Data Engineering Re-skill Program

Training Manual

Trainer: Deirdre Geary	Introduction to Data Engineering

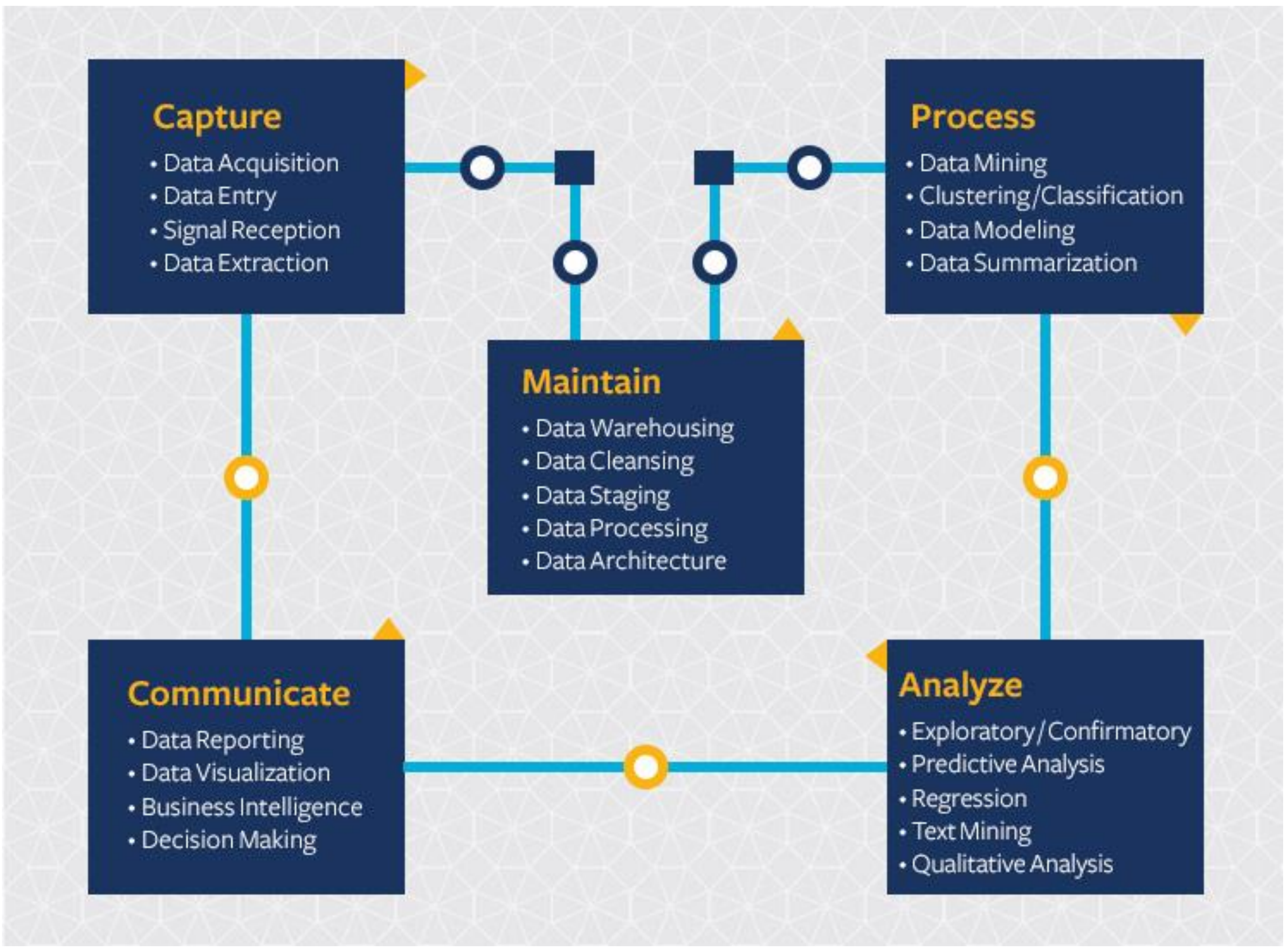
Table of Contents

PART I – DATA SCIENCE

Lesson 1.1: What is data science?

Data Science is the ability to take data, to be able to understand it, to process it, to extract value from it, to visualize it and to communicate it. Effective data scientists are able to identify relevant questions, collect data from a multitude of different data sources, organize the information, translate results into solutions, and communicate their findings in a way that positively affects business decisions.

Data Science Processes



1. Capture, (data acquisition, data entry, signal reception, data extraction);
2. Maintain (data warehousing, data cleansing, data staging, data processing, data architecture);
3. Process (data mining, clustering/classification, data modeling, data summarization);
4. Analyze (exploratory/confirmatory, predictive analysis, regression, text mining, qualitative analysis);
5. Communicate (data reporting, data visualization, business intelligence, decision making).

Lesson 1.2: What is role of data scientist?

The role of a data scientist is to turn raw data into actionable insights. Much of the world's raw data–from electronic medical records to customer transaction histories–lives in organized collections of tables called relational databases. Therefore, to be an effective data scientist, you must know how to wrangle and extract data from these databases using SQL. Data scientists examine which questions need answering and where to find the related data. They have business acumen and analytical skills as well as the ability to mine, clean, and present data. Businesses use data scientists to source, manage, and analyze large amounts of unstructured data. Results are then synthesized and communicated to key stakeholders to drive strategic decision-making in the organization.

Skills needed: Programming skills (R, Python), statistical and mathematical skills, storytelling and data visualization, Hadoop, SQL, machine learning.

Lesson 1.3: What is the Team Data Science Process?

The Team Data Science Process (TDSP) is an agile, iterative data science methodology to deliver predictive analytics solutions and intelligent applications efficiently. TDSP helps improve team collaboration and learning. It contains a distillation of the best practices and structures from Microsoft and others in the industry that facilitate the successful implementation of data science initiatives.

Data science lifecycle

The Team Data Science Process (TDSP) provides a lifecycle to structure the development of your data science projects. The lifecycle outlines the steps, from start to finish, that projects usually follow when they are executed.

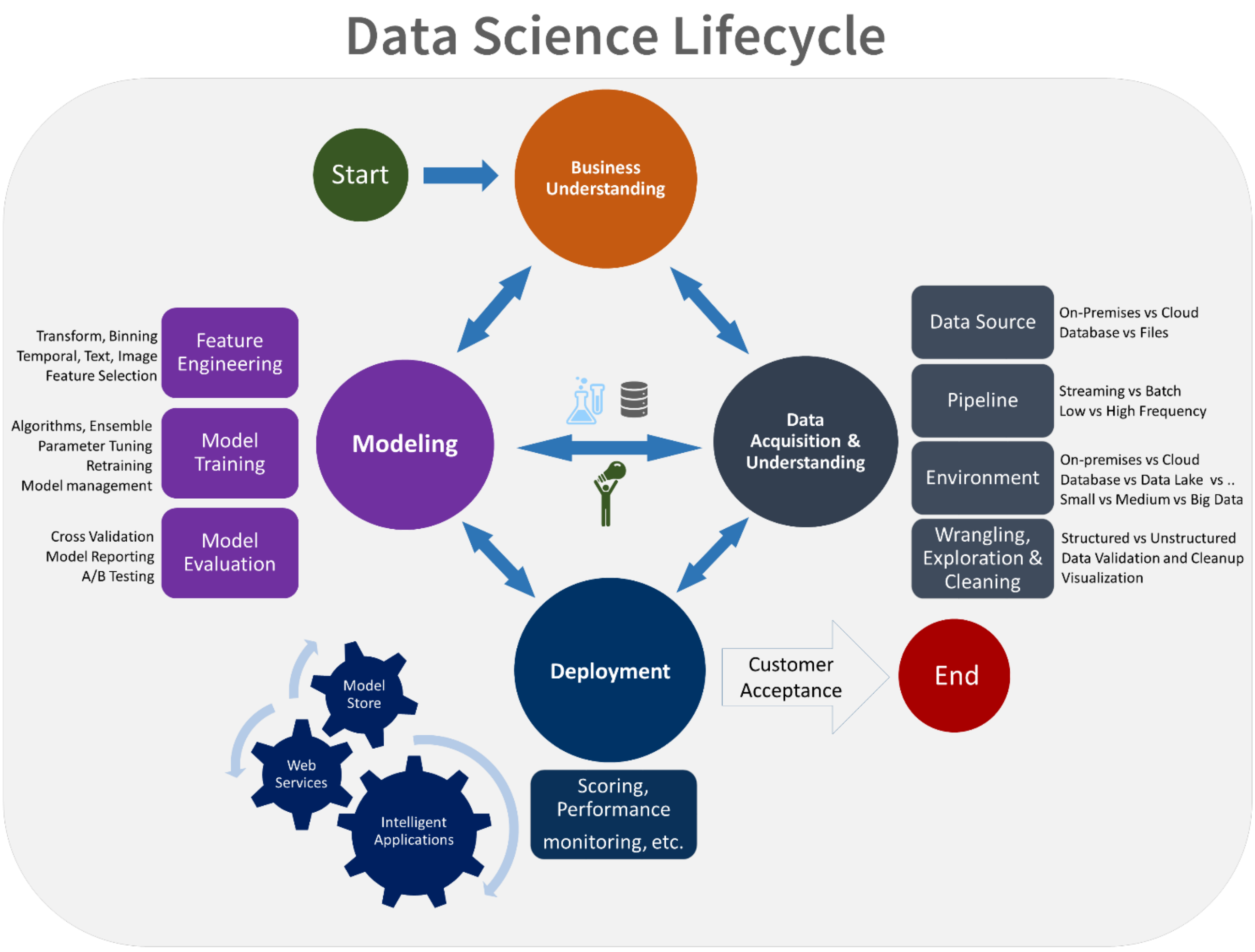
If you are using another data science lifecycle, such as CRISP-DM, KDD or your organization's own custom process, you can still use the task-based TDSP in the context of those development lifecycles. At

a high level, these different methodologies have much in common.

This lifecycle has been designed for data science projects that ship as part of intelligent applications. These applications deploy machine learning or artificial intelligence models for predictive analytics. Exploratory data science projects or ad hoc analytics projects can also benefit from using this process. But in such cases, some of the steps described may not be needed.

The lifecycle outlines the major stages that projects typically execute, often iteratively:

- **Business Understanding**
- **Data Acquisition and Understanding**
- **Modeling**
- **Deployment**
- **Customer Acceptance**


Data Science Lifecycle
Start
Business Understanding
Data Acquisition & Understanding
Data Source
On-Premises vs Cloud
Database vs Files
Pipeline
Streaming vs Batch
Low vs High Frequency
Environment
On-premises vs Cloud
Database vs Data Lake vs ..
Small vs Medium vs Big Data
Wrangling, Exploration & Cleaning
Structured vs Unstructured
Data Validation and Cleanup
Visualization
Modeling
Feature Engineering
Transform, Binning
Temporal, Text, Image
Feature Selection
Model Training
Algorithms, Ensemble
Parameter Tuning
Retraining
Model management
Model Evaluation
Cross Validation
Model Reporting
A/B Testing
Deployment
Scoring, Performance monitoring, etc.
Model Store
Web Services
Intelligent Applications
Customer Acceptance
End

PART II – DATA MODELING

Section 1: Data Modeling Fundamentals

In this section you will cover the following topics:

- What is data modeling?
- Why model data?
- How are data models used in practice?
- A brief overview of different types of models
- Conceptual Data Models
- Logical Data Models
- Physical Data Models
- Comparison of CDM, LDM and PDM
- Overview of four common data modelling notations
- Comparing the syntax of the data modeling notations

Lesson 1.1: What is data modeling?

Data modelling is the analysis of data objects and their relationships to other data objects. It is often the first step in database design and object-oriented programming as the designers first create a conceptual model of how data items relate to each other. Data modeling involves a progression from conceptual model to logical model to physical schema.

Lesson 1.2: Why model data?

The concept of data modeling can be better understood if we compare the development cycle of a data model to the construction of a house. For example, a couple are planning to build a house (database) and call the architect (data modeler) and talk through their building requirements (business requirements). The architect (data modeler) develops the plans (data model) and gives it to couple. Once the plans have been agreed, the couple calls the builders (DBA) to build the house (database).

Why does the architect develop plans for a new house? Probably for the same reason that the data modeller does for a business's data requirements. Pictures help people understand concepts and ideas. They help us visualize what things look like before they are completed.

Models facilitate communication between systems people and end users so that both parties can validate and confirm what the requirements are. A model is easy to change because it is just a picture rather than a fully developed system that has taken a long time to develop. Mistakes are costly, and if communication discrepancies and ideas can be fully defined in advance, the possibility of potential error later in the project can be minimized. Models empower and provide end users with a sense of ownership because a picture is something tangible and can be used as documentation throughout the project. Often, end users do not know what the requirements are until they can see it on paper. Getting your users involved early in the project and engaged in the building and validation of the model will increase the quality and adoption of the system after it is built.

The aim of a data model is to make sure that all data objects provided by the data modeller are completely and accurately represented. The data model needs to be detailed enough to be used by the DBA to build the physical database. The information contained in the model will be used to define the significance of business, relational tables, primary and foreign keys, stored procedures, and triggers so the data model can be used to facilitate communication within the business.

Lesson 1.3: A brief overview of different types of models

As mentioned, you are likely to see three basic styles of data model:

Conceptual: describes WHAT the system contains - This data model includes all major entities, relationships and will not contain much detail

about the attributes and is often used in the initial planning phase.

Logical: describes HOW the system will be implemented regardless of the DBMS you will use - This is the actual implementation and extension of a conceptual data model into a logical data model. A logical data model is the version of the model that represents the business requirements of an organization.

Physical: describe HOW the system will be implemented using a specific DBMS – in our case an Oracle 11g database. - This is a complete model that includes all required tables, columns, relationship, database properties for the physical implementation of the database

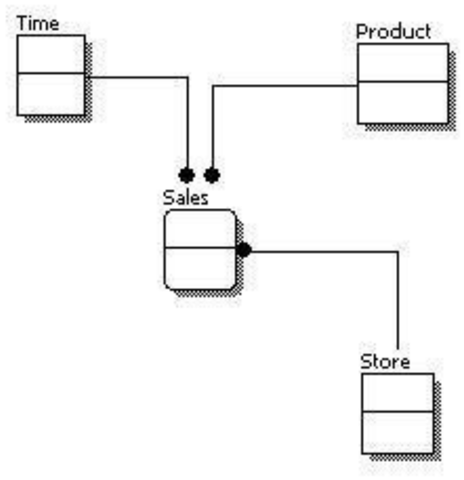
We can see that the complexity increases from conceptual to logical to physical. This is why we always first start with the conceptual data model (so we understand at high level what are the different entities in our data and how they relate to one another), then move on to the logical data model (so we understand the details of our data without worrying about how they will actually implemented), and finally the physical data model (so we know exactly how to implement our data model in the database of choice).

It is worth noting that there are also other modelling styles. For example, some modelling tools, such as SQL Developer Data Modeler also introduce a relational model. It is an intermediate model between the logical model and the physical model. It supports relational design decisions independent of the constraints of the target physical platform.

Lesson 1.5: Conceptual Data Models

A conceptual data model identifies the highest-level relationships between the different entities. Features of conceptual data model:

- Includes the important entities and the relationships among them.
- No attribute is specified.
- No primary key is specified.



So why create a Conceptual Model? Well the most important reason is that a conceptual model facilitates the discussion on the shape of the future system. It helps communication between the data modeller and their client as well as between them and their colleagues. A model also forms a basis for the default design of the physical database, and it is relatively cheap to make and very cheap to change. Last but not least, it forms an important part of your 'ideal system' documentation.

Conceptual modeling is similar to the work of an architect—transforming things that only exist in people's minds into a design that is sufficiently substantial to eventually be created physically.

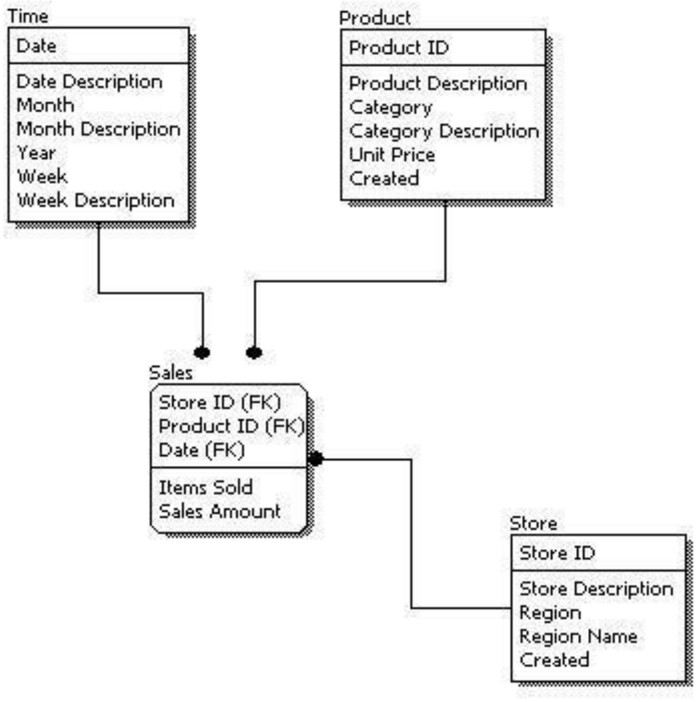
Lesson 1.6: Logical Data Models

A logical data model describes the data in as much detail as possible, disregarding how they will be physical implemented in the database. Features of a logical data model:

- Includes all entities and relationships among them.
- All attributes for each entity are specified.
- The primary key for each entity is specified.
- Foreign keys (keys identifying the relationship between different entities) are specified.
- Normalization occurs at this level.

The basic steps for designing the logical data model are as follows:

- Specify primary keys for all entities.
- Find the relationships between different entities.
- Find all attributes for each entity.
- Resolve many-to-many relationships.
- Normalization.



The Logical Data model is independent of the hardware or software to be used for implementation and is typically developed by data architects or analysts. The diagram that is used to build the Logical Data model is called the Entity Relationship Diagram (ERD). The basic components of an ERD include:

1. **Entities:** Things of significance about which information must be held
2. **Relationships:** How the things of significance are related
3. **Attributes:** Specific information that must be held

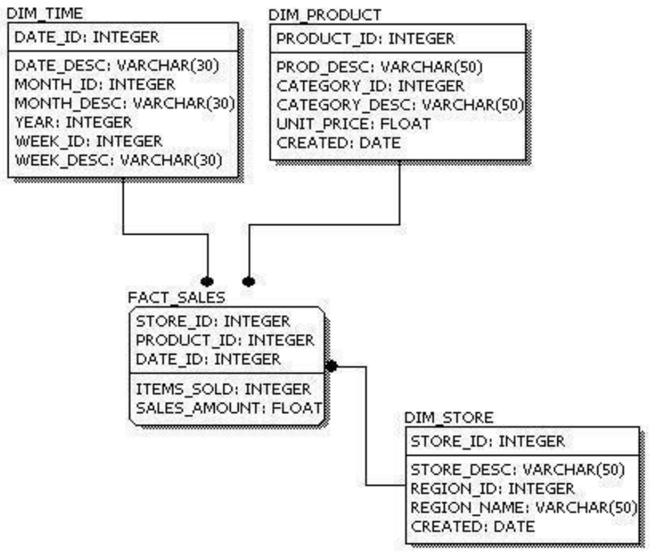
Lesson 1.7: Physical Data Models

Physical data model represents how the model will be built in the database. A physical database model shows all table structures, including column name, column data type, column constraints, primary key, foreign key, and relationships between tables. Features of a physical data model include:

- Specification all tables and columns.
- Foreign keys are used to identify relationships between tables.
- Denormalization may occur based on user requirements.
- Physical considerations may cause the physical data model to be quite different from the logical data model.
- Physical data model will be different for different RDBMS. For example, data type for a column may be different between MySQL and SQL Server.

The steps for physical data model design are as follows:

- Convert entities into tables.
- Convert relationships into foreign keys.
- Convert attributes into columns.
- Modify the physical data model based on physical constraints / requirements.


DIM_TIME
DATE_ID: INTEGER
DATE_DESC: VARCHAR(30)
MONTH_ID: INTEGER
MONTH_DESC: VARCHAR(30)
YEAR: INTEGER
WEEK_ID: INTEGER
WEEK_DESC: VARCHAR(30)
DIM_PRODUCT
PRODUCT_ID: INTEGER
PROD_DESC: VARCHAR(50)
CATEGORY_ID: INTEGER
CATEGORY_DESC: VARCHAR(50)
UNIT_PRICE: FLOAT
CREATED: DATE
FACT_SALES
STORE_ID: INTEGER
PRODUCT_ID: INTEGER
DATE_ID: INTEGER
ITEMS_SOLD: INTEGER
SALES_AMOUNT: FLOAT
DIM_STORE
STORE_ID: INTEGER
STORE_DESC: VARCHAR(50)
REGION_ID: INTEGER
REGION_NAME: VARCHAR(50)
CREATED: DATE

Lesson 1.7: Comparison of CDM, LDM and PDM

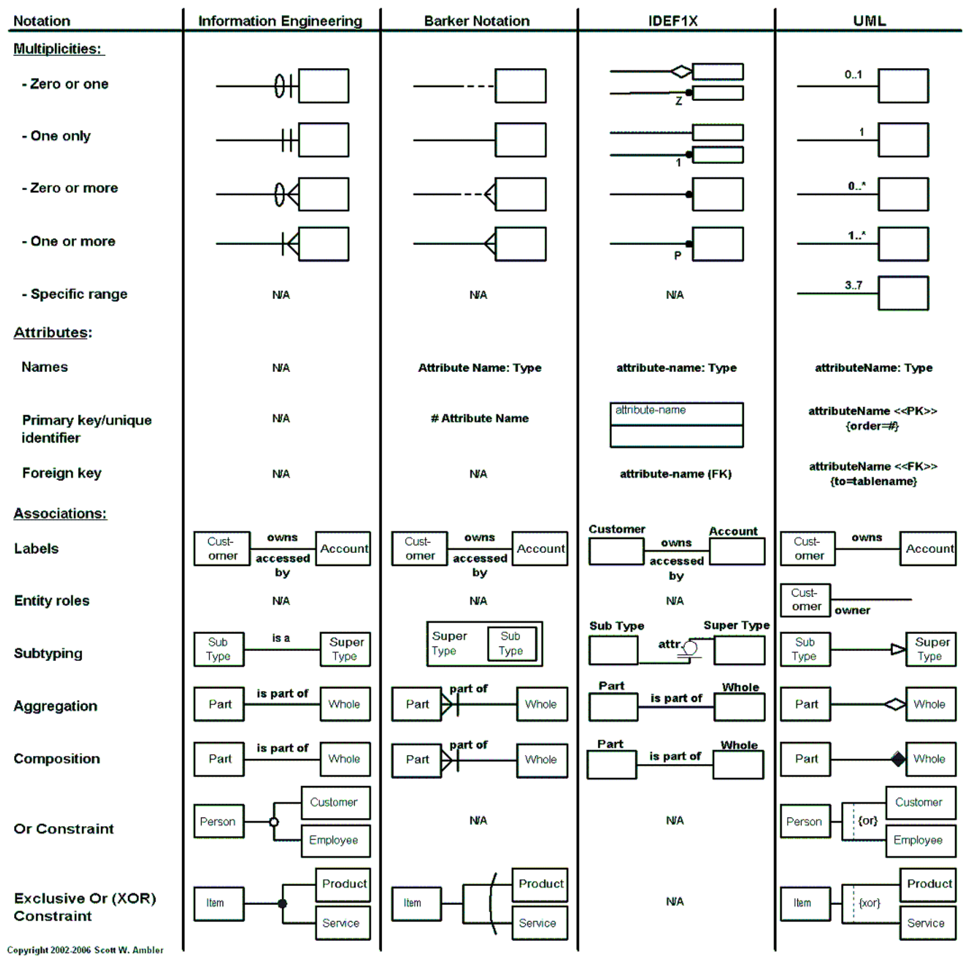
The three levels of data modeling, conceptual data model, logical data model, and physical data model, were discussed in prior sections. Here we compare these three types of data models. The table below compares the different features:

Feature	Conceptual	Logical	Physical
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓

Although LDMs and PDMs sound very similar, and they in fact are, the level of detail that they model can be significantly different. This is because the aims for each diagram is different – you use an LDM to explore domain concepts with your stakeholders and the PDM to define your database design.

Lesson 1.10: Comparing the syntax of the data modeling notations

The following diagram shows a summary of the syntax of four data modeling notations: Information Engineering (IE), Barker, IDEF1X, and the Unified Modeling Language (UML). For more information see David Hay's paper 'A Comparison of Data Modeling Techniques' (http://www.essentialstrategies.com/publications/modeling/compare.htm).



Section 2: How to model data

In this section you will cover the following topics:

- Introduction to the main players: Entities, Attributes and Relationships
- Components of a Relationship
- Many-to-One and One-to-Many Relationships
- Many-to-Many Relationships
- One-to-One Relationships
- Recursive Relationships
- Naming the Relationship
- Determining the Relationship’s Cardinality
- Examples of Unique Identifiers
- Identifying Relationships
- Identifying Relationships with Multiple Entities
- Non-Identifying Relationship
- Primary and Secondary Unique Identifiers
- Searching for Unique Identifiers
- Normalize to reduce data redundancy overview
- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Apply naming conventions

Lesson 2.1: Introduction to the main players: Entities, Attributes and Relationships

The goal of a logical data model is to develop an entity relationship diagram that represents the information requirements of the business. Logical data modeling is independent of the hardware or software to be used for the implementation. A logical data model includes both graphic and textual components. On the surface of the model, you can view the relationships among the data. In the textual descriptions, you can understand the definitions behind the data and the details of the relationships among the objects.

Using an entity relationship diagram to convey the information requirements is a useful communication mechanism. Users can easily understand and change the model as it is developed. It is better to spend time in advance to validate the business rules before making implementation decisions.

An entity relationship diagram has three main components:

- Entity: An object or concept about which you want to store information
- Attributes: Descriptions of entities and specific pieces of information that
- must be known
- Relationship: A natural association that exists between two or more entities

Lesson 2.2: What are entities?

An entity is something of interest. Entities are categories of things that are important for a business and about which information must be kept. Entities contain facts and information that the business must know and remember. Some examples of entities might include the following:

- Person: e.g. employee, customer
- Place: e.g. state, country, municipality
- Thing: e.g. inventory item, vehicle, product
- Concept: e.g. policy, risk, coverage, job
- Organization: e.g. agency, department
- Event: e.g. service request, claim, election

Entity characteristics include the following:

- Is represented by a rectangular box
- Has a unique name, usually in noun form
- Has name in uppercase, with no hyphens or underscores

Ideally an entity should be normal, the data modeling world's version of cohesive. A normal entity depicts one concept, just like a cohesive class models one concept. For example, customer and order are clearly two different concepts; therefore, it makes sense to model them as separate entities.

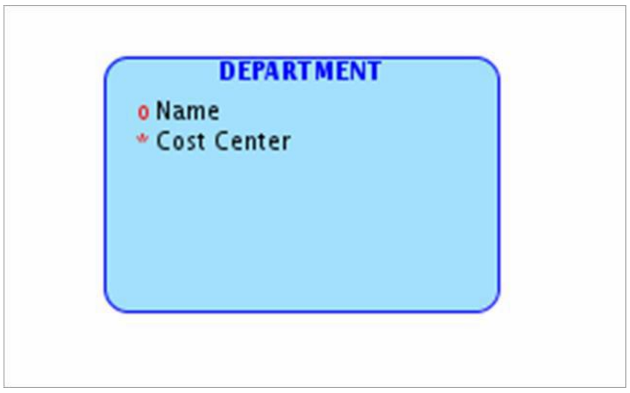
The following questions can help determine whether or not a valid entity has been discovered:

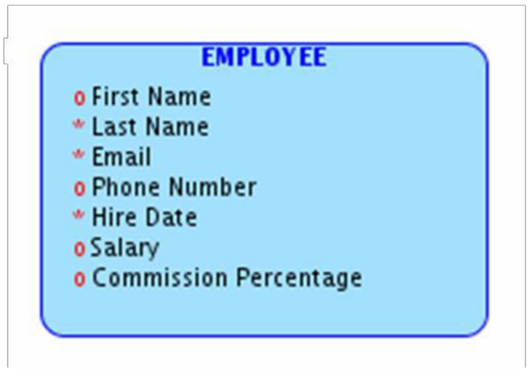
- ✓ Does the entity have business significance to *this Business Area*?
- ✓ Does the *entity have characteristics* that this Business Area needs to use?
- ✓ Should each occurrence of the entity be *uniquely identified*?
- ✓ Does the entity have any *relationship* to another entity?

If the answer to each of the 4 questions is 'Yes,' then a valid entity has been discovered.

Lesson 2.5: What are attributes?

Attributes are information about an entity that must be known or held. Attributes describe an entity by qualifying, identifying, classifying, quantifying, or expressing the state of the entity. Attributes represent a type of description or detail, not an instance. Attribute names are singular and are represented within the entity box.





Attributes have the following characteristics:

- Attributes are displayed within the entity box on the ERD.
- Attribute names should be singular and in mixed case or lowercase.
- Attributes are qualified with name of the entity. Therefore, the attribute name should not include the name of the entity. For example, rather than employee_phone_number, use phone_number.
- Attributes are classified as one of the following:
  - Not null: Indicated by the asterisk * symbol next to the attribute
  - Optional (nulls allowed)**:** Indicated by the o (optional) symbol next to the attribute

The following questions can help determine whether or not an attribute of an entity has been discovered:

✓ Does it describe the entity further?

✓ Does it have significance to *this* Business Area?

✓ Does it represent an *atomic* piece of data?

✓ Does it apply only to *this* particular entity?

If the answer to each of the 4 questions is 'Yes,' then a valid attribute has been discovered for an entity.

Class Exercise: Identify Entities and Attributes

In this practice, you identify and model the entities from the following requirements.

The Training Manager in CoursesRUs wants to model the entity and attributes for a set of requirements. The entity and attributes must store the following:

- Course information (including its title, duration, and fee)
- Information about an instructor (such as name, phone number, and address)
- Information about each student (such as name, phone number, and address)
- Information about what courses a student took
- Information about what courses an instructor taught
- Training location information (such as name, location, and phone number)

Lesson 2.6: Components of a Relationship

A relationship represents the business rules that link entities. There are always two business rules for each relationship. For example :

- A DEPARTMENT may contain one or many EMPLOYEEs.
- An EMPLOYEE must be assigned to one and only one DEPARTMENT.

Each direction of a relationship has:

- A name (for example, 'contain one' or 'assigned to')
- An optionality (for example, either 'must be' or 'may be')
- A degree (for example, either 'one and only one' or 'one or more')

The components of the relationship include the following:

- Name: The label that appears close to the entity it is assigned to. Make sure that all relationship names are in lower case (examples: 'assigned to'' or 'responsible for').
- Cardinality: The minimum and maximum number of values in the relationship
    - Minimum values can be either optional (zero) or mandatory (at least one).
    - Maximum values can be either one or many.

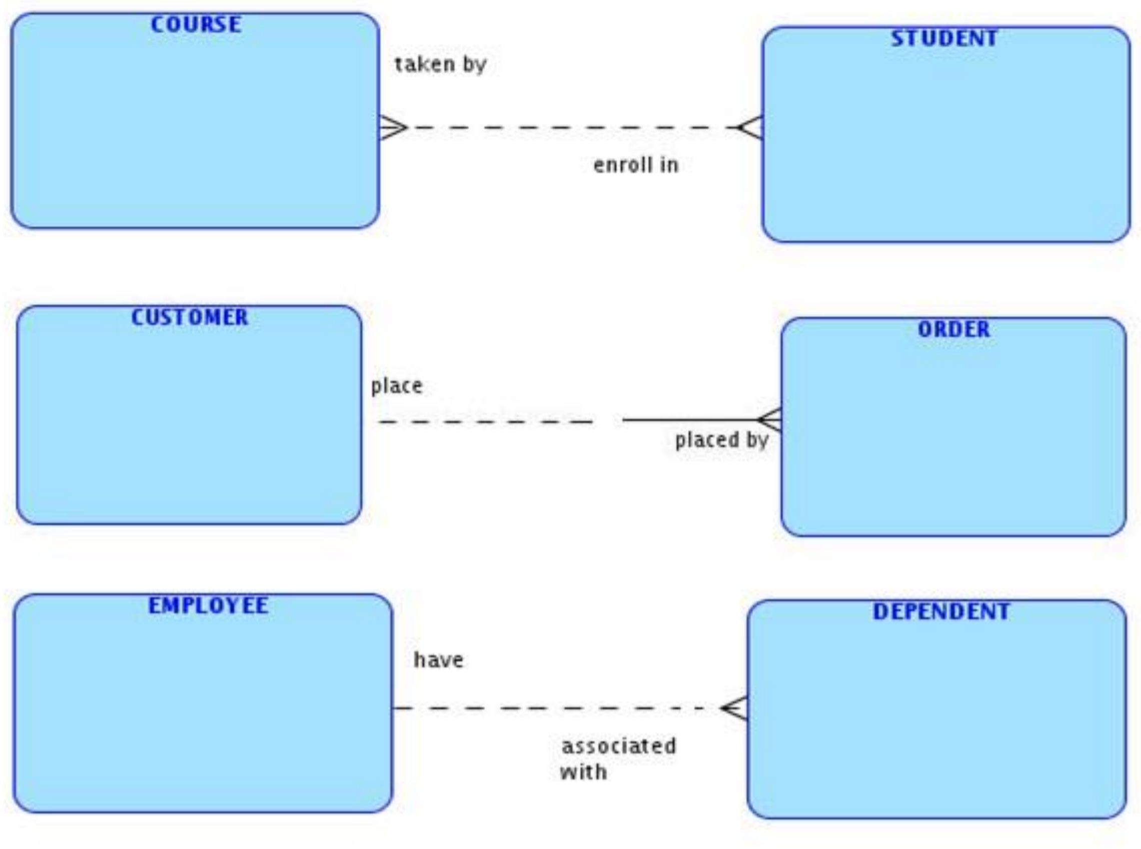
When reading the business rule sentence, use the following words for the minimum values:

- Optional: Use 'may be' or 'may.'
- Mandatory: Use 'must be' or must.'

When reading the business rule sentence, use the following words for the maximum values:

- Line: Use 'one and only one.
- Crow's feet: Use 'one or more.'

Lesson 2.7: Relationships: Some Examples



In order to establish the business rules for each relationship, first read a relationship in one direction and then read the relationship in the other direction.

In the above examples note the following:

Between STUDENT and COURSE, the business rules are:

- A COURSE may be taken by one or more STUDENTs.
- A STUDENT may be enrolled in one or more COURSEs.

Between CUSTOMER and ORDER, the business rules are:

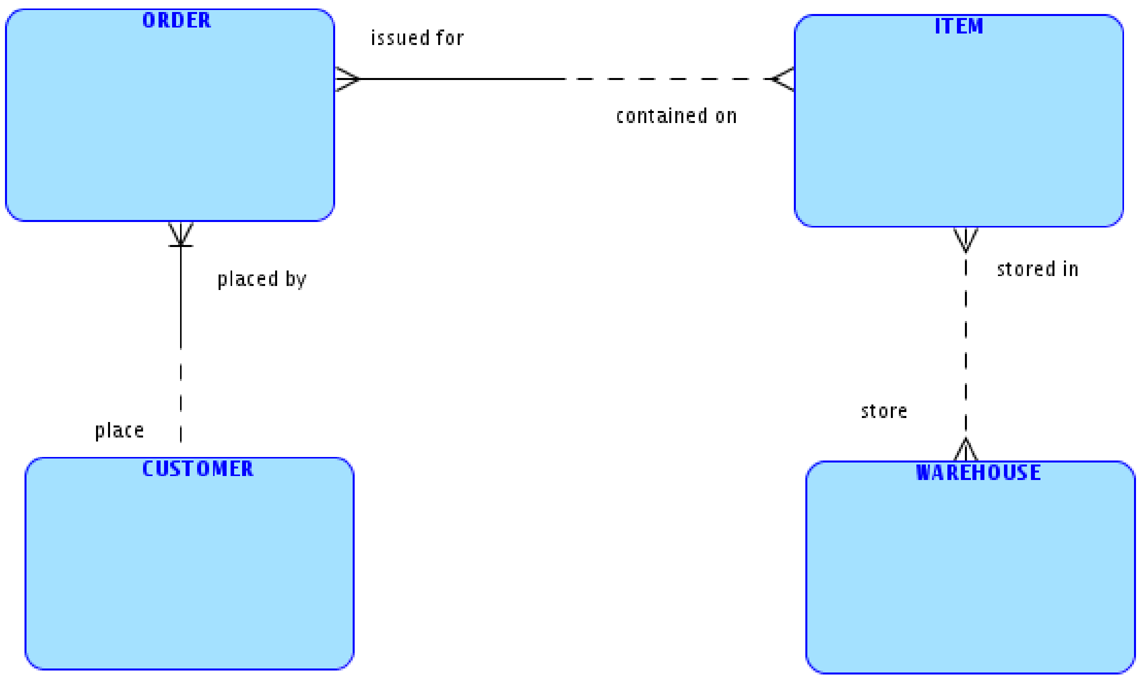
- A CUSTOMER may place one or more ORDERs.
- An ORDER must be placed by one and only one CUSTOMER.

Between EMPLOYEE and DEPENDENT, the business rules are:

- An EMPLOYEE may have one-to-many DEPENDENTs.
- A DEPENDENT may be associated with one and only one EMPLOYEE.

Class Exercise: Define Business Rules

In this practice, write the business rule sentences for the following ERD:



Lesson 2.8: Overview of Relationship Types

There are three types of relationships:

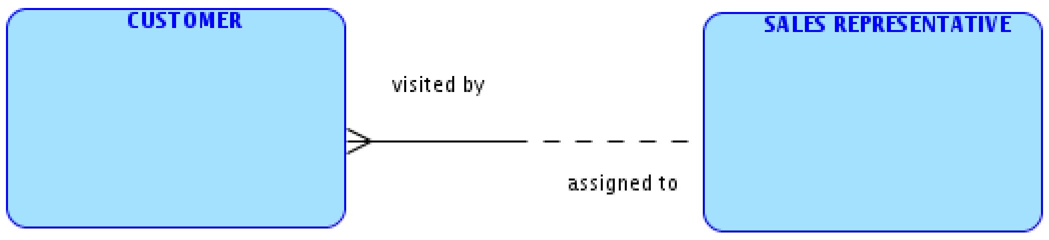
- **Many-to-one (M:1)** or **one-to-many (1:M):** There are crow’s feet on one side of the relationship. The direction of the crow’s feet determines whether the relationship is M:1 or 1:M. This type of relationship is the most common.
- **Many-to-many (M:M):** There are crow’s feet on both sides of this relationship. It is common to see M:M relationships in a high level ERD at the beginning of a project.
- **One-to-one (1:1):** This type of relationship is a line without crow's feet on either end. These types of relationships are rare.

In Oracle SQL Developer Data Modeler, the notation is slightly different:

- One-to-many is 1:N.
- Many-to-many is M:N.

Lesson 2.9: Many-to-One and One-to-Many Relationships

Many-to-one and one-to-many relationships (M:1 and 1:M) have cardinality of *one or more* in one direction and *one and only one* in the other direction.

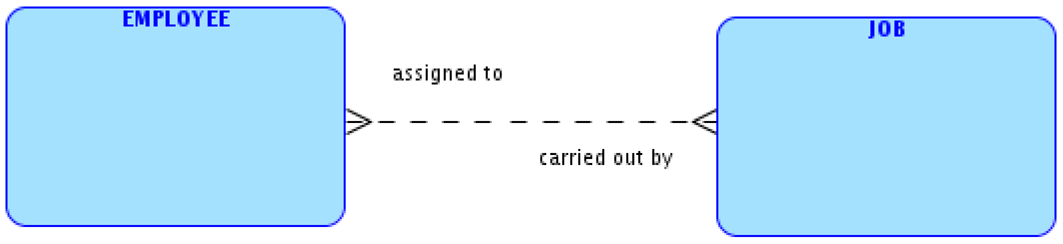


Business rules:

- Each CUSTOMER must be visited by *one and only one* SALES REPRESENTATIVE.
- Each SALES REPRESENTATIVE may be assigned to *one or more* CUSTOMERs.

Lesson 2.10: Many-to-Many Relationships

Many-to-many relationships (M:M) have cardinality of *one or more* in both directions.



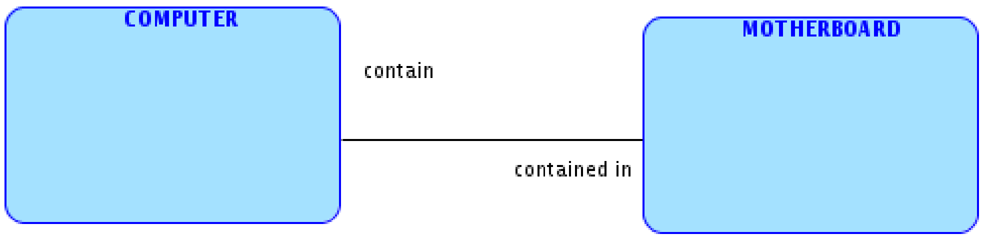
Business rules:

- Each EMPLOYEE may be assigned to *one or more* JOBs.
- Each JOB may be carried out by *one or more* EMPLOYEEs

Lesson 2.11: One-to-One Relationships

One-to-one relationships (1:1) have cardinality of *one and only one* in both directions. The example below is a one-to-one relationship because the cardinality is a line with no crow's feet in either direction.

These types of relations are the least common because they may instead be one entity with attributes contained in that entity.

