur1`
	`INTO :emp_no;`

Header Files SQLNOTFOUND

Informix Dynamic Server ESQL/C header files define SQLNOTFOUND. In Oracle, you must explicitly define SQLNOTFOUND as either +100 (ANSI mode) or +1403 (Oracle mode) depending on the mode being used in the Oracle precompiler.

CURSOR Declaration

In Informix Dynamic Server ESQL/C, the CURSOR can be declared with WITH HOLD options, so that a CURSOR is not closed by COMMIT or ROLLBACK. This does not comply with the ANSI standard, but Oracle supports it provided you select the MODE=ORACLE precompiler option. For this reason, you modify the program to COMMIT after closing the CURSOR.

DECLARE CURSOR Statement

Informix Dynamic Server ESQL/C DECLARE CURSOR statements can have INTO clauses. You can specify the host variables in which to fetch the data in the DECLARE CURSOR statement, and then use the cursor name in the FETCH statement. This does not comply with the ANSI standard, and is not supported by Oracle. As a result, you change all DECLARE statements with INTO clauses to have the INTO clause in the FETCH statement. The following table compares the DECLARE CURSOR statement in Informix Dynamic Server and Oracle:

Database Language	Example

Informix Dynamic Server ESQL/C	`FETCH statement.`
	`EXEC SQL DECLARE CURSOR cur1 FOR`
	`SELECT login_no`
	`INTO :login_no`
	`FROM users`
	`WHERE user_name = 'PAM';`
	
	`EXEC SQL FETCH cur1;`

Database Language	Example
Oracle Pro*C	EXEC SQL DECLARE CURSOR cur1 FOR SELECT login_no FROM users WHERE user_name='PAM'; EXEC SQL FETCH cur1 INTO :login_no;

FOR UPDATE Option

Informix Dynamic Server ESQL/C cannot lock individual rows. To prevent a row from being modified, the existing code must declare a cursor with the FOR UPDATE option, open the cursor, fetch it, and then close it. Oracle can lock a selected row by using the FOR UPDATE option, without requiring an explicit cursor declaration. You must change the logic of some programs to take advantage of the Oracle method. The following table compares the FOR UPDATE option in Informix Dynamic Server and Oracle:

Database Language	Example
Informix Dynamic Server ESQL/C	EXEC SQL DECLARE CURSOR cur1 FOR SELECT login_no INTO :login_no FROM users WHERE user_name = 'PAM'; FOR UPDATE; EXEC SQL OPEN cur1; EXEC SQL FETCH cur1; EXEC SQL CLOSE cur1;
Oracle Pro*C	EXEC SQL SELECT login_no INTO :login_no FROM users WHERE user_name = 'PAM' FOR UPDATE;

Application Conversion Issues

Oracle and Informix Dynamic Server use temporary tables. The difference being that Oracle creates temporary tables once, and the data is kept separate between

sessions. You must manually create temporary tables in Oracle, and separate from the application. The Migration Workbench marks instances of this detected by the converter as errors.

The following table compares the TEMP TABLE statement option in Informix Dynamic Server and Oracle:

Database Language	Example
Informix Dynamic Server ESQL/C	`EXEC SQL CREATE TEMP TABLE tab2 (fname CHAR(15), lname CHAR(15)) WITH NO LOG;`
Oracle Pro*C	`/* SPCONV-WRN:(TEMP TABLE): It will be more performant to create the temporary table separately. */` `EXEC SQL CREATE GLOBAL TEMPORARY TABLE tab2(fname CHAR(15), lname CHAR(15)) ON COMMIT PRESERVE ROWS;`

The Oracle Pro*C Preprocessor

For more information on Oracle Pro*C including command line options for the preprocessor refer to the *Pro*C/C++ Precompiler Programmer's Guide.*

Conversion Errors and Warnings

The ESQL/C to Oracle Pro*C Converter shares most of the errors and warnings it generates with the stored procedure parser. For more information on stored procedures refer to the "Triggers, Packages, and Stored Procedures" chapter. Additional errors and warnings are explained in "ESQL/C to Oracle Pro*C Converter Errors" on page 5 -14 and ESQL/C to Oracle Pro*C Warnings on page 5 -15.

ESQL/C to Oracle Pro*C Converter Errors

The cause of the Informix Dynamic Server ESQL/C to Oracle Pro*C converter errors require manual investigation and correction by the user. Table 5–1 lists details of possible error messages.

Note: In Table 5–1 SPCONV refers to errors shared with the stored procedure parser. ESQL refers to errors specific to the embedded SQL parser.