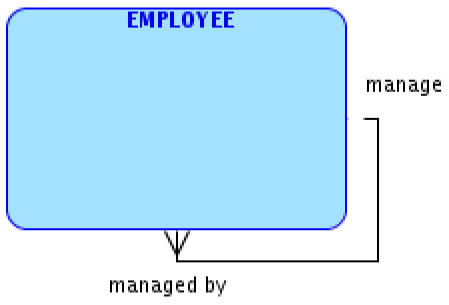


Business rules:

- Each COMPUTER must contain *one and only one* MOTHERBOARD.
- Each MOTHERBOARD must be contained in *one and only one* COMPUTER.

Lesson 2.12: Recursive Relationships

One additional relationship that must be mentioned is a recursive relationship. Recursive relationships are relationships with an entity and itself. In the example below, there is a recursive relationship with the EMPLOYEE entity. There are still two business rules for this type of relationship.



Business rules:

- Each EMPLOYEE may manage *one or more* EMPLOYEEs.
- Each EMPLOYEE must be managed by *one and only one* EMPLOYEE.

Lesson 2.16: Determining the Relationship's Cardinality

The first question to answer is what is the minimum cardinality for each direction of the relationship?

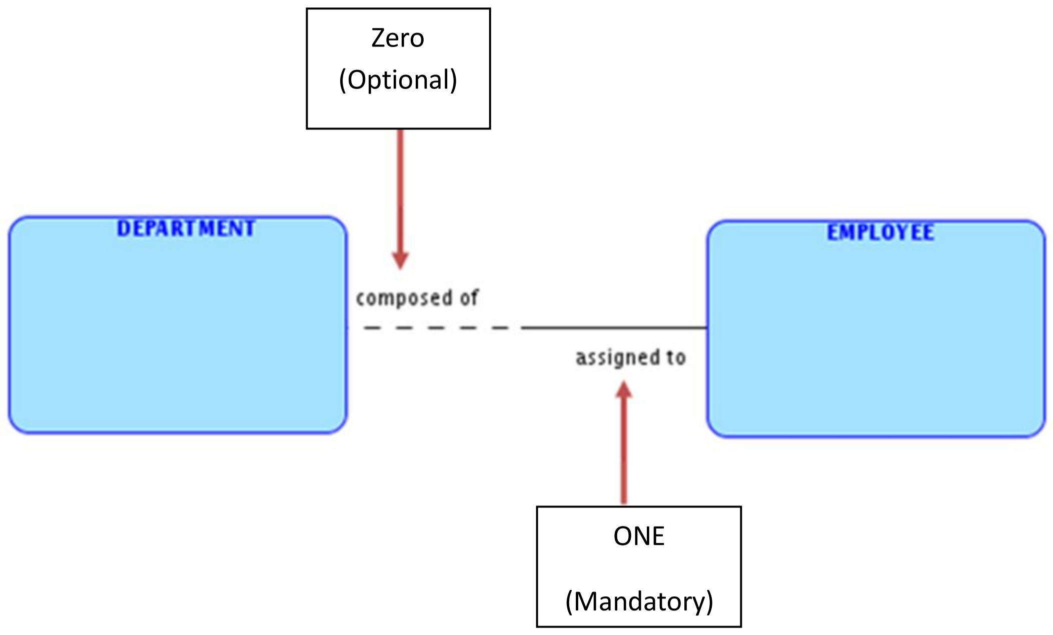
In the following example, answer the following questions:

1. Must an EMPLOYEE be assigned to a DEPARTMENT?
   Always.
2. Is there any situation in which an EMPLOYEE will not be assigned to a DEPARTMENT?
   No, an EMPLOYEE must always be assigned to a DEPARTMENT. (Mandatory)
3. Must a DEPARTMENT be composed of an EMPLOYEE?

   No, a DEPARTMENT does not have to be composed of an EMPLOYEE (Optional)

When the minimum cardinality is optional, the value could be zero. When the minimum cardinality is mandatory, the value must be at least one.

Note that the relationship line in the diagram was intentionally drawn without the maximum cardinality.



The next question to answer is what is the maximum cardinality for each direction of the relationship?

In the following example, answer the following questions:

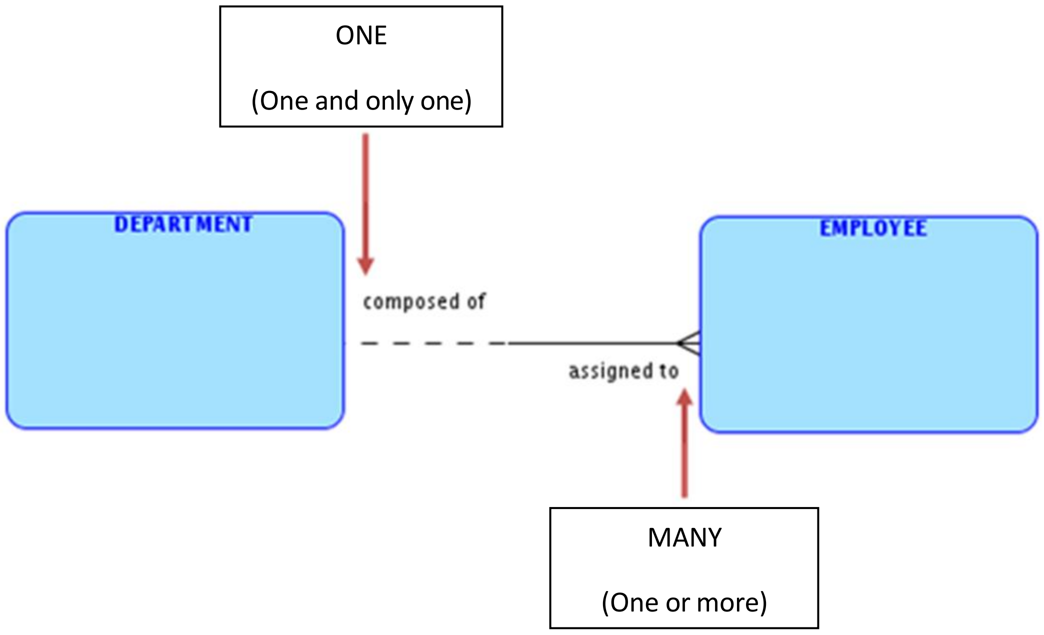
a. Must an EMPLOYEE be assigned to more than one DEPARTMENT?

   No, an EMPLOYEE must always be assigned to one and only one DEPARTMENT. (One)

b. May a DEPARTMENT be composed of more than one EMPLOYEE?

Yes, a DEPARTMENT may be composed of one or more EMPLOYEEs (Many)

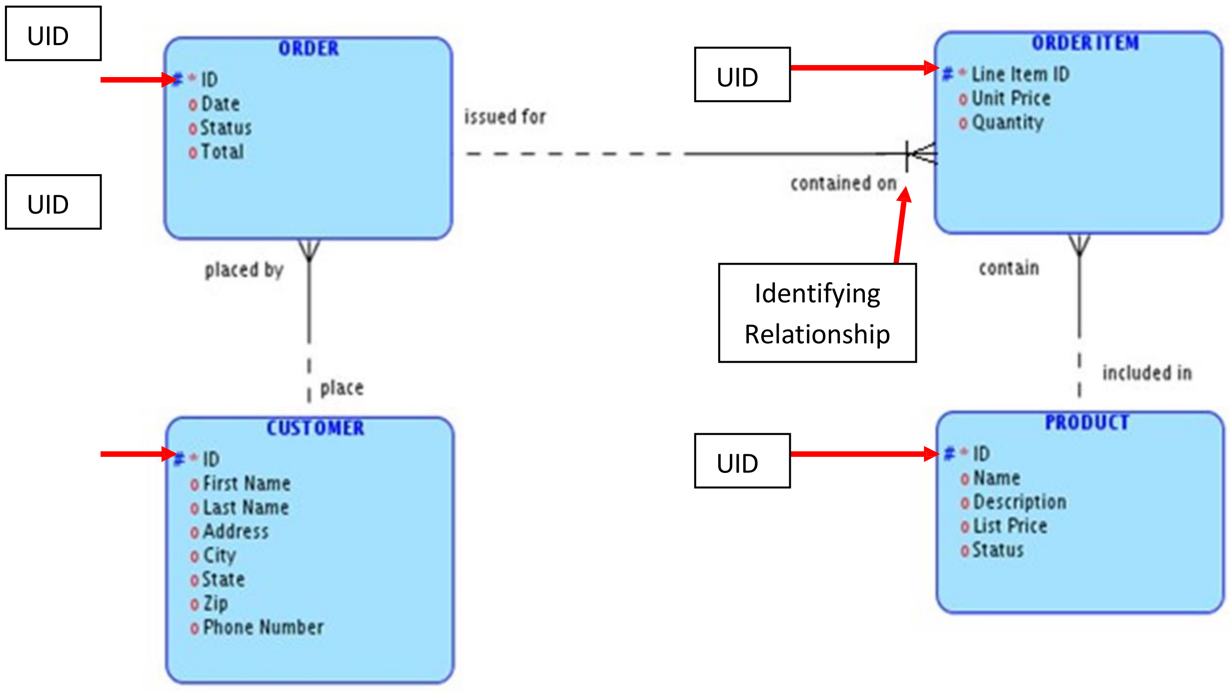
When the maximum cardinality is one, the value can only be one. When the maximum cardinality is many, the value can be one or more.



Lesson 2.18: Assigning keys - Unique Identifiers

A unique identifier (UID) is a special attribute (or group of attributes) that uniquely identifies a particular instance of an entity. A unique identifier attribute is designated with a # symbol. Each component of a unique identifier must be mandatory.

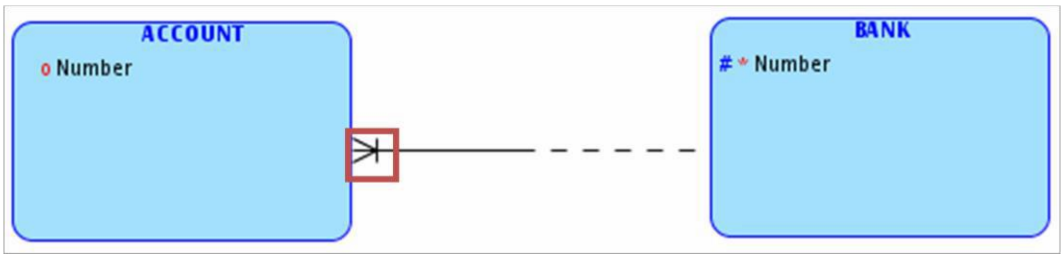
In the following example the unique identifier for ORDER is Order ID, for CUSTOMER it is CUSTOMER ID, and for PRODUCT it is PRODUCT ID. The unique identifier for ORDER ITEM is a composite between Line Item ID and the relationship with the ORDER entity. The vertical line on the relationship indicates that it is part of the unique identifier in the ORDER ITEM entity (also called an *identifying relationship*). This concept is discussed later in this lesson.



Lesson 2.20: Identifying Relationships

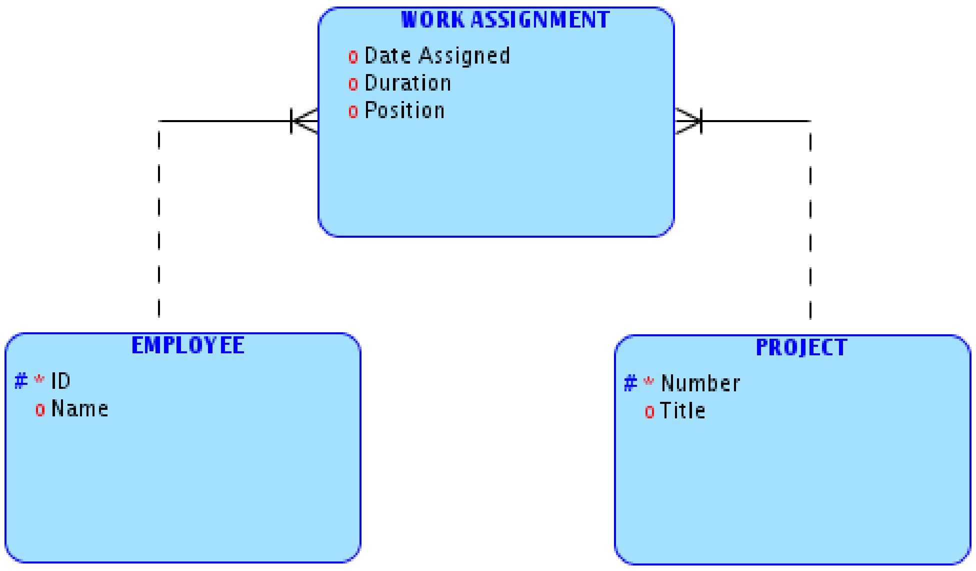
An identifying relationship is created when the unique identifier for an entity includes the relationship with another entity for it to be unique. In the example below, the unique identifier for the ACCOUNT entity is the ACCOUNT number as well as the relationship between BANK and ACCOUNT. The unique identifier requires both the ACCOUNT Number and the relationship between BANK and ACCOUNT. The identifying relationship is depicted with a vertical bar on the relationship line.

Note that a relationship included in a unique identifier must be mandatory and one-and-only-one in the direction that participates in the unique identifier.



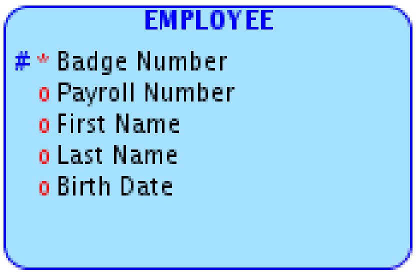
Lesson 2.21: Identifying Relationships with Multiple Entities

An entity may be uniquely identified through multiple relationships. In the example below, an EMPLOYEE and PROJECT are needed to make WORK ASSIGNMENT unique, so both relationships are included in the unique identifier for WORK ASSIGNMENT.



Lesson 2.23: Primary and Secondary Unique Identifiers

An entity can have more than one unique identifier. In the example below, there are two candidate unique identifiers for the EMPLOYEE entity: badge number and payroll number. When this situation occurs, select one candidate unique identifier to be the primary unique identifier, and the others to be secondary unique identifiers.



Lesson 2.25: Normalization

Normalization is the process of organizing the attributes and tables of a relational database to minimize data redundancy. Data redundancy may be where you have an attribute repeated in two or more tables.

Edgar Codd, the inventor of the relational model, first introduced the concept of normalization and what we now know as the First normal form (1NF) in 1970. He went on to define the Second normal form (2NF) and Third normal form(3NF) in 1971. A relational database table is often described as ‘normalized’ if it is in the Third Normal Form.

An example of normalization is that an entity's unique identifier is stored everywhere in the system, but its name is held in only one table. The name can be updated more easily in one row of one table. For example, if we updated a department’s name from ‘HR’ to ‘Human Resources’. The update would be done in one place and immediately the correct name would be displayed throughout the system.

Below are Codd’s rules of normalization – we will go through them in more detail in the following pages.

First Normal Form (1NF)	An entity is in the first normal form if it has *no multivalued attributes*
Second Normal Form (2NF)	An entity is in second normal form if it is already in first normal form and all its attributes are fully dependent on the *concatenated unique identifier*
Third Normal Form (3NF)	An entity is in third normal form if it is already in second normal form and all its attributes are fully dependent on the *unique identifier*. No attribute may “determine” any other attribute (no transitive dependencies)

The tables created during the design will conform to the rules of normalization. Each formal normalization rule from the relational database design has a corresponding data model interpretation. The interpretations that can be used to validate the placement of attributes in an ERD are shown in the table above.

The following are benefits of normalization:

- Normalization ensures that each attribute appropriately belongs to the entity to which it has been assigned and not another entity.
- Normalization eliminates redundant storage of information; this simplifies application logic, because developers do not need to think about multiple copies of the same piece of information.
- Normalization ensures that you have one attribute in one place, with one name, with one value, at any one time.

Lesson 2.26: First Normal Form (1NF)

First normal form validates that each attribute has a single value for each occurrence of the entity. There should be no attribute that has a repeating value.

In the example below, we have some unnormalized data relating to Purchase Orders we issue: Purchase

Order No

Purchase Order Date Supplier
Id

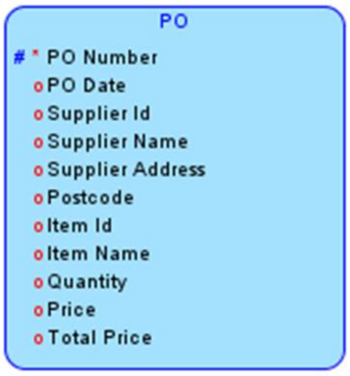
Supplier Name Supplier
Address Postcode

Item Id Item
Name
Quantity
Price

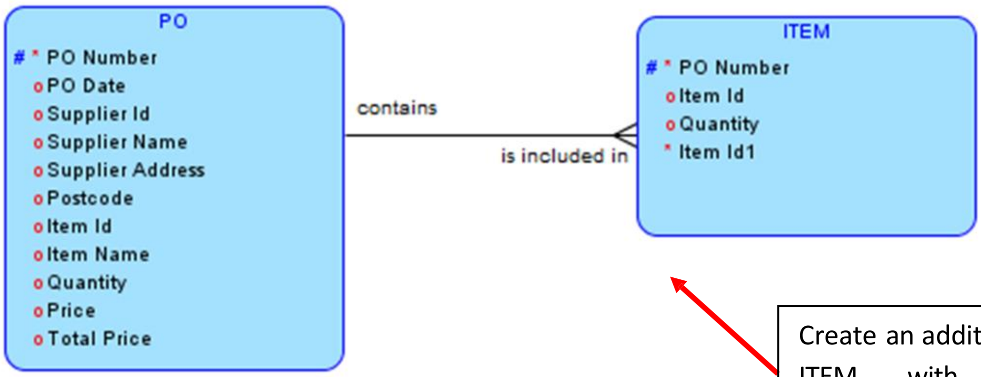
Total Price

The ITEM ID, ITEM NAME, QUANTITY and PRICE attributes could have more than one value for each PURCHASE ORDER NO. That is we have repeating values so the data is not in 1NF. You must perform the following:

- Create another entity and move the attributes that are repeating to the new entity.
- Create an identifying 1:M relationship with the new entity.



The attributes ITEM ID, ITEM NAME, QUANTITY and PRICE have multiple values. Therefore, this is not in 1NF



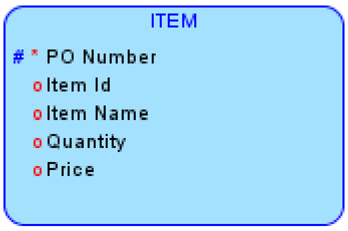
Create an additional entity ITEM with a 1:M relationship to PO

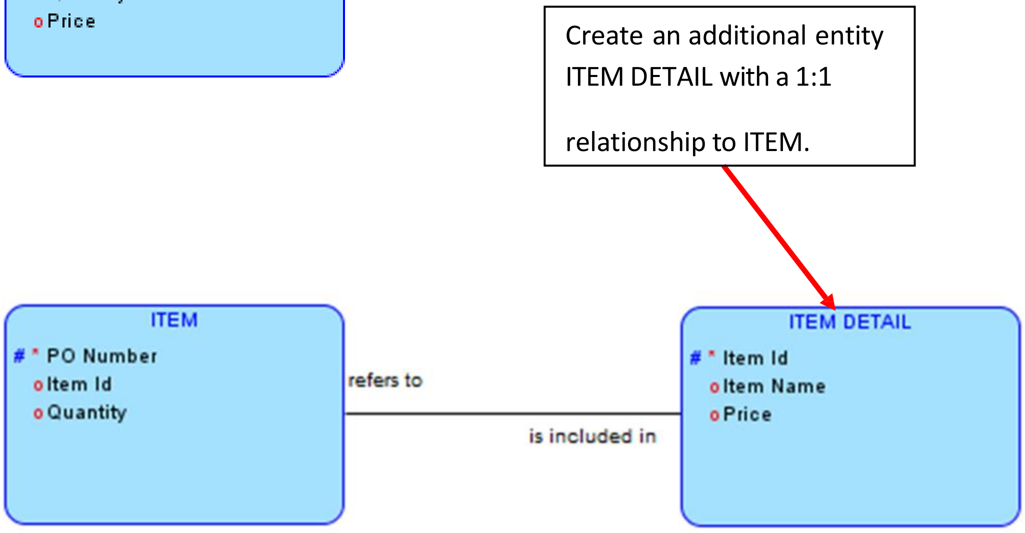
In addition to having no multi-valued attributes (repeating values) repeating INF also decrees that the attributes should only contain atomic values. An atomic value is a value that cannot be divided. ‘Non-atomic’ attributes should be split into more than 1 field because they are hiding detail. For example, SUPPLIER ADDRESS could be divided into STREET, TOWN and COUNTY. The decision on whether to divided the attributes or not is down to your requirements – if it is likely you will want to find all suppliers from a certain town then it would be useful to divide the address as we have done here.

Lesson 2.27: Second Normal Form (2NF)

Second normal form validates that each attribute is dependent on its entity's unique identifier. Each specific instance of the UID must determine a single instance of each attribute. Each attribute is not dependent on only part of its entity's UID.

Keeping to our Purchase Order example, if we look at the ITEM entity then you will see that the Item Name and the Price are dependent on the Item Id but not on the PO No. The Quantity is dependent on the PO No and the Item No. Therefore, the attributes must be moved and a new entity, ITEM DETAIL, is created as shown below:

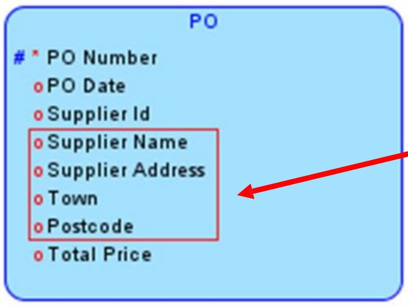




Lesson 2.28: Third Normal Form (3NF)

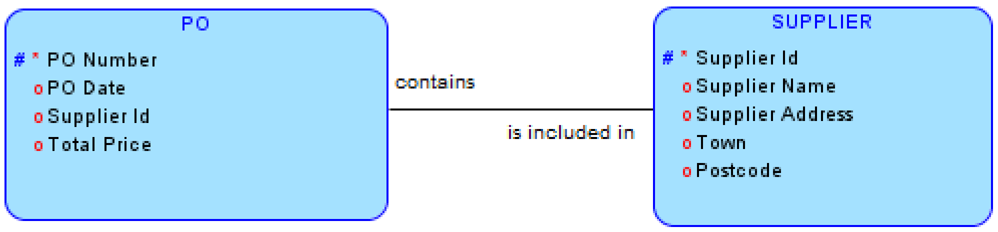
Third normal form validates that each attribute depends only on the UID of its entity (and on nothing else). You need to move any non-UID attribute that is dependent on another non-UID attribute into a new entity.

Following on from the previous example, in the PO entity the Supplier Name, Supplier Address, Town and Postcode attributes are dependent on the Supplier Id attribute. Because these attributes are dependent in part on a non-UID attribute, the attributes, along with the non-UID attribute (Supplier Id), should be moved to a new entity and an identifying relationship should be created.



The Supplier Name, Supplier Address, Town and Postcode attributes are dependent on the Supplier ID (the non-unique identifier).

Therefore, this is not 3NF



Create a new SUPPLIER entity. Move the Supplier Name, Supplier Address, Town and Postcode attributes to the new entity, and then create an identifying relationship.

In the ITEM entity the only non-UID is Quantity and as such it cannot be dependent on any other non-UID attributes as they are not any, so this entity is already in 3NF. In the ITEM DETAIL entity, the Item Name and Price are dependent on the Item Id and not on each other, so this entity is already in 3NF.

Class Exercise: Taking unnormalised data and converting it into 3NF

Here is some unnormalized data – go through the steps of converting it first into 1NF, then into 2NF and finally into 3NF.

Project Code	Project Title	Project Manager	Project Budget	Employee ID	Employee Name	Department ID	Department Name	Hourly Rate
IT010	Pensions	K Jones	123000	100	P Lewis	10	IT	23.43
IT010	Pensions	K Jones	123000	102	S Smith	40	HR	34.21
IT010	Pensions	K Jones	123000	104	H Finn	40	HR	35.21
IT060	Salaries	H Evans	345000	107	G Bale	20	DBA	23.24
IT060	Salaries	H Evans	345000	332	L Jones	30	ACCOUNTS	45.22
IT060	Salaries	H Evans	345000	201	P Smith	30	ACCOUNTS	19.34
IT060	Salaries	H Evans	345000	103	T Woods	30	ACCOUNTS	23.51
IT089	HR	L Adams	100020	111	D Banks	10	IT	24.45
IT089	HR	L Adams	100020	107	G Bale	20	DBA	22.33
IT089	HR	L Adams	100020	110	J Wills	30	ACCOUNTS	33.12

Section 3: Denormalization

In this section you will cover the following topics:

- What Is Denormalization?
- Different Denormalization Techniques
  - Storing Derivable Values
  - Pre-Joining Tables
  - Hard-Coded Values
  - Keeping Details with the Master Table
  - Repeating Current Detail with the Master Table
  - Adding an END_DATE Column
  - Adding a CURRENT_ROW_INDICATOR Column
  - Hierarchy Level Indicator
  - Short Circuit Keys
- Denormalize verses Normalization
-

Lesson 3.1: What Is Denormalization?

Denormalization is the process of adding redundancy to the data to improve performance. You first of all start with a normalized model and then add redundancy to the design. This reduces the integrity of the design and often means you need application code to compensate.

It is worth considering denormalization to systematically add redundancy to the database to help improve performance after other options such as indexing have failed. Denormalization can improve certain types of data access dramatically, but there is no guaranteed success and there is always a cost. Denormalization makes the data model less robust, and it will always slow down data manipulation language (DML). It complicates processing and introduces the possibility of data integrity problems. It always requires additional programming to maintain the denormalized data.

Here are some guidelines for denormalizing:

- Always create a conceptual data model that is completely normalized.
- Consider denormalization as the last option to boost performance. Never presume that denormalization is required.
- Do denormalization during the database design.
- After performance objectives have been met, do not implement any further denormalization.
- Fully document all denormalization, stating what was done to the tables and what application code was added to compensate for the denormalization.

Over the next few pages we will look at some of the different techniques that can be used to denormalize data.

Lesson 3.2: Storing Derivable Values

When a calculation is frequently executed during queries, it can be worthwhile to store the results of that calculation. If the calculation involves detail records, you should store the derived calculation in the master table. For example, adding an ORDER_TOTAL column to an ORDERS table to store the derived value of an order. Make sure to write application code to recalculate the value each time that DML is executed against the detail records. In all situations of storing derivable values, it is important to check that the denormalized values cannot be directly updated. They should always be recalculated by the system.

This denormalizing technique is appropriate when:

- Source values are in multiple records or tables
- Derivable values are frequently needed, and source values are not needed
- Source values are infrequently changed

The advantages of using this technique include the following:

- Source values do not need to be looked up every time the derivable value is required.
- The calculation does not need to be performed during a query or report.

However, there are some disadvantages in using this approach such as:

- DML against the source data will require recalculation or adjustment of the derivable data.
- Data duplication introduces the possibility of data inconsistencies.

Lesson 3.3: Pre-Joining Tables

You can pre-join tables by including a non-key column in a table, when the actual value of the primary key—and consequentially the foreign key—has no business meaning. By including a non-key column that has business meaning, you can avoid joining tables, thus speeding up specific queries.

You must include application code that updates the denormalized column each time the 'master' column value changes in the referenced record.

This denormalizing technique is appropriate when:

- Frequent queries against many tables are required
- Slightly stale data is acceptable

The advantages of using this technique include the following:

- Time-consuming joins can be avoided.
- Updates can be postponed when stale data is acceptable.

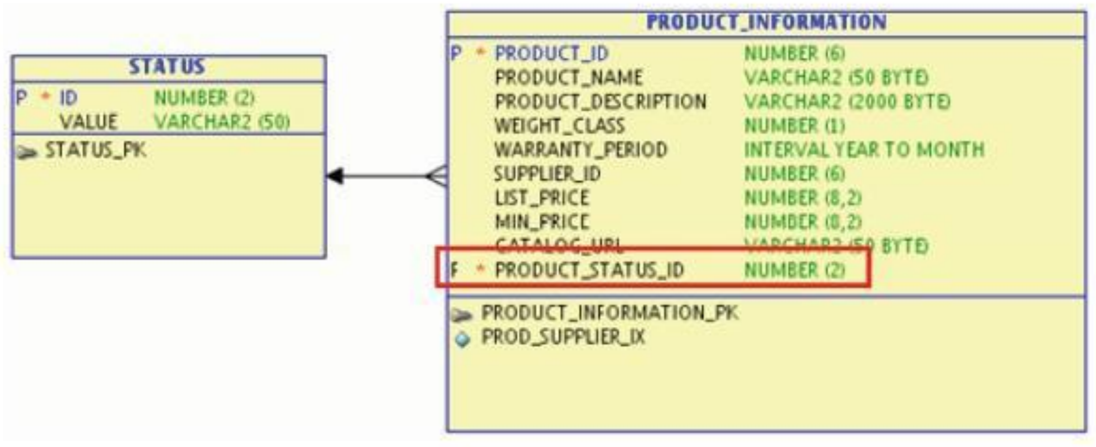
However, there are some disadvantages in using this approach such as:

- Extra DML is needed to update the original non-denormalized column.
- Extra columns and possibly larger indexes require more working and disk space.

Lesson 3.4: Hard-Coded Values

If a reference or lookup table contains records that remain constant, you can consider hard-coding those values into the application code. This means that you will not need to join tables to retrieve the list or reference values. This is a special type of denormalization, when values are kept outside the table in the database.

In the following example, the PRODUCT_STATUS column contains values that are constant. In this case, you can create a check constraint on the PRODUCT_STATUS column that will validate the values. It worth mentioning that a check constraint, though it resides in the database, is still a form of hard coding. Whenever a new value of PRODUCT_STATUS is needed, the constraint must be modified.



This denormalizing technique is appropriate when:

- The set of allowable values can reasonably be considered to be static during the life cycle of the system
- The set of possible values is small (perhaps less than 30)

The advantages of using this technique include the following:

- Implementing a look-up table is not necessary.
- Joins to a look-up table are not necessary.

However, changing look-up values requires recoding and retesting.

Lesson 3.5: Keeping Details with the Master Table

In a situation where the number of detail records per master is a fixed value (or has a fixed maximum) and where usually all detail records are usually queried with the master, you may consider adding the detail columns to the master table. This denormalization technique works best when the number of records in the detail table is small. In this way, you reduce the number of joins during queries.

This denormalizing technique is appropriate when:

- The number of detail records for all masters is fixed and static
- The number of detail records multiplied by the number of columns of the detail is small (perhaps less than 30)

The advantages of using this technique include the following:

- No joins are required.
- It saves space because keys are not propagated.

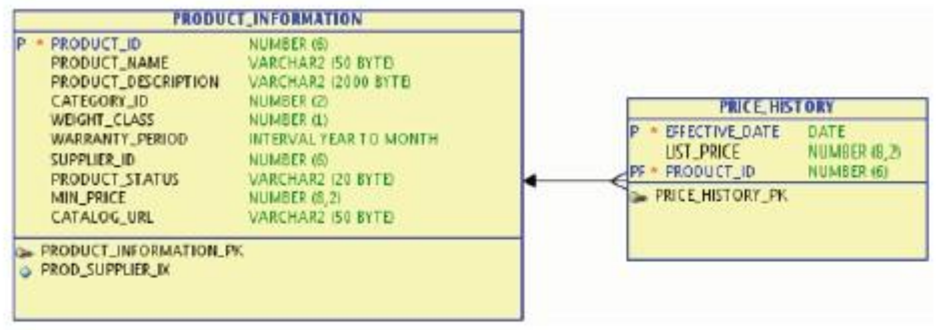
However, the disadvantages of using this approach include the following:

- It increases the complexity of DML and SELECTs across detail values.
- Checks must be repeated for each repeating column.

Lesson 3.6: Repeating Current Detail with the Master Table

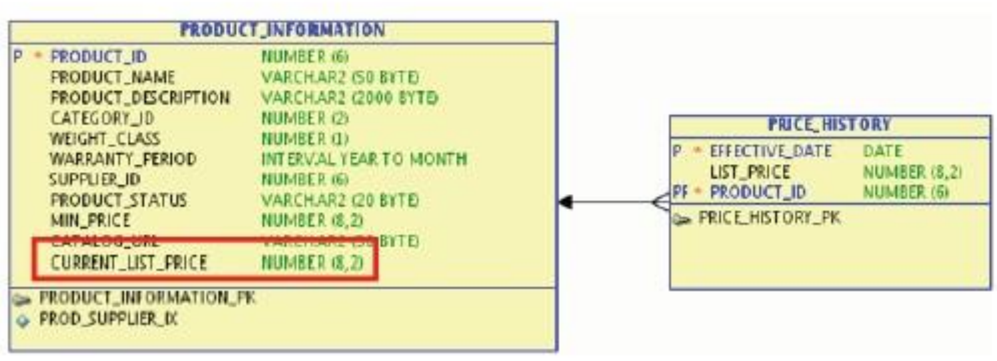
Often when the storage of historical data is necessary, many queries require only the most current record. You can add a new foreign key column to store this single detail with its master. Make sure you add code to change the denormalized column any time a new record is added to the history table. Additional code must be written to maintain the duplicated single detail value for the master record.

In the following example, the price for a product is maintained in the PRICE_HISTORY table by the EFFECTIVE_DATE column.



PRODUCT_INFORMATION
P * PRODUCT_ID NUMBER (6)
PRODUCT_NAME VARCHAR2 (50 BYTE)
PRODUCT_DESCRIPTION VARCHAR2 (2000 BYTE)
CATEGORY_ID NUMBER (2)
WEIGHT_CLASS NUMBER (1)
WARRANTY_PERIOD INTERVAL YEAR TO MONTH
SUPPLIER_ID NUMBER (6)
PRODUCT_STATUS VARCHAR2 (20 BYTE)
MIN_PRICE NUMBER (8,2)
CATALOG_URL VARCHAR2 (50 BYTE)
PRODUCT_INFORMATION_PK
PROD_SUPPLIER_IX
PRICE_HISTORY
P * EFFECTIVE_DATE DATE
LIST_PRICE NUMBER (8,2)
PF * PRODUCT_ID NUMBER (6)
PRICE_HISTORY_PK

The CURRENT_LIST_PRICE column is added to the PRODUCT_INFORMATION table to store the current detail information.



PRODUCT_INFORMATION
P * PRODUCT_ID NUMBER (6)
PRODUCT_NAME VARCHAR2 (50 BYTE)
PRODUCT_DESCRIPTION VARCHAR2 (2000 BYTE)
CATEGORY_ID NUMBER (2)
WEIGHT_CLASS NUMBER (1)
WARRANTY_PERIOD INTERVAL YEAR TO MONTH
SUPPLIER_ID NUMBER (6)
PRODUCT_STATUS VARCHAR2 (20 BYTE)
MIN_PRICE NUMBER (8,2)
CATALOG_URL VARCHAR2 (50 BYTE)
CURRENT_LIST_PRICE NUMBER (8,2)
PRODUCT_INFORMATION_PK
PROD_SUPPLIER_IX
PRICE_HISTORY
P * EFFECTIVE_DATE DATE
LIST_PRICE NUMBER (8,2)
PF * PRODUCT_ID NUMBER (6)
PRICE_HISTORY_PK

This denormalizing technique is appropriate when:

- Detail records per master have a property such that one record can be considered ‘current’ and others can be considered ‘historical’
- Queries frequently need this specific single detail and only occasionally need the other details
- The master often has only one single detail record

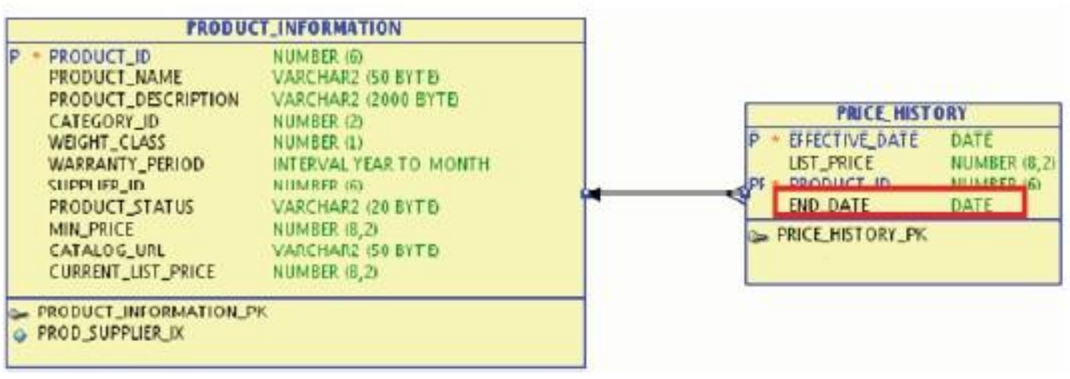
The advantage of using this technique means no join is required for queries that need only the specific single detail. Whereas the disadvantage is the possibility of data inconsistencies due to the detail value being repeated

Lesson 3.7: Adding an END_DATE Column

One of the most common denormalization techniques is to store the end date for periods that are consecutive. As a result, the end date for a period can be derived from the start date of the previous period. Using this technique, you can find a detail record for a particular date without using a complex query.

In the following example, the END_DATE column is added to the PRICE_HISTORY table.

As a result, if you want to check the price between a certain set of dates, you can create a query using the BETWEEN clause.



This denormalizing technique is appropriate when queries are needed from tables with long lists or records that are historical, and you are interested in the most current record

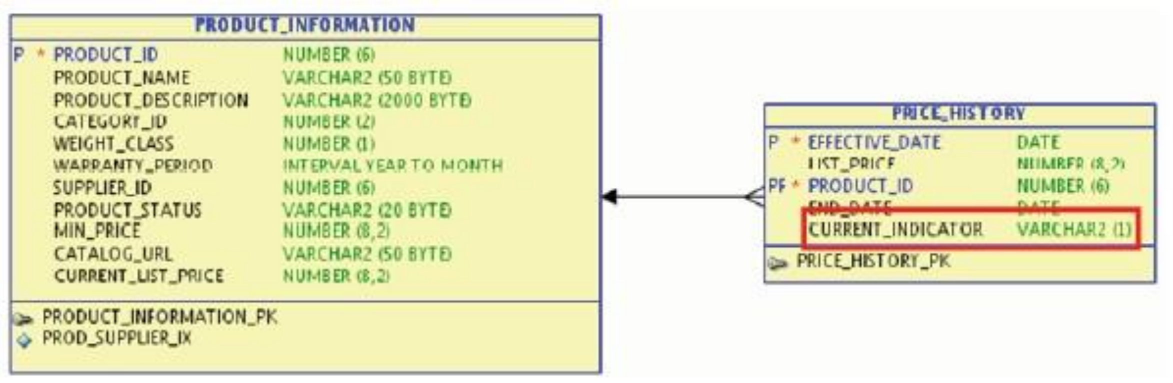
The main advantage of using this technique is that you can use the BETWEEN operator for date- selection queries instead of a potentially time-consuming synchronizing subquery.

On the downside extra code is needed to populate the END_DATE column with the value found in the previous start date record.

Lesson 3.8: Adding a CURRENT_ROW_INDICATOR Column

This denormalization technique enables you to quickly find the most current detail record by adding a new indicator column to the details table to represent the currently active row. You would need to add code to update the indicator column each time that you insert a new record.

In the following example, the CURRENT_INDICATOR column is added to the PRICE_HISTORY table to enable you to query the currently active row more easily.


PRODUCT_INFORMATION
P * PRODUCT_ID NUMBER (6)
PRODUCT_NAME VARCHAR2 (50 BYTE)
PRODUCT_DESCRIPTION VARCHAR2 (2000 BYTE)
CATEGORY_ID NUMBER (2)
WEIGHT_CLASS NUMBER (1)
WARRANTY_PERIOD INTERVAL YEAR TO MONTH
SUPPLIER_ID NUMBER (6)
PRODUCT_STATUS VARCHAR2 (20 BYTE)
MIN_PRICE NUMBER (8,2)
CATALOG_URL VARCHAR2 (50 BYTE)
CURRENT_LIST_PRICE NUMBER (8,2)
PRODUCT_INFORMATION_PK
PROD_SUPPLIER_IX
PRICE_HISTORY
P * EFFECTIVE_DATE DATE
LIST_PRICE NUMBER (8,2)
PF * PRODUCT_ID NUMBER (6)
END_DATE DATE
CURRENT_INDICATOR VARCHAR2 (1)
PRICE_HISTORY_PK

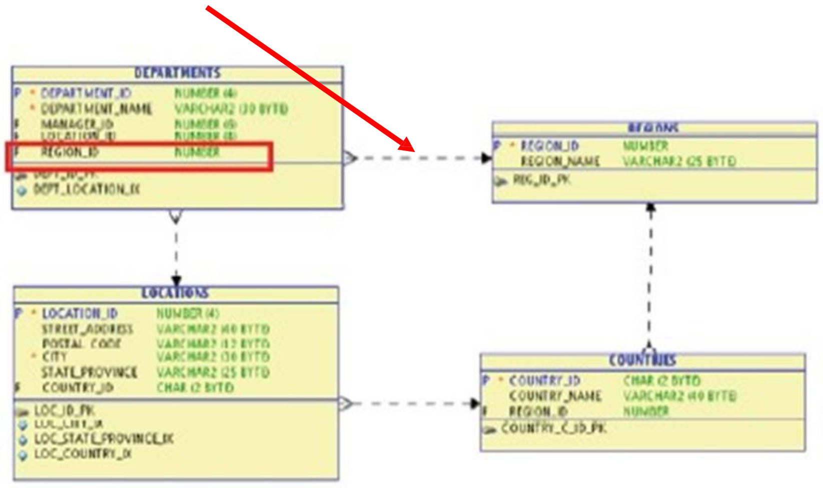
This denormalizing technique is appropriate when the situation requires retrieving the most current record from a long list.

The main advantages of using this technique means that queries and subqueries are less complicated. However, extra column and application code is needed to maintain the column and the concept of "current" makes it impossible to make data adjustments ahead of time.

Lesson 3.10: Short Circuit Keys

For database designs that contain three (or more) levels of master detail where there is a need to query only the lowest and highest-level records, consider creating a short circuit key. These new foreign key definitions directly link the lowest-level detail records to higher-level grandparent records. The result can produce fewer table joins when queries execute.

In the following example, you frequently want to know which region a particular department is in. As a result, you can create a foreign key between the DEPARTMENTS and REGIONS table.



Lesson 3.11: Denormalize verses Normalization

There are advantages and disadvantages with both of these techniques.

Normalized tables are usually smaller and take up less storage as the data is divided vertically among many tables. This allows them to perform better as they are small enough to get fit into the buffer. Their updates and inserts are very fast because the data to be updated is located at a single place and there are no duplicates. And selects are fast in cases where data has to be fetched from a single table, as normally normalized tables are small enough to fit into the buffer. And finally, as the data is not duplicated in normalization there is less need for GROUP BY or DISTINCT queries.

The main cause of concern with fully normalized tables is that normalized data means joins between tables. Even with efficient indexing strategies querying data suffers.

Denormalized databases tend to do better with querying as the data is present in the same table so there is no need for any joins, hence the selects tend to be faster than with normalized tables.

A single table with all the required data allows much more efficient index usage. If the columns are indexed properly, then results can be filtered and sorted by utilizing the same index. While in the case of a normalized table, since the data would be spread out in different tables, this would not be possible.

Although selects can be very fast on denormalized tables, because the data is duplicated, the updates and inserts become complex and costly.

So, in summary neither one of the above approaches can be entirely ignored as a real world application is going to have both read and write loads. Therefore, the correct way would be to use both the normalized and denormalized approaches depending on the situation.

Section 4: Data warehousing concepts

In this section you will cover the following topics:

- What Is a Data Warehouse?
- What are the Data Warehouse components?
- Star and Snowflake schemas
- Facts, Dimensions and Measures
- Slowly Changing Dimensions

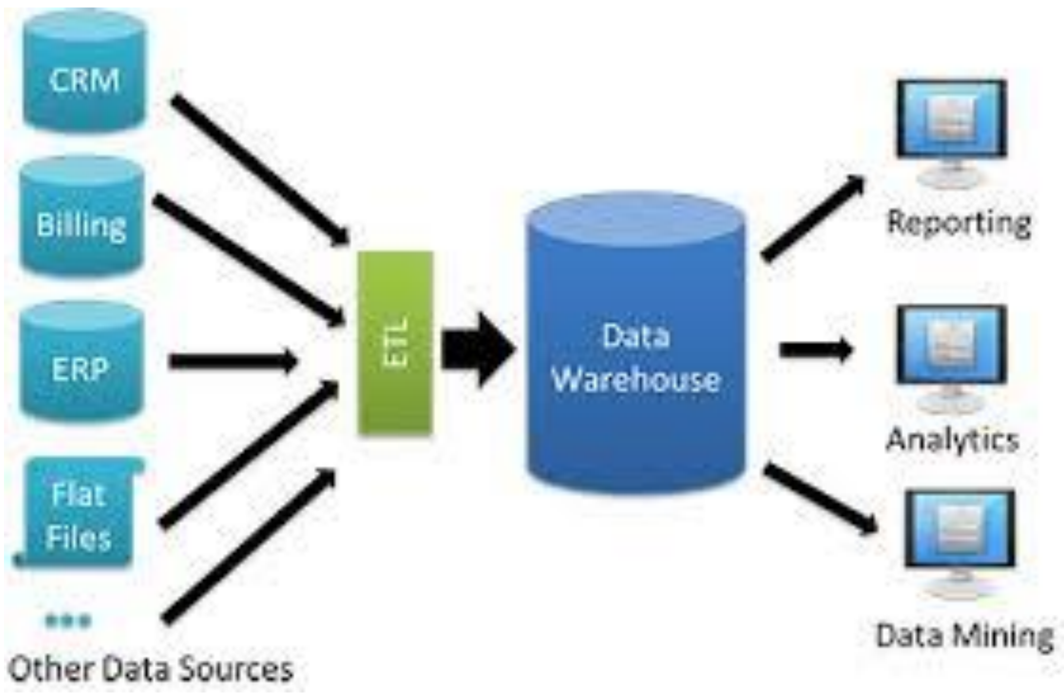
Lesson 4.1: What Is a Data Warehouse?

There are many definitions for the term "data warehouse" and disagreements over specific implementation details. It is generally agreed, however, that a data warehouse is a centralized store of business data that can be used for reporting and analysis to inform key decisions.

Typically, a data warehouse:

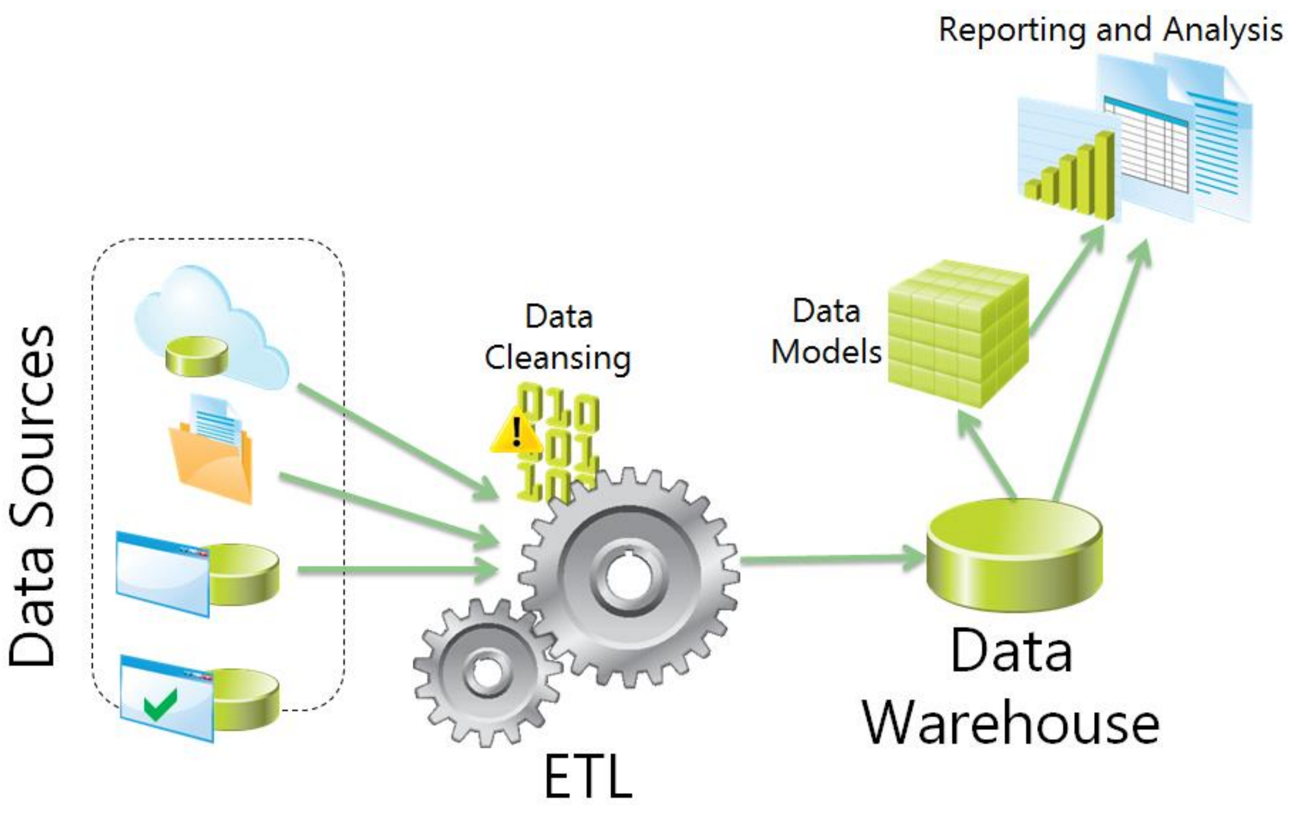
- Contains a large volume of data that relates to historical business transactions.
- Is optimized for read operations that support querying the data. This is in contrast to a typical Online Transaction Processing (OLTP) database that is designed to support data insert, as well as update and delete operations.
- Is loaded with new or updated data at regular intervals.
- Provides the basis for enterprise BI applications.

As shown below a data warehouse is constructed by integrating data from multiple heterogeneous sources that support analytical reporting, structured and/or ad hoc queries, and decision making. Data warehousing involves data cleaning, data integration, and data consolidations.



Lesson 4.2: Components of a Data Warehouse

A data warehousing solution usually consists of the following elements:



Data sources

Sources of business data for the data warehouse, often including OLTP application databases, flat files, XML files and data exported from proprietary systems, such as accounting applications.

An Extract, Transform, and Load (ETL) process

A process for accessing data in the data sources, modifying it to conform to the data model for the data warehouse, and loading it into the data warehouse.

Data staging areas

Intermediary locations where the data to be transferred to the data warehouse is stored. It is prepared here for import and loading into the data warehouse.

A data warehouse

A relational database designed to provide high-performance querying of historical business data for reporting and analysis.

Many data warehousing solutions also include:

Data cleansing and deduplication

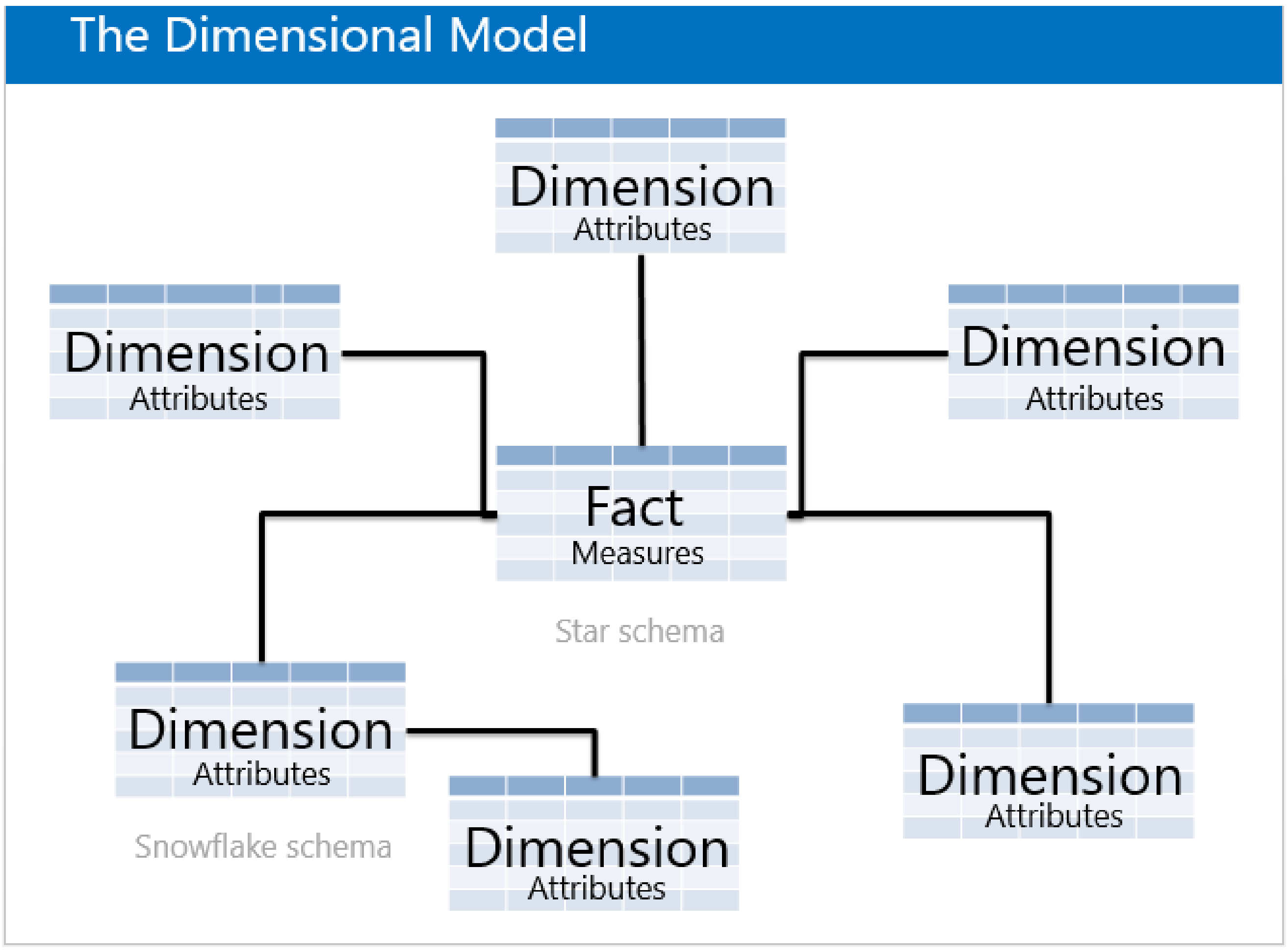
A solution for resolving data quality issues before it is loaded into the data warehouse.

Master Data Management (MDM)

A solution that provides an authoritative data definition for business entities used by multiple systems across the organization.

Lesson 4.3: The Dimensional Model

Although data warehouses can be implemented as normalized, relational database schemas, most designs are based on the dimensional model advocated by Ralph Kimball. In the dimensional model, the numeric business measures that are analyzed and reported are stored in fact tables, which are related to multiple dimension tables, in which the attributes by which the measures can be aggregated are stored. For example, a fact table might store sales order measures, such as revenue and profit, and be related to dimension tables representing business entities such as product and customer. These relationships make it possible to aggregate the sales order measures by the attributes of a product for example, to find the total profit for a particular product model.



Dimension

A dimension is a structure that categorizes facts and measures in order to enable users to answer business questions. Commonly used dimensions are people, products, place and time.

Fact

A Fact table consists of the measurements, metrics or facts of a business process. It is located at the center of a star schema or a snowflake schema surrounded by dimension tables. Where multiple fact tables are used, these are arranged as a fact constellation schema. A fact table typically has two types of columns: those that contain facts and those that are a foreign key to dimension tables. The primary key of a fact table is usually a composite key that is made up of all of its foreign keys.

Star Schema

Ideally, a dimensional model can be implemented in a database as a “star” schema, in which each fact table is directly related to its relevant dimension tables. A common approach is to design a star schema in which numerical measures are stored in fact tables that have foreign keys to multiple dimension tables containing the business entities by which the measures can be aggregated. Before designing your data warehouse, you must know which dimensions your business users employ when aggregating data, which measures need to be analyzed and at what granularity, and which facts include those measures. You must also carefully plan the keys that will be used to link facts to dimensions, and consider whether your data warehouse must support the use of dimensions that change over time. For example, handling dimension records for customers who change their address.

Snowflake Schema

However, in some cases, one or more dimensions may be normalized into a collection of related tables to form a “snowflake” schema. Generally, you should avoid creating snowflake dimensions because, in a typical data warehouse workload, the performance benefits of a single join between fact and dimension tables outweigh the data redundancy reduction benefits of normalizing the dimension data.

Note

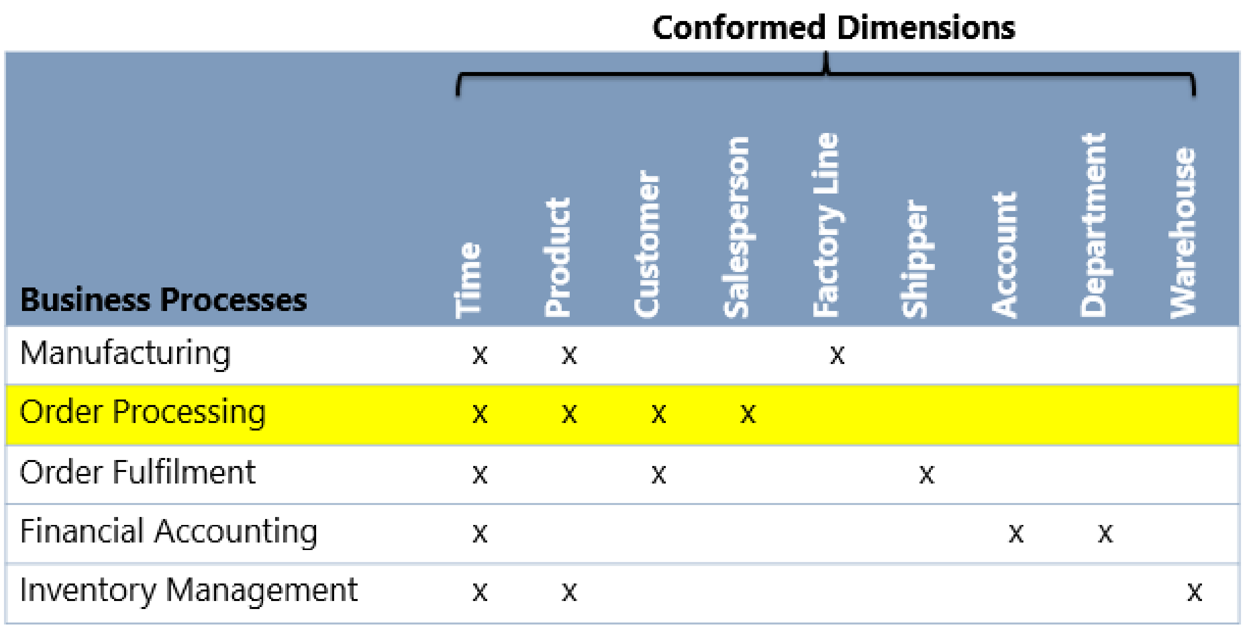
You must also consider the physical implementation of the database, because this will affect the performance and manageability of the data warehouse. It is common to use table partitioning to distribute large fact data across multiple physical disk. You should also consider the appropriate indexing strategy for your data, and whether to use data compression when storing it.

Lesson 4.4: The Data Warehouse Design Process

Although every project has its unique considerations, there is a commonly-used process for designing a dimensional data warehouse that many BI professionals have found effective. The method is largely based on the data warehouse design patterns identified and documented by Ralph Kimball and the Kimball Group, though some BI professionals may adopt a varied approach to each task.

1. Determine analytical and reporting requirements
2. Identify the business processes that generate the required data
3. Examine the source data for those business processes
4. Conform dimensions across business processes
5. Prioritize processes and create a dimensional model for each
6. Document and refine the models to determine the database logical schema
7. Design the physical data structures for the database

After you identify the business processes and conformed dimensions, you can document them in a matrix



	Conformed Dimensions								
Business Processes	Time	Product	Customer	Salesperson	Factory Line	Shipper	Account	Department	Warehouse
Manufacturing	x	x			x				
Order Processing	x	x	x	x					
Order Fulfilment	x		x			x			
Financial Accounting	x						x	x	
Inventory Management	x	x							x

- **Grain**: 1 row per order item
- **Dimensions**: Time (order date and ship date), Product, Customer, Salesperson
- **Facts**: Item Quantity, Unit Cost, Total Cost, Unit Price, Sales Amount, Shipping Cost

You can then use the matrix to select each business process based on priority, and design a dimensional model by performing the following steps:

Identify the grain

The grain of a dimensional model is the lowest level of detail at which you can aggregate the measures. It is important to choose the level of grain that will support the most granular of reporting and analytical requirements, so typically the lowest level possible from the source data is the best option. For example, an order processing system might record data at two levels. There may be order-level data, such as the order date, salesperson, customer, and shipping cost, as well as line item-level data like the products included in the order and their individual quantities, unit costs, and selling prices. To support the most granular analysis and reporting, the grain should be declared at the line item level, so the fact table will contain one row per line item.

Select the required dimensions

Next, determine which of the dimensions related to the business process should be included in the model. The selection of dimensions depends on the reporting and analytical requirements, specifically on the business entities by which users need to aggregate the measures. Almost all dimensional models include a time-based dimension, and the others generally become obvious as you review the requirements.

Identify the facts.

Finally, identify the facts that you want to include as measures. These are numeric values that can be expressed at the level of the grain chosen earlier and aggregated across the selected dimensions. Some facts will be taken directly from source systems, and others might be derived from the base facts. For example, you might choose **Quantity** and **Unit Price** facts from an order processing source system, and then calculate a total **Sales Amount**.

Lesson 4.5: Designing Dimension Tables

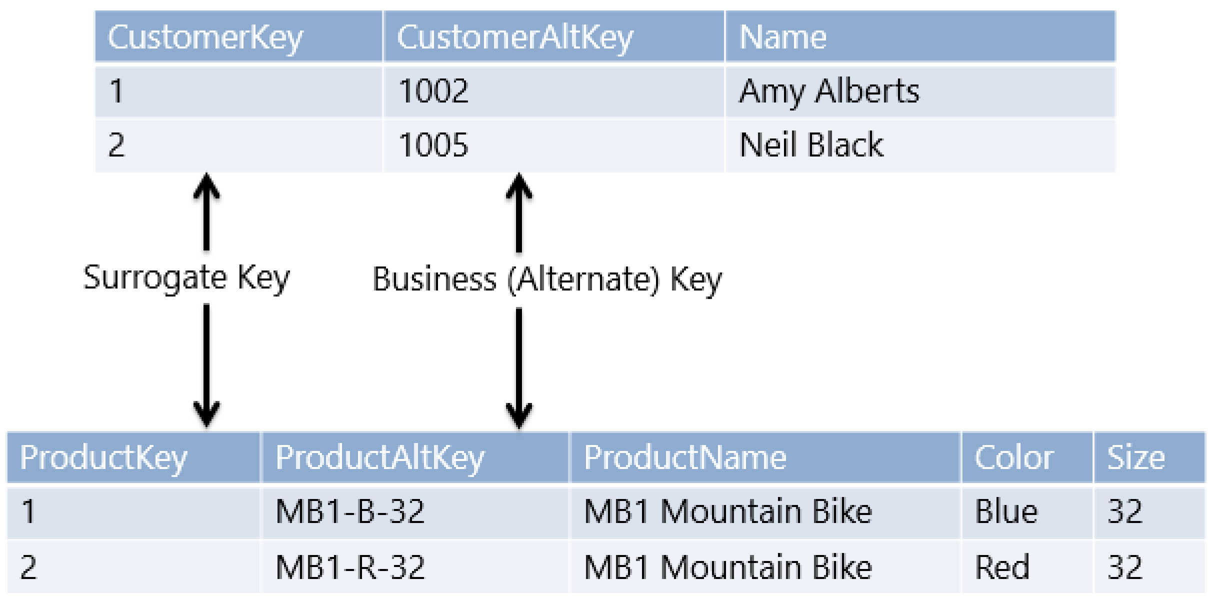
After designing the dimensional models for the data warehouse, you can translate the design into a logical schema for the database. However, before you design dimension tables, it is important to consider some common design patterns and apply them to your table specifications.

Dimension Keys

Each row in a dimension table represents an instance of a business entity by which the measures in the fact table can be aggregated. Like other tables in a database, a key column uniquely identifies each row in the dimension table. In many scenarios, the dimension data is obtained from a source system in which a key is already assigned (sometimes referred to as the "business" key). When designing a data warehouse, however, it is standard practice to define a new "surrogate" key that uses an integer value to identify each row.

A surrogate key is recommended for the following reasons:

- The data warehouse might use dimension data from multiple source systems, so it is possible that business keys are not unique.
- Some source systems use non-numeric keys, such as a globally unique identifier (GUID), or natural keys, such as an email address, to uniquely identify data entities. Integer keys are smaller and more efficient to use in joins from fact tables.
- Each row in a dimension table represents a specific version of a business entity instance. If the dimension table supports "Type 2" slowly-changing dimensions, the table might need to contain multiple rows that represent different versions of the same entity. These rows will have the same business key and won't be uniquely identifiable without a surrogate key.



CustomerKey	CustomerAltKey	Name
1	1002	Amy Alberts
2	1005	Neil Black

ProductKey	ProductAltKey	ProductName	Color	Size
1	MB1-B-32	MB1 Mountain Bike	Blue	32
2	MB1-R-32	MB1 Mountain Bike	Red	32

Typically, the business key is retained in the dimension table as an “alternate” key. Business keys that are based on natural keys can be familiar to users analyzing the data. For example, a **ProductCode** business key that users will recognize might be used as an alternate key in the **Product** dimension table. However, the main reason for retaining a business key is to make it easier to manage slowly-changing dimensions when loading new data into the dimension table. The ETL process can use the alternate key as a lookup column to determine whether an instance of a business entity already exists in the dimension table.

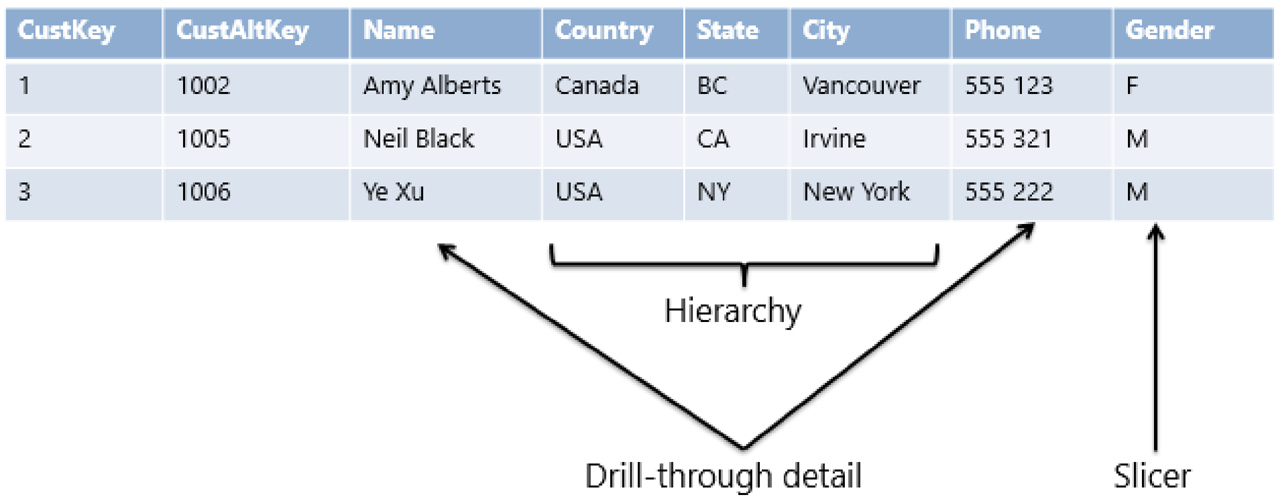
Dimension Attributes and Hierarchies

In addition to the surrogate and alternate key columns, a dimension table includes a column for each attribute of the business entity that is needed to support reporting and analytical requirements. When designing a dimension table, you need to identify and include attributes that will be used in reports and analysis. Typically, dimension attributes are used in one of the following three ways:

Hierarchies. Multiple attributes can be combined to form hierarchies that enable users to drill down into deeper levels of detail. For example, the **Customer** table in the slide includes **Country**, **State**, and **City** attributes that can be combined to form a natural geographical hierarchy. Business users can view aggregated fact data at each level, for example, to see sales order revenue by country. They can then access a specific country to see a breakdown by state, before drilling further into a specific state to see sales revenue by city.

Slicers. Attributes do not need to form hierarchies to be useful in analysis and reporting. Business users can group or filter data based on single-level hierarchies to create analytical sub-groupings of data. For example, the **Gender** attribute in the **Customer** table can be used to compare sales revenue for male and female customers.

Drill-through detail. Some attributes have little value as slicers or members of a hierarchy. For example, it may be unlikely that a business user will need to analyze sales revenue by customer phone number. However, it can be useful to include entity-specific attributes to facilitate drill-through functionality in reports or analytical applications. For example, in a sales order report that enables users to drill down to the individual order level, users might want to double-click an order and drill through to see the name and phone number of the customer who placed it.



CustKey	CustAltKey	Name	Country	State	City	Phone	Gender
1	1002	Amy Alberts	Canada	BC	Vancouver	555 123	F
2	1005	Neil Black	USA	CA	Irvine	555 321	M
3	1006	Ye Xu	USA	NY	New York	555 222	M

Note

As a general rule, try to design your data warehouse in a way that eliminates, or at least minimizes, NULL values, particularly in fact table key columns that reference dimension tables. NULL values make it easy to accidentally eliminate rows from reports and produce misleading totals.

Lesson 4.6: Designing Slowly Changing Dimensions

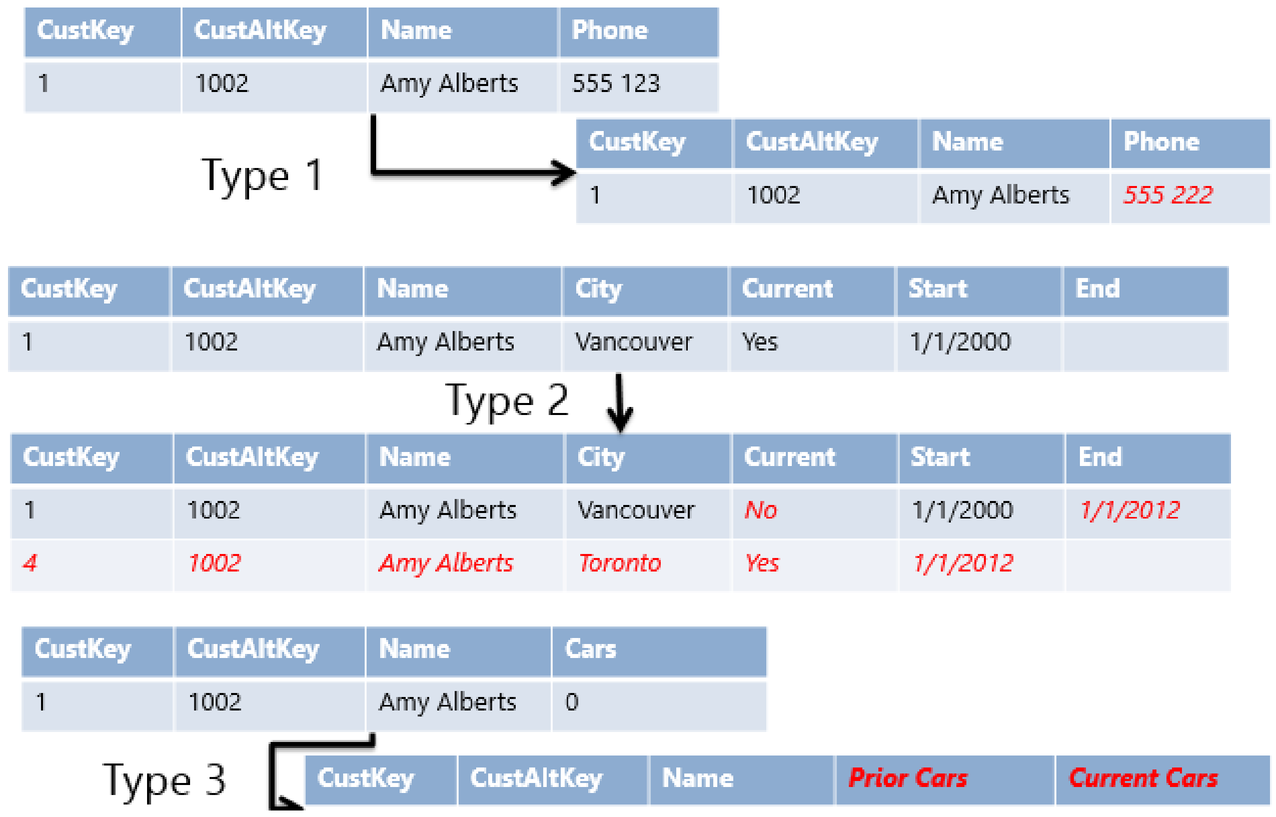
Slowly changing dimensions (SCDs) are a significant consideration in the design of dimension tables. You should try to identify requirements for maintaining historic dimension attribute values as early as possible in the design process.

There are three common techniques used to handle attribute value changes in SCDs:

Type 1. These changes are the simplest type of SCD to implement. Attribute values are updated directly in the existing dimension table row and no history is maintained. This makes Type 1 changes suitable for attributes that are used to provide drill-through details, but unsuitable for analytical slicers or hierarchy members where historic comparisons must reflect the attribute values as they were at the time of the fact event.

Type 2. These changes involve the creation of a fresh version of the dimension entity in the form of a new row. Typically, a bit column in the dimension table is used as a flag to indicate which version of the dimension row is the current one. Additionally, datetime columns are often used to indicate the start and end of the period for which a version of the row was (or is) current. Maintaining start and end dates makes it easier to assign the appropriate foreign key value to fact rows as they are loaded so they are related to the version of the dimension entity that was current at the time the fact occurred.

Type 3. These changes are rarely used. In a Type 3 change, the previous value (or sometimes a complete history of previous values) is maintained in the dimension table row. This requires modifying the dimension table schema to accommodate new values for each tracked attribute, and can result in a complex dimension table that is difficult to manage.



CustKey	CustAltKey	Name	Phone
1	1002	Amy Alberts	555 123

Type 1

CustKey	CustAltKey	Name	Phone
1	1002	Amy Alberts	*555 222*

CustKey	CustAltKey	Name	City	Current	Start	End
1	1002	Amy Alberts	Vancouver	Yes	1/1/2000	

Type 2

CustKey	CustAltKey	Name	City	Current	Start	End
1	1002	Amy Alberts	Vancouver	*No*	1/1/2000	*1/1/2012*
4	*1002*	*Amy Alberts*	*Toronto*	*Yes*	*1/1/2012*	

CustKey	CustAltKey	Name	Cars
1	1002	Amy Alberts	0

Type 3

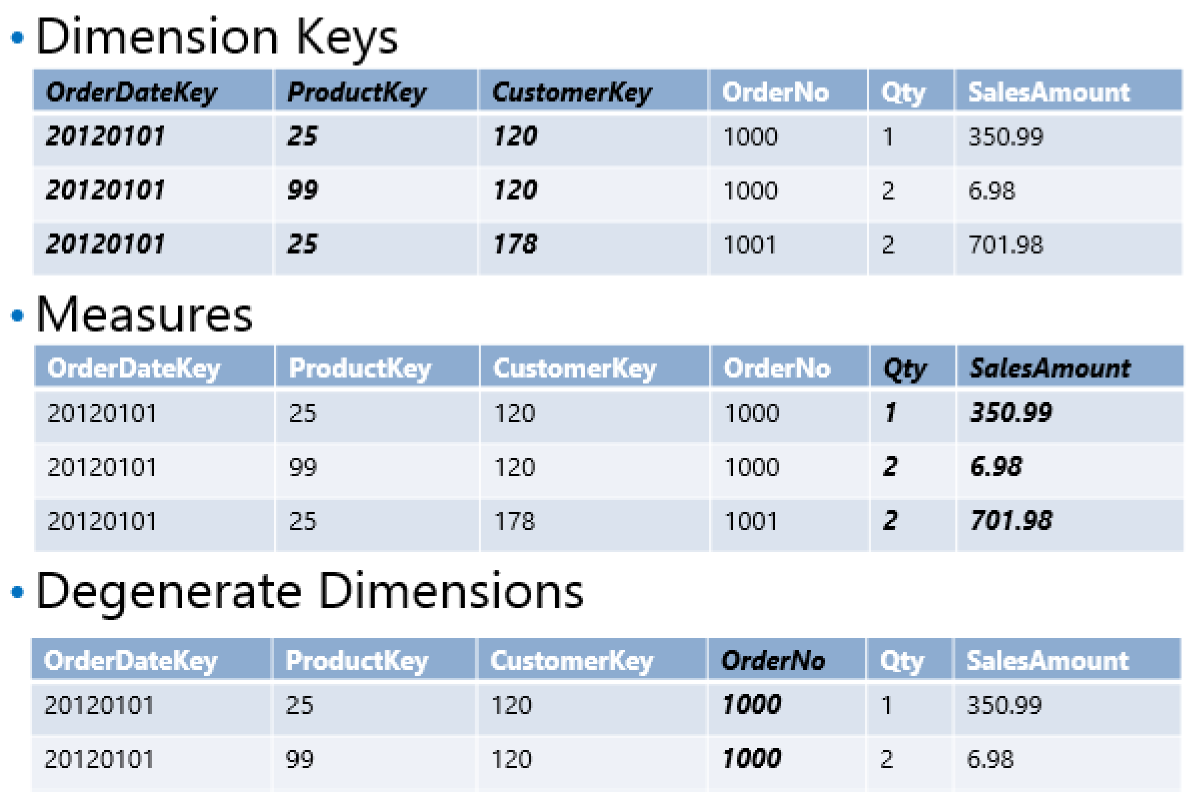
CustKey	CustAltKey	Name	***Prior Cars***	***Current Cars***

Lesson 4.7: Designing Fact Tables

Fact tables contain the numeric measures that can be aggregated across the dimensions in your dimensional model and can become extremely large.

Fact Table Columns

A fact table usually consists of the following kinds of columns:



- Dimension Keys

OrderDateKey	***ProductKey***	***CustomerKey***	OrderNo	Qty	SalesAmount
20120101	***25***	***120***	1000	1	350.99
20120101	***99***	***120***	1000	2	6.98
20120101	***25***	***178***	1001	2	701.98

- Measures

OrderDateKey	ProductKey	CustomerKey	OrderNo	***Qty***	***SalesAmount***
20120101	25	120	1000	***1***	***350.99***
20120101	99	120	1000	***2***	***6.98***
20120101	25	178	1001	***2***	***701.98***

- Degenerate Dimensions

OrderDateKey	ProductKey	CustomerKey	***OrderNo***	Qty	SalesAmount
20120101	25	120	***1000***	1	350.99
20120101	99	120	***1000***	2	6.98

Dimension keys. Fact tables reference dimension tables by storing the surrogate key for each related dimension

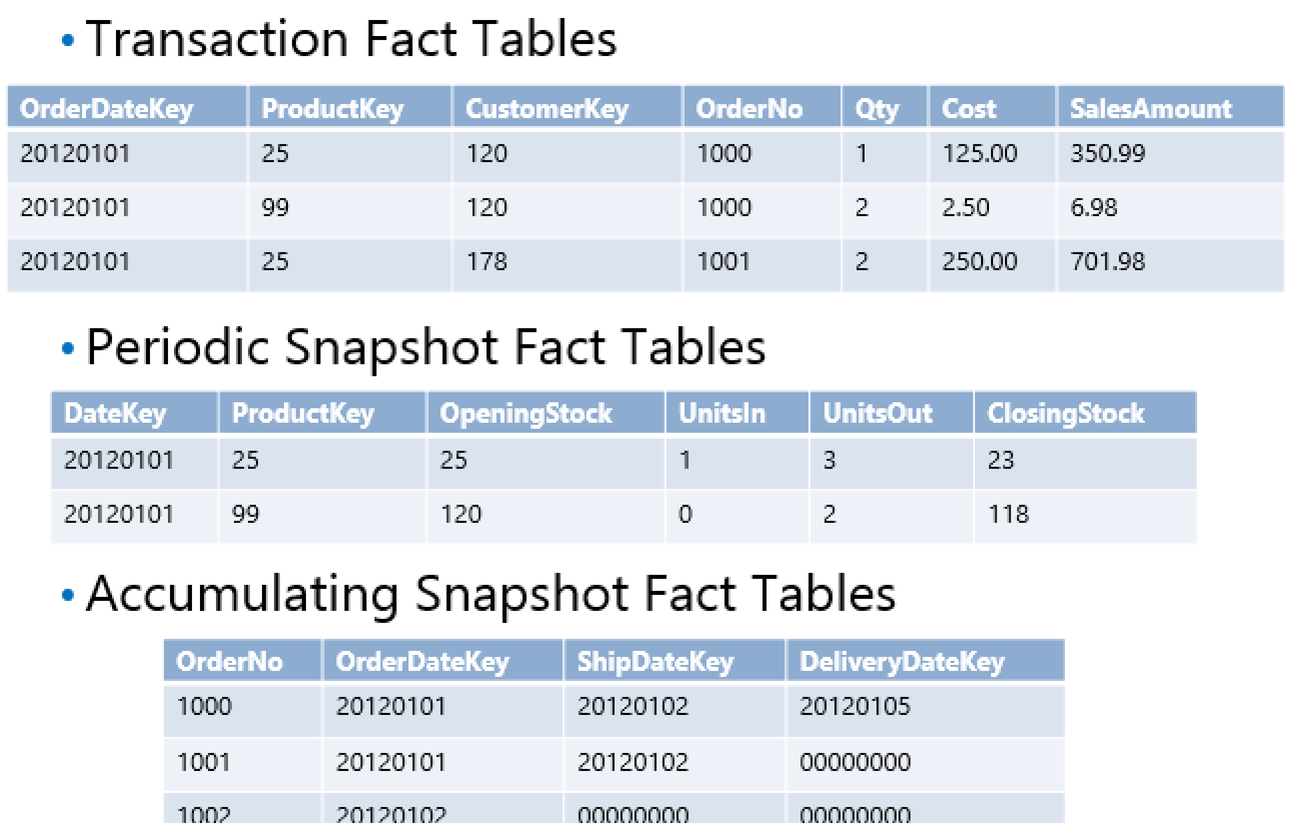
Measures. In most cases, a fact table is primarily used to store numeric measures that can be aggregated by the related dimensions. A fact table with no numeric measure columns is sometimes referred to as a "factless fact table".