*Table 5–1 ESQL to Oracle Pro*C Converter Errors*

Error Message	Description
`ESQL-CONV-ERR:(EXEC SQL ..) The converter will not parse this expression correctly.`	The converter failed to understand this statement. It places it in a comment. You must manually convert it.
`SPCONV-ERR:(**) Statement ignored. Manual conversion required.`	The (**) statement (for example the dynamic PUT statement) is not automatically converted. You must manually convert it. Most Set statements also require manual conversion.
`ESQL-CONV-ERR:(CONNECT) Manual conversion of the username, password and database required.`	The CONNECT string changes when you move it from Informix Dynamic Server to Oracle.
`ESQL-CONV-ERR:(DATABASE) Manual conversion of the username, password and database required.`	The CONNECT string changes when you move it from Informix Dynamic Server to Oracle.
`SPCONV-ERR:(ALTER INDEX) Statement ignored. Manual conversion may be required.`	The ALTER statement clause is unlikely to occur in ESQL/C and Oracle Pro*C. If it occurs, you must manually add it.

`/* SPCONV-MSG:(EXEC SQL ..) The converter will not parse this expression correctly. */`
`/**********************`
`ERROR STATEMENT COMMENTED`
`**********************`
`exec sql <Statement not parsed goes here>`
`*****************************/`
	The converter has failed on this statement and has continued with the next statement.

`/* ESQL-CONV-ERR:(DYNAMIC SQL) Conversion not supported in this release. Manual conversion may be required. */`	Release 9.2.0.1.0 of the Migration Workbench does not support Dynamic SQL. However, some commands do work in ANSI mode.

ESQL/C to Oracle Pro*C Warnings

Warning messages are for information purposes and may not require intervention. Table 5–2 lists possible warning messages.

Note: In Table 5–2 SPCONV refers to errors shared with the stored procedure parser. ESQL refers to warnings specific to the embedded SQL parser.

Table 5–2 ESQL/C to Oracle Pro*C Warnings

Warning	Description
`ESQL-CONV-MSG:(CLOSE) Statement not required when compilation mode = ORACLE.`	Oracle mode is the default Oracle Pro*C setting so the close statement is not required. It places it in a comment.
`ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives.`	An informational message to note that the Oracle WHENEVER statement has additional options that may be of use.
`ESQL-CONV-MSG:(INTO) Clause removed from cursor definition and integrated into FETCH statement.`	A reminder that the INTO clause has moved, as shown in the syntax of the "DECLARE CURSOR Statement" example.
`ESQL-CONV-MSG:(DYNAMIC SQL) Unsupported in this release. Manual conversion may be required.`	Most commands for dynamic SQL statements are similar in Oracle and Informix Dynamic Server. The generated SQL statements should be similar, but the converter makes no attempt to convert the dynamic SQL statements, however simple commands are converted.
`/* SPCONV-MSG:(**) Statement passed to DDL file. */`	Some commands may be executed before running the new Oracle Pro*C application. For example, you create Oracle temporary tables before running the application. The data is saved separately each time a session is run.
`/* SPCONV-WRN:(=>) Oracle requires Positional parameter notation to precede Named parameter notation. Manual conversion required.*/`	This warning indicates complications and variations in the CALL statement syntax.
`/* ESQL-CONV-MSG:(CLOSE) Statement not required when compilation mode = ORACLE. */`	You should close a cursor before reopening it. However, if you specify the Oracle mode (default), you do not need to close the cursor. Choosing the oracle mode can increase performance.

*Table 5–2 ESQL/C to Oracle Pro*C Warnings (Cont.)*

Warning	Description
`/* ESQL-CONV-MSG:(COMMIT) Statement will automatically close all cursors referenced in a CURRENT OF clause when compilation mode = ORACLE. Other cursors are unaffected. */`	Statement automatically closes all cursors included in a CURRENT OF clause when compiled in Oracle mode.
`"/* ESQL-CONV-MSG:(COMMIT) Statement will automatically close all explicit cursors when compilation mode = ORACLE. */`	Informational message about the behaviors of cursors on COMMIT. Refer to the Oracle Pro*C documentation for more details.
`/* SPCONV-MSG:(CONTINUE **) Statement emulated using GOTO statement and LABEL definition.*/`	An informational message about how Oracle emulates Informix Dynamic Server behavior.
`/* SPCONV-MSG:(WITH RESUME) Collating results for REF CURSOR return.*/`	An informational message referring to the use of REF CURSOR. For further information refer to the "Triggers, Packages, and Stored Procedures"chapter.
`/* SPCONV-MSG:(WITH RESUME) Initialising GLOBAL TEMPORARY TABLE used to store Procedures interim results. */`	An informational message about the use of temporary tables in the emulation of the Informix Dynamic Server WITH RESUME option.
`/* ESQL-CONV-MSG:(SCROLL) Scroll cursors not available in Oracle. Manual conversion may be required. */`	Oracle does not have SCROLL cursors. You can manually move the data from a cursor into a PL/SQL table.
`/* ESQL-CONV-MSG:(WITH HOLD) Unsupported in Oracle. Manual conversion may be required. */`	The WITH HOLD option is not available in Oracle. You must manually convert it.
`/* ESQL-CONV-MSG:(INTO) Clause removed from cursor definition and integrated into FETCH statement. */`	An informational message. Move the INTO clause to the FETCH statement in Oracle.
`/* ESQL-CONV-MSG:(SELECT FIRST n) Emulated using FOR clause in FETCH statement. Manual conversion may be required. */`	An informational message. Additional declare cursor facilities not available in Oracle.