Degenerate dimensions. Sometimes, a fact has associated attributes that are neither keys nor measures, but which can be useful to group or filter facts in a report or analysis. You might include this column in the fact table where client applications can use it as a "degenerate dimension" by which the fact data can be aggregated.

Note: Unlike most database tables, a fact table does not necessarily require a primary key.

Types of Fact Table

Generally, data warehouses include fact tables that are one of the following three types:



- Transaction Fact Tables

OrderDateKey	ProductKey	CustomerKey	OrderNo	Qty	Cost	SalesAmount
20120101	25	120	1000	1	125.00	350.99
20120101	99	120	1000	2	2.50	6.98
20120101	25	178	1001	2	250.00	701.98

- Periodic Snapshot Fact Tables

DateKey	ProductKey	OpeningStock	UnitsIn	UnitsOut	ClosingStock
20120101	25	25	1	3	23
20120101	99	120	0	2	118

- Accumulating Snapshot Fact Tables

OrderNo	OrderDateKey	ShipDateKey	DeliveryDateKey
1000	20120101	20120102	20120105
1001	20120101	20120102	00000000
1002	20120102	00000000	00000000

Transaction fact tables. The most common kind of fact table is a "transaction" fact table, in which each row records a transaction or event at an appropriate grain. For example, a fact table might record sales orders at the line item grain, in which each row records the purchase of a specific item.

Periodic snapshot tables. These record measure values at a specific point in time. For example, a fact table might record the stock movement for each day, including the opening and closing stock count figures.

Accumulating snapshot fact tables. These can be used in scenarios where you might want to use a fact table to track the progress of a business process through multiple stages. For example, a fact table might track an order from initial purchase through to delivery by including a date dimension key field for the order date, shipping date, and delivery date. The **ShipDate** and **DeliveryDate** columns for orders that have been placed but not yet shipped will contain the dimension key for an "Unknown" or "None" row in the time dimension table.

Section 5: Fundamentals of SQL Developer Data Modeler and PowerDesigner Viewer

In this section you will cover the following topics:

- Introduction to PowerDesigner Viewer
- Create a new Design using SQL Developer Data Modeler
- Create a logical data model using SQL Developer Data Modeler
- Create a physical data model from the logical data model using SQL Developer Data Modeler
- Generate the DDL using SQL Developer Data Modeler
- Reverse engineer from a database schema using SQL Developer Data Modeler

Lesson 5.1: Introduction to PowerDesigner Viewer

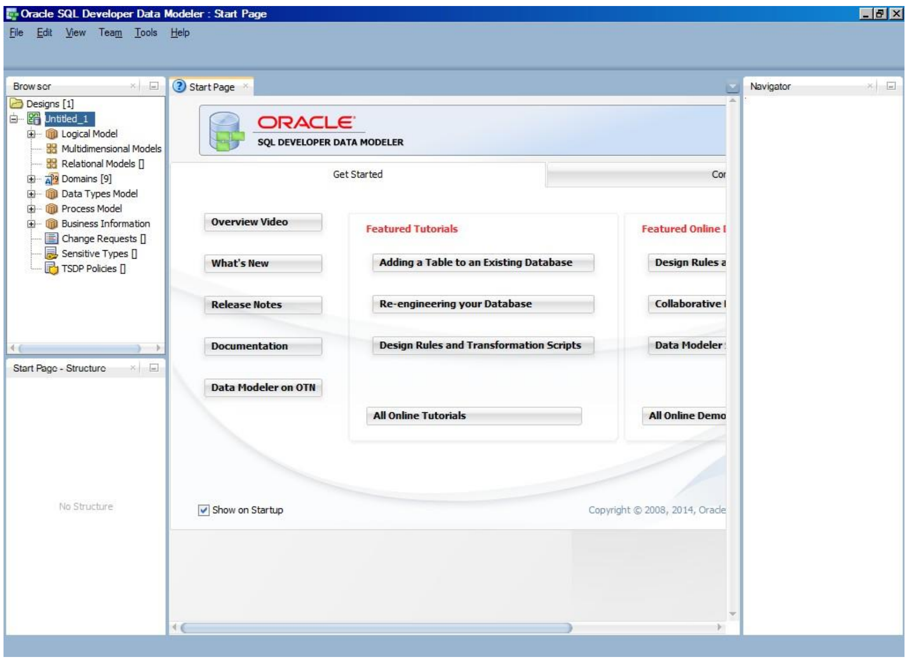
PowerDesigner is an end-to-end modeling tool produced by Sybase and is now owned by SAP. It was initially developed to design Oracle databases in the mid 1990s, but has now evolved to also support MS SQL Server, PostgeSQL, MySQL, DB2 and Informix. It runs on a Windows OS as a standalone tool. PowerDesigner supports standard methodologies and notations and provides automated code- reverse engineering and generation through customizable templates.

PowerDesigner viewer is a free tool that can open, read, and navigate any PowerDesigner model from past or current versions, as well as generate and print reports based on the contents of the models.

Lesson 5.1: Create a new Design using SQL Developer Data Modeler

SQL Developer Data Modeler is data modeling tool developed by Oracle. Using the tool users can create, browse and edit, logical, relational and physical models.

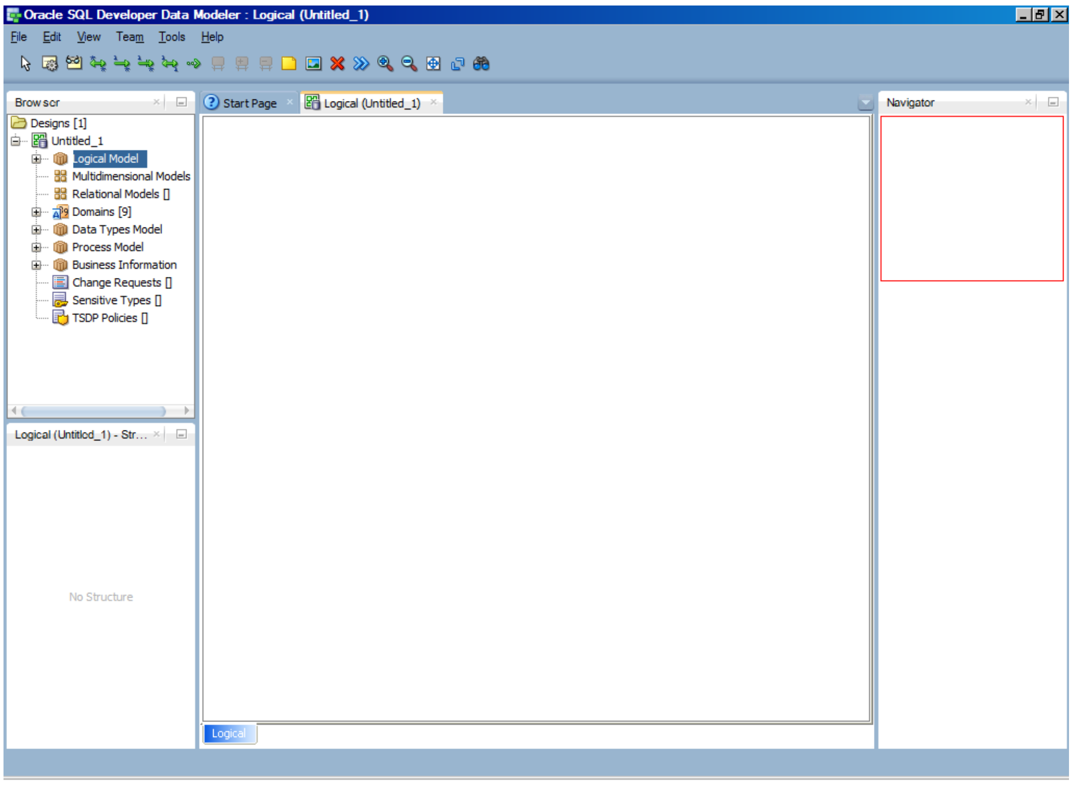
When you start SQL Developer Data Modeler it will take you to this start page:



As with PowerDesigner there is a browser on the left and a work area in the middle. The equivalent of PowerDesigner's Workspaces are Designs in SQL Developer Data Modeler. A new empty Design is created automatically for you when you first start up the tool.

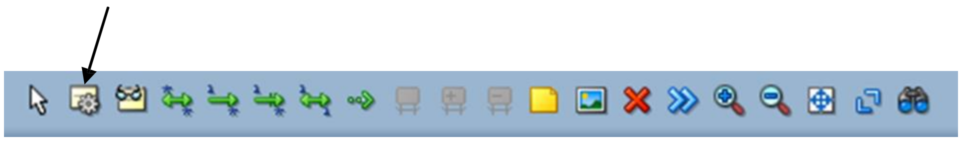
Lesson 5.3: Create a logical data model using SQL Developer Data Modeler

In the Browser, right click on Logical Model node and select Show. Notice that the toolbar changes to display tools specifically for working with entity relationship diagrams.

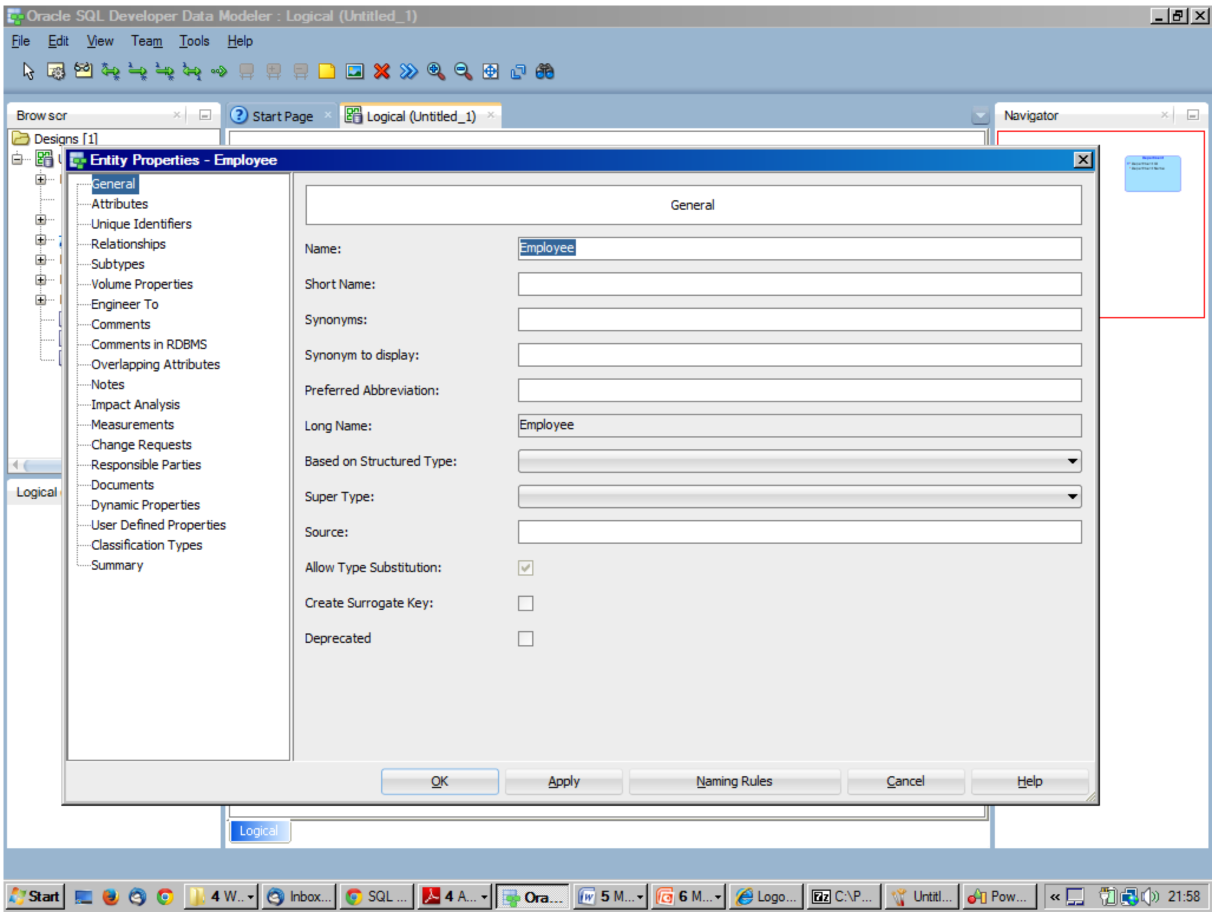


To create an entity:

On the toolbar, click the New Entity tool and click anywhere in the white space of the Logical pane



The Entity Properties window appears in which you can enter the name of the entity and other relevant information.


Oracle SQL Developer Data Modeler : Logical (Untitled_1)

File Edit View Team Tools Help

Browser | Start Page | Logical (Untitled_1) | Navigator

Designs [1]

Entity Properties - Employee

General
Attributes
Unique Identifiers
Relationships
Subtypes
Volume Properties
Engineer To
Comments
Comments in RDBMS
Overlapping Attributes
Notes
Impact Analysis
Measurements
Change Requests
Responsible Parties
Documents
Dynamic Properties
User Defined Properties
Classification Types
Summary

General

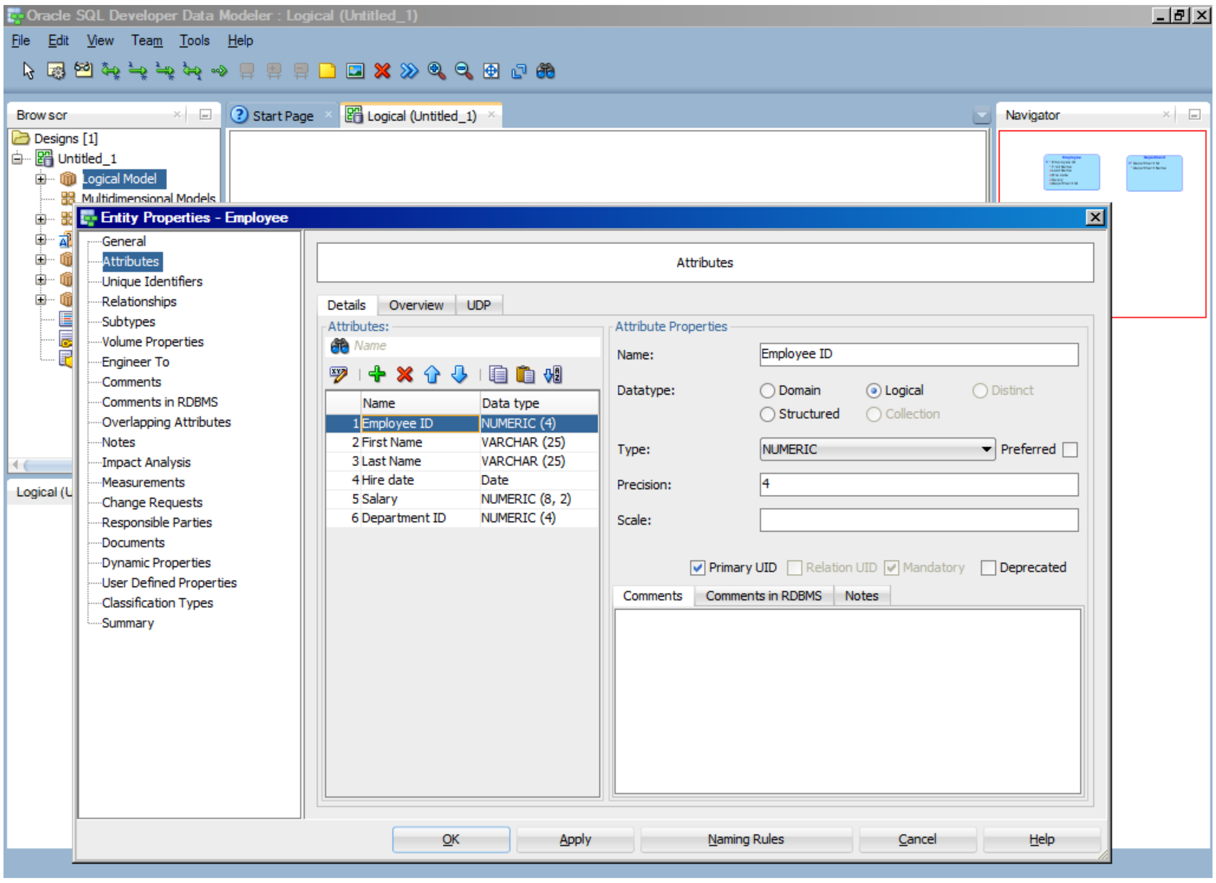
Name:	Employee
Short Name:	
Synonyms:	
Synonym to display:	
Preferred Abbreviation:	
Long Name:	Employee
Based on Structured Type:	
Super Type:	
Source:	
Allow Type Substitution:	✓
Create Surrogate Key:	☐
Deprecated	☐

OK | Apply | Naming Rules | Cancel | Help

Logical

Next add the attributes to the entity – click on Attributes link. Click the Add an Attribute icon

In the Name field, enter a name for the attribute, its data type, type, and identifier (if appropriate) and any other relevant data and click on Apply. Repeat these steps until you have created all the attributes for this entity. You can use the up and down arrows to reorder the columns. When all the attributes have been created, click OK to create the entity and its attributes.



Attributes

	Name	Data type
1	Employee ID	NUMERIC (4)
2	First Name	VARCHAR (25)
3	Last Name	VARCHAR (25)
4	Hire date	Date
5	Salary	NUMERIC (8, 2)
6	Department ID	NUMERIC (4)

Name: Employee ID

Datatype: Domain [✓ Logical] Distinct Structured Collection

Type: NUMERIC Preferred ☐

Precision: 4

Scale:

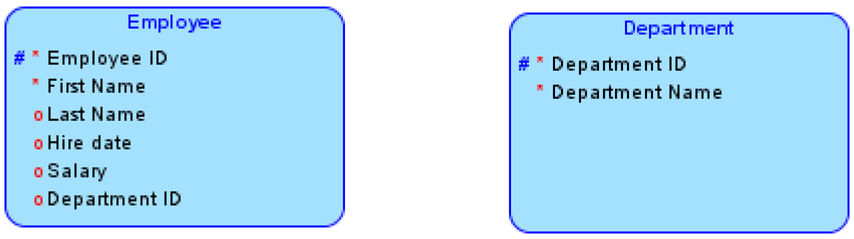
[✓ Primary UID] ☐ Relation UID [✓ Mandatory] ☐ Deprecated

To define the relationships between the click the desired relationship type on the toolbar.

If you hover over each icon a tooltip will tell you which relationship it creates. Next

click the source entity and then the target entity to create a relationship. In our example we

have two entities EMPLOYEE and DEPARTMENT and the relationship is between the Department ID in both entities.

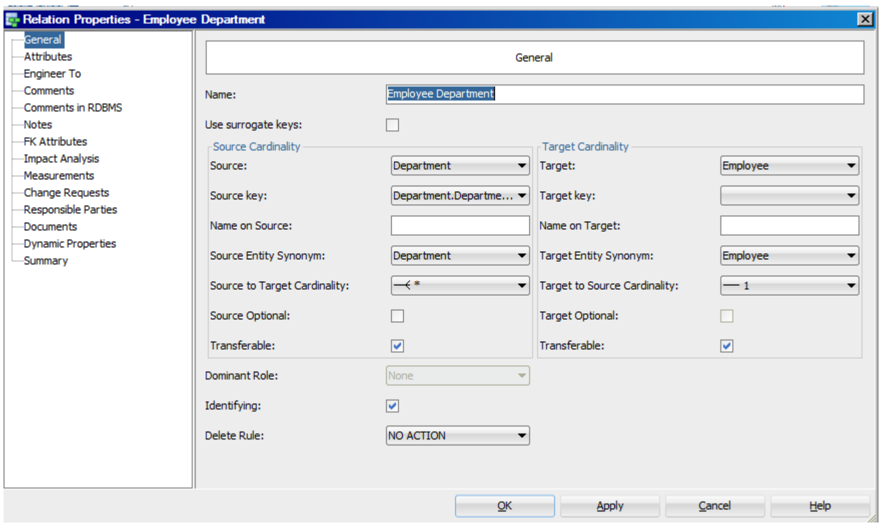


One department can employee many employees but one employee can only work for one department – so this will be a one-to-many (1:N) relationship with the EMPLOYEE entity being the target and the DEPARTMENT being the source. Click on the DEPARTMENT entity first and then the EMPLOYEE one. This will automatically display Relation Properties box as shown below. Specify the name of the relationship.

Specify the source and target names for the relationship. Note that these names will be specified in the diagram and will be used to validate the business rules for the relationship.

Specify the minimum and maximum cardinality for the relationship. The Source Optional option controls whether the source entity in the relation must contain one or more instances. The Target Optional option controls whether the target entity in the relationship must contain one or more instances. In the example in the slide, the check box for Target Optional was deselected because there must be a DEPARTMENT for each EMPLOYEE.

Specify whether this is an identifying relationship by selecting the Identifying option.



General

Name: Employee Department

Use surrogate keys: ☐

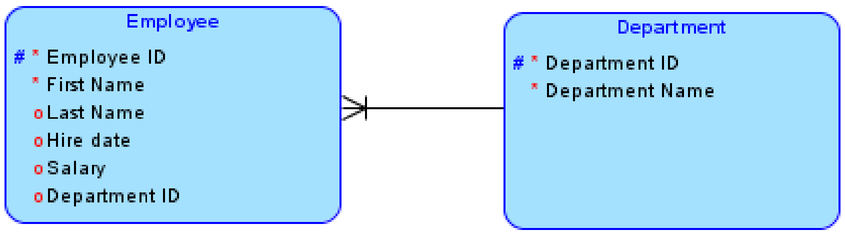
Source Cardinality		Target Cardinality	
Source:	Department	Target:	Employee
Source key:	Department.Departme...	Target key:	
Name on Source:		Name on Target:	
Source Entity Synonym:	Department	Target Entity Synonym:	Employee
Source to Target Cardinality:	—< *	Target to Source Cardinality:	— 1
Source Optional:	☐	Target Optional:	☐
Transferable:	✓	Transferable:	✓

Dominant Role: None

Identifying: ✓

Delete Rule: NO ACTION

Click Apply and then OK

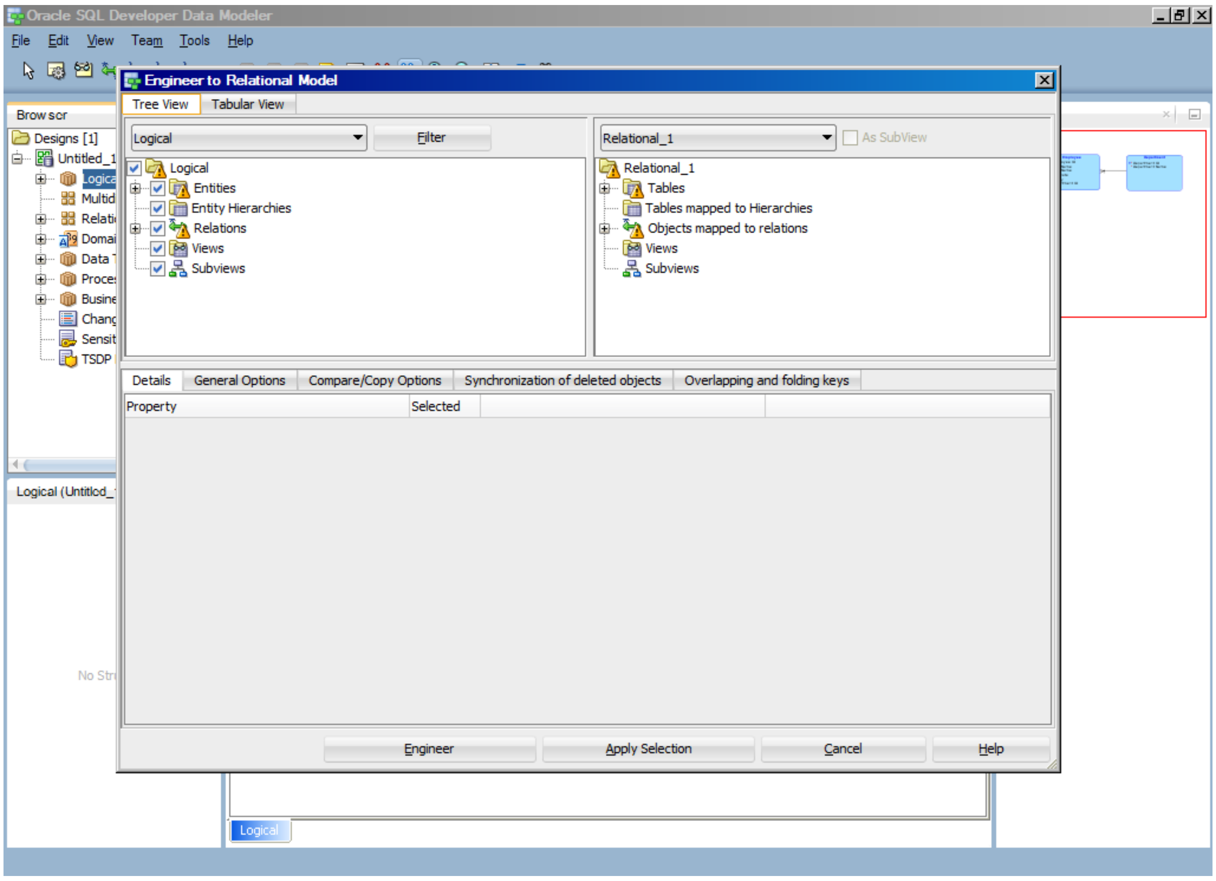


To move a created entity to a different position right click on the entity hold the mouse button down and drag it into position. Release the button when it is in the correct position.

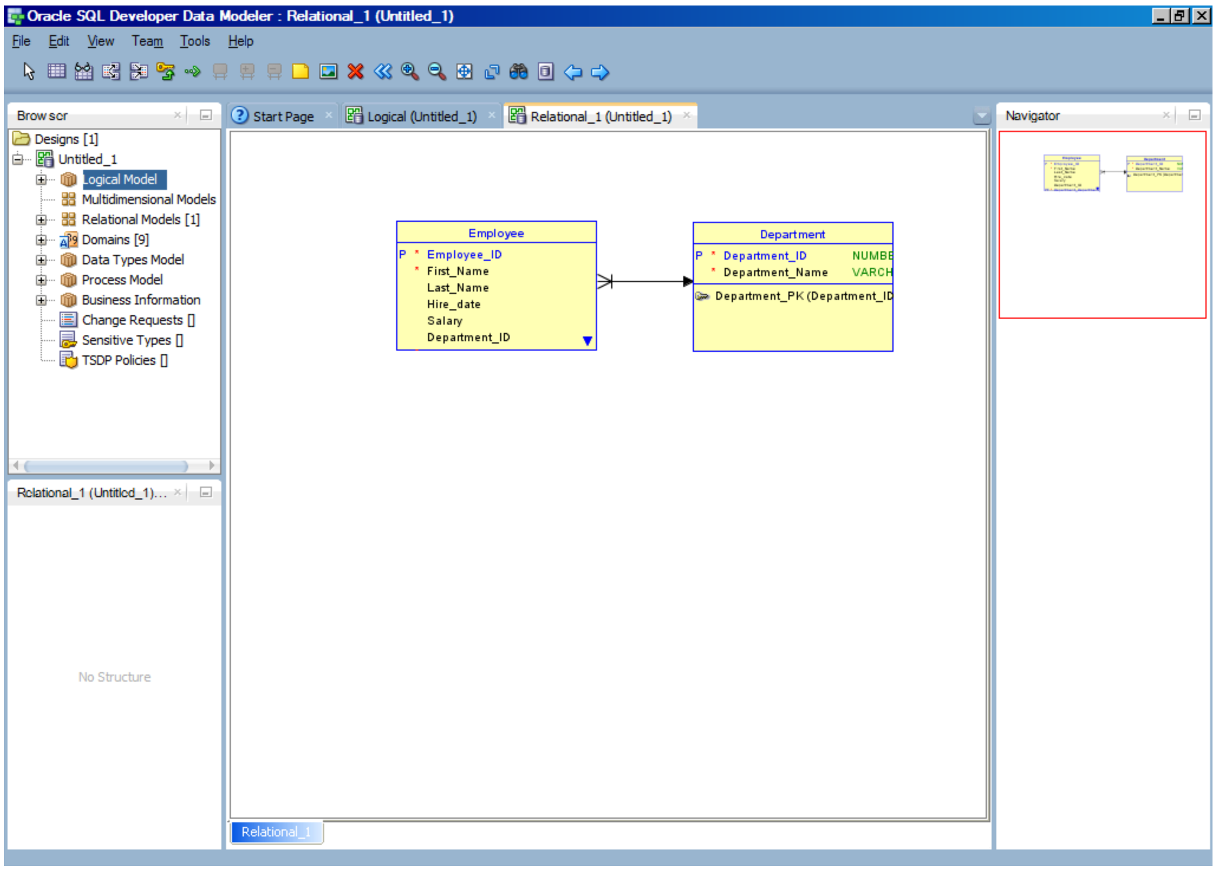
Lesson 5.4: Create a physical data model from the logical data model using SQL Developer Data Modeler

You can develop the relational model as follows:

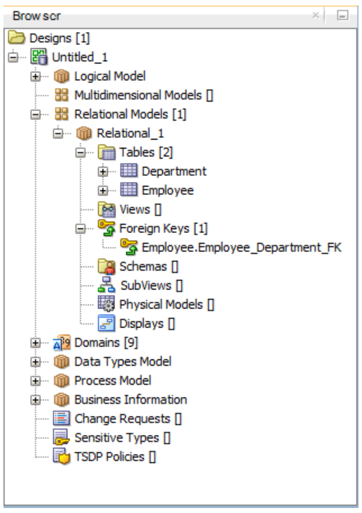
Click on the Logical Model in the Browser, and click the Engineer to Relational Model icon on the toolbar. The Engineering dialog box is displayed.



For now, accept all defaults (i.e. do not filter), and click Engineer. This causes the Relational_1 model to be populated with tables and other objects that reflect the logical model.

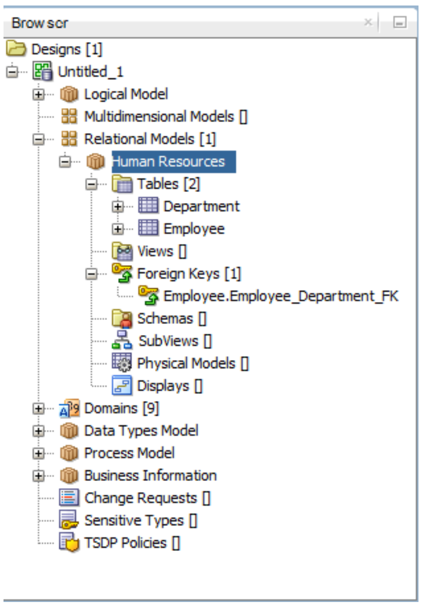


Expand the Relational Models node in the object browser on the left side of the window and expand Relational_1 and optionally nodes under it that contain any entries (such as Tables and Columns), to view the objects created.



Change the name of the relational model from Relational_1 to something more meaningful for diagram displays, such as Human Resources. To do this right click the Relational_1 node in the hierarchy display, select Properties, in the General pane of the Model Properties – Relational_1

dialog box specify Name as Human Resources, and click OK.



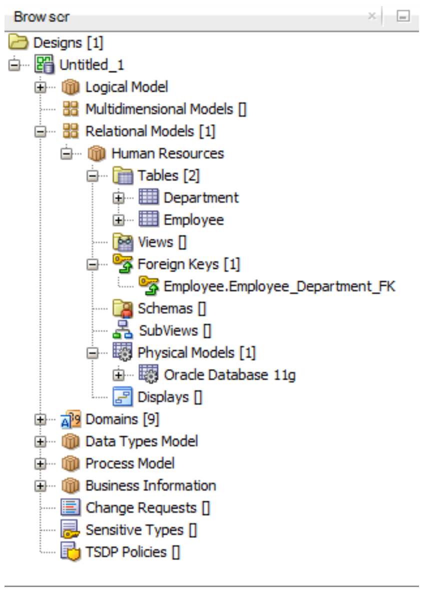
Lesson 5.5: Generate the DDL using SQL Developer Data Modeler

Now we will look at how you can generate Data Definition Language (DDL) statements that you can use to create database objects that reflect the models that you have designed. The DDL statements will implement the physical model (ie type of database) such as Oracle Database 11g) that you specify.

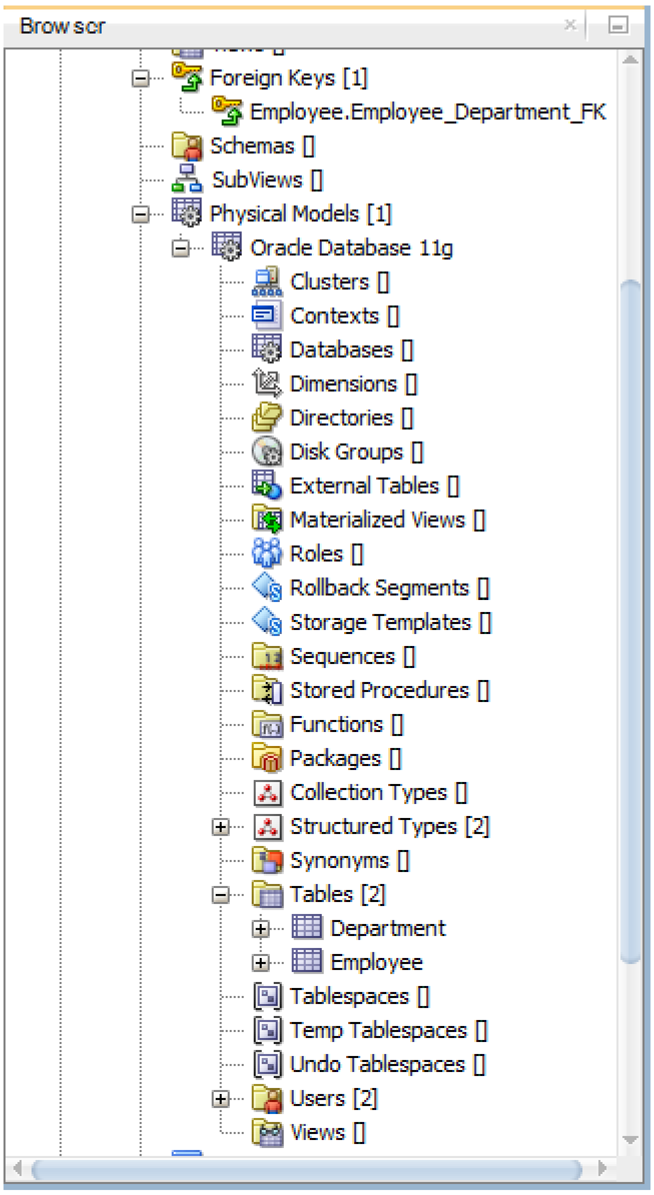
You can develop the physical model by following the steps outlined below.

Expand the Relational Model in the Browser and right click the Physical Models node and select New. A dialog box is displayed for selecting the type of database for which to create the physical model. Specify the type of database (for example, Oracle Database 11g), and click OK.

A physical model reflecting the type of database is created under the Physical Models node.



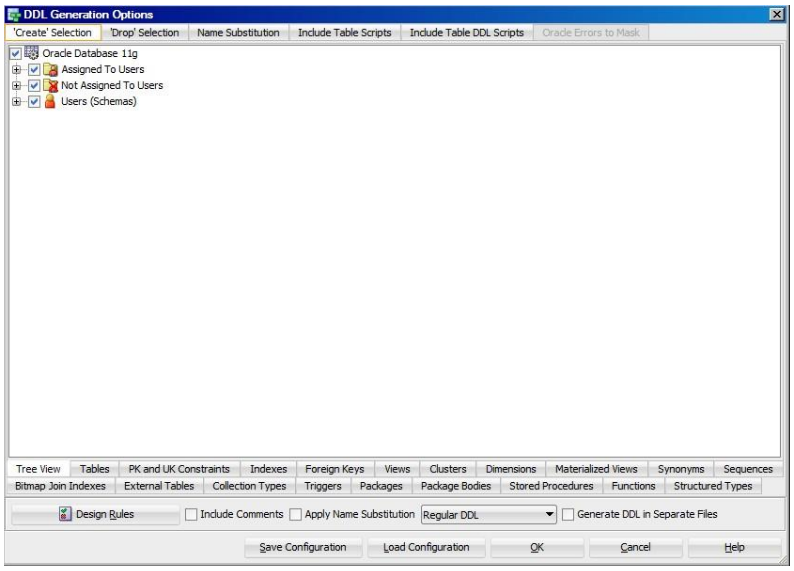
Expand the Physical Models node under the Library relational model, and expand the newly created physical model and the Tables node under it, to view the table objects that were created.



```
Browser
  Foreign Keys [1]
    Employee.Employee_Department_FK
  Schemas []
  SubViews []
  Physical Models [1]
    Oracle Database 11g
      Clusters []
      Contexts []
      Databases []
      Dimensions []
      Directories []
      Disk Groups []
      External Tables []
      Materialized Views []
      Roles []
      Rollback Segments []
      Storage Templates []
      Sequences []
      Stored Procedures []
      Functions []
      Packages []
      Collection Types []
      Structured Types [2]
      Synonyms []
      Tables [2]
        Department
        Employee
      Tablespaces []
      Temp Tablespaces []
      Undo Tablespaces []
      Users [2]
      Views []
```

Click File, then Export, then DDL File.

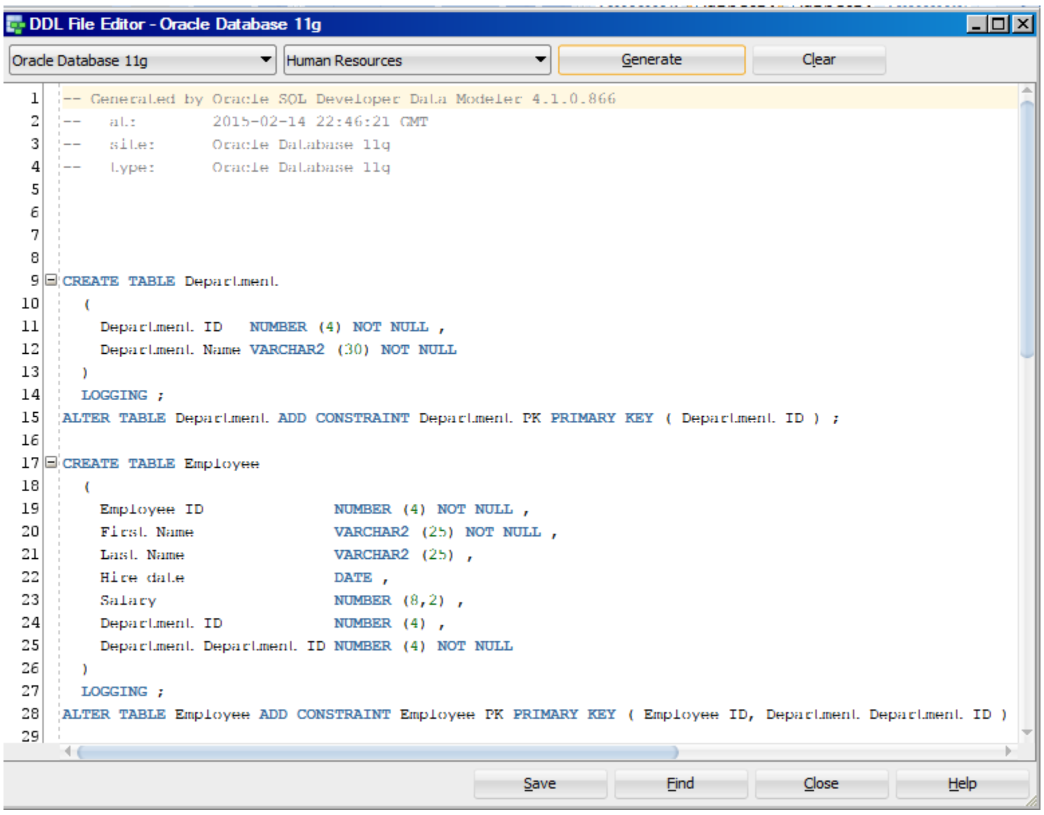
Select the database type (for example, Oracle Database 11g) and click Generate. The DDL Generation Options dialog box is displayed.



✓ Oracle Database 11g
✓ Assigned To Users
✓ Not Assigned To Users
✓ Users (Schemas)

☐ Include Comments ☐ Apply Name Substitution Regular DDL ☐ Generate DDL in Separate Files

Accept all defaults, and click OK. A DDL file editor is displayed, with SQL statements to create the tables and add constraints.



```
-- Generated by Oracle SQL Developer Data Modeler 4.1.0.866
--   at:        2015-02-14 22:46:21 GMT
--   site:      Oracle Database 11g
--   type:      Oracle Database 11g




CREATE TABLE Department
  (
    Department ID   NUMBER (4) NOT NULL ,
    Department Name VARCHAR2 (30) NOT NULL
  )
  LOGGING ;
ALTER TABLE Department ADD CONSTRAINT Department PK PRIMARY KEY ( Department ID ) ;

CREATE TABLE Employee
  (
    Employee ID                NUMBER (4) NOT NULL ,
    First Name                 VARCHAR2 (25) NOT NULL ,
    Last Name                  VARCHAR2 (25) ,
    Hire date                  DATE ,
    Salary                     NUMBER (8,2) ,
    Department ID              NUMBER (4) ,
    Department Department ID NUMBER (4) NOT NULL
  )
  LOGGING ;
ALTER TABLE Employee ADD CONSTRAINT Employee PK PRIMARY KEY ( Employee ID, Department Department ID )
```

Click Save to save the statements to a .sql script file (for example, create_hr_objects.sql) on your local system.

Later, run the script (for example, using a database connection and SQL Worksheet in SQL Developer) to create the objects in the desired database.

Click Close to close the DDL file editor.

To save the design right click on Untitled_1 in the Browser and select Save Design, give the file a name eg hrDesign and click Save. This creates a folder called hrDesign to hold the detailed information about the design and a file called hrDesign.dmd. It is this file that you open when you want to view or change the models. The file loads the whole design into Data Modeler.

Lesson 5.6: Reverse engineer from a database schema using SQL Developer Data Modeler

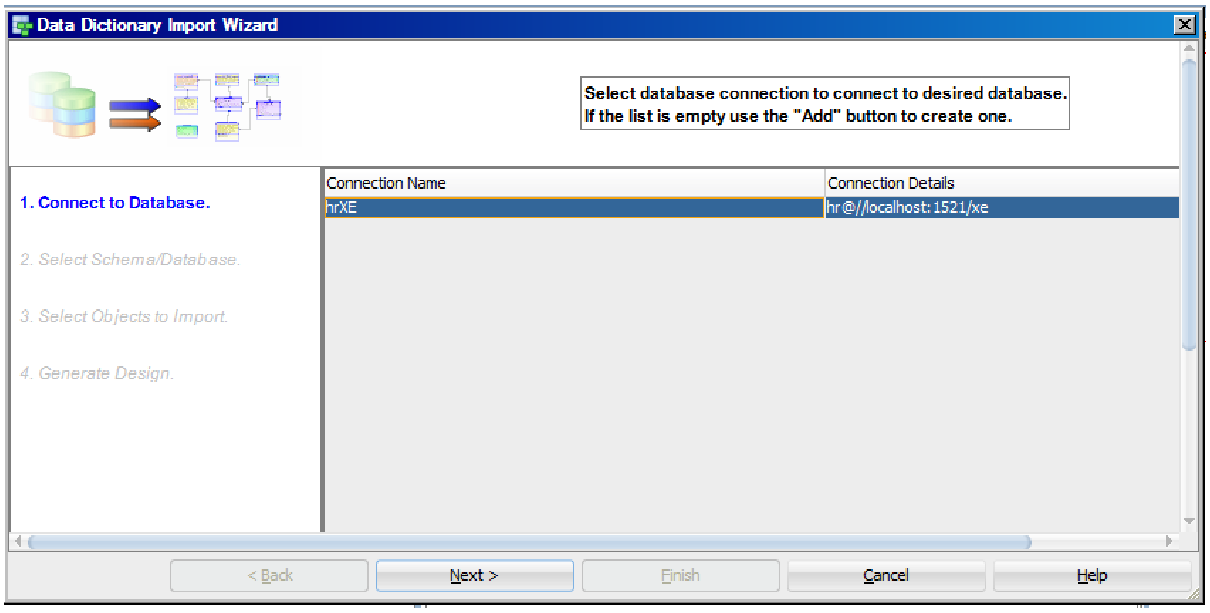
Reverse engineer in this context means we can take an existing data base schema and create a relational model based on it. We are going to use the HR schema which already exists in your database.

First of all, close any open Designs.

This process requires the following actions:

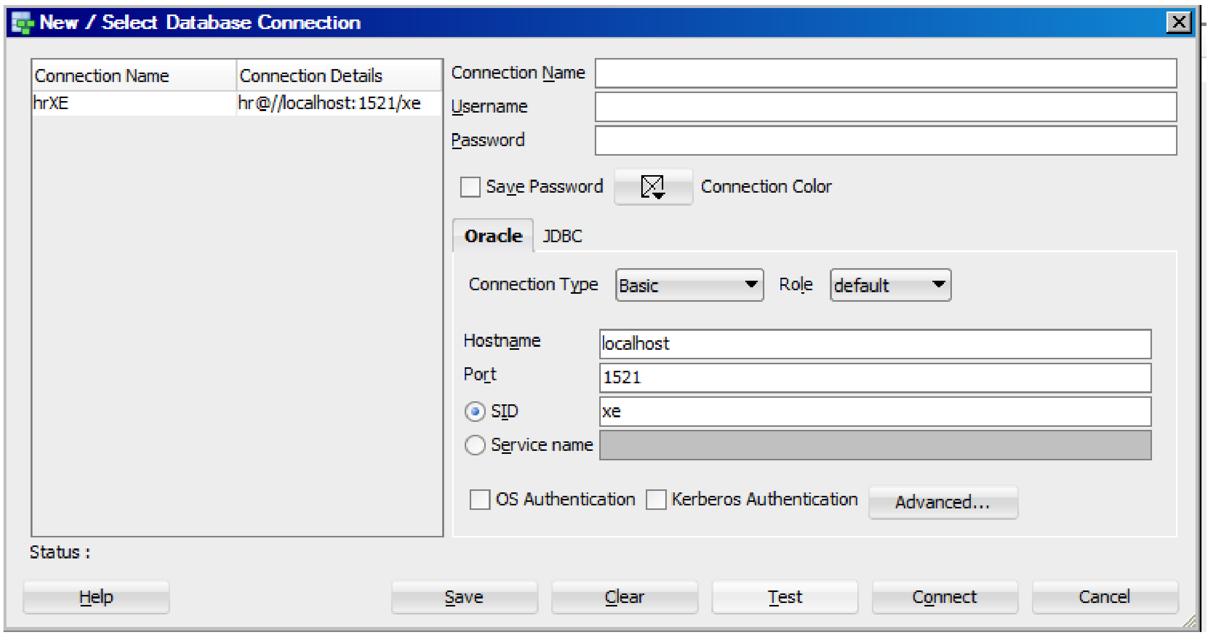
1. Connect to the database
2. Select the Schema or Database you want to use
3. Select the objects you want to include
4. Generate the design

Start by select File **>** Import **>** Data Dictionary. Select the connection you wish to use and click Next. If the connection does not already exist, you will need to create it first.


Data Dictionary Import Wizard
Select database connection to connect to desired database.
If the list is empty use the "Add" button to create one.
1. Connect to Database.
2. Select Schema/Database.
3. Select Objects to Import.
4. Generate Design.
< Back
Next >
Finish
Cancel
Help

Connection Name	Connection Details
hrXE	hr@//localhost:1521/xe

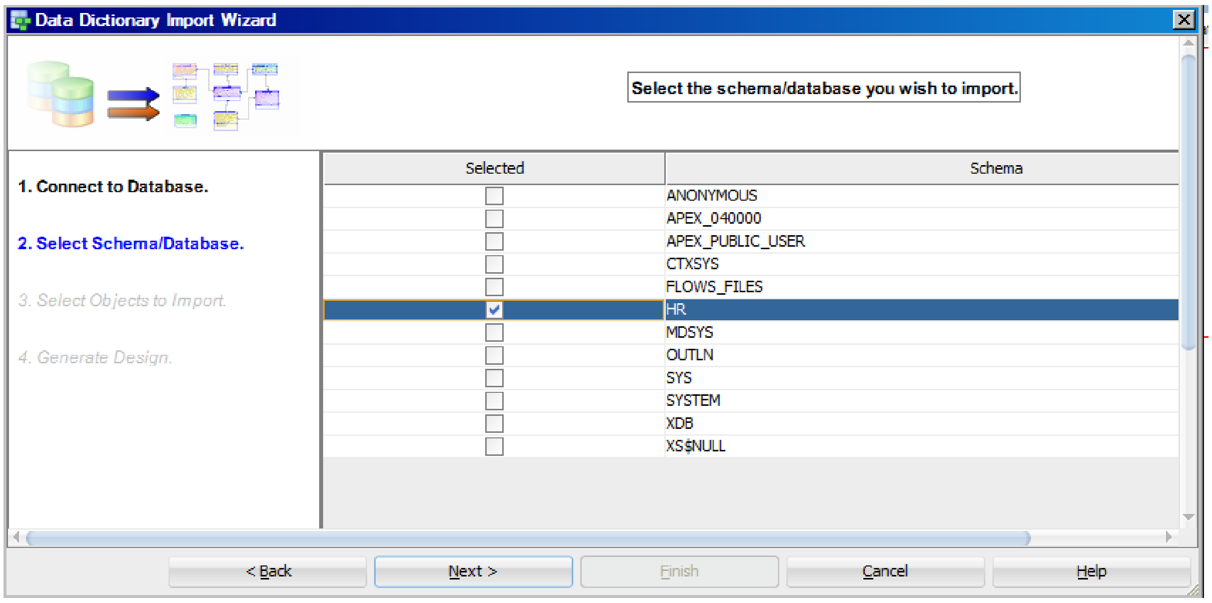
Your trainer will give you the connection information for the class. Click on Add and complete the form.



Connection Name	Connection Details
hrXE	hr@//localhost:1521/xe

Connection Name:
Username:
Password:
Save Password [ ] Connection Color
Oracle | JDBC
Connection Type: Basic
Role: default
Hostname: localhost
Port: 1521
SID [●]: xe
Service name:
OS Authentication [ ] Kerberos Authentication [ ] Advanced...
Status :

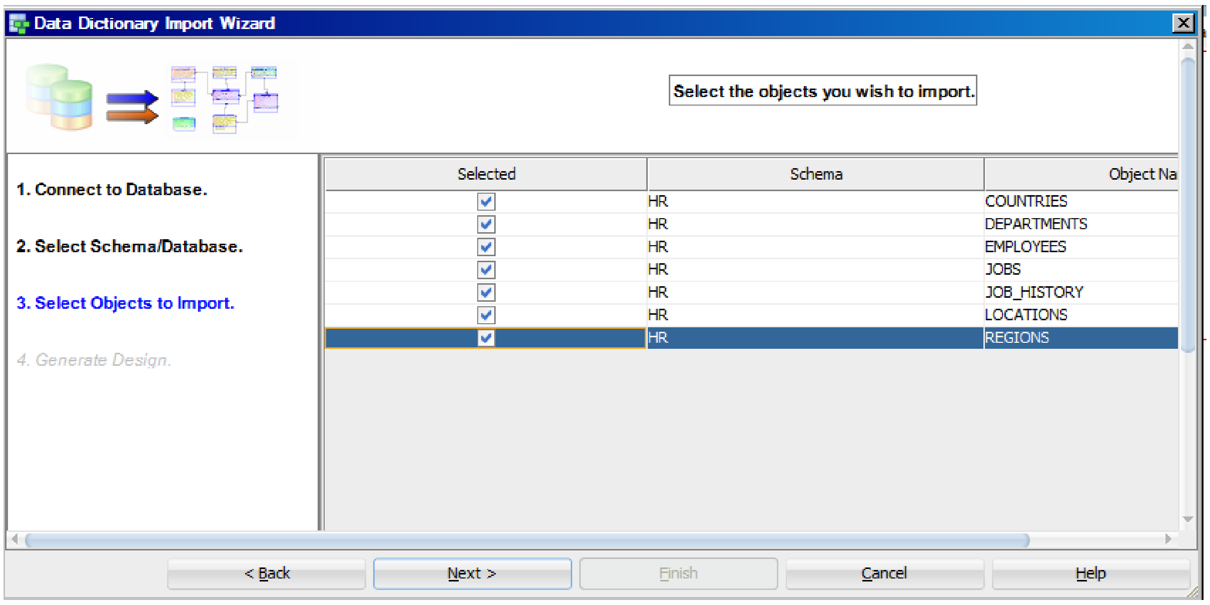
Once the connection is completed successfully you will be able to select the schema you want to import as shown below


Data Dictionary Import Wizard
1. Connect to Database.
2. Select Schema/Database.
3. Select Objects to Import.
4. Generate Design.
< Back
Next >
Finish
Cancel
Help

Select the schema/database you wish to import.

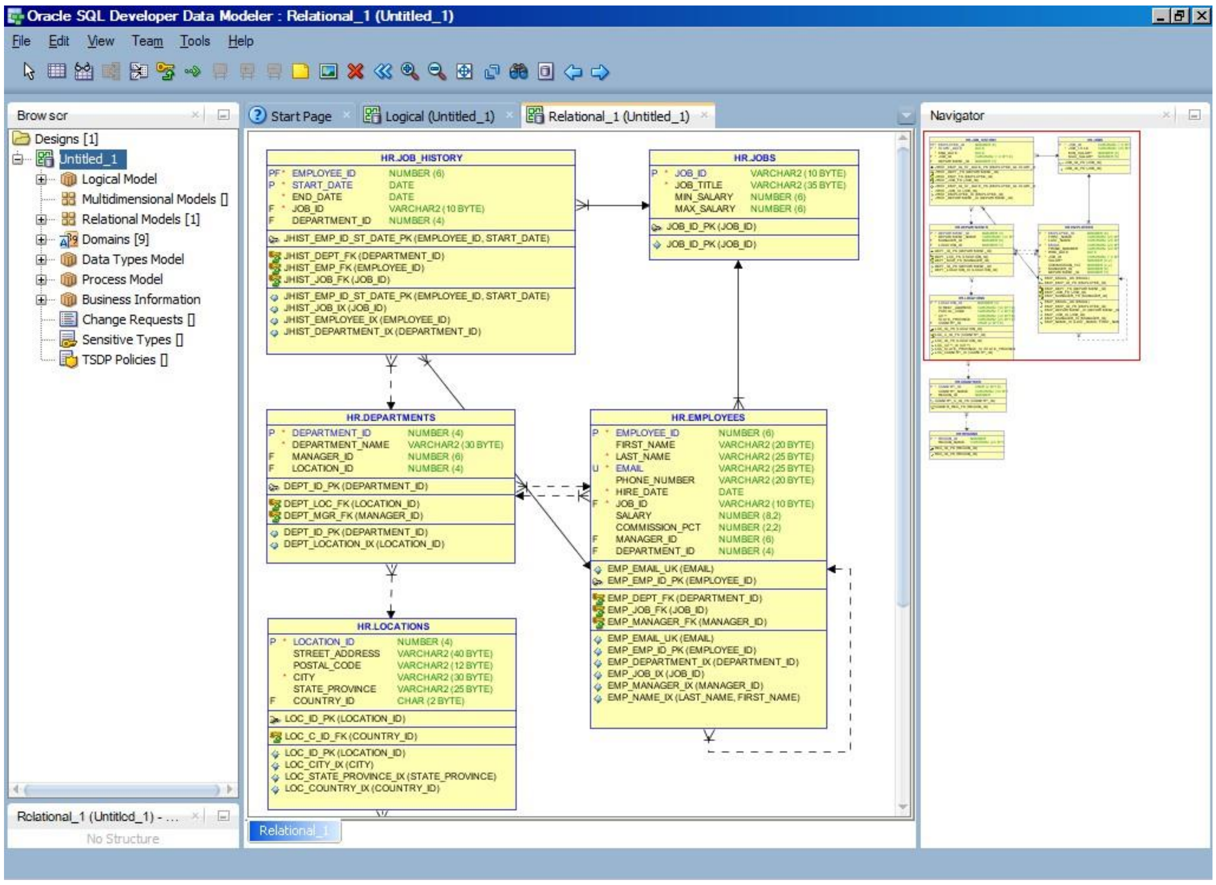
Selected	Schema
☐	ANONYMOUS
☐	APEX_040000
☐	APEX_PUBLIC_USER
☐	CTXSYS
☐	FLOWS_FILES
✓	HR
☐	MDSYS
☐	OUTLN
☐	SYS
☐	SYSTEM
☐	XDB
☐	XS$NULL

Click Next. Select the objects you want to import.


Data Dictionary Import Wizard
Select the objects you wish to import.
1. Connect to Database.
2. Select Schema/Database.
3. Select Objects to Import.
4. Generate Design.
< Back
Next >
Finish
Cancel
Help

Selected	Schema	Object Na
✓	HR	COUNTRIES
✓	HR	DEPARTMENTS
✓	HR	EMPLOYEES
✓	HR	JOBS
✓	HR	JOB_HISTORY
✓	HR	LOCATIONS
✓	HR	REGIONS

Click Next. View the summary and click Finish. The Design will be generated:



HR.JOB_HISTORY

PF *	EMPLOYEE_ID	NUMBER (6)
P *	START_DATE	DATE
*	END_DATE	DATE
F *	JOB_ID	VARCHAR2 (10 BYTE)
F	DEPARTMENT_ID	NUMBER (4)

JHIST_EMP_ID_ST_DATE_PK (EMPLOYEE_ID, START_DATE)
JHIST_DEPT_FK (DEPARTMENT_ID)
JHIST_EMP_FK (EMPLOYEE_ID)
JHIST_JOB_FK (JOB_ID)
JHIST_EMP_ID_ST_DATE_PK (EMPLOYEE_ID, START_DATE)
JHIST_JOB_IX (JOB_ID)
JHIST_EMPLOYEE_IX (EMPLOYEE_ID)
JHIST_DEPARTMENT_IX (DEPARTMENT_ID)

HR.JOBS

P *	JOB_ID	VARCHAR2 (10 BYTE)
*	JOB_TITLE	VARCHAR2 (35 BYTE)
	MIN_SALARY	NUMBER (6)
	MAX_SALARY	NUMBER (6)

JOB_ID_PK (JOB_ID)
JOB_ID_PK (JOB_ID)

HR.DEPARTMENTS

P *	DEPARTMENT_ID	NUMBER (4)
*	DEPARTMENT_NAME	VARCHAR2 (30 BYTE)
F	MANAGER_ID	NUMBER (6)
F	LOCATION_ID	NUMBER (4)

DEPT_ID_PK (DEPARTMENT_ID)
DEPT_LOC_FK (LOCATION_ID)
DEPT_MGR_FK (MANAGER_ID)
DEPT_ID_PK (DEPARTMENT_ID)
DEPT_LOCATION_IX (LOCATION_ID)

HR.EMPLOYEES

P *	EMPLOYEE_ID	NUMBER (6)
	FIRST_NAME	VARCHAR2 (20 BYTE)
*	LAST_NAME	VARCHAR2 (25 BYTE)
U *	EMAIL	VARCHAR2 (25 BYTE)
	PHONE_NUMBER	VARCHAR2 (20 BYTE)
*	HIRE_DATE	DATE
F *	JOB_ID	VARCHAR2 (10 BYTE)
	SALARY	NUMBER (8,2)
	COMMISSION_PCT	NUMBER (2,2)
F	MANAGER_ID	NUMBER (6)
F	DEPARTMENT_ID	NUMBER (4)

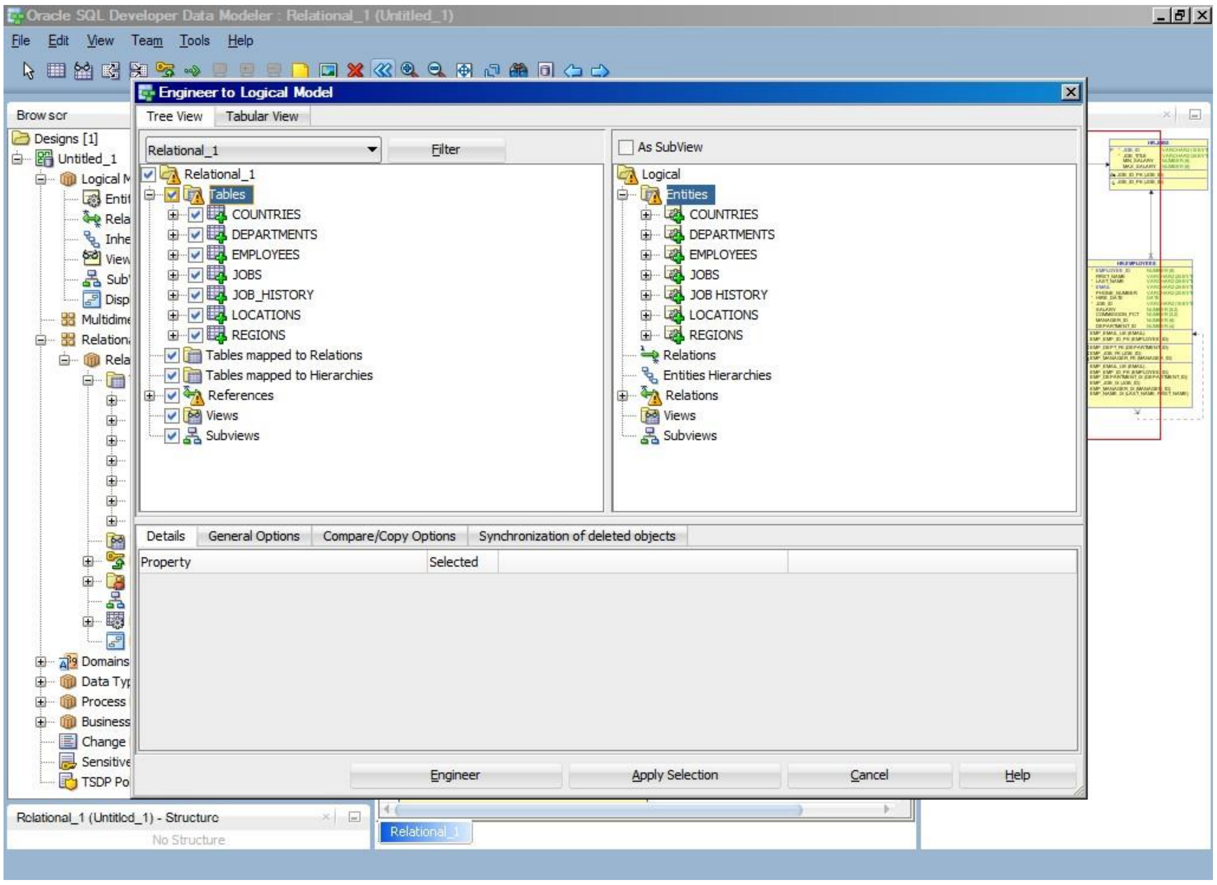
EMP_EMAIL_UK (EMAIL)
EMP_EMP_ID_PK (EMPLOYEE_ID)
EMP_DEPT_FK (DEPARTMENT_ID)
EMP_JOB_FK (JOB_ID)
EMP_MANAGER_FK (MANAGER_ID)
EMP_EMAIL_UK (EMAIL)
EMP_EMP_ID_PK (EMPLOYEE_ID)
EMP_DEPARTMENT_IX (DEPARTMENT_ID)
EMP_JOB_IX (JOB_ID)
EMP_MANAGER_IX (MANAGER_ID)
EMP_NAME_IX (LAST_NAME, FIRST_NAME)

HR.LOCATIONS

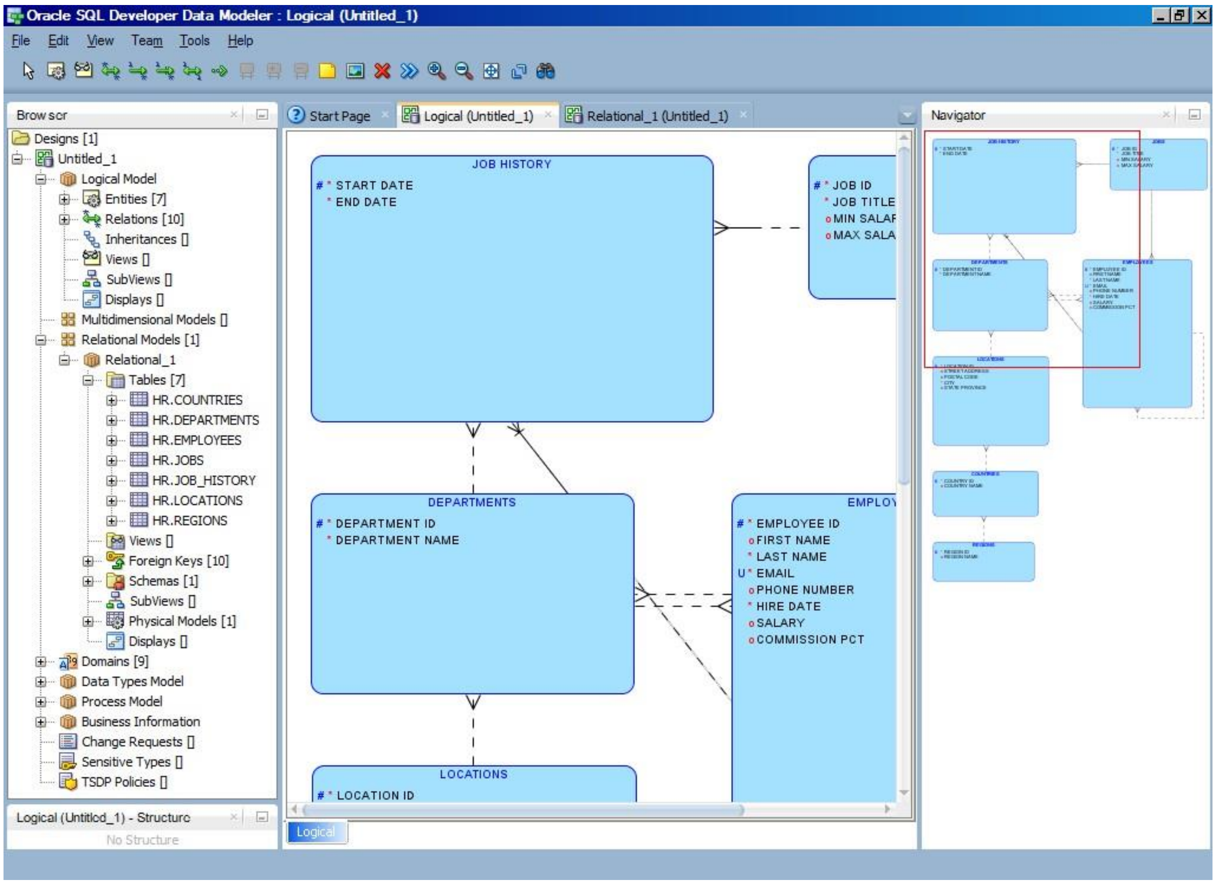
P *	LOCATION_ID	NUMBER (4)
	STREET_ADDRESS	VARCHAR2 (40 BYTE)
	POSTAL_CODE	VARCHAR2 (12 BYTE)
*	CITY	VARCHAR2 (30 BYTE)
	STATE_PROVINCE	VARCHAR2 (25 BYTE)
F	COUNTRY_ID	CHAR (2 BYTE)

LOC_ID_PK (LOCATION_ID)
LOC_C_ID_FK (COUNTRY_ID)
LOC_ID_PK (LOCATION_ID)
LOC_CITY_IX (CITY)
LOC_STATE_PROVINCE_IX (STATE_PROVINCE)
LOC_COUNTRY_IX (COUNTRY_ID)

Next we want to Reverse Engineer the Relational Model to a Logical Model. Select the Engineer to Logical Model icon . An engineering window opens. The warning icons indicate that objects are different between the relational and logical model. Expand the Tables object.

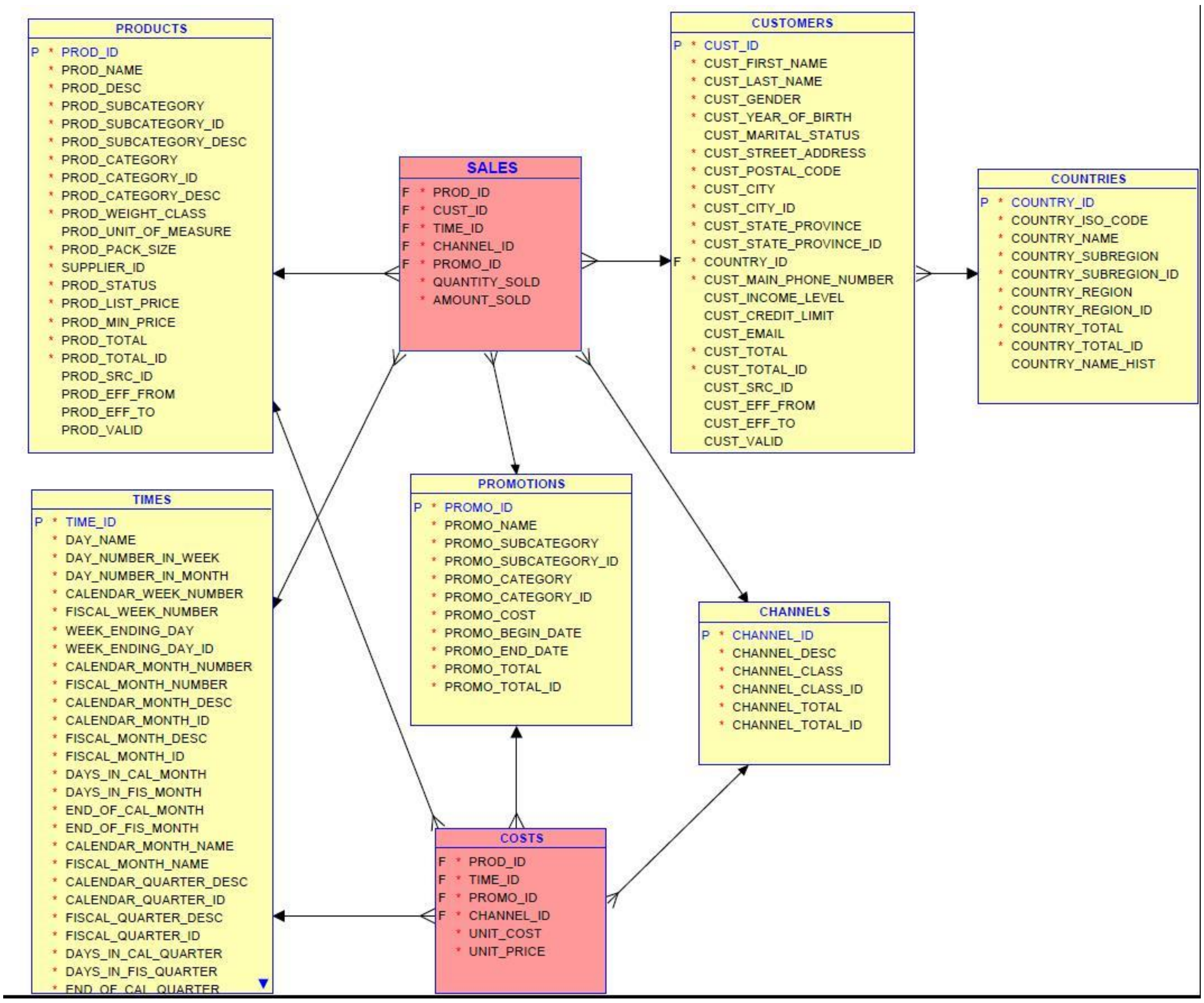


The plus sign icon indicates that the tables will be added to the logical model. Click Engineer**.** You should now see the logical model created successfully.


Oracle SQL Developer Data Modeler : Logical (Untitled_1)
File Edit View Team Tools Help
Browser
Designs [1]
Untitled_1
Logical Model
Entities [7]
Relations [10]
Inheritances []
Views []
SubViews []
Displays []
Multidimensional Models []
Relational Models [1]
Relational_1
Tables [7]
HR.COUNTRIES
HR.DEPARTMENTS
HR.EMPLOYEES
HR.JOBS
HR.JOB_HISTORY
HR.LOCATIONS
HR.REGIONS
Views []
Foreign Keys [10]
Schemas [1]
SubViews []
Physical Models [1]
Displays []
Domains [9]
Data Types Model
Process Model
Business Information
Change Requests []
Sensitive Types []
TSDP Policies []
Logical (Untitled_1) - Structure
No Structure
Start Page
Logical (Untitled_1)
Relational_1 (Untitled_1)
Navigator
JOB HISTORY
# * START DATE
* END DATE
# * JOB ID
* JOB TITLE
o MIN SALAR
o MAX SALA
DEPARTMENTS
# * DEPARTMENT ID
* DEPARTMENT NAME
EMPLOY
# * EMPLOYEE ID
o FIRST NAME
* LAST NAME
U * EMAIL
o PHONE NUMBER
* HIRE DATE
o SALARY
o COMMISSION PCT
LOCATIONS
# * LOCATION ID
Logical

Lesson 5.6: Create a multidimensional model using SQL Developer Data Modeller

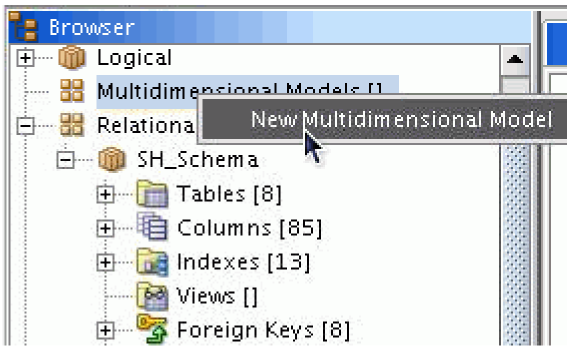
In this lesson the SH schema will be used. SH is the Sales History schema of the Oracle sample schemas.



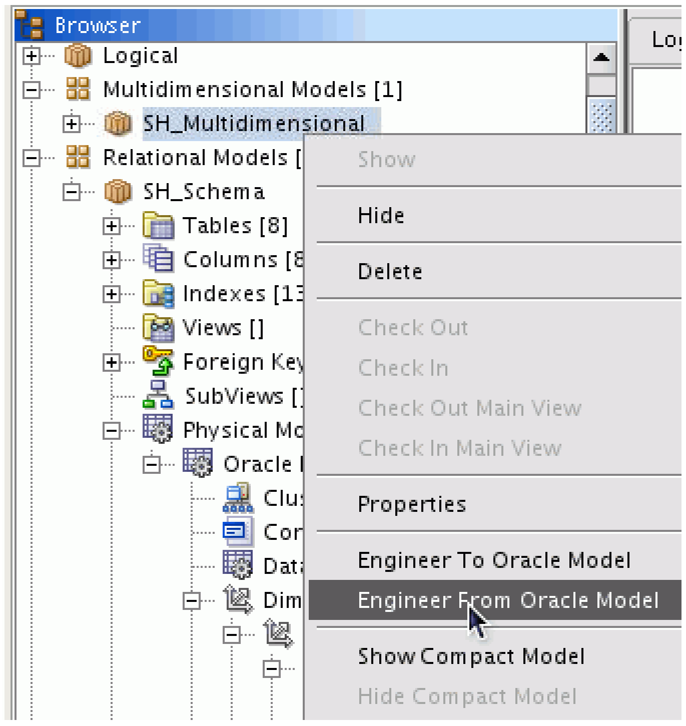
Generating a Multidimensional Model

To generate a multidimensional model, you engineer the relational model to a logical model and then generate the multidimensional model

You can create a multidimensional model. In the left navigator, right-click **Multidimensional Models** and select **New Multidimensional Model**.

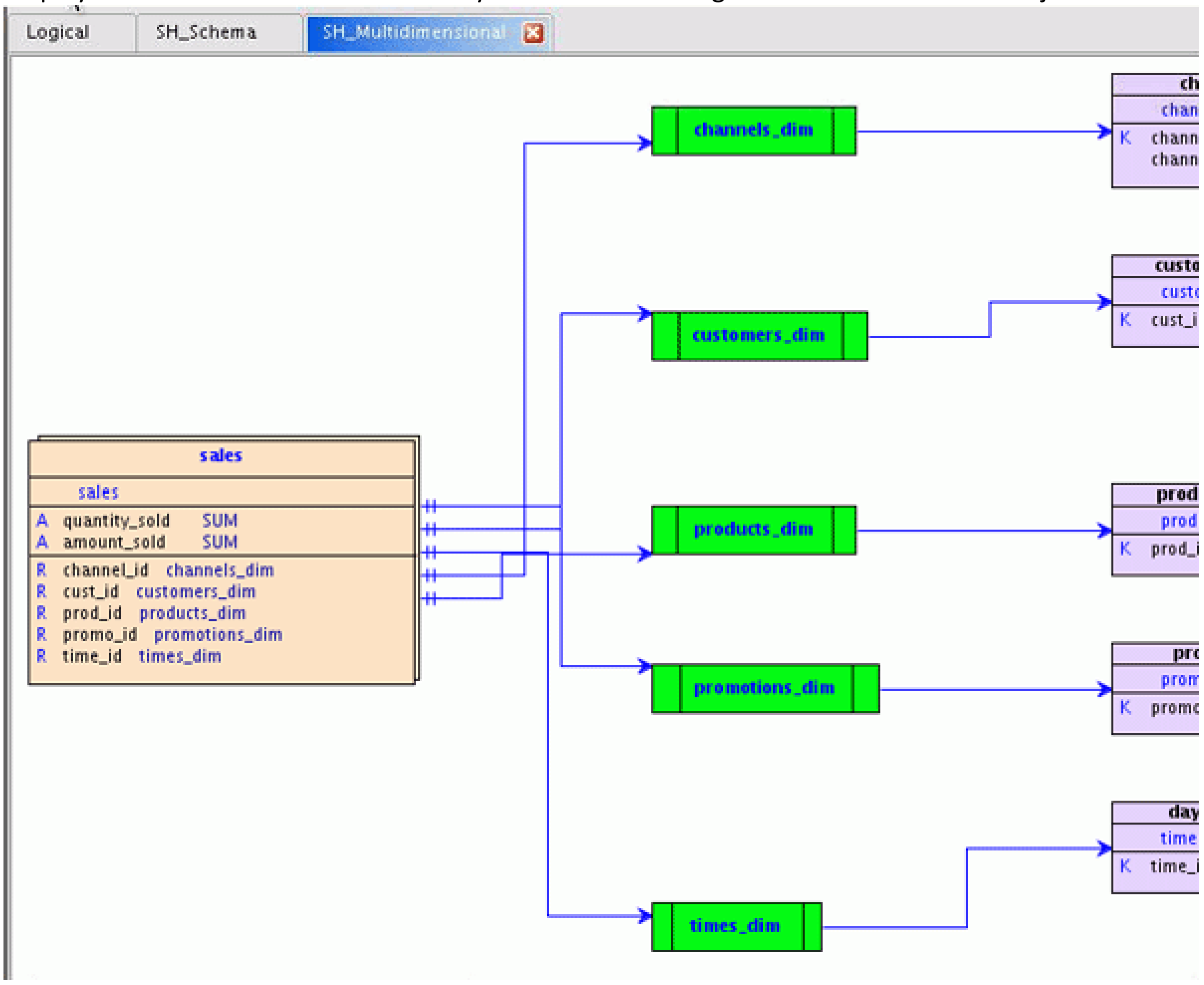


To create the multidimensional model right-click **SH_Multidimensional** and select **Engineer From Oracle Model**.



On the ‘Select Oracle Model’ form, select ‘SH_Schema’ for Relational Model, ‘Use Foreign Keys’ for Match Method. Click ‘OK’

Display the Multidimensional Model. Layout the model using Auto Route and Resize Object to Visible.



Part III – Relational Databases using Oracle

Section 1: Development Tools

In this section you will learn:

- What SQL Developer is
- How to use SQL Developer

Lesson 1.1: Overview SQL Developer

The main development tool used in this training is Oracle SQL Developer. SQL*Plus is available as an optional development tool. This is appropriate for a 10g and 11g audience.

What is Oracle SQL Developer?

Oracle SQL Developer is free and fully supported graphical tool that enhances productivity and simplifies database development tasks. Using SQL Developer, users can browse, edit and create database objects, run SQL statements, edit and debug PL/SQL statements and build PL/SQL unit tests, run reports, and place files under version control.

Getting Started with Oracle SQL Developer

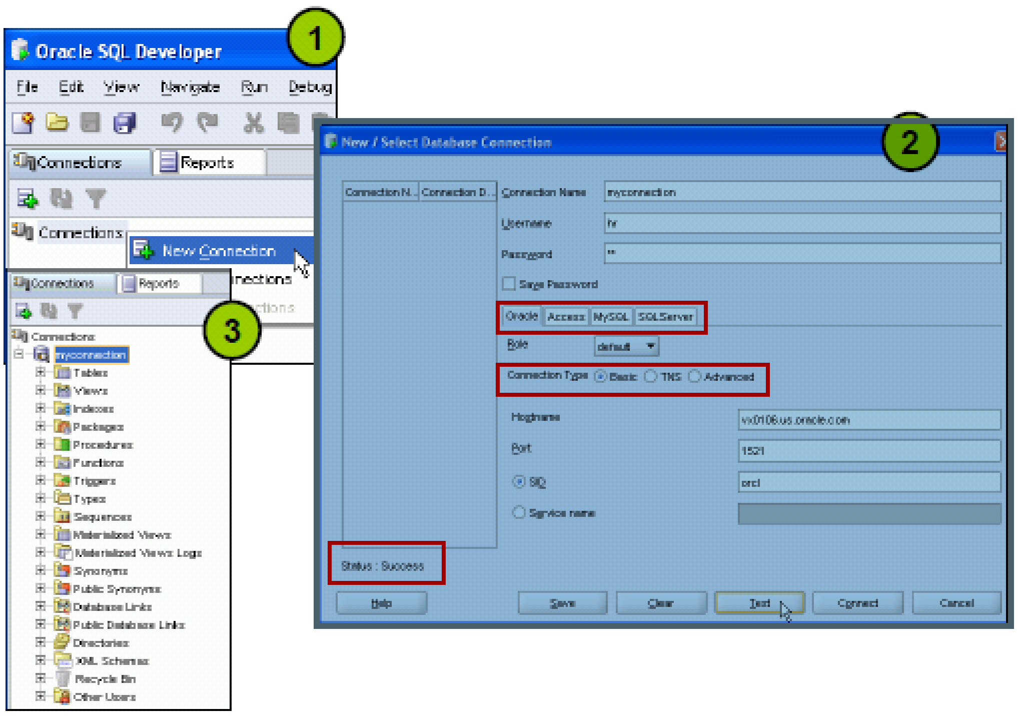
Download the latest Oracle SQL Developer from

http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html

Creating a Database Connection

A connection is an Oracle SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use Oracle SQL Developer, you must have at least one database connection, which may be existing, created, or imported .You can create and test connections for multiple databases and for multiple schemas .By default, the tnsnames.ora file is located in the $ORACLE_HOME/network/admin directory. But, it can also be in the directory specified by the TNS_ADMIN environment variable or the registry value. When you start Oracle SQL Developer and display the Database Connections dialog box, Oracle SQL Developer automatically imports any connections defined in the tnsnames.ora file on your system.