Table 5–2 ESQL/C to Oracle Pro*C Warnings (Cont.)

Warning	Description
`/* ESQL-CONV-MSG:(MODE=ANSI) ANSI compliant variable declaration generated. */`	`EXEC SQL BEGIN DECLARE SECTION;` and `EXEC SQL END DECLARE SECTION;` statements added.
`/*ESQL-CONV-MSG:(MODE=ORACLE) Non ANSI compliant variable declaration generated. */`	Declare section header and footer not required in Oracle mode.
`/* SPCONV-MSG:(GLOBAL **) Global Variable definition moved to globalPkg Package.*/`	Global variables moved to the OMWB_ EMULATION user package GLOBALPKG.
`/* SPCONV-MSG:(DROP DATABASE) Statement ignored. */`	Oracle databases are seldom dropped in embedded SQL. If a database is dropped it is ignored.
`/* SPCONV-MSG:(DROP **) OMWB_ Emulation.DDL_MANAGER procedure used to execute DDL statement.*/`	Release 9.2.0.1.0 of the Migration Workbench does not support DROP ** statements. These statements are ignored.
`/* ESQL-CONV-MSG:(BEGIN .. END) Embedded PL/SQL code block generated for Stored Procedure call. */`	An informational message describing how Oracle code contains an embedded PL/SQL code block.
`/* ESQL-CONV-MSG:(SELECT) Statement illegal as a procedure parameter in Oracle. */`	Statement illegal as a procedure parameter in Oracle.
`/* ESQL-CONV-MSG:(SELECT) Statement removed from procedure call. */`	Statement removed from procedure call.
`/* ESQL-CONV-MSG:(*) Manual conversion of the generated variable TYPE may be required.*/`	SELECT statement in Informix Dynamic Server converted into a SELECT variable, which may have the wrong type. Manual conversion may be required.
`/* ESQL-CONV-MSG:(INTO) Procedure call converted to function call as only one value returned. */`	An Informix Dynamic Server procedure returning one value converts to an Oracle function. PL/SQL functions must return a value into a variable, for example `a:= func(); just func();` will create an error.

Table 5–2 ESQL/C to Oracle Pro*C Warnings (Cont.)

Warning	Description
`/* ESQL-CONV-MSG:(**) Statement emulated using Oracle FOR syntax within cursor declaration. */`	An informational message describing emulation in Oracle.
`/* ESQL-CONV-WRN:(**) Oracle has no equivalent cursor action. Manual conversion required.*/`	Oracle does not have various cursor actions. You can manually move the data from a cursor into a PL/SQL table.
`/* ESQL-CONV-MSG:(INTO) Clause originally declared within cursor declaration. */`	The INTO clause was originally in the DECLARE section but was moved to the FETCH statement.
`/* ESQL-CONV-MSG:(FREE) Statement not required in ORACLE. */`	EXEC SQL FREE CURSORID statement placed in a comment.
`/* ESQL-CONV-MSG:(DYNAMIC SQL) Unsupported in this release. Manual conversion may be required.*/`	Release 9.2.0.1.0 of the Migration Workbench does not support Dynamic SQL, but simple SQL and simple ANSI dynamic SQL are supported. Convert substitution variables to `:var1`.
`/* ESQL-CONV-MSG:(GET DIAGNOSTICS) Manual conversion required.*/`	This call is significantly different between Oracle and Informix Dynamic Server. You must manually convert it.
`/* SPCONV-MSG:(SUBQUERY) Subquery within IF statement emulated by using Boolean variable.*/`	An informational message describing emulation in Oracle.
`/* SPCONV-MSG:(LOCK TABLE) Please see 'Oracle 8i Server SQL reference' for details of other LOCK options.*/`	An informational message. Refer to the Oracle 8i Server SQL documentation for more information.
`/* SPCONV-MSG:(WITH RESUME) Statement emulated through use of GLOBAL TEMPORARY TABLES.*/`	Oracle does not support the WITH RESUME statement. It is emulated using temporary tables. Manual conversion may be required.
`/* SPCONV-MSG:(RETURNING) Informix RETURNING clause parameters converted to Oracle OUT parameters.*/`	If an Informix Dynamic Server function has more than one returning parameter, these are converted into Oracle OUT parameters.