Creating a Database Connection



Connection Name: myconnection
Username: hr
Password: **
Save Password
Role: default
Connection Type: (●) Basic ( ) TNS ( ) Advanced
Hostname: vx0106.us.oracle.com
Port: 1521
(●) SID: orcl
( ) Service name
Status : Success

To create a database connection, perform the following steps:

1. On the Connections tabbed page, right-click Connections and select New Connection.

2. In the New/Select Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to.

   1. From the Role drop-down list, you can select either `default` or SYSDBA

   2. You can select the connection type as:

   - **Basic**: In this type, you enter the host name and system identifier (SID) for the database that you want to connect to. The Port is already set to 1521. Or, you can also enter the Service name directly if you are using a remote database connection.

   - **TNS**: You select any one of the database aliases imported from the `tnsnames.ora`

3. Click Test to make sure that the connection has been set correctly.

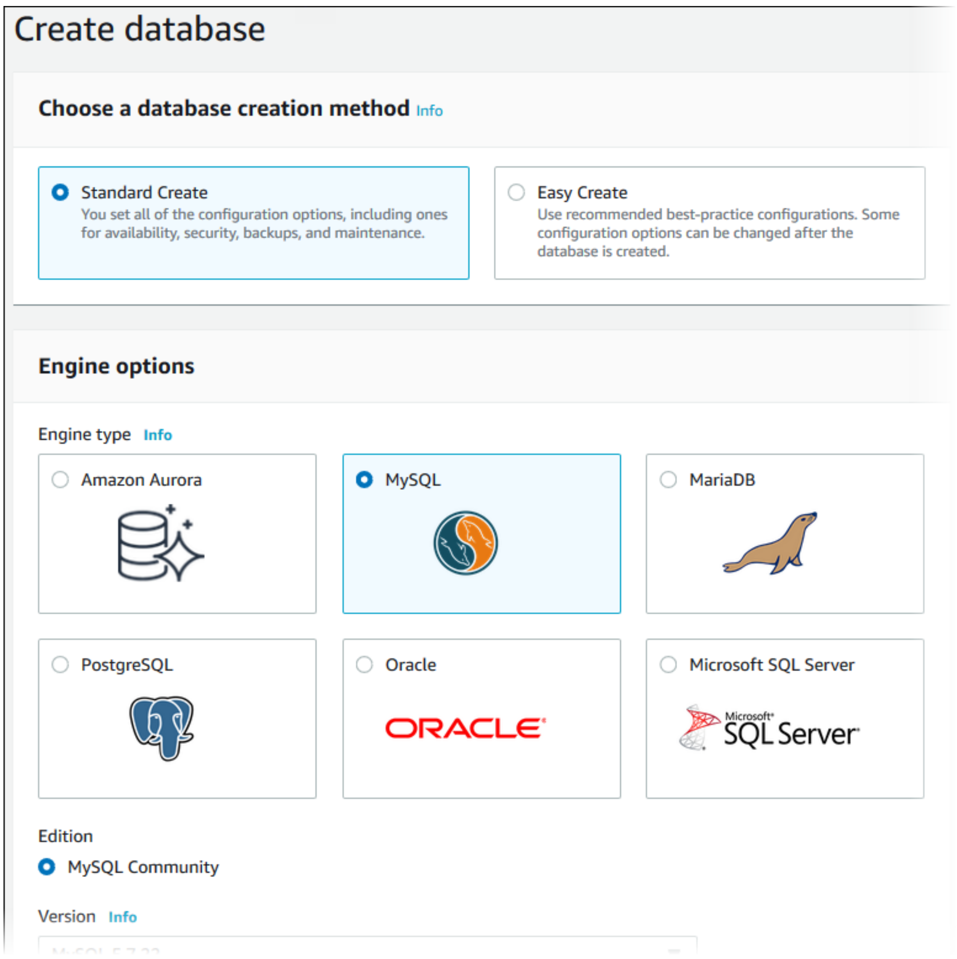
4. Click Connect.

Lesson 1.2: Oracle RDS

Oracle® Database is a relational database management system developed by Oracle. Amazon RDS makes it easy to set up, operate, and scale Oracle Database deployments in the cloud. With Amazon RDS, you can deploy multiple editions of Oracle Database in minutes with cost-efficient and re-sizable hardware capacity. Amazon RDS frees you up to focus on application development by managing time-consuming database administration tasks including provisioning, backups, software patching, monitoring, and hardware scaling.

To launch an Oracle DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region in which you want to create the DB instance. This example uses the US West (Oregon) Region.
3. In the navigation pane, choose **Databases**.
4. Choose **Create database**.
5. On the **Create database** page, shown following, make sure that the **Standard Create** option is chosen, and then choose **Oracle**.



Create database

Choose a database creation method Info

(●) Standard Create
You set all of the configuration options, including ones for availability, security, backups, and maintenance.

( ) Easy Create
Use recommended best-practice configurations. Some configuration options can be changed after the database is created.

Engine options

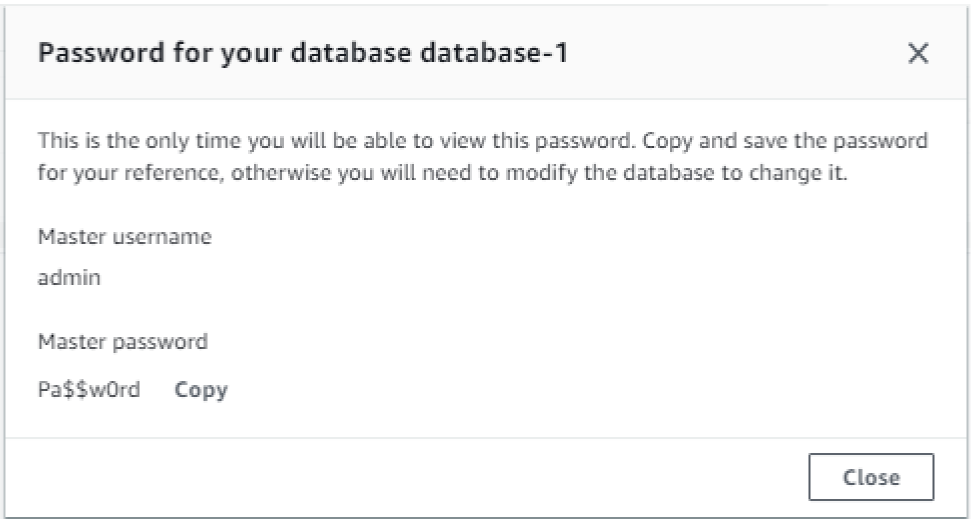
Engine type Info

( ) Amazon Aurora
(●) MySQL
( ) MariaDB
( ) PostgreSQL
( ) Oracle
( ) Microsoft SQL Server

Edition
(●) MySQL Community

Version Info

6. In the **Templates** section, choose **Dev/Test**.
7. In the **Settings** section, set these values:
   - **DB instance identifier** – `database1-db-instance`
   - **Master username** – Admin
   - **Auto generate a password** – Disable the option
   - **Master password** – Choose a password.
   - **Confirm password** – Retype the password.


Password for your database database-1

This is the only time you will be able to view this password. Copy and save the password for your reference, otherwise you will need to modify the database to change it.

Master username

admin

Master password

Pa$$w0rd Copy

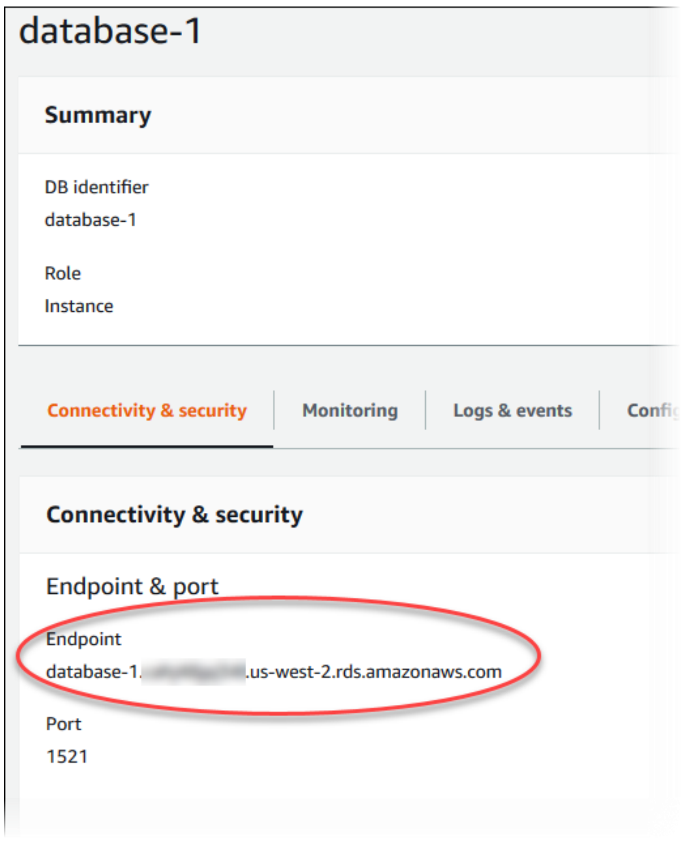
Close

8. In the **Templates** section, choose **Dev/Test**.
9. Choose required database setting and select **Create Database**

Lesson 1.3: Connecting to a DB Instance Running the Oracle Database Engine

Each Amazon RDS DB instance has an endpoint, and each endpoint has the DNS name and port number for the DB instance. To connect to your DB instance using a SQL client application, you need the DNS name and port number for your DB instance.

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the console, choose the AWS Region of your DB instance.
3. Find the DNS name and port number for your DB Instance.
    a. Choose **Databases** to display a list of your DB instances.
    b. Choose the Oracle DB instance name to display the instance details.
    c. On the **Connectivity & security** tab, copy the endpoint. Also, note the port number. You need both the endpoint and the port number to connect to the DB instance.


database-1

Summary

DB identifier
database-1

Role
Instance

Connectivity & security

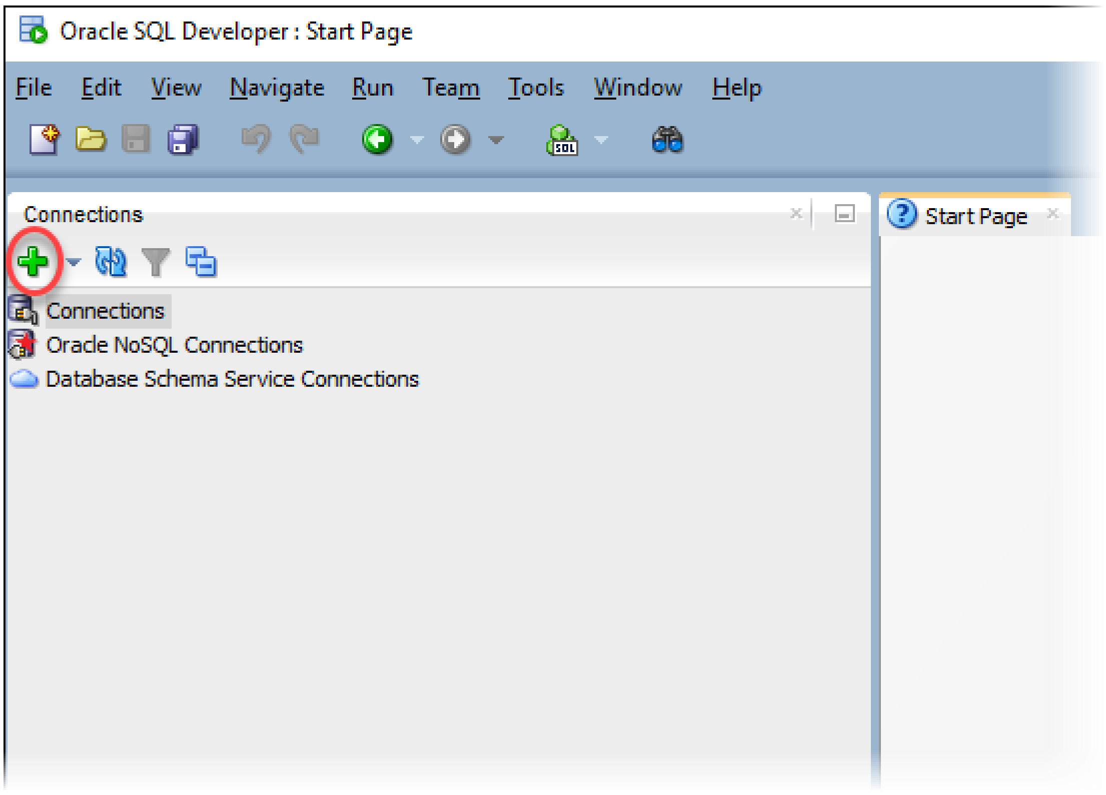
Endpoint & port

Endpoint
database-1.[redacted].us-west-2.rds.amazonaws.com

Port
1521

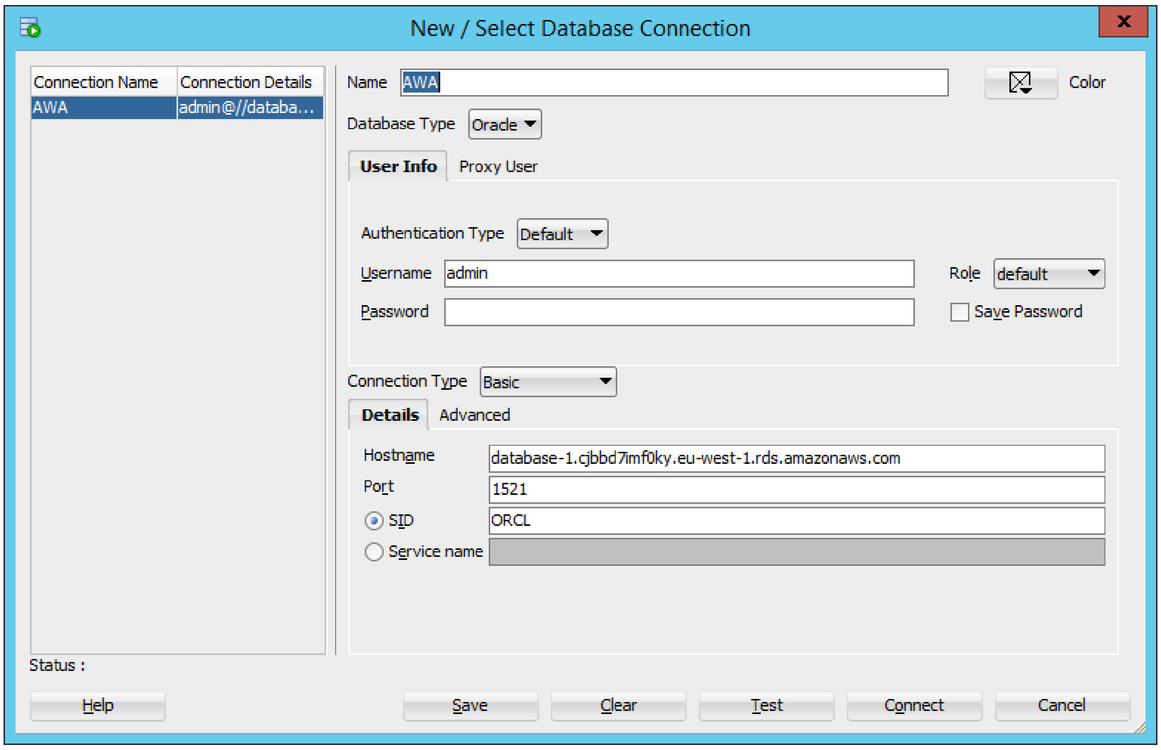
Connecting to Your DB Instance Using Oracle SQL Developer

1. Start Oracle SQL Developer.
2. On the **Connections** tab, choose the **add (+)** icon.



3. In the **New/Select Database Connection** dialog box, provide the information for your DB instance:
   - For **Connection Name**, enter a name that describes the connection, such as `Oracle-RDS`.
   - For **Username**, enter the name of the database administrator for the DB instance.
   - For **Password**, enter the password for the database administrator.
   - For **Hostname**, enter the DNS name of the DB instance.
   - For **Port**, enter the port number.
   - For **SID**, enter the Oracle database SID.

The completed dialog box should look similar to the following.


New / Select Database Connection

Connection Name	Connection Details
AWA	admin@//databa…

Name: AWA

Database Type: Oracle

User Info | Proxy User

Authentication Type: Default

Username: admin

Role: default

Password:

Save Password: ☐

Connection Type: Basic

Details | Advanced

Hostname: database-1.cjbbd7imf0ky.eu-west-1.rds.amazonaws.com

Port: 1521

SID: ORCL [✓]

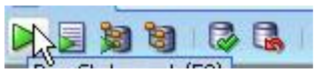
Service name:

Status :

4. Choose **Connect**.

Lesson 1.4 Writing queries SQL Developer

Once you have a database connection, you are ready to browse the schema, query and modify data.



You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

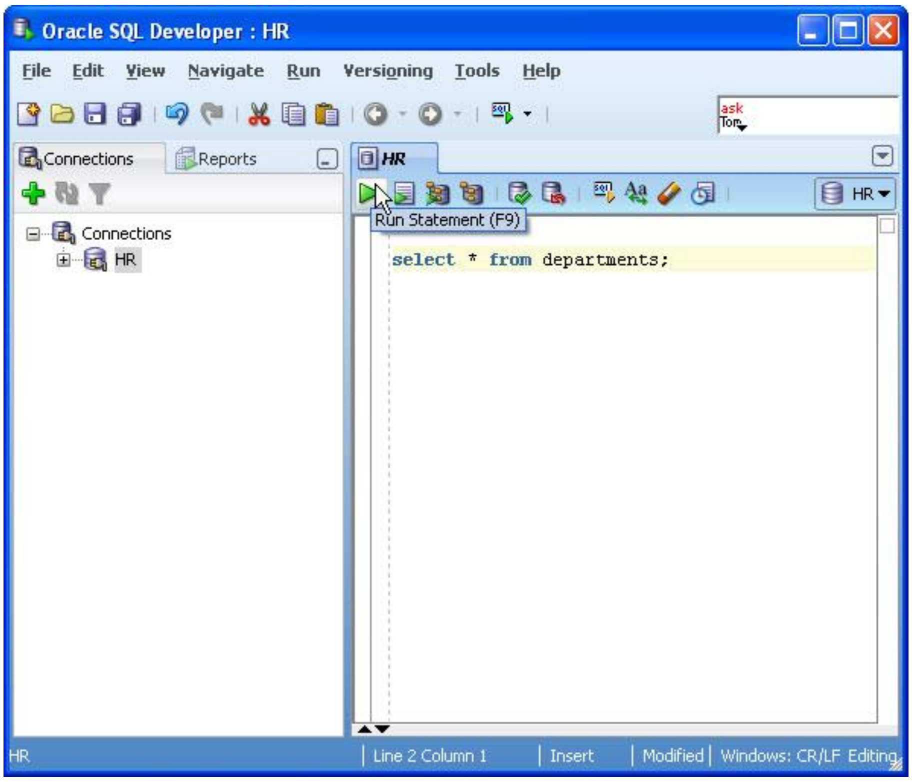
Execute Statement:	This executes the statement at the cursor in the Enter SQL Statement box. Alternatively, you can press [F9]. The output is generally shown in a formatted manner in the Results tab page.
Run Script:	This executes all statements in the Enter SQL Statement box using the Script Runner. The output is generally shown in the conventional script format in the Scripts tab page.
Commit:	This writes any changes to the database and ends the transaction.
Rollback:	This discards any changes to the database, without writing them to the database, and ends the transaction.

Query all the data in the DEPARTMENTS table. Enter the select statement below and click Execute Statement or F9.

SQL

```
SELECT * from departments;
```

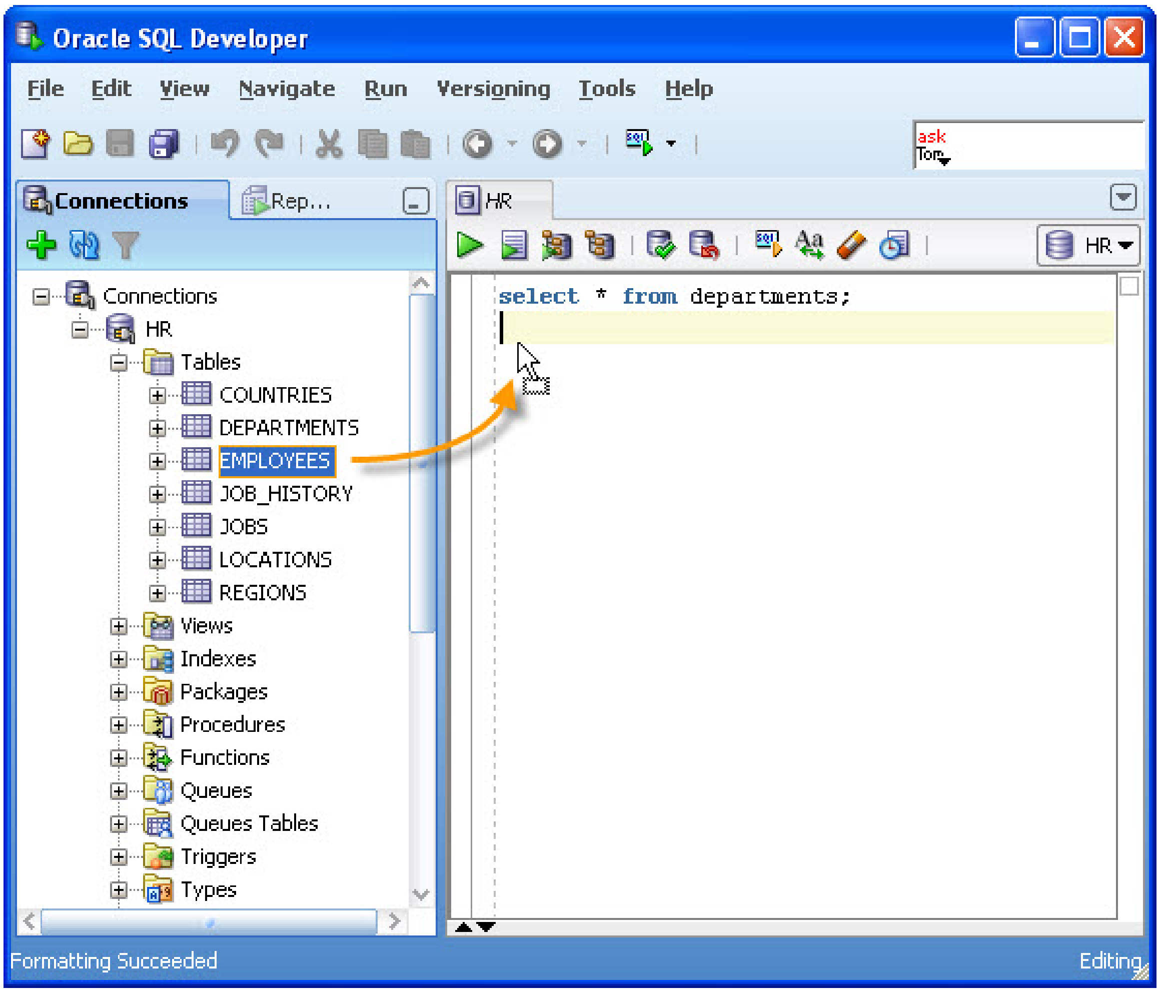
Query Results



```
select * from departments;
```

In the SQL Worksheet, you can use the Enter SQL Statement box to enter a single or multiple SQL statements. For a single statement, the semicolon at the end is optional. When you enter the statement, the SQL keywords are automatically highlighted. To execute a SQL statement, ensure that your cursor is within the statement and click the Execute Statement icon. Alternatively, you can press [F9]. To execute multiple SQL statements and see the results, click the Run Script icon. Alternatively, you can press [F5].

Instead of selecting all the columns from a table, you can itemize them, selecting only the data you require. Instead of typing in each column name, you can just drag the table name from the Connection Navigator. Expand the Tables node and drag the EMPLOYEES table onto the worksheet.



```
select * from departments;
```

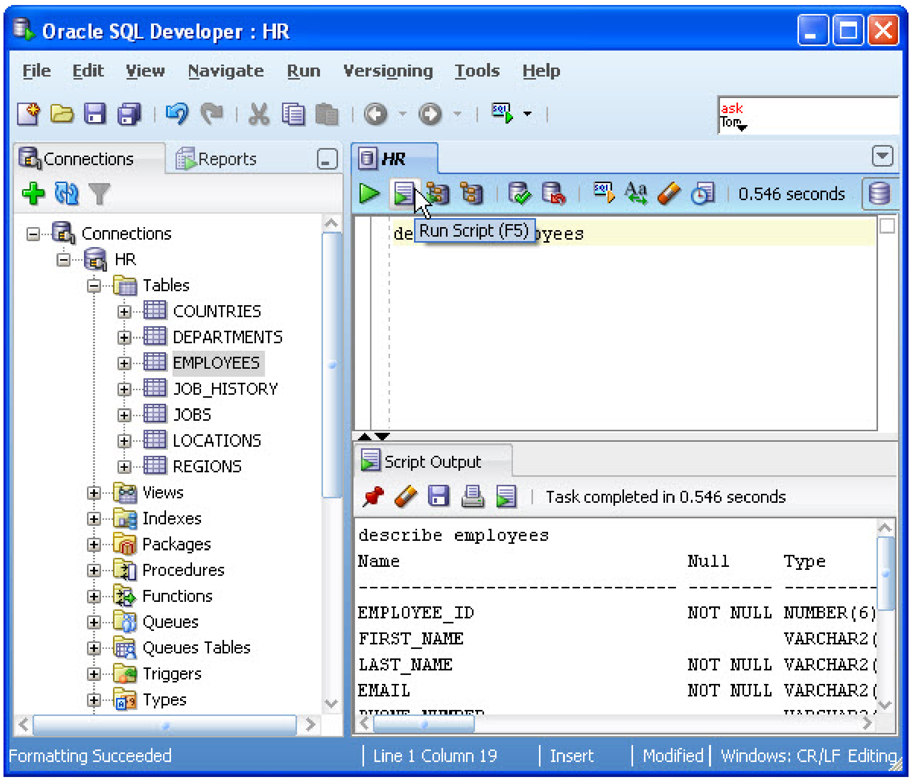
Use SQL*Plus commands

The SQL Worksheet allows you to use a selection of SQL*Plus commands. SQL*Plus commands have to be interpreted by the SQL Worksheet before being passed to the database. Some commands are not supported and are hence ignored and are not SENT to the Oracle database.

Instead of CLICKING on F9, select Run Script or F5.

SQL describe EMPLOYEES

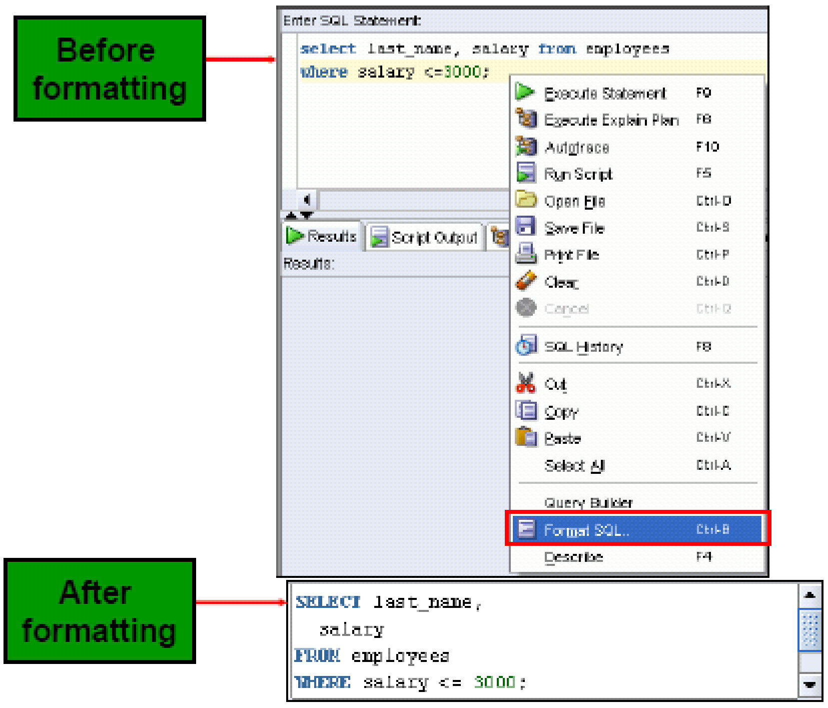
Query results



```
describe employees
Name                           Null     Type
------------------------------ -------- ---------
EMPLOYEE_ID                    NOT NULL NUMBER(6)
FIRST_NAME                              VARCHAR2(
LAST_NAME                      NOT NULL VARCHAR2(
EMAIL                          NOT NULL VARCHAR2(
```

Formatting the SQL Code

You may want to apply indentation, spacing, capitalization, and the line separation of the SQL code. Oracle SQL Developer has the feature for formatting the SQL code. To format the SQL code, right-click in the statement area, and select Format SQL.



```
select last_name, salary from employees
where salary <=3000;
```

```
SELECT last_name,
  salary
FROM employees
WHERE salary <= 3000;
```

Saving SQL Statements

1. From the File menu, select Save or Save As or [CTRL] + [S].

2. In the Save dialog box, enter the appropriate filename. Make sure the extension is `.sql` or the File type is selected as SQL Script (*.sql). Click Save.

3. The SQL Worksheet is renamed to the filename that you saved the script as. Make sure you do not enter any other SQL statements in the same worksheet. To continue with other SQL queries, open a new worksheet.

Section 2: Introduction to Oracle database

In this section you will learn:

- Discuss the basic design, theoretical, and physical aspects of a relational database
- Categorize the different types of SQL statements
- Describe the data set used by the course

Lesson 2.1: Discuss the basic design, theoretical, and physical aspects of a relational database

Relational and Object Relational Database Management Systems

The Oracle server supports both the relational and the object relational database models. The Oracle server extends the data-modeling capabilities to support an object relational database model that provides object-oriented programming, complex data types, complex business objects, and full compatibility with the relational world.

A **relational database** is a database that conforms to the relational model. A relational database stores data in a set of simple relations. A **relation** is a set of tuples. A **tuple** is an unordered set of **attribute** values.

The relational model has the following major aspects:

Structures

Primary Key

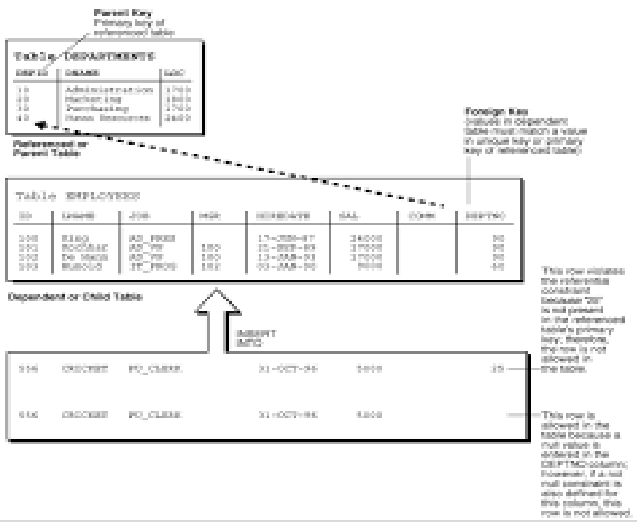
The primary key constraint uniquely identifies each record in a database table. Primary keys must contain UNIQUE values. A primary key column cannot contain NULL values.

Foreign Key

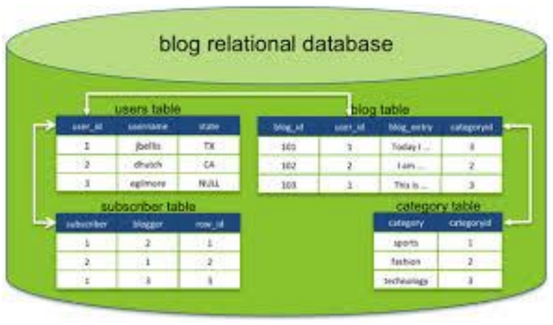
A foreign key in one table points to a primary key in another table.

Table

A **table** is a two-dimensional representation of a relation in the form of rows (tuples) and columns (attributes). Each row in a table has the same set of columns. A relational database is a database that stores data in relations (tables). For example, a relational database could store information about company employees in an employee table, a department table, and a salary table.



Relational database example:



user_id	username	state
1	jbellis	TX
2	dhutch	CA
3	egilmore	NULL

blog_id	user_id	blog_entry	categoryid
101	1	Today I ...	1
102	2	I am ...	2
103	1	This is ...	3

subscriber	blogger	row_id
1	2	1
2	1	2
1	3	3

category	categoryid
sports	1
fashion	2
technology	3

The relational model is the basis for a relational database management system (RDBMS). Essentially, an RDBMS moves data into a database, stores the data, and retrieves it so that it can be manipulated by applications. An RDBMS distinguishes between the following types of operations:

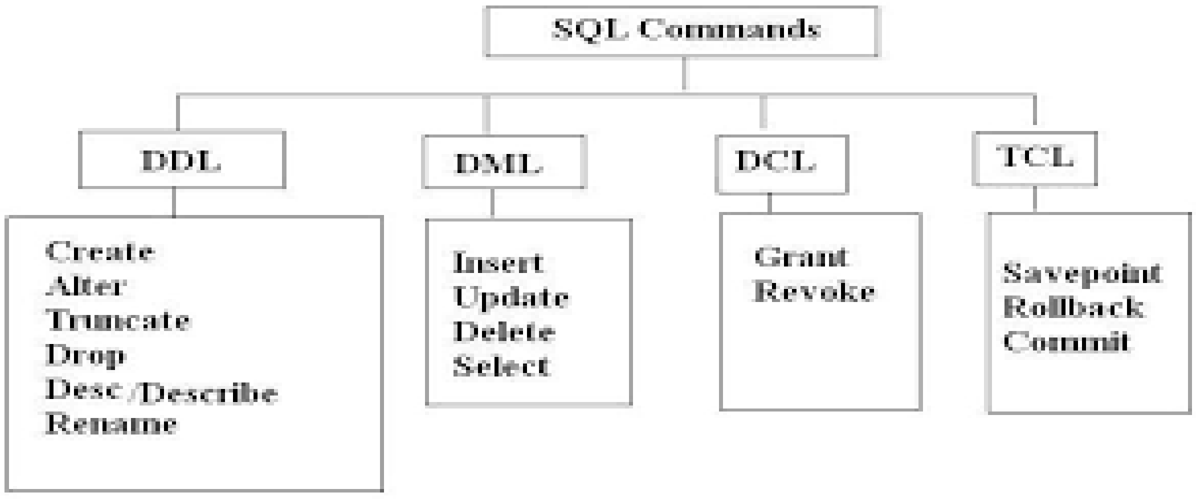
Logical operations

In this case, an application specifies *what* content is required. For example, an application requests an employee name or adds an employee record to a table.

Physical operations

In this case, the RDBMS determines *how* things should be done and carries out the operation. For example, after an application queries a table, the database may use an index to find the requested rows, read the data into memory, and perform many other steps before returning a result to the user. The RDBMS stores and retrieves data so that physical operations are transparent to database applications.

Structured Query Language (SQL)



SQL is a set-based declarative language that provides an interface to a RDBMS such as Oracle Database.

The SQL language is a very large language and can be split into sub languages.

Data Manipulation Language (DML)

The most commonly used language in SQL is the Data Manipulation Language. This is used to

- retrieve the data from the database
- create new data and modify existing data
- remove existing data

Later in this manual we shall return to the DML language.

Data Definition Language (DDL)

This sub language is used to:

- build new database structure i.e. tables
- alter existing structures
- delete existing structure
- give access to the data and security to logon to the database

Data Control Language (DCL)

This sub language is used to:

- Grant and revoke permissions

Transaction Control Language (DCL)

This sub language is used to:

- Control new transactions performed against data

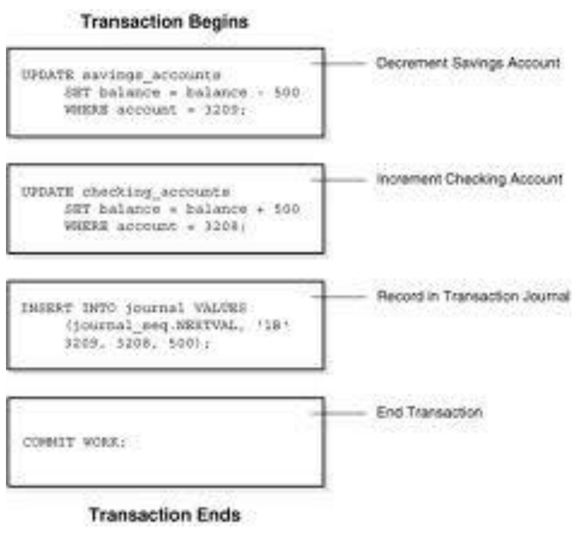
PL/SQL

PL/SQL is a procedural extension to Oracle SQL. PL/SQL is integrated with Oracle Database, enabling you to use all of the Oracle Database SQL statements, functions, and data types. You can use PL/SQL to control the flow of a SQL program, use variables, and write error-handling procedures

Transactions

An RDBMS must be able to group SQL statements so that they are either all committed, which means they are applied to the database, or all rolled back, which means they are undone. A transaction is a logical, atomic unit of work that contains one or more SQL statements.

An illustration of the need for transactions is a funds transfer from a savings account to a checking account.


Transaction Begins
Decrement Savings Account
Increment Checking Account
Record in Transaction Journal
End Transaction
Transaction Ends

```
UPDATE savings_accounts
    SET balance = balance - 500
    WHERE account = 3209;
```

```
UPDATE checking_accounts
    SET balance = balance + 500
    WHERE account = 3208;
```

```
INSERT INTO journal VALUES
    (journal_seq.NEXTVAL, '1B'
    3209, 3208, 500);
```

```
COMMIT WORK;
```

The transfer consists of the following separate operations:

- Decrease the savings account.
- Increase the checking account.
- Record the transaction in the transaction journal.

Oracle Database guarantees that all three operations succeed or fail as a unit. For example, if a hardware failure prevents a statement in the transaction from executing, then the other statements must be rolled back. The basic principle of a transaction is "all or nothing": an atomic operation succeeds or fails as a whole.

Data Concurrency

A requirement of a multiuser RDBMS is the control of concurrency, which is the simultaneous access of the same data by multiple users. Without concurrency controls, users could change data improperly, compromising data integrity. For example, one user could update a row while a different user simultaneously updates it.

If multiple users access the same data, then one way of managing concurrency is to make users wait. However, the goal of a DBMS is to reduce wait time, so it is either nonexistent or negligible. All SQL statements that modify data must proceed with as little interference as possible. Destructive interactions, which are interactions that incorrectly update data or alter underlying data structures, must be avoided.

Oracle Database uses locks to control concurrent access to data. A lock is a mechanism that prevents destructive interaction between transactions accessing a shared resource. Locks help ensure data integrity while allowing maximum concurrent access to data.

Data Consistency

In Oracle Database, each user must see a consistent view of the data, including visible changes made by a user's own transactions and committed transactions of other users. For example, the database must prevent dirty reads, which occur when one transaction sees uncommitted changes made by another concurrent transaction.

Oracle Database always enforces statement-level read consistency, which guarantees that the data returned by a single query is committed and consistent with respect to a single point in time. Depending on the transaction isolation level, this point is the time at which the statement was opened or the time the transaction began. The Flashback Query feature enables you to specify this point in time explicitly.

The database can also provide read consistency to all queries in a transaction, known as transaction-level read consistency. In this case, each statement in a transaction sees data from the same point in time, which is the time at which the transaction began.

Lesson 2.2: Overview to Oracle Architecture

A database server is the key to information management. In general, a server reliably manages a large amount of data in a multiuser environment so that users can concurrently access the same data. A database server also prevents unauthorized access and provides efficient solutions for failure recovery.