Table 5–2 ESQL/C to Oracle Pro*C Warnings (Cont.)

Warning	Description
`/* ESQL-CONV-MSG:(ROLLBACK) Statement expanded to utilise Oracle SAVEPOINTS. */`	An informational message describing how the ROLLBACK statement is used to go to the previous SAVE'POINT, if that parser option is used.
`/* ESQL-CONV-MSG:(ROLLBACK) Statement automatically closes all cursors referenced in a CURRENT OF clause when compilation mode = ORACLE. */`	An informational message describing how cursors are closed in Oracle mode.
`/* ESQL-CONV-MSG:(ROLLBACK) Statement closes all explicit cursors when compilation mode = ANSI. */`	An informational message describing how cursors are closed in ANSI mode.
`/* ESQL-CONV-MSG:(ROLLBACK) Extra ROLLBACK option available in Oracle. Please see the Oracle 8i users guide for more information. */`	An informational message. Refer to the Oracle 8i Server SQL documentation for more information.
`/* ESQL-CONV-MSG:(OUTER) Simple OUTER joins may not be fully converted. Manual conversion may be required. */`	Simple OUTER joins may not be automatically converted.
`/* ESQL-CONV-MSG:(MATCHES) Complex search patterns not fully converted. Manual conversion may be required. */`	An informational message describing how complex search patterns are not fully converted. Manual conversion may be required.
`/* ESQL-CONV-MSG:(NOWAIT) Keyword added to emulate Informix SET LOCK MODE statement. */`	Informix Dynamic Server sets NOWAIT in a SET statement, but Oracle places it in the SELECT FOR UPDATE statement.
`/* ESQL-CONV-MSG:(GROUP BY) Oracle does not enable literal numbers to be used in the GROUP BY clause. Manual conversion may be required. */`	Replace `GROUP BY 1,3;` with `GROUP BY col1, col3;`

Table 5–2 ESQL/C to Oracle Pro*C Warnings (Cont.)

Warning	Description
`/* SPCONV-MSG:(SET DEBUG FILE) OMWB_emulation.utilities Package introduced to mimic Informix functionality.*/`	Debugging enabled by use of the `OMWB_EMULATION.UTILITIES` package.
`/* ESQL-CONV-MSG:(NOT WAIT) Option may be emulated by implementing the Oracle NOWAIT SELECT statement option. */`	Informix Dynamic Server sets NOWAIT in a SET statement, but Oracle places it in the SELECT FOR UPDATE statement.
`/* SPCONV-MSG:(SYSTEM) Emulating Informix SYSTEM statement by using OMWB_emulation.SHELL Procedure.*/`	The `OMWB_EMULATION.SHELL` procedure can emulate the system command, along with a small C program.
`/* SPCONV-MSG:(TRACE) OMWB_ emulation.utilities Package introduced to mimic Informix functionality.*/`	Trace facilities are provided by the `OMWB_EMULATION.UTILITIES.DEBUG()` procedure.
`/* ESQL-CONV-MSG:(ONLY) No equivalent available in Oracle. Statement ignored. */`	ONLY statement is not supported in Oracle. You must manually convert it.
`/* SPCONV-MSG:(UPDATE STATISTICS) Statement ignored.*/`	UPDATE statistics statement is not supported in Oracle. You must manually convert it.
`/*ESQL-CONV-MSG:(WHENEVER) Oracle supports additional directives. */`	An informational message.
`/*SPCONV-WRN:(ALTER TABLE) Unable to convert ALTER TABLE statement. Manual conversion required*/`	The ALTER TABLE options used do not directly convert to Oracle syntax. You must manually convert it.
`/* SPCONV-WRN:(TEMP TABLE): It will be more performant to create the temporary table separately .*/`	Remove the CREATE TABLE statement and run separately, before running the Oracle Pro*C application. Oracle temporary tables hold per-session information but they cannot be separately created for each session.
`/* SPCONV-WRN:(**) Conversion of remote Database links not supported. Manual conversion required. */`	Use Oracle database links to simulate remote database links. Refer to the Oracle9*i* documentation for more details.

Table 5–2 ESQL/C to Oracle Pro*C Warnings (Cont.)

Warning	Description
`/* SPCONV-WRN:(REFERENCES BYTE) Converted to Oracle BLOB datatype. Restrictions apply.*/`	Informix Dynamic Server BLOB and CLOB datatype support differs from Oracle BLOB and CLOB datatype support. Manual conversion required. In this example, the BYTE datatype is converted to BLOB.
`/* SPCONV-WRN:(REFERENCES TEXT) Converted to Oracle CLOB datatype. Restrictions apply.*/`	Informix Dynamic Server BLOB and CLOB datatype support differs from Oracle BLOB and CLOB datatype support. Manual conversion required. In this example, the TEXT datatype is converted to CLOB.
`/* SPCONV-MSG:(DROP **) Statement passed to DDL file.*/`	One-off statements that are best used in a SQL script.
`/* SPCONV-WRN:(DBINFO) Unable to convert function call. Manual conversion required.*/`	Some DBINFO calls cannot be directly converted to Oracle.
`/* SPCONV-WRN:(**) Manual conversion required if the procedure returns more than one value.*/`	An informational message.
`/* SPCONV-WRN:(EXCEPTION) Emulation of Informix Exceptions incomplete.*/`	Refer to Informix Dynamic Server documentation for more information.
`/* ESQL-CONV-WRN:(RECOVER) Statement Ignored. */`	RECOVER table statement is ignored.
`/* SPCONV-WRN:(RETURN) Collating results for REF CURSOR return.*/`	Results collated into temporary table are selected out in a single result set.
`/* SPCONV-WRN:(FOR READ ONLY) Statement Ignored. */`	Default Oracle behavior is FOR READ ONLY.
`/* SPCONV-WRN:(SYSTEM) Statement Ignored. Parse option turned off.*/`	The SYSTEM emulation option switched off so SYSTEM is ignored. You must manually convert it.
`/* SPCONV-WRN: ( TRACE **) Currently not supported. Manual conversion required.*/`	Some TRACE options are not converted by the converter. Refer to the Oracle documentation for an overview of the Oracle tracing facilities.

*Table 5–2 ESQL/C to Oracle Pro*C Warnings (Cont.)*

Warning	Description
`/* SPCONV-WRN:(MATCHES) Converted to an Oracle LIKE, BETWEEN or IN statement. Manual conversion of the search pattern may be required.*/`	The MATCHES statement is not precisely converted to Oracle. You must manually convert the search pattern.

Restrictions

There are some restrictions with the Informix Dynamic Server ESQL/C to Oracle Pro*C Converter. The following converter restrictions apply:

Renaming Reserved Words

The converter adds an underscore after reserved words so `EXEC SQL INCLUDE datetime.h` becomes `EXEC SQL INCLUDE datetime_.h`, which in this case is not useful. This behavior is useful where a column name is an Oracle reserved word and the Migration Workbench schema conversion renames the column by adding an underscore. Then `COLUMN YEAR` becomes `YEAR_` in both the embedded SQL and schema creation part of the Migration Workbench.

Header Files

There may be differences between header files used in Informix Dynamic Server ESQL/C and Oracle Pro*C. You must remove Informix Dynamic Server specific files and replace them with the Oracle equivalent.

Using multiple connections for different transactions

The converter does not support multiple connections. It replaces more complicated Informix Dynamic Server CONNECT statements with a simple Oracle CONNECT statement. For information on how to manage multiple contexts in Oracle, refer the following commands in the Oracle Pro*C documentation:

```
EXEC SQL CONTEXT ALLOCATE :ctx;
EXEC SQL CONTEXT USE :ctx;
EXEC SQL CONTEXT FREE :ctx;
```

Using Demonstration Code

To create a user in Oracle, enter the following commands:

```
>sqlplus SYSTEM/MANAGER
SQL>CREATE USER examp IDENTIFIED BY examp;
SQL>GRANT CONNECT, RESOURCE TO examp;
SQL>CONNECT examp/examp
SQL>CREATE TABLE emp (      EMPNO NUMBER(4),
                        ENAME VARCHAR2(10),
                        JOB VARCHAR2(9),
                        MGR NUMBER(4),
                        HIREDATE DATE,
                        SAL NUMBER(7,2),
                        COMM NUMBER(7,2),
                        DEPTNO NUMBER(2));
SQL>INSERT INTO EMP VALUES
        (7499, 'ALLEN',  'SALESMAN',  7698,
        SYSDATE, 1600,  300, 30);
SQL>INSERT INTO EMP VALUES
        (7521, 'WARD',   'SALESMAN',  7698,
        SYSDATE, 1250,  500, 30);
SQL>INSERT INTO EMP VALUES
        (7566, 'JONES',  'MANAGER',   7839,
        SYSDATE,  2975, NULL, 20);
SQL>INSERT INTO EMP VALUES
        (7839, 'KING',   'PRESIDENT', NULL,
        SYSDATE, 5000, NULL, 10);
SQL>COMMIT;
```

The executable relies on a database populated by data. The following example shows how to produce an executable:

1. Precompile the code from Oracle Pro*C to a C file using the Oracle Pro*C executable `proc esqlproc1.pc`.
2. Compile the C code using a suitable development environment. The Oracle Pro*C example shipped by with Oracle contains project files for the Visual C++ development environment. The project contains details of the libraries and include files required to build a small executable based on Oracle Pro*C. The project file used is `%ORACLE_HOME%\precomp\demo\proc\sample.dsp`. You have to add the C file `esqlproc1.c` to the project.