Database and Instance

An Oracle database server consists of a database and at least one database instance (commonly referred to as simply an instance). Because an instance and a database are so closely connected, the term Oracle database is sometimes used to refer to both instance and database. In the strictest sense the terms have the following meanings:

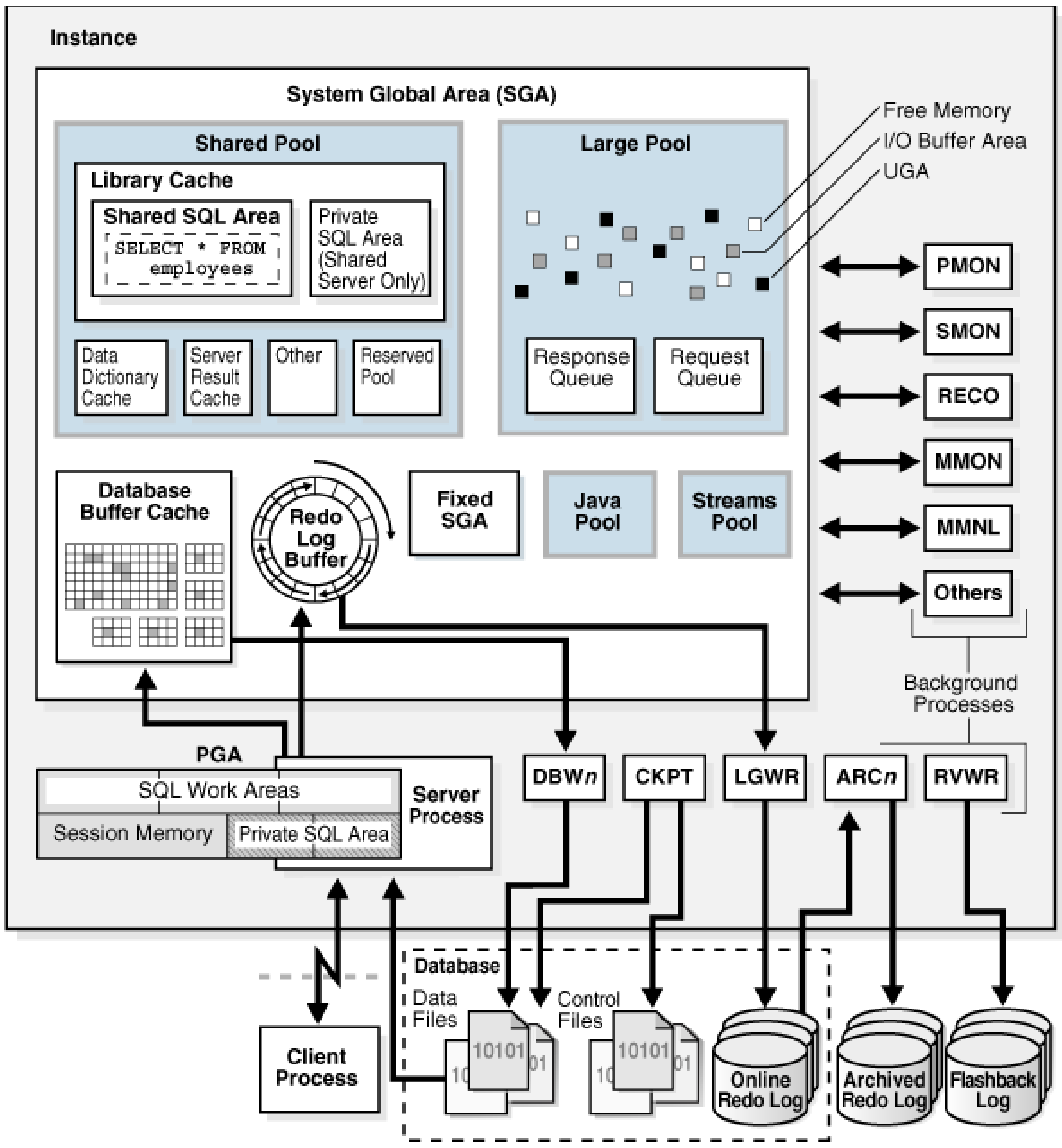
Database

A database is a set of files, located on disk, that store data. These files can exist independently of a database instance.

Database instance

An instance is a set of memory structures that manage database files. The instance consists of a shared memory area, called the system global area (SGA), and a set of background processes. An instance can exist independently of database files.

The diagram shows a database and its instance. For each user connection to the instance, the application is run by a client process. Each client process is associated with its own server process. The server process has its own private session memory, known as thee program global area (PGA).



The physical database structures are the files that store the data.

Data files

Every Oracle database has one or more physical data files, which contain all the database data. The data of logical database structures, such as tables and indexes, is physically stored in the data files.

Control files

Every Oracle database has a control file. A control file contains metadata specifying the physical structure of the database, including the database name and the names and locations of the database files.

Online redo log files

Every Oracle Database has an online redo log, which is a set of two or more online redo log files. An online redo log is made up of redo entries (also called **redo records**), which record all changes made to data.

Instance Memory Structures

Oracle Database creates and uses memory structures for purposes such as memory for program code, data shared among users, and private data areas for each connected user. The following memory structures are associated with an instance:

System Global Area (SGA)

The SGA is a group of shared memory structures that contain data and control information for one database instance. Examples of SGA components include cached data blocks and shared SQL areas.

Program Global Areas (PGA)

A PGA is a memory region that contains data and control information for a server or background process. Access to the PGA is exclusive to the process. Each server process and background process has its own PGA.

Overview of Schemas and Common Schema Objects

A schema is a collection of database objects. A schema is owned by a database user and has the same name as that user. Schema objects are the logical structures that directly refer to the database's data. Schema objects include structures like tables, views, and indexes

Some of the most common schema objects are defined in the following section.

Tables

Tables are the basic unit of data storage in an Oracle database. Database tables hold all user-accessible data. Each table has columns and rows.

Indexes

Indexes are optional structures associated with tables. Indexes can be created to increase the performance of data retrieval. Just as the index in a book helps you quickly locate specific information, an Oracle index provides an access path to table data. When processing a request, Oracle can use some or all of the available indexes to locate the requested rows efficiently. Indexes are useful when applications frequently query a table for a range of rows (for example, all employees with a salary greater than 1000 dollars) or a specific row.

Indexes are created on one or more columns of a table. After it is created, an index is automatically maintained and used by Oracle. Changes to table data (such as adding new rows, updating rows, or deleting rows) are automatically incorporated into all relevant indexes with complete transparency to the users.

Views

Views are customized presentations of data in one or more tables or other views. A view can also be considered a stored query. Views do not actually contain data. Rather, they derive their data from the tables on which they are based, referred to as the base tables of the views.

Like tables, views can be queried, updated, inserted into, and deleted from, with some restrictions. All operations performed on a view actually affect the base tables of the view.

Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table. They also hide data complexity and store complex queries.

Synonyms

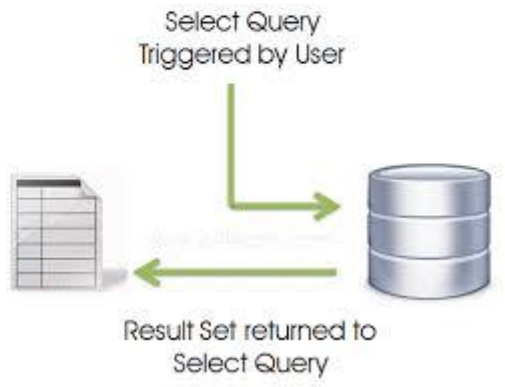
A synonym is an alias for any table, view, materialized view, sequence, procedure, function, package, type, Java class schema object, user-defined object type, or another synonym. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

Section 3: Retrieve Data using the SQL SELECT Statement

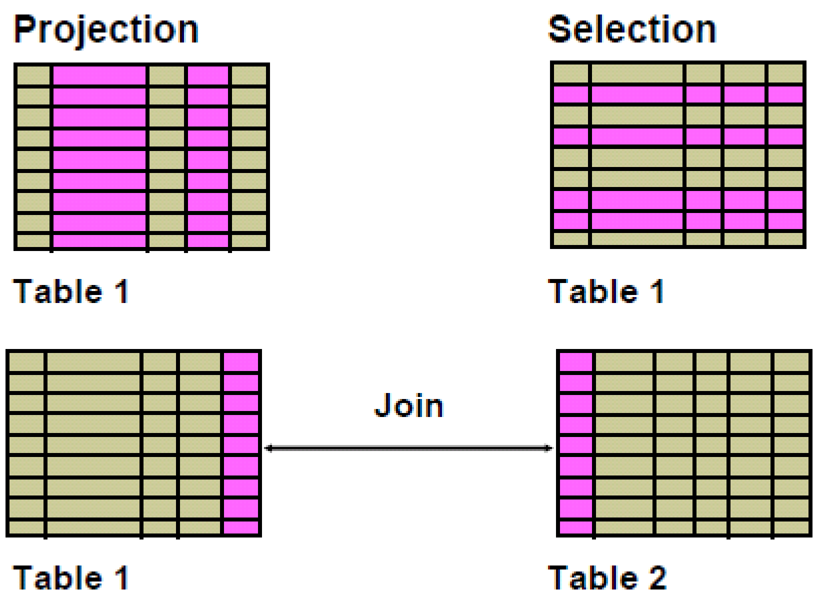
In this section you will learn:

- The capabilities of SQL SELECT statements
- Generate a report of data from the output of a basic SELECT statement
- Select All Columns
- Select Specific Columns
- Column Heading Defaults
- Arithmetic Operators
- Understand Operator Precedence
- DESCRIBE command to display the table structure
- Oracle Data dictionary

Lesson 3.1: Capabilities of SELECT Statement



The SELECT statement is used to retrieve data from the Oracle database. It is possible to reference either tables directly or indirectly using views in our SELECT statements. The SELECT statements are the most commonly used command in the SQL language, therefore they are an appropriate concept to cover right at the start of using the SQL language.



With a `SELECT` statement, you can use the following capabilities:

Projection

Select the columns in a table that are returned by a query. Select as few or as many of the columns as required.

Selection

Select the rows in a table that are returned by a query. Various criteria can be used or restrict the rows that are retrieved.

Joining

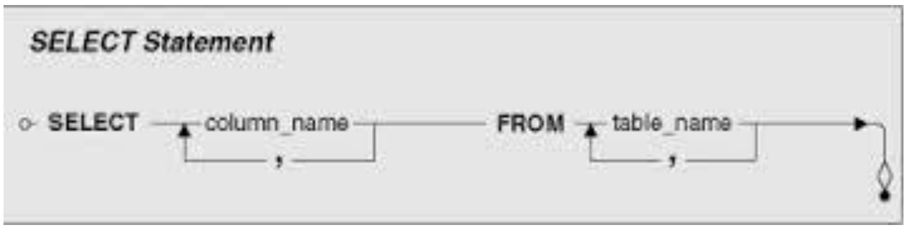
Bring together data that is stored in different tables by specifying the link between them.

Lesson 3.2: Basic SELECT statements

SELECT statements consist of several keywords or commands, some of these are mandatory while others are optional.

The SELECT statement require a minimum of 2 keywords

SELECT	Lists the columns or expressions to be returned
FROM	Lists the tables or views that the columns or expressions listed in the SELECT command.



Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table;
```

SQL Example:

```
SELECT      last_name,
            salary
FROM        employees;
```

Query Results:

```
LAST_NAME   SALARY
King        24000
Kochhar     17000
De Haan     17000
Hunold      9000
Ernst       6000
…
```

Explanation: This statement lists last_name and salary columns from the employees table.

Code file: Code3_2_1.sql

Note 1:

Throughout this course, the words `keyword`, `clause`, and `statement` are used as follows:

• A `keyword` refers to an individual SQL element.

For example, `SELECT` and `FROM` are keywords.

• A `clause` is a part of a SQL statement.

For example, `SELECT employee_id`, `last_name`, and so on is a clause.

• A *`statement`* is a combination of two or more clauses.

For example, `SELECT * FROM employees` is a SQL statement.

Note 2:

- SQL statements are not case-sensitive.
- SQL statements can be entered on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indents are used to enhance readability.
- In SQL Developer, SQL statements can optionally be terminated by a semicolon (;). Semicolon

Lesson 3.3: SELECT all columns

It is possible to select all of the columns rather than specifying individual columns using the "*".

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table;
```

SQL Example:

```
SELECT     *
FROM       employees;
```

Query Results:

```
EMPLOYEE_ID    FIRST_NAME    LAST_NAME    EMAIL
100            Steven        King         SKING
101            Neena         Kochhar      NKOCHHAR
102            Lex           De Haan      LDEHAAN
103            Alexander     Hunold       AHUNOLD
```

Explanation: This statement select all of the columns from the EMPLOYEES table

Code file: Code3_3_1.sql

Lesson 3.4: Column Aliases

By default the headings display by a select will display the column name or the expression. It is possible to provide an alias to each of the columns or expressions to replace the default heading

Syntax:

```
SELECT *|{[DISTINCT] column|expression [as][alias],...}
FROM table;
```

SQL Example:

```
SELECT     EMPLOYEE_ID as "Employee No",
           LAST_NAME as Name
FROM       employees;
```

Query Results:

```
Employee No         NAME
174                 Abel
166                 Ande
130                 Atkinson
105                 Austin
204                 Baer
…
```

Explanation: This statement aliases Employee_ID to Employee No.

Code file: Code3_4_1.sql

If we place the alias name in double quotes it is literal translated. This means that the alias could be more than one word and use different cases. If the alias is not in double quote it can only be one word or it would require an underscore to join multiple words.

Lesson 3.5: Retrieving Distinct Values

When building a SELECT statement it is possible to use the DISTINCT keyword to only return the unique values from a column.

Syntax:

```
SELECT DISTINCT  column_name
FROM             table or view;
```

SQL Example:

```
SELECT     DISTINCT JOB_ID
FROM       employees;
```

Query Results:

```
JOB_ID
AC_ACCOUNT
AC_MGR
AD_ASST
AD_PRES
AD_VP
…
```

Explanation: This statement selects the JOB_ID column from the EMPLOYEES table. The DISTINCT returns one instance of each job id

File Code3_5_1.sql

Lesson 3.6: Literal Strings

In the SQL language single quotes are more commonly used than double quotes. Single quotes are known as literal strings which mean the contents of the single quotes are literally translated. For example, if text is placed in single quotes in lower case it will be used as lower case in expressions. This is important when we are searching for data as we may search for a value using a literal string in lower case, but values required are held in the database in upper case. For example: if we search for 'Jones' but the database column hold 'JONES' we shall not find a match.

Literal strings are used with text and dates in SQL.

Syntax:

```
SELECT      column_or_expression1 [as] [alias]["alias"]

FROM        table or view;
```

SQL Example:

```
SELECT    last_name ||' is a '||job_id AS "Employee Details"
FROM      employees;
```

Query Results:

```
Employee Details
Abel is a SA_REP
Ande is a SA_REP
Atkinson is a ST_CLERK
Austin is a IT_PROG
 …….
```

Explanation: This statement returns the last name and salary for each employee are concatenated with a literal, to give the returned rows more meaning

File: Code3_5_1.sql

Lesson 3.7: Introduction to SQL Operators

In SQL statements we can use various operators to manipulate the data that is return.

Concatenation Operator

This operator allows columns or expression to attached together to create one string.

Arithmetic Operator

These operators allow basic arithmetic functionality to be performed. For example, multiplication, addition etc.

Logical Operators

These operators allow multiple conditions to be combined. For example, we may wish to display the name of employees would work in admin department and earn more than £12000. We use logical operators to combine the conditions of working in admin department and earning more than £12000.

Comparison Operators

These operators allow compare multiple values and return a Boolean result. For example, we may wish to show employees who have a job title of manager. We can use a comparison operator to check if the value of manager exists in the job title column.

Set Operators

These operators allow multiple data sets to be combined to create one data set. For example, we may have two sales tables one for the north and one for the south. The set operators can be used to combine the two sales tables to create one sales table.

Concatenation Operator

This operator allows columns or expression to attached together to create one string. You can link columns to other columns, arithmetic expressions, or constant values to create a character expression by using the concatenation operator (||). Columns on either side of the operator are combined to make a single output column.

Syntax:

```
SELECT   column|expression || column|expression   as [alias],...}
FROM table;
```

SQL Example:

```
SELECT    last_name ||' is a '||job_id AS "Employee Details"
FROM      employees;
```

Query Results:

```
Employee Details
Abel is a SA_REP
Ande is a SA_REP
Atkinson is a ST_CLERK
Austin is a IT_PROG
 …….
```

Explanation: This statement concatenates last_name and job_id columns in the employees table.

Code file: Code3_6_1.sql

Alternative Quote (q) Operator

Many SQL statements use character literals in expressions or conditions. If the literal itself contains a single quotation mark, you can use the quote (`q`) operator and select your own quotation mark delimiter. You can choose any convenient delimiter, single-byte or multibyte, or any of the following character pairs: [ ], { }, ( ), or < >.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table;
```

SQL Example:

```
SELECT department_name || ' Department' || q'['s Manager
Id: ]'|| manager_id
        AS "Department and Manager"
FROM departments;
```

Query Results:

```
Department and Manager
Administration Department's Manager Id: 200
Marketing Department's Manager Id: 201
Purchasing Department's Manager Id: 114
Human Resources Department's Manager Id: 203
Shipping Department's Manager Id: 121
…
```

Explanation: This statement shows the string contains a single quotation mark, which is normally interpreted as a delimiter of a character string. By using the `q` operator, however, brackets [] are used as the quotation mark delimiters. The string between the brackets delimiters is interpreted as a literal character string.

Code file: Code3_6_2.sql

Arithmetic Operator

These operators allow basic arithmetic functionality to be performed. For example, multiplication, addition etc.

There are 4 arithmetic operators:

Operator	Name
+	addition
-	subtraction
/	divide
*	multiplication

Syntax:

```
SELECT *|{[DISTINCT] column|expression * - / + [alias],...}
FROM table;
```

SQL Example:

```
SELECT    last_name ,(salary * 1.02)/12 as NewSalary
FROM      employees;
```

Query Results:

```
LAST_NAME  NEWSALARY
King       2040
Kochhar    1445
De Haan    1445
Hunold     765
Ernst      510
….
```

Explanation: This statement uses arithmetic operators to calculate an increased monthly salary for each employee.

Code file: Code3_6_3.sql

Operator Precedence

If an arithmetic expression contains more than one operator, multiplication and division are evaluated first. If operators in an expression are of the same priority, then evaluation is done from left to right. You can use parentheses to force the expression that is enclosed by the parentheses to be evaluated first.

Rules of Precedence:

- Multiplication and division occur before addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to override the default precedence or to clarify the statement.

Lesson 3.8: DESCRIBE statement

In SQL Developer, you can display the structure of a table by using the `DESCRIBE` command. The command displays the column names and the data types, and it shows you whether a column *must* contain data (that is, whether the column has a `NOT NULL` constraint).In the syntax, *table name* is the name of any existing table, view, or synonym that is accessible to the user.

Syntax `DESCRIBE` *table|View| synonym;*

SQL Example

```
DESCRIBE employees;
```

Query Results

```
DESCRIBE employees
Name              Null      Type
--------------  --------  ------------
EMPLOYEE_ID     NOT NULL NUMBER(6)
FIRST_NAME               VARCHAR2(20)
LAST_NAME       NOT NULL VARCHAR2(25)
EMAIL           NOT NULL VARCHAR2(25)
PHONE_NUMBER             VARCHAR2(20)
HIRE_DATE       NOT NULL DATE
JOB_ID          NOT NULL VARCHAR2(10)
SALARY                   NUMBER(8,2)
COMMISSION_PCT           NUMBER(2,2)
MANAGER_ID               NUMBER(6)
DEPARTMENT_ID            NUMBER(4)
```

Explanation: This statement displays the structure of the employees table.

Code file: Code3_7_1.sql

Lesson 3.9: Oracle Data Dictionary

The Oracle Database can contain many objects such as tables, views, indexes etc. It may be necessary for us to find out information about the object. Therefore, we have the Oracle Data Dictionary which contains metadata i.e. data about the Oracle database. There are many data dictionary tables or view and the number that we have access to will depend on our role. For example, in Oracle 9i a DBA may have access to approximately 1700 tables or views while a developer may have access to approximately 600 tables. The vast majority of these tables or views may never be used by DBA or Developers.

Most of the tables used will have the following prefixes:

- USER_
- ALL_
- DBA
- V$

USER_ tables or views show information about objects owned by the current user.

ALL_ tables or views show information about objects owned by the current user or that the current user has been given access to.

DBA_ table or views show information about all the object in the database.

V$ views are used by DBA's primarily as part of tuning the Oracle database.

DICT View

The DICT view shows a list of the all the table you are able to access. As many of the name may not be that easy to interpret a comments field with a description of the table or view is provided.

As there are many tables or views described in the DICT view it would be advisable to provide a where clause condition to restrict the data.

For example:

```
SELECT *
FROM dict
WHERE table_name LIKE '%TABLES%'
```

USER_TABLES : Description of the user's own relational tables

USER_OBJECT_TABLES :Description of the user's own object tables

USER_ALL_TABLES :Description of all object and relational tables owned by the all users

For example, the following shows all user tables:

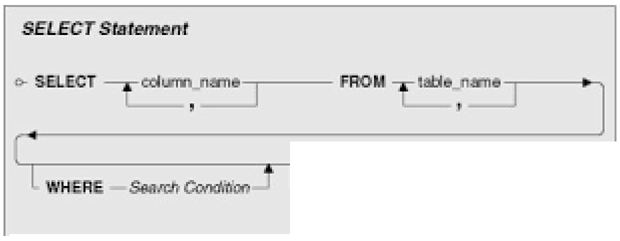
```
SELECT *
FROM USER_TABLES
```

Section 4: Restrict and Sort Data

- Write queries that contain a WHERE clause to limit the output retrieved
- List the comparison operators and logical operators that are used in a WHERE clause
- Describe the rules of precedence for comparison and logical operators
- Use character string literals in the WHERE clause
- Write queries that contain an ORDER BY clause to sort the output of a SELECT statement
- Sort output in descending and ascending order

Lesson 4.1 : WHERE clause

The WHERE Clause is one of the main conditions applied to a SQL statement. The WHERE clause is written after the FROM clause in the statement. You can restrict the rows that are returned from the query by using the WHERE clause. A WHERE clause contains a condition that must be met and it directly follows the FROM clause. If the condition is true, the row meeting the condition is returned.



The WHERE clause can compare values in columns, literal, arithmetic expressions, or functions. It consists of three elements:

1. Column name
2. Comparison condition
3. Column name, constant, or list of values

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     last_name, salary
FROM       employees
WHERE      DEPARTMENT_ID = 90;
```

Query Results:

```
LAST_NAME  SALARY
King       24000
Kochhar    17000
De Haan    17000
…
```

Explanation: This statement lists last_name and salary columns from the employees table for department ID 90.

Code file: Code4_1_1.sql

Lesson 4.2: Comparison operators and logical operators

Comparison operators are used to limit the data returned. Comparison operators are made against columns within the tables referenced in the FROM clause.

Operator	Name
=	equals
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
IS NULL	Is a null value
LIKE	Match a character pattern
IN(set)	Match any of a list of values
BETWEEN…AND…	Between two values (inclusive)
'<>', '!=' or '^='	Not Equal to

Equals

The equals '=' sign is used when there is an exact match between one condition and another

Syntax

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example

```
SELECT     last_name, salary
FROM       employees
WHERE      DEPARTMENT_ID = 90;
```

Query Results

```
LAST_NAME  SALARY
King       24000
Kochhar    17000
De Haan    17000
…
```

Explanation: This statement lists last_name and salary columns from the employees table for department ID 90.

Code file: Code4_2_1.sql

Not Equals

Not equals is used when searching for data does not equal a certain condition. This can be used for both text and numerical comparisons. The text match needs to be in the same case as the column value. The condition is can use these signs '<>', '!=' or '^='

Syntax

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example

```
SELECT     last_name, salary
FROM       employees
WHERE      DEPARTMENT_ID <> 90;
```

Query Results

```
LAST_NAME  SALARY
Hunold     9000
Ernst      6000
Austin     4800
Pataballa  4800
Lorentz    4200
…
```

Explanation: This statement lists last_name and salary columns from the employees table for department ID not equal 90.

Code file: Code4_2_2.sql

Greater Than

The greater than sign '>' is used to query back data which is above a numerical value.

Greater Than or Equal To

The greater than or equal to sign '>=' is used in the same way as greater than but will also compare with the condition value.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     last_name, salary
FROM       employees
WHERE      salary >=9000;
```

Query Results:

```
LAST_NAME  SALARY
King       24000
Kochhar    17000
De Haan    17000
Hunold     9000
Greenberg  12008
…
```

Explanation: This statement lists last_name and salary columns from the employees table for salary >=9000.

Code file: Code4_2_3.sql

Less Than

The less than sign '<' is used to query back data which is below a numerical value.

Less Than or Equal To

'<=' As above but compares the current value and any value below.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     last_name, salary
FROM       employees
WHERE      salary <=9000;
```

Query Results:

```
LAST_NAME  SALARY
Hunold     9000
Ernst      6000
Austin     4800
Pataballa  4800
Lorentz    4200
…
```

Explanation: This statement lists last_name and salary columns from the employees table for salary <=9000.

Code file: Code4_2_4.sql

IN

The IN command is used to compare multiple number of characters in one condition instead of using several equals conditions. The condition defined using the `IN` operator is also known as the *`membership condition`*.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     last_name, salary, department_id
FROM       employees
WHERE      DEPARTMENT_ID IN (90,70,10);
```

Query Results:

```
LAST_NAME   SALARY     DEPARTMENT_ID
Whalen      4400       10
Baer        10000      70
King        24000      90
Kochhar     17000      90
...
```

Explanation: This statement lists last_name and salary columns from the employees table for department ID 10, 70 and 90.

Code file: Code4_2_5.sql

NOT IN

This condition does the opposite of IN and brings back rows that do not satisfy the conditions made.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     last_name, salary, department_id
FROM       employees
WHERE      department_id NOT IN (90,70,10);
```

Query Results:

```
LAST_NAME   SALARY     DEPARTMENT_ID
Hunold      9000       60
```

```
Ernst      6000       60
Austin     4800       60
Pataballa  4800       60
Lorentz    4200       60
…
```

Explanation: This statement lists last_name and salary columns from the employees table for department ID not in 10, 70 and 90.

Code file: Code4_2_6.sql

BETWEEN

The between condition can be used for dates and numerical comparisons to return data that occurs between the two conditions.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     last_name, salary
FROM       employees
WHERE      salary BETWEEN 9000 AND 15000;
```

Query Results:

```
LAST_NAME  SALARY
Hunold     9000
Greenberg  12008
Faviet     9000
Raphaely   11000
Russell    14000
…
```

Explanation: This statement lists last_name and salary columns from the employees table for salary is in the range of 9000 to 15000.

Code file: Code4_2_7.sql

NOT BETWEEN

NOT BETWEEN will query for data that is not included in the between statement.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     last_name, salary
FROM       employees
WHERE      salary NOT BETWEEN 9000 AND 15000;
```

Query Results:

```
LAST_NAME  SALARY
King       24000
Kochhar    17000
De Haan    17000
Ernst      6000
```

```
Austin     4800
…
```

Explanation: This statement lists last_name and salary columns from the employees table for salary is in the range of 9000 to 15000.

Code file: Code4_2_8.sql

LIKE

The like condition is used to compare text fields. The like condition can be used with the wildcard character '%' or '_' . '%' represents 0 to more characters and the '_' represents individual characters.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     first_name, last_name
FROM       employees
WHERE      first_name like 'S%';
```

Query Results:

```
FIRST_NAME LAST_NAME
Sundar     Ande
Shelli     Baida
Sarah      Bell
Shelley    Higgins
Steven     King
…
```

Explanation: The `SELECT` statement returns the first name from the `EMPLOYEES` table for any employee whose first name begins with the letter "S." Note the uppercase "S." Consequently, names beginning with a lowercase "s" are not returned.

Code file: Code4_2_9.sql

The `LIKE` operator can be used as a shortcut for some `BETWEEN` comparisons.

Syntax

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example

```
SELECT     last_name, hire_date
FROM       employees
WHERE      hire_date LIKE '%02';
```

Query Results

```
LAST_NAME  HIRE_DATE
Greenberg  17-AUG-02
Faviet     16-AUG-02
Raphaely   07-DEC-02
Mavris     07-JUN-02
Baer       07-JUN-02
…
```

Explanation: This statement lists last_name and salary columns from the employees table.

Code file: Code4_2_10.sql

NOT LIKE

The condition 'NOT LIKE' will return the every except the matches within the condition.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     first_name, last_name
FROM       employees
WHERE      first_name NOT LIKE 'S%';
```

Query Results:

```
FIRST_NAME LAST_NAME
Ellen      Abel
Mozhe      Atkinson
David      Austin
Hermann    Baer
Amit       Banda
…
```

Explanation: The `SELECT` statement returns the first name from the `EMPLOYEES` table for any employee whose first name does not begin with the letter S

Code file: Code4_2_11a.sql

Combining Wildcard Characters

The `%` and `_` symbols can be used in any combination with literal characters.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     first_name, last_name
FROM       employees
WHERE      last_name  LIKE '_o%';
```

Query Results:

```
FIRST_NAME LAST_NAME
Karen      Colmenares
Louise     Doran
Tayler     Fox
Charles    Johnson
Vance      Jones
…
```

Explanation: The `SELECT` statement returns the first name from the `EMPLOYEES` table for any employee whose last name first letter is anything, second letter is o and has anything after it

Code file: Code4_2_11b.sql

IS NULL

The 'IS NULL' condition will bring back data where the relevant column referenced in the WHERE clause is null. The NULL conditions include the IS NULL condition and the IS NOT NULL condition. The IS NULL condition tests for nulls. A null value means that the value is unavailable, unassigned, unknown, or inapplicable. Therefore, you cannot test with =, because a null cannot be equal or unequal to any value.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     first_name, last_name, commission_pct
FROM       employees
WHERE      commission_pct IS NULL;
```

Query Results:

```
FIRST_NAME LAST_NAME  COMMISSION_PCT
Steven     King
Neena      Kochhar
Lex        De Haan
Alexander  Hunold
…
```

Explanation: The `SELECT` statement returns the first name, last name, and commission for all employees who are `not` entitled to receive a commission

Code file: Code4_2_12a.sql

IS NOT NULL

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     first_name, last_name, commission_pct
FROM       employees
WHERE      commission_pct IS NOT NULL;
```

Query Results:

```
FIRST_NAME LAST_NAME  COMMISSION_PCT
John       Russell    0.4
Karen      Partners   0.3
Alberto    Errazuriz  0.3
Gerald     Cambrault  0.3
…
```

Explanation: The `SELECT` statement returns the first name, last name and commission for all employees who are entitled to receive a commission

Code file: Code4_2_12b.sql

Logical Operators

To create multiple conditions within a WHERE clause a logical operator is required to separate the conditions. Logical operators consist of AND, OR and NOT. The result of each condition is either TRUE or FALSE depending if the condition is satisfied.

Operator	Name
AND	Returns `TRUE` if *both* component conditions are true
OR	Returns `TRUE` if *either* component condition is true
NOT	Returns `TRUE` if the condition is false

AND

The AND operator is the most used operator. The AND operator allows the user to add as many conditions to a query as they want in a sequence.

AND Truth Table

The following table shows the results of combining two expressions with `AND`:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     first_name, last_name, commission_pct, salary
FROM       employees
WHERE      commission_pct IS NULL
AND salary >=10000;
```

Query Results:

```
FIRST_NAME LAST_NAME  COMMISSION_PCT   SALARY
Steven     King                        24000
Neena      Kochhar                     17000
Lex        De Haan                     17000
Nancy      Greenberg                   12008
…
```

Explanation: The `SELECT` statement returns the first name, last name, commission and salary for all employees who are `not` entitled to receive a commission and have a salary over 10000

Code file: Code4_2_13a.sql

OR

OR Truth Table

The following table shows the results of combining two expressions with `OR`:

OR	**TRUE**	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Syntax:

```
SELECT     *|{[DISTINCT] column|expression [alias],...}
FROM       table
[WHERE     condition(s)];
```

SQL Example:

```
SELECT     first_name, last_name, commission_pct, salary
FROM       employees
WHERE      commission_pct IS NULL
OR         salary >=10000;
```

Query Results:

```
FIRST_NAME LAST_NAME  COMMISSION_PCT   SALARY
Steven     King                        24000
Neena      Kochhar                     17000
Lex        De Haan                     17000
Alexander  Hunold                      9000
Bruce      Ernst                       6000
…
```

Explanation: The `SELECT` statement returns the first name, last name, commission and salary for all employees who are *not* entitled to receive a commission or have a salary over 10000

Code file: Code4_2_13b.sql

NOT

As the name suggest the NOT operator returns anything that does not equal the conditions.

NOT Truth Table

The following table shows the result of applying the NOT operator to a condition:

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

Note: The NOT operator can also be used with other SQL operators, such as BETWEEN, LIKE, and NULL.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     first_name, last_name, commission_pct, salary
FROM       employees
WHERE      last_name NOT LIKE '%A%';
```

Query Results:

```
FIRST_NAME LAST_NAME  COMMISSION_PCT   SALARY
Steven     King                        24000
Neena      Kochhar                     17000
Lex        De Haan                     17000
Alexander  Hunold                      9000
Bruce      Ernst                       6000
…
```

Explanation: The SELECT statement returns the first name, last name, commission and salary for all employees whose last name does not start with A

Code file: Code4_2_14.sql

Lesson 4.3: Rules of precedence for comparison and logical operators

SQL allows the user to use AND, OR and NOT as often as required. The use of brackets '(' ')' allows the user to organise the conditions in different ways with different results. The rules of precedence determine the order in which expressions are evaluated and calculated.

Rules of precedence

Operator	Name
1	Arithmetic operators
2	Concatenation operator
3	Comparison conditions
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	Not equal to
7	`NOT` logical condition
8	AND logical condition
9	`OR` logical condition

Rules of precedence

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

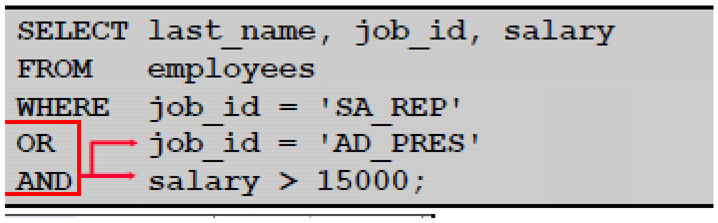
SQL Example:

```
SELECT     last_name, job_id, salary
FROM       employees
WHERE      job_id = 'SA_REP'
OR         job_id = 'AD_PRES'
AND        salary > 15000;
```

Query Results:

```
LAST_NAME  JOB_ID     SALARY
King       AD_PRES    24000
Tucker     SA_REP     10000
Bernstein  SA_REP     9500
Hall       SA_REP     9000
Olsen      SA_REP     8000
…
```

Explanation:



```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id = 'SA_REP'
OR     job_id = 'AD_PRES'
AND    salary > 15000;
```

The first condition is that the job ID is AD_PRES and the salary is greater than 15,000. The second condition is that the job ID is SA_REP. Therefore, the SELECT statement reads as follows:

"Select the row if an employee is a president and earns more than 15,000, or if the employee is a sales representative."

Code file: Code4_2_15.sql

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

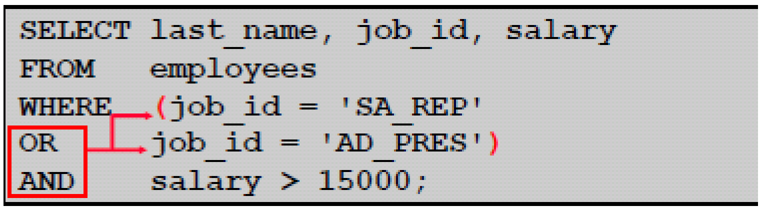
SQL Example:

```
SELECT     last_name, job_id, salary
FROM       employees
WHERE      (job_id = 'SA_REP'
OR         job_id = 'AD_PRES')
AND        salary > 15000;
```

Query Results:

```
LAST_NAME  JOB_ID     SALARY
King       AD_PRES    24000
```

Explanation:



```
SELECT last_name, job_id, salary
FROM   employees
WHERE  (job_id = 'SA_REP'
OR     job_id = 'AD_PRES')
AND    salary > 15000;
```

The first condition is that the job ID is AD_PRES or SA_REP. The second condition is that the salary is greater than 15,000. Therefore, the SELECT statement reads as follows:

"Select the row if an employee is a president or a sales representative, and if the employee earns more than 15,000."

Code file: Code4_2_16.sql

Lesson 4.4: Character string literals

Character strings `WHERE` clause must be enclosed with single quotation marks (''). Number constants, however, should not be enclosed with single quotation marks. All character searches are case-sensitive. Use of singlerow

Functions UPPER and LOWER to override the case sensitivity

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     first_name, last_name, commission_pct, salary
FROM       employees
WHERE      last_name ='King';
```

Query Results:

```
FIRST_NAME LAST_NAME  COMMISSION_PCT   SALARY
Janette    King       0.35             10000
Steven     King                        24000
```

Explanation: The `SELECT` statement returns the first name, last name, commission and salary for all employees whose last name is King

Code file: Code4_4_1.sql

Lesson 4.5: Date queries

Dates in the `WHERE` clause must be enclosed with single quotation marks (`''`). Oracle databases store dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds. The default date display is in the `DD-MON-RR` format.

To check the default date format in effect for your session at any given time, issue the following query against the NLS_SESSION_PARAMETERS data dictionary view:

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT *
FROM nls_session_parameters;
```

Query Results:

```
NLS_LANGUAGE                AMERICAN
NLS_TERRITORY               AMERICA
NLS_CURRENCY                $
NLS_ISO_CURRENCY            AMERICA
NLS_NUMERIC_CHARACTERS      .,
NLS_CALENDAR                GREGORIAN
NLS_DATE_FORMAT             DD-MON-RR
NLS_DATE_LANGUAGE           AMERICAN
…
```

Explanation: The `SELECT` statement returns the default date format in effect for your session at any given time

Code file:

Use the ALTER SESSION command to specify a session-level default date format. The following example works in Oracle8*i* or higher, and sets the default date format to DD-MON-YYYY:

```
ALTER SESSION SET NLS_DATE_FORMAT=”DD-MON-YYYY”
```

Syntax: SELECT *|{[DISTINCT] *column|expression* [*alias*],...}
FROM *table*
[WHERE *condition(s)*];

SQL Example:

```
SELECT first_name, last_name, salary, hire_date
FROM employees
WHERE hire_date BETWEEN '01-JAN-2005' AND '31-DEC-2005';
```

Query Results:

```
FIRST_NAME LAST_NAME  SALARY     HIRE_DATE
Neena      Kochhar    17000      21-SEP-05
David      Austin     4800       25-JUN-05
John       Chen       8200       28-SEP-05
Ismael     Sciarra    7700       30-SEP-05
Shelli     Baida      2900       24-DEC-05…
```

Explanation: The SELECT statement returns the first name, last name, salary and hire date for all employees whose hire date was in 2005

Code file: Code4_5_1.sql

Lesson 4.6 : Use of functions in the WHERE clause and performance considerations

Singlerow functions can be used in the WHERE clause.

Note:

The use of functions in the WHERE clause can have a negative impact on performance as the index may not be used.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT first_name, last_name, commission_pct, salary
FROM employees
WHERE UPPER(last_name) ='KING';
```

Query Results:

```
FIRST_NAME LAST_NAME  COMMISSION_PCT   SALARY
Janette    King       0.35             10000
Steven     King                        24000
```

Explanation: The `SELECT` statement returns the first name, last name, commission and salary for all employees whose last name is King. The UPPER function override the case sensitivity.

Code file: Code4_6_1.sql

Lesson 4.7: ORDER BY clause

ORDER BY clause allows the user to sort the data into an appropriate order. The ORDER BY clause does not require that the column which the data is to be sorted by is included in the SELECT list.

The syntax for ORDER BY is

```
ORDER BY {column, expr, numeric_position} [ASC|DESC]]
```

The ASC and DESC refer to whether the query returns the sorted column in Ascending or Descending order.

Syntax:

```
SELECT expr
FROM table
[WHERE condition(s)]
[ORDER BY {column, expr, numeric_position} [ASC|DESC]];
```

SQL Example:

```
SELECT      first_name, last_name,  department_id
FROM        employees
WHERE       last_name NOT LIKE '%A%'
ORDER BY    department_id, last_name, first_name;
```

Query Results:

```
FIRST_NAME LAST_NAME  DEPARTMENT_ID
Jennifer   Whalen     10
Pat        Fay        20
Michael    Hartstein  20
Shelli     Baida      30
Karen      Colmenares 30
…
```

Explanation: The `SELECT` statement returns the first name, last name and department id for all employees whose last name does not start with A. The resultset is sorted by department id then by last name and then by first name.

Code file: Code4_7_1.sql

Sorting by using the column’s numeric position

Syntax:

```
SELECT expr
FROM table
[WHERE condition(s)]
[ORDER BY {column, expr, numeric_position} [ASC|DESC]];
```

SQL Example:

```
SELECT first_name, last_name,  department_id
FROM employees
WHERE last_name NOT LIKE '%A%'
ORDER BY 3, 2, 1;
```

Query Results:

```
FIRST_NAME LAST_NAME  DEPARTMENT_ID
Jennifer   Whalen     10
Pat        Fay        20
Michael    Hartstein  20
Shelli     Baida      30
Karen      Colmenares 30

…
```

Explanation: The `SELECT` statement returns the first name, last name and department id for all employees whose last name does not start with A. The resultset is sorted by department id then by last name and then by first name.

Code file: Code4_7_2.sql

Sorting with NULLS

It is possible sort data on a column that contain null values. If the sort is in ascending order the null values are placed at the end of the result set. If the sort is in descending order the null values are placed at the top of the result set.

Syntax:

```
SELECT expr
FROM table
[WHERE condition(s)]
[ORDER BY {column, expr, numeric_position} [ASC|DESC]];
```

SQL Example:

```
SELECT first_name, last_name,  department_id, commission_pct
FROM       employees
WHERE      last_name  LIKE '%A%'
ORDER BY   commission_pct;
```

Query Results:

```
FIRST_NAME LAST_NAME  DEPARTMENT_ID    COMMISSION_PCT
Sundar     Ande       80               0.1
Ellen      Abel       80               0.3
David      Austin     60
Mozhe      Atkinson   50

…
```

Explanation: The `SELECT` statement returns the first name, last name and department id for all employees whose last name starts with A. The resultset is sorted by commission_pct. Nulls are at the end of the resultset

Code file: Code4_7_3.sql

Lesson 4.8: Substitution Variables

By using a substitution variable in place of the exact values in the WHERE clause, you can run the same query for different values.

You can create reports that prompt users to supply their own values to restrict the range of data returned, by using substitution variables. You can embed substitution variables in a command file or in a single SQL statement. A variable can be thought of as a container in which values are temporarily stored. When the statement is run, the stored value is substituted.