For more information on Oracle Pro*C / C++ refer to the *Pro*C/C++ Precompiler Programmer's Guide.*

6

Disconnected Source Model Loading

The Disconnected Source Model Load feature of the Migration Workbench allows consultants to work on a customer's database migration without having to install and run the Migration Workbench at the customer site.

To perform the disconnected source model load option a customer must generate delimited flat files containing schema metadata from the database to be migrated. You generate the flat file by running a predefined Migration Workbench script against the source database. The flat files are sent to a consultant who uses the Migration Workbench to load the metadata files into a source and Oracle model. You can then map this schema to Oracle.

Generating Database Metadata Flat Files

Informix Dynamic Server databases use the Bulk Copy Program (BCP) to generate delimited metadata flat files. Predefined scripts installed with the Migration Workbench invoke the BCP, and generate the flat files for each database. The BCP outputs delimited metadata files from the database with a `.dat` extension. However, for a successful migration of a database the .dat metadata files are converted into XML files by the Migration Workbench. The Migration Workbench converts the .dat files when the source metadata files are selected during the capture phase of the migration, and outputs the generated `.xml` files to the same root directory as the source `.dat` files.

Flat File Generation Scripts

The predefined script file for Informix Dynamic Server is in the `%ORACLE_HOME%\Omwb\DSML_Scripts\informix7` directory.

Running the Script

To run the script file from the %ORACLE_HOME%\DSML_scripts\informix7 directory, use the following command line:

```
IDS7_DSML_SCRIPT <root dir> <servername>
```

A

Code Samples

This appendix contains a sample of an Oracle package used to convert TRACE statements.

OMWB_Emulation Utilities Package

For release 9.2.0.1.0 of the Migration Workbench a user has to be added manually, or the OMWB_emulation references in the generated code should be removed:

```
REM
REM Message : Created User :omwb_emulation
REM User :
CREATE USER omwb_emulation IDENTIFIED BY oracle
;
GRANT CONNECT,RESOURCE TO omwb_emulation
;
CREATE TABLE OMWB_emulation.debug_table(log_date DATE,log_user VARCHAR(100),log_
message
VARCHAR(4000));

CONNECT Omwb_emulation/oracle
REM
REM Message : Created Package : UTILITIES_1
REM User : omwb_emulation
CREATE OR REPLACE PACKAGE utilities AS
  DebugFile   VARCHAR2(20) DEFAULT 'trace.log';
  /* The following variable DebugDir should be edited to
     DEFAULT to a valid UTL_FILE_DIR entry within the
     destination databases init.ora initialization file. */
  DebugDir    VARCHAR2(50); /* DEFAULT ''; */
  DebugOut    INTEGER DEFAULT 3;
```

```
  PROCEDURE DEBUG(debug_statement VARCHAR2);
  PROCEDURE DEBUG_TO_TABLE(debug_statement VARCHAR2);
  PROCEDURE DEBUG_TO_DBMS(debug_statement VARCHAR2);
  PROCEDURE DEBUG_TO_FILE(debug_statement VARCHAR2);
  PROCEDURE RESET_DEBUG_TABLE;
  PROCEDURE RESET_DEBUG_FILE;

  FUNCTION HEX (n pls_integer)
    RETURN VARCHAR2;
  FUNCTION MDY (month_in pls_integer,
                day_in   pls_integer,
                year_in  pls_integer)
    RETURN DATE;
  FUNCTION DAY (date_in DATE)
    RETURN INTEGER;
  FUNCTION MONTH (date_in DATE)
    RETURN INTEGER;
  FUNCTION YEAR(date_in DATE)
    RETURN INTEGER;
  FUNCTION WEEKDAY(date_in DATE)
    RETURN INTEGER;
END utilities;

/

REM
REM Message : Created Package : GLOBALPKG_1
REM User : omwb_emulation
CREATE OR REPLACE PACKAGE globalPkg AUTHID CURRENT_USER AS
/* The following are T/SQL specific global variables. */
  identity  INTEGER;
  trancount INTEGER;
  TYPE RCT1 IS REF CURSOR;/*new weak cursor definition*/
  PROCEDURE incTrancount;
  PROCEDURE decTrancount;
END globalPkg;
/

REM
REM End Packages for omwb_emulation
REM

REM
REM Start Stored Procedures for omwb_emulation
REM
```

```
REM
REM Message : Created Procedure : UTILITIES
REM User : omwb_emulation
CREATE OR REPLACE PACKAGE BODY utilities AS

PROCEDURE DEBUG (debug_statement IN VARCHAR2) IS
BEGIN
  /* Call the appropriate sub procedure depending on the
     value of the utilities.DebugOut variable.
     This variable should be set within the utilities
     package header. */
  IF(debug_statement IS NULL) THEN
    RETURN;
  END IF;
  IF    (utilities.DebugOut = 1) THEN
    DEBUG_TO_FILE(debug_statement);
  ELSIF (utilities.DebugOut = 2) THEN
    DEBUG_TO_DBMS(debug_statement);
  ELSE
    DEBUG_TO_TABLE(debug_statement);
  END IF;
END DEBUG;

PROCEDURE DEBUG_TO_TABLE (debug_statement IN VARCHAR2) IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO OMWB_emulation.debug_table
  VALUES(SYSDATE,
         USER,
         debug_statement);
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20108,'utilities.DEBUG_TO_TABLE : Error raised when
attempting to insert row into OMWB_Emulation.debug_table table.');
END DEBUG_TO_TABLE;

PROCEDURE DEBUG_TO_DBMS(debug_statement VARCHAR2) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(debug_statement);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.ENABLE(1000000);
    DBMS_OUTPUT.PUT_LINE(debug_statement);
```

```
END DEBUG_TO_DBMS;

PROCEDURE DEBUG_TO_FILE(debug_statement VARCHAR2) IS
fileID         UTL_FILE.FILE_TYPE;
BEGIN
  fileID := UTL_FILE.FOPEN(utilities.DebugDir,
                           utilities.DebugFile,
                           'a');
  UTL_FILE.PUT_LINE(fileID,
                    SYSDATE
                    || ' '
                    || USER
                    || ' '
                    || debug_statement);
  UTL_FILE.FCLOSE(fileID);
EXCEPTION
  WHEN UTL_FILE.INVALID_OPERATION THEN
    RAISE_APPLICATION_ERROR(-20100,'utilities.DEBUG_TO_FILE raised : Invalid
operation.');
  WHEN UTL_FILE.INVALID_FILEHANDLE THEN
    RAISE_APPLICATION_ERROR(-20101,'utilities.DEBUG_TO_FILE raised : Invalid
file handle.');
  WHEN UTL_FILE.WRITE_ERROR THEN
    RAISE_APPLICATION_ERROR(-20102,'utilities.DEBUG_TO_FILE raised : Write
Error.');
  WHEN UTL_FILE.INVALID_PATH THEN
    RAISE_APPLICATION_ERROR(-20103,'utilities.DEBUG_TO_FILE raised : Invalid
path.');
  WHEN UTL_FILE.INVALID_MODE THEN
    RAISE_APPLICATION_ERROR(-20104,'utilities.DEBUG_TO_FILE raised : Invalid
mode.');
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20105,'utilities.DEBUG_TO_FILE raised : Unhandled
Exception.');
END DEBUG_TO_FILE;

PROCEDURE RESET_DEBUG_TABLE IS
BEGIN
  DELETE FROM OMWB_Emulation.debug_table;
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20107,'utilities.RESET_DEBUG_TABLE : Error raised
when attempting to clear the OMWB_Emulation.debug_table table.');
END RESET_DEBUG_TABLE;
```

```
PROCEDURE RESET_DEBUG_FILE IS
fileID         UTL_FILE.FILE_TYPE;
BEGIN
  fileID := UTL_FILE.FOPEN(utilities.DebugDir,
                           utilities.DebugFile,
                           'w');
  UTL_FILE.PUT_LINE(fileid,
                    'Log file creation :'
                    || SYSDATE);
  UTL_FILE.FCLOSE(fileID);
EXCEPTION
  WHEN UTL_FILE.INVALID_OPERATION THEN
    RAISE_APPLICATION_ERROR(-20100,'utilities.RESET_DEBUG_FILE  raised : Invalid
operation.');
  WHEN UTL_FILE.INVALID_FILEHANDLE THEN
    RAISE_APPLICATION_ERROR(-20101,'utilities.RESET_DEBUG_FILE  raised : Invalid
file handle.');
  WHEN UTL_FILE.WRITE_ERROR THEN
    RAISE_APPLICATION_ERROR(-20102,'utilities.RESET_DEBUG_FILE  raised : Write
Error.');
  WHEN UTL_FILE.INVALID_PATH THEN
    RAISE_APPLICATION_ERROR(-20103,'utilities.RESET_DEBUG_FILE  raised : Invalid
path.');
  WHEN UTL_FILE.INVALID_MODE THEN
    RAISE_APPLICATION_ERROR(-20104,'utilities.RESET_DEBUG_FILE  raised : Invalid
mode.');
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20105,'utilities.RESET_DEBUG_FILE  raised :
Unhandled Exception.');
END RESET_DEBUG_FILE;

FUNCTION HEX(n pls_integer)
RETURN VARCHAR2 IS
BEGIN
  IF n > 0 THEN
    RETURN HEX (TRUNC (n / 16)) || SUBSTR ('0123456789ABCDEF', MOD (n, 16) + 1,
1);
  ELSE
    RETURN NULL;
  END IF;
END HEX;

FUNCTION MDY(month_in pls_integer,
             day_in   pls_integer,
             year_in  pls_integer)
```

```
RETURN DATE IS
bad_day   EXCEPTION;
bad_month EXCEPTION;
bad_year  EXCEPTION;
BEGIN
  IF month_in < 0 OR month_in > 12 THEN
    RAISE bad_month;
  END IF;
  IF day_in < 0 OR day_in > 31 THEN
    RAISE bad_day;
  END IF;
  IF year_in < 999 THEN
    RAISE bad_year;
  END IF;
  RETURN TO_DATE(TO_CHAR(month_in)
                 || '-'
                 || TO_CHAR(day_in)
                 || '-'
                 || TO_CHAR(year_in),
                 'MM-DD-YYYY');
EXCEPTION
  WHEN bad_day THEN
    RETURN NULL;
  WHEN bad_year THEN
    RETURN NULL;
  WHEN bad_month THEN
    RETURN NULL;
END MDY;

FUNCTION DAY(date_in DATE)
RETURN INTEGER IS
BEGIN
  IF date_in IS NULL THEN
    RETURN NULL;
  END IF;
  RETURN TO_NUMBER(TO_CHAR(date_in,'DD'));
END DAY;

FUNCTION MONTH(date_in DATE)
RETURN INTEGER IS
BEGIN
  IF date_in IS NULL THEN
    RETURN NULL;
  END IF;
  RETURN TO_NUMBER(TO_CHAR(date_in,'MM'));
```

```
END MONTH;

FUNCTION YEAR(date_in DATE)
RETURN INTEGER IS
BEGIN
  IF date_in IS NULL THEN
    RETURN NULL;
  END IF;
  RETURN TO_NUMBER(TO_CHAR(date_in,'YYYY'));
END YEAR;

FUNCTION WEEKDAY(date_in DATE)
RETURN INTEGER IS
BEGIN
  IF date_in IS NULL THEN
    RETURN NULL;
  END IF;
  RETURN TO_NUMBER(TO_CHAR(date_in,'D'));
END WEEKDAY;

END utilities;

/
REM
REM Message : Created Procedure : SHELL
REM User : omwb_emulation
CREATE OR REPLACE PROCEDURE SHELL(os_command VARCHAR)
AUTHID CURRENT_USER AS
BEGIN
/* This is a dummy stored procedure added by the migration
   workbench. Please see the Migration Workbench users
   guide for information on how to configure this procedure
   for use. */
   NULL;
END SHELL;
/

REM
REM Message : Created Procedure : GLOBALPKG
REM User : omwb_emulation
CREATE OR REPLACE PACKAGE BODY globalPkg AS
/* This is a dummy package body added by the migration
   workbench in order to emulate T/SQL specific global variables. */
PROCEDURE incTrancount IS
BEGIN
```

```
  trancount := trancount + 1;
END incTrancount;
PROCEDURE decTrancount IS
BEGIN
  trancount := trancount - 1;
END decTrancount;
END globalPkg;
/

REM
REM Message : Created Procedure : DDL_MANAGER
REM User : omwb_emulation
CREATE OR REPLACE PROCEDURE DDL_Manager(ddl_statement VARCHAR)
AUTHID CURRENT_USER IS
BEGIN
  EXECUTE IMMEDIATE ddl_statement;
EXCEPTION
  WHEN OTHERS THEN
    RAISE;
END DDL_Manager;
/

REM
REM End Stored Procedures for omwb_emulation
REM

GRANT EXECUTE ON utilities TO public;

 GRANT SELECT, INSERT ON Omwb_emulation.debug_table TO PUBLIC;

GRANT EXECUTE ON SHELL TO public;

GRANT EXECUTE ON globalPkg TO public;

GRANT EXECUTE ON DDL_Manager TO public;
```

Index

Symbols

A

B

C

D

E

F

I

K

L

M

N

O

P

R

S

T

U

V

W