You can use single-ampersand (`&`) substitution variables to temporarily store values. You can also predefine variables by using the `DEFINE` command. `DEFINE` creates and assigns a value to a variable.

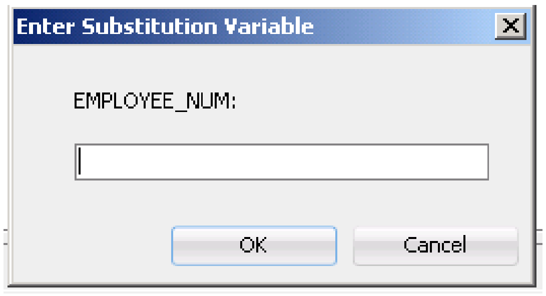
Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     employee_id, last_name, salary, department_id
FROM       employees
WHERE      employee_id = &employee_num ;
```

Query Results:


Enter Substitution Variable

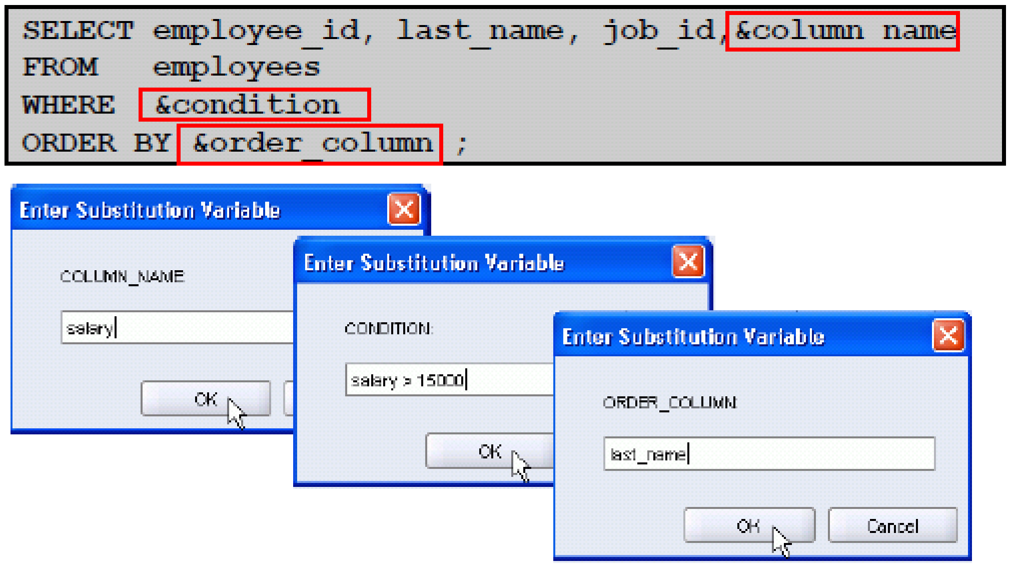
EMPLOYEE_NUM:

Explanation: The `SELECT` statement prompts the user for an employee number

Code file: Code4_8_1.sql

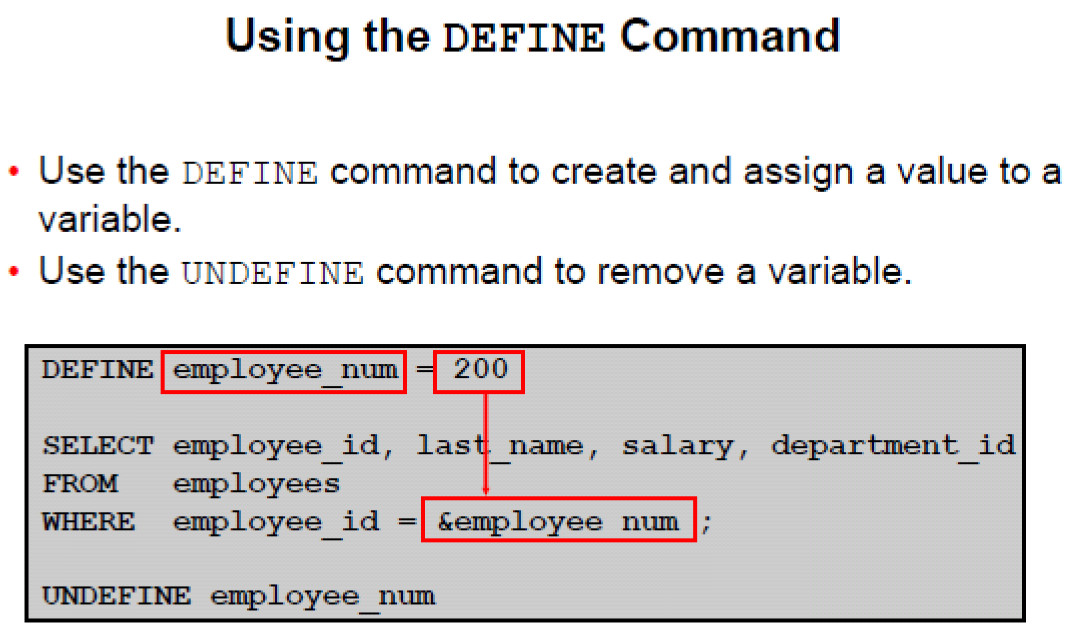
Other uses of Substitution Variables

You can use the substitution variables not only in the `WHERE` clause of a SQL statement, but also as substitution for column names, expressions, or text.



```
SELECT employee_id, last_name, job_id,&column_name
FROM   employees
WHERE  &condition
ORDER BY &order_column ;
```

Enter Substitution Variable
COLUMN_NAME
salary
OK
Enter Substitution Variable
CONDITION:
salary > 15000
OK
Enter Substitution Variable
ORDER_COLUMN
last_name
OK
Cancel



Using the DEFINE Command

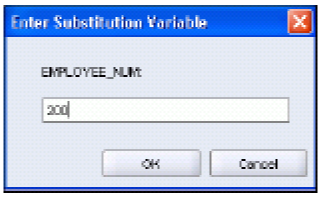
- Use the `DEFINE` command to create and assign a value to a variable.
- Use the `UNDEFINE` command to remove a variable.

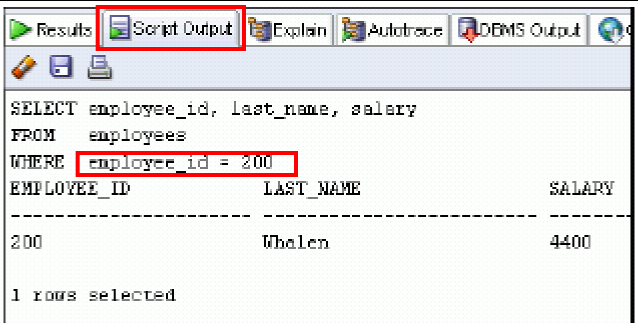
```
DEFINE employee_num = 200

SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num ;

UNDEFINE employee_num
```

Using the `VERIFY` Command

Use the `VERIFY` command to toggle the display of the substitution variable, both before and after SQL Developer replaces substitution variables with values:

```
SET VERIFY ON
SELECT employee_id, last_name, salary
FROM   employees
WHERE  employee_id = &employee_num;
```





```
SELECT employee_id, last_name, salary
FROM   employees
WHERE  employee_id = 200
EMPLOYEE_ID            LAST_NAME                 SALARY
---------------------- ------------------------- --------
200                    Whalen                    4400

1 rows selected
```

Section 5: Single-Row Functions

- Describe the differences between single row and multiple row functions
- Manipulate strings with character function in the SELECT and WHERE clauses
- Manipulate numbers with the ROUND, TRUNC, and MOD functions
- Perform arithmetic with date data
- Manipulate dates with the DATE functions

Lesson 5.1: Single row and multiple row functions

Functions are Oracle PL/SQL objects that are created by Oracle and included in the SQL language. They will return a result that can be used in the SQL statement. Functions can accept arguments which are used to produce a value to be returned.

It is possible to use SQL functions in SQL and also PL/SQL statements to manipulate and use resultant data.

There are two types of functions:

- Single-row functions
- Multiple-row functions

Single row functions

Single row functions have the following features

- Manipulate data items
- Accept arguments and return one value
- Act on each row that is returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments that can be a column or an expression
- Can be used in WHERE clause and SELECT clause

Syntax

```
function_name [(arg1, arg2,...)]
```

Single row functions can be categories as follows

- STRING (Character) Functions
- DATE functions
- NULL functions
- NUMERIC functions
- CONVERSION functions
- MISCELLANEOUS functions

Multiple-Row Functions

Functions can manipulate groups of rows to give one result per group of rows. These functions are also known as *`group functions`*

Lesson 5.2: Usage of functions in the SELECT and WHERE clauses

Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row that is returned by the query. Can be used in WHERE clause and SELECT clause

Syntax:

```
function_name [(arg1, arg2,...)]
```

SQL Example:

```
SELECT  first_name,  UPPER(last_name),  commission_pct,
salary
FROM employees
WHERE UPPER(last_name) ='KING';
```

Query Results:

```
FIRST_NAME UPPER(LAST_NAME) COMMISSION_PCT  SALARY
Steven     KING                             24000
Janette    KING             0.35            10000
…
```

Explanation: The `SELECT` statement returns the first name, last name, commission and salary for all employees whose last name is King. The UPPER function overrides the case sensitivity.

Code file: Code5_2_1.sql

Lesson 5.3: Manipulate strings with character functions

Function	Description	Example	Result
CONCAT	Joins values together	`SELECT CONCAT('Hello', 'World') FROM DUAL`	`HelloWorld`
SUBSTR	Extracts a string of determined length	`SELECT SUBSTR('HelloWorld',1,5) FROM DUAL`	Hello
LENGTH	Shows the length of a string as a numeric value	`SELECT LENGTH('HelloWorld') FROM DUAL`	10
INSTR	Finds the numeric position of a named character	`SELECT INSTR('HelloWorld', 'W') FROM DUAL`	6
LPAD	Returns an expression left-padded to the length of *n* characters with a character expression	`SELECT LPAD(14000,10,'*') FROM DUAL`	`*****14000`
RPAD	Returns an expression right-padded to the length of *n* characters with a character expression	`SELECT RPAD(14000, 10, '*') FROM DUAL`	`14000*****`
REPLACE	Replace a string with another string	`SELECT REPLACE ('JACK and JUE','J','BL') FROM DUAL`	`BLACK and BLUE`
TRIM	Trims leading or trailing characters	`SELECT TRIM('H' 'HelloWorld') FROM DUAL`	`elloWorld`

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT employee_id, CONCAT(first_name, last_name)AS NAME,
LENGTH (last_name), INSTR(last_name, 'a') "Contains 'a'?"
FROM employees
WHERE SUBSTR(last_name, -1, 1) = 'n';
```

Query Results:

```
EMPLOYEE_ID      NAME  LENGTH(LAST_NAME)      Contains
'a'?
102   LexDe Haan       7                      5
105   DavidAustin      6                      0
110   JohnChen         4                      0
112   Jose ManuelUrman 5                      4
123   ShantaVollman    7                      6
…
```

Explanation: The `SELECT` statement returns the first name and last name joined, the length of the last name, position of a in the last name.

Code file: Code5_3_1.sql

Lesson 5.4: Manipulate numbers with the numeric functions

Function	Description	Example	Result
MOD	Accept 2 values as arguments and return the remainder when the first value is divided by the second	`SELECT mod(3,2),mod(18,10) FROM dual;`	1 and 8
FLOOR	The FLOOR function accepts a value as an argument and rounds the value down to the nearest integer value	`SELECT FLOOR(15.7) "Floor" FROM DUAL`	15
GREATEST	Returns the greatest of the list of *exprs* (expressions).	`SELECT GREATEST('HARRY','HARRIOT','HAROLD') "GREATEST" FROM DUAL;`	HARRY
MOD	Returns the remainder of *m* divided by *n*. Returns *m* if *n* is 0.	`SELECT MOD (26,11) FROM DUAL`	4
ROUND	Returns *n* rounded to *m* places to the right of the decimal point	`SELECT ROUND (54.339, 2) FROM DUAL;`	54.34
TRUNC	Returns *n* truncated to *m* decimal places, where *m* and *n* are numeric arguments.	`SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;`	15.7

There are several other numeric functions that can be used in SQL. For example we may require to perform a trigonometry calculation. The table below lists some of the other functions that may be encountered.

Function Name / Syntax
ABS (value)
ACOS (value)
ASIN (value)
ATAN (value)
COS (value)
COSH (value)
EXP (value)
LOG (value)
POWER (value1, value2))

SIGN (value)
SIN (value)
SINH (value)
SQRT (value)
TAN (value)
TANH (value)

Lesson 5.5: Manipulate dates with the date functions

Function	Description	Example	Result
TRUNC	Returns the date *d* with its time portion truncated to the time unit specified by the format model *fmt*.	`SELECT TRUNC(TO_DATE('27-OCT-92', 'DD-MON-YY'), 'YEAR') "First Of The Year" FROM DUAL;`	1992-01-01
SYSDATE	Returns the current date and time.	`SELECT TO_CHAR(SYSDATE, 'MM-DD-YYYY HH24:MI:SS') NOW FROM DUAL`	04-12-2014 19:13:48
TIMESTAMPADD	Adds a date and time value to the current timestamp.	`SELECT {fn TIMESTAMPADD (SQL_TSI_DAY, 1, {fn NOW()})} FROM DUAL;`	adds one day to the current
NEXT_DAY	Returns the date of the first weekday named by *char* that is later than the date *d*.	`SELECT NEXT_DAY('15-MAR-92','TUESDAY') "NEXT DAY" FROM DUAL;`	1992-03-17

Section 6: Invoke Conversion Functions and Conditional Expressions

In this section you will:

- Describe implicit and explicit data type conversion
- Use the TO_CHAR, TO_NUMBER, and TO_DATE conversion functions
- Nest multiple functions
- Apply the NVL, NULLIF, and COALESCE functions to data
- Use conditional IF THEN ELSE logic in a SELECT statement

Lesson 6.1: Describe implicit and explicit data type conversion

In some cases, the Oracle server receives data of one data type where it expects data of a different data type. When this happens, the Oracle server can automatically convert the data to the expected data type. This data type conversion can be done `implicitly` by the Oracle server or `explicitly` by the user.

Implicit Data Type Conversion

Oracle server can automatically perform data type conversion in an expression. For example, the expression hire_date > '01-JAN-90' results in the implicit conversion from the string '01-JAN-90' to a date. Therefore, a VARCHAR2 or CHAR value can be implicitly converted to a number or date data type in an expression.

For expression evaluation, the Oracle server can automatically convert the following:

NUMBER to VARCHAR2 or CHAR

DATE to VARCHAR2 or CHAR

Explicit Data Type Conversion

Explicit data type conversions are done by using the conversion functions. Conversion functions convert a value from one data type to another. Generally, the form of the function names follows the convention data type **TO** data type. The first data type is the input data type and the second data type is the output.

Lesson 6.2: Conversion functions

SQL provides three functions to convert a value from one data type to another

TO_CHAR

TO_NUMBER

TO_DATE

TO_CHAR

The TO_CHAR function will convert either a date or number to a text string. The format of the new string is dependent on the format argument provided. It is possible to pass the national language to function.

Syntax:

```
    TO_CHAR([date][number][,format][,language])
```

The format arguments can accept several different codes that can be built up to provide a format mask used to convert the data. This is especially important for the conversion of dates to characters. In Oracle dates hold both date and time information however when a date column is selected it is display in the default date style i.e. DD-Mon-YY 01-JAN-2005. If we require viewing a date in a different format we require converting it to a string using the TO_CHAR function. Before looking at some examples of working with the TO_CHAR function we shall consider the codes that can be used.

Codes for Date formats

Code	Description
DD	The number of the day in a month 1,2,..31
Day	The full spelling of a name of the day in the specified case e.g. DAY – MONDAY or Day – Monday
DY	The first 3 character abbreviation for a day e.g. Mon
D	The number of the day in a week i.e. 1 – 7. Where Sunday is day 1.
DDD	The number of the day in the year i.e. 1 -366
Th	Display a day as an ordinal number i.e. 1st, 2nd, 3rd, 4th
J	Julian days since 4712BC i.e. 2453524
MM	The number of the month in two digit format i.e. 01,02 – 12
Month	The full spelling of the name of the month in the specified case e.g. MONTH – MAY or Month – May
Mon	The first 3 character abbreviation for the month in the case specified i.e. MON – JUN or Mon – Jun
YY	The 2 digit year number i.e. 05
YYY	The 3 digit year number i.e. 005

YYYY	The 4 digit year number i.e. 2005
Y,YYY	The 4 digit year number with a comma 2,005
Year	The full spelling of the year in the specified case e.g. Year – Two Thousand Five or YEAR – TWO THOUSAND FIVE
IYYY,IYY, IY	Displays the years based on the ISO standards for year start date. See below.
RR	Displays 2 digit year rounded to a century. If the year in date provide is <50 and current year is >=50 rounded to next century i.e. 22nd. If the year in date provide is >=50 but the current year is <50 the century to rounded down. For example current year is 2005 and date provided is 73 the RR format would use 1973 rather than 2073.
RRRR	As per the RR but displays a 4 digit year.
AD or BC	AD or BC is added to the format mask for the year provided i.e. 2005 AD
A.D. or B.C.	A.D. or B.C. is added to the format mask for the year provided i.e. 2005 A.D.
CC	The century for the date provided i.e. 21st century
SCC	The century for the date provide with a minus displayed if the century is BC.
Q	The Quarter number of the year starting at the 1st January e.g. 1-4
WW	The week number of the year i.e. 1 -53
W	The week number of the month i.e. 1-5
HH or HH12	The hours display as numbers between 1 -12
HH24	The hours displayed as numbers between 1- 24
MI	The minutes displayed as numbers in an hour 1 – 60
SS	The seconds displayed as numbers in a minute 1 -60
SSSSS	The seconds displays as number of day.
FF	The fraction of a second displayed to 6 digit value e.g. 743521
AM or PM	AM or PM is added to the selected time
A.M. or P.M.	A.M. or P.M. is added to the selected time
Sp	Spell a number e.g. dd
Fm	Fill mode removes and blank spaces from the date picture
TZD	
TZH	Display the hours offset of the time zone for example +12,-5
TZM	Display the minutes offset of the time zone. This is used with the TZH format e.g. TZH:TZM may return +10:00
TZR	Displays the time zone region

It is also possible to place punctuation marks such as full stops, hyphens etc. and spaces in the format mask for example:

‘ dd-mon-yyyy hh:mi’

In the above table we mentioned that date can use the ISO standard. The ISO 8601 standard determine start date to use for the current year. It determines if the during the first week of the year most of the days are in the new or previous year. For example if the first of January is a Friday, Saturday or Sunday then it is included in the previous year however if the first of January is a Monday, Tuesday, Wednesday or Thursday then the

1first of January is part of the new year. There dependent of this the IYYY format may result in a different year displayed than the YYYY format.

Codes for Number Formats

The table below shows the format codes that can be used when converting numbers to character strings using the TO_CHAR function.

Code	Description
L	Display the local currency symbol i.e. £
9	Represents a single digit in a format mask e.g. 9999 = 1234
0	Display a leading or trailing zero in a format mask e.g. 0099 = 0012
,	Display a comma in a format mask e.g. 9,999 = 1234
.	Display a period in a format mask e.g. 9,999.00 = 1234.00
PR	Display negative numbers in brackets e.g. -9999 = <1234>
MI	Display negative numbers with a trailing minus
FM	Remove any leading or trailing blank values e.g. FM999.99 = 12.1

Usage of TO_CHAR

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     TO_CHAR(hire_date),

           TO_CHAR(hire_date, 'fmdd-Month-YYYY'),

           TO_CHAR(hire_date, 'hh24:mi:ss'),

           TO_CHAR(salary, 'L999,999')

FROM       employees;
```

Query Results:

```
HD1        HD2                   HD3        HD4
17-JUN-03  17-June-2003          00:00:00   $24,000
21-SEP-05  21-September-2005     00:00:00   $17,000
13-JAN-01  13-January-2001       00:00:00   $17,000
03-JAN-06  3-January-2006        00:00:00   $9,000
21-MAY-07  21-May-2007           00:00:00   $6,000
 …
```

Explanation: This statement uses the TO_CHAR function to format dates and numbers

Code file: Code6_2_1.sql

NLS

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT     TO_CHAR (sysdate, 'Day',  'NLS_DATE_LANGUAGE =
French' ),

TO_CHAR(1994.50,'C9,999.99', 'NLS_ISO_CURRENCY=France')

FROM       dual;
```

Query Results:

```
Dimanche        EUR1,994.50…
```

Explanation: This statement uses the TO_CHAR function and NLS to format dates and numbers

Code file: Code6_2_2.sql

TO_DATE

The TO_DATE function allows numbers or character strings to be converted to dates. This could be useful during data loads where dates are held in a different format from standard Oracle date format.

Usage of TO_DATE

Syntax:

```
TO_DATE([text][number][,format][,national_language_optio
n])
```

SQL Example:

```
SELECT     to_date('01/01/2002','dd/mm/yyyy')

FROM       dual;
```

Query Results:

```
TO_DATE('01/01/2002','DD/MM/YYYY')
01-JAN-02
```

Explanation: This statement uses the TO_DATE function to format dates

Code file: Code6_2_3.sql

TO_NUMBER

The TO_NUMBER function allows characters to be converted into numbers. This could be useful during data loads where a column is held as a character rather than a number. This could be due to the number having a currency symbol place in front of it which has converted it to a character string.

Usage of TO_NUMBER

Syntax:

```
TO_NUMBER([text][,format][,national_language_option])
```

SQL Example:

```
SELECT to_number('$12,500','L99,999')
 FROM dual
```

Query Results:

```
TO_NUMBER('$12,500','L99,999')
12500
```

Explanation: This statement uses the TO_NUMBER function to convert text to number

Code file: Code6_2_4.sql

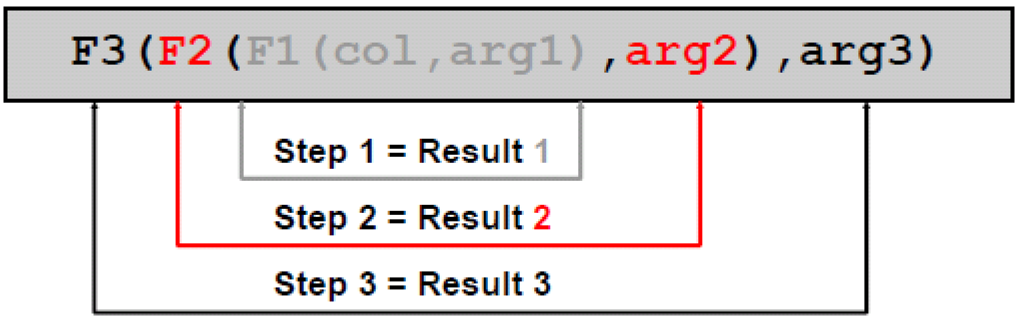
Other Miscellaneous Conversion Functions

Function Name/Syntax	Description
ASCIISTR(string)	Accepts a text string and converts it to ASCII characters. New in Oracle 9i.
CAST(value AS datatype)	Converts an item into any data type. For example, a number into a varchar2 or a date into a varchar2. e.g. CAST(1234 AS varchar2(20))
CHARTOROWID(STRING)	Converts a text into the ROWID data type.
COMPOSE(STRING)	Converts a string to a Unicode string. Can be used to store data that may use symbols such as acutes, tidle etc. . `SELECT 'ol' || COMPOSE('e' || UNISTR('\0301')) FROM dual;`
DECOMPOSE	Converts a Unicode string to a string.

Lesson 6.3: Nest multiple functions

Single-row functions can be nested to any depth. Nested functions are evaluated from the innermost level to the outermost level. Some examples follow to show you the flexibility of these functions.

- Single-row functions can be nested to any level.
- Nested functions are evaluated from the deepest level to the least deep level.



Syntax:

SELECT *|{[DISTINCT] *column|expression* [*alias*],...}
FROM *table*
[WHERE *condition(s)*];

SQL Example:

```
SELECT TRIM(UPPER(first_name))
FROM employees;
```

Query Results:

```
TRIM(UPPER(FIRST_NAME))
ELLEN
SUNDAR
MOZHE
DAVID
HERMANN …
```

Explanation: The `SELECT` statement returns the trimmed uppercase first name, for all employees

Code file: Code6_3_1.sql

Lesson 6.4: Apply the DECODE, NVL, NULLIF, and COALESCE functions to data

DECODE

The DECODE function is similar to IF statements if other programming languages and it one of the most commonly used functions in Oracle SQL.

Syntax:

```
DECODE (exp1, search, result
           [search2, result]
           [search3, result]
           ….[,result])
```

SQL Example:

```
SELECT region_name,
      decode(region_id,1,'Region A',
     2,'Region B',
      'Region C')
    FROM regions;
```

Query Results:

```
REGION_NAME                 REGION
Europe                      Region A
Americas                    Region B
Asia                        Region C
Middle East and Africa      Region C
```

Explanation: The `SELECT` statement returns Region A if the region id is 1, Region B if the ID is 2 etc

Code file: Code6_4_1.sql

NVL

The NVL function is used to return a replacement value when a null value is found. Null values can cause problems in calculations i.e. a null plus a value will return the result null. Therefore, the NVL function can be used to ensure the correct result is achieved.

Syntax:

```
NVL(exp1, exp2)
```

SQL Example:

```
SELECT      last_name,
            salary,
            salary * commission_pct as Total,
            salary * nvl(commission_pct,0) as Total2
FROM        employees;
```

Query Results:

```
LAST_NAME        SALARY     TOTAL      TOTAL2
King             24000                 0
Kochhar          17000                 0
De Haan          17000                 0
Hunold           9000                  0
Ernst            6000                  0
```

Explanation: The `SELECT` statement converts a null to 0 in the Total2 result

Code file: Code6_4_2.sql

COALESCE

The coalesced function is similar to using the NVL function. If expression 1 is null, then another specified value will be returned.

Syntax:

```
Coalesce(exp1,exp2,expn…)
```

SQL Example:

```
SELECT last_name, employee_id,
COALESCE(TO_CHAR(commission_pct),TO_CHAR(manager_id),
'No commission and no manager') as Status
FROM employees;
```

Query Results:

```
LAST_NAME  EMPLOYEE_ID      STATUS
King       100              No commission and no manager
Kochhar    101              100
De Haan    102              100
Hunold     103              102
Ernst      104              103
…
```

Explanation: The `SELECT` statement shows that if the `manager_id` value is not null, it is displayed. If the `manager_id` value is null, then the `commission_pct` is displayed. If the `manager_id` and `commission_pct` values are null, then “No commission and no manager” is displayed.

Code file: Code4_2_14.sql

NULLIF

The `NULLIF` function compares two expressions. If they are equal, the function returns a null. If they are not equal, the function returns the first expression. However, you cannot specify the literal `NULL` for the first expression.

Syntax: `NULLIF (`*expr1*`,` *expr2*`)`

SQL Example:

```
SELECT first_name, LENGTH(first_name) "expr1",
last_name, LENGTH(last_name) "expr2",
NULLIF(LENGTH(first_name), LENGTH(last_name)) result
FROM employees;
```

Query Results:

```
FIRST_NAME expr1 LAST_NAME  expr2 RESULT
Ellen      5     Abel       4     5
Sundar     6     Ande       4     6
Mozhe      5     Atkinson   8     5
David      5     Austin     6     5
Hermann    7     Baer       4     7
…
```

Explanation: The `SELECT` statement shows that if the length of the first name in the `EMPLOYEES` table is compared to the length of the last name in the `EMPLOYEES` table. When the lengths of the names are equal, a null value is displayed. When the lengths of the names are not equal, the length of the first name is displayed.

Code file: Code6_4_4.sql

Lesson 6.5: Use conditional CASE statement

The CASE statement in Oracle can be used for the same purpose as the DECODE function.

Simple CASE functions

The simple case statement is very similar to the DECODE statement used in Oracle, the CASE statement will search for a value and return a different value, it will also return a default value for any search values that do not match.

Syntax:

```
CASE expr  WHEN comparison_expr1  THEN return_expr1
[WHEN comparison_expr2  THEN return_expr2
WHEN comparison_exprn  THEN return_exprn
ELSE else_expr]
END;
```

SQL Example:

```
SELECT     first_name,
       last_name,department_id,
       (CASE department_id
        WHEN 90 THEN 'SOUTH ENGLAND'
        WHEN 20 THEN 'NORTH ENGLAND'
        ELSE 'SCOTLAND'
        END) AS AREA
   FROM       employees;
```

Query Results:

```
FIRST_NAME LAST_NAME  DEPARTMENT_ID    AREA
Steven     King       90               SOUTH ENGLAND
Neena      Kochhar    90               SOUTH ENGLAND
Lex        De Haan    90               SOUTH ENGLAND
Alexander  Hunold     60               SCOTLAND
Bruce      Ernst      60               SCOTLAND
…
```

Explanation: The `SELECT` uses a CASE statement to evaluate the department ID and return a department name.

Code file: Code6_5_1.sql

Searched CASE functions

The searched case statement looks more like an IF statement and allows us to place logical conditionality on searches.

Syntax:

```
CASE WHEN comparison_expr1  THEN return_expr1
[WHEN comparison_expr2  THEN return_expr2
WHEN comparison_exprn  THEN return_exprn
ELSE else_expr]
END;
```

SQL Example:

```
SELECT  first_name,
      last_name,
    salary,
      CASE WHEN salary >= 4000
      AND salary <=20000 THEN 23
      WHEN salary>20000 THEN 40
     ELSE 0
     END AS TAX_RATE
  FROM  employees
```

Query Results:

```
FIRST_NAME       LAST_NAME  SALARY     TAX_RATE
Steven           King       24000      40
Neena            Kochhar    17000      23
Lex              De Haan    17000      23
Alexander        Hunold     9000       23
Bruce            Ernst      6000       23
…
```

Explanation: The `SELECT` statement uses a searched case to determine the tax rate.

Code file: Code6_5_2.sql

Section 7: Aggregate Data Using the Group Functions

In this section you will:

- Use the aggregation functions to produce meaningful reports
- Divide the retrieved data in groups by using the GROUP BY clause
- Exclude groups of data by using the HAVING clause

Lesson 7.1: Aggregation functions

Unlike single-row functions, group functions operate on sets of rows to give one result per group. These sets may comprise the entire table or the table split into groups. Group functions are used to carry out aggregating of data. This is done through the use of aggregating functions, GROUP BY clause, the HAVING clause and extensions to the GROUP clause, ROLLUP, CUBE and GROUPING SETS.

Aggregation Functions

An aggregate function summarizes the results of an expression over a number of rows, returning a single value. Aggregate functions are calculations.

Syntax :

```
aggregate_function ([DISTINCT | ALL] expression)
```

aggregate_function -

Gives the name of the function – e.g SUM, COUNT, etc

DISTINCT -

Specifies that the aggregate function should consider only distinct values of the argument expression

ALL -

Specifies that the aggregate function should consider all values, including duplicate values.

expression -

Specifies a column, or any other expression, on which you want to perform aggregation.

SUM

Returns the sum of values in a column or expression over the rows returned by the query

Syntax: `SUM(column_expr)`

SQL Example:

```
SELECT sum(salary)
FROM employees;
```

Query Results:

```
SUM(SALARY)
691416
```

Explanation: The `SELECT` statement returns the sum of the salary field

Code file: Code7_1_1.sql

AVG

Returns the average value of a column or expression over the rows returned by a query.

Syntax: `AVG(column_expr)`

SQL Example:

```
SELECT AVG(salary)
FROM employees;
```

Query Results:

```
6461.831775700934579439252336448598130841
```

Explanation: The `SELECT` statement returns the average of the salary field

Code file: Code7_1_1.sql

COUNT

Returns the number of values in a column or expression over the rows returned by the query.

Syntax:

```
COUNT([DISTINCT] column_expr)
```

SQL Example:

```
SELECT count(employee_id)
FROM employees;
```

Query Results:

```
COUNT(EMPLOYEE_ID)
107
```

Explanation: The `SELECT` statement returns the count of the employee id field

Code file: Code7_1_1.sql

MAX / MIN

Returns the highest (MAX) or lowest (MIN) of values in a column or expression over the rows returned by the query

Syntax:

```
MAX(column_expr)

MIN(column_expr
```

SQL Example:

```
SELECT MAX(salary), MIN(SALARY)
FROM employees;
```

Query Results:

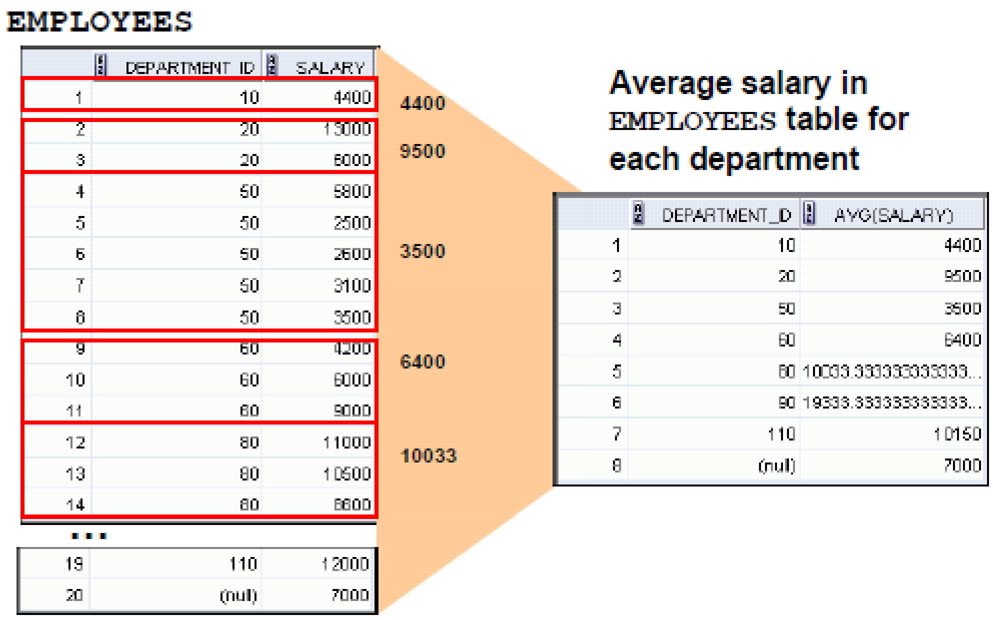
```
MAX(SALARY)      MIN(SALARY)
24000            2100
```

Explanation: The `SELECT` statement returns the highest and lowest salary

Code file: Code7_1_1.sql

Lesson 7.2: GROUP BY clause

The GROUP BY clause, groups a result set into multiple groups and then produces a single row of summary information for each group. The GROUP BY clause does not require the GROUP BY field to be in the SELECT list. The GROUP BY clause is used when referencing columns other than the aggregate function columns.



EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	50	5800
5	50	2500
6	50	2600
7	50	3100
8	50	3500
9	60	4200
10	60	6000
11	60	9000
12	80	11000
13	80	10500
14	80	8600
...		
19	110	12000
20	(null)	7000

	DEPARTMENT_ID	AVG(SALARY)
1	10	4400
2	20	9500
3	50	3500
4	60	6400
5	80	10033.3333333333333...
6	90	19333.3333333333333...
7	110	10150
8	(null)	7000

Syntax:

```
SELECT column, group_function(column)
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[ORDER BY column]
```

SQL Example:

```
SELECT     department_id, AVG(salary)
FROM       employees
GROUP BY   department_id;
```

Query Results:

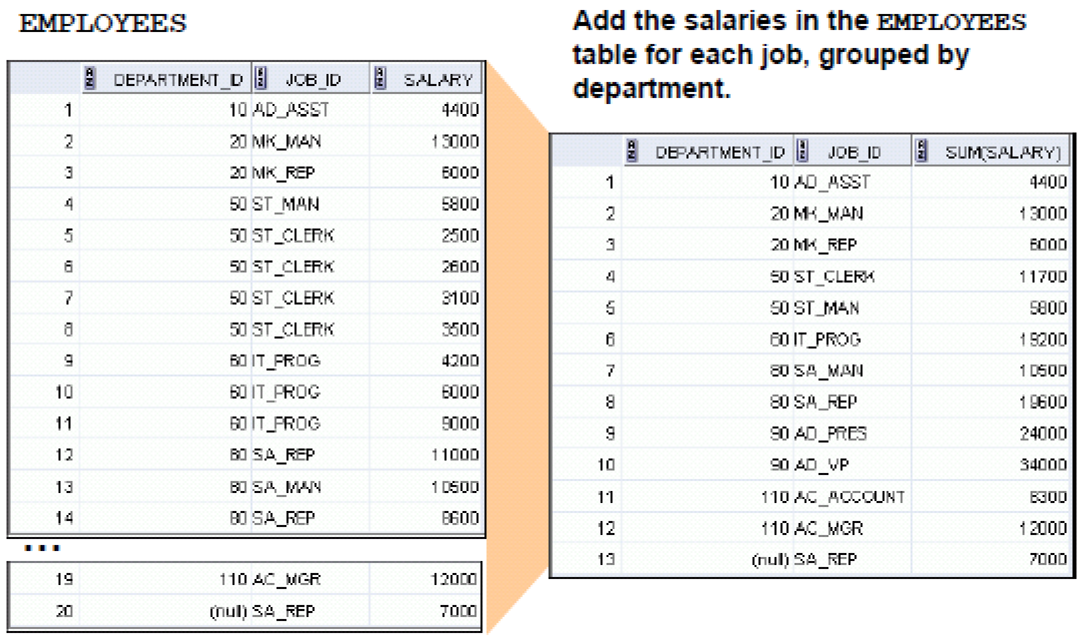
```
DEPARTMENT_ID    AVG(SALARY)
100              8601.3333333333333
30               4150
                 7000
90               19333.33333333333333
…
```

Explanation: The `SELECT` statement returns the department number and the average salary for each department

Code file: Code7_2_1.sql

Grouping by More than One Column

Sometimes you need to see results for groups within groups. The diagram shows a report that displays the total salary that is paid to each job title in each department.



EMPLOYEES

	DEPARTMENT_ID	JOB_ID	SALARY
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_MAN	5800
5	50	ST_CLERK	2500
6	50	ST_CLERK	2600
7	50	ST_CLERK	3100
8	50	ST_CLERK	3500
9	60	IT_PROG	4200
10	60	IT_PROG	6000
11	60	IT_PROG	9000
12	80	SA_REP	11000
13	80	SA_MAN	10500
14	80	SA_REP	8600
...			
19	110	AC_MGR	12000
20	(null)	SA_REP	7000

Add the salaries in the EMPLOYEES table for each job, grouped by department.

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_CLERK	11700
5	50	ST_MAN	5800
6	60	IT_PROG	19200
7	80	SA_MAN	10500
8	80	SA_REP	19600
9	90	AD_PRES	24000
10	90	AD_VP	34000
11	110	AC_ACCOUNT	8300
12	110	AC_MGR	12000
13	(null)	SA_REP	7000

Syntax:

```
SELECT column, group_function(column)
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[ORDER BY column];
```

SQL Example:

```
SELECT department_id dept_id, job_id, SUM(salary)
FROM employees
GROUP BY department_id, job_id
ORDER BY department_id;
```

Query Results:

```
DEPT_ID     JOB_ID      SUM(SALARY)
10          AD_ASST     4400
20          MK_MAN      13000
20          MK_REP      6000
30          PU_CLERK    13900
30          PU_MAN      11000…
```

Explanation: The `SELECT` statement shows how the rows are grouped by the department number and secondly by job ID. The salary is summed per group.

Code file: Code7_2_4.sql

Lesson 7.3: HAVING clause

The having clause is associated with the group by clause. The HAVING clause is used to put a filter on the groups created by the GROUP BY clause. If a query has a HAVING clause along with a GROUP BY clause the result set will need to satisfy the conditions placed within the HAVING clause

Syntax:

```
SELECT column, group_function
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[HAVING group_condition]
[ORDER BY column];
```

SQL Example:

```
SELECT      department_id dept_id, job_id, SUM(salary)
FROM        employees
GROUP BY    department_id, job_id
HAVING      SUM(SALARY) > 10000
ORDER BY    department_id;
```

Query Results:

```
DEPT_ID     JOB_ID      SUM(SALARY)
20          MK_MAN      13000
30          PU_CLERK    13900
30          PU_MAN      11000
50          SH_CLERK    64300
50          ST_CLERK    55700
…
```

Explanation: The `SELECT` statement displays the dept ID, job ID and total monthly salary for each job that has a total payroll exceeding 10,000.

Code file: Code7_3_1.sql

Lesson 7.4:Introduction of Analytical Functions

Oracle has enhanced SQL's analytical processing capabilities by introducing a new family of analytic SQL functions. These analytic functions enable you to calculate:

- Rankings and percentiles
- Moving window calculations
- Lag/lead analysis
- First/last analysis
- Linear regression statistics

Ranking functions include cumulative distributions, percent rank, and N-tiles. Moving window calculations allow you to find moving and cumulative aggregations, such as sums and averages. Lag/lead analysis enables direct inter-row references so you can calculate period-to-period changes. First/last analysis enables you to find the first or last value in an ordered group.

RANK

RANK calculates the rank of a value in a group of values. Rows with equal values for the ranking criteria receive the same rank. Oracle then adds the number of tied rows to the tied rank to calculate the next rank. Therefore, the ranks may not be consecutive numbers.

Syntax :

```
RANK ( expr [, expr]... ) WITHIN GROUP
( ORDER BY
  expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]...
```

SQL Example:

```
SELECT employee_id,
  department_id,
  Salary,
  RANK() OVER (PARTITION BY department_id ORder by salary
DESC) Rk
FROM employees
WHERE department_id IN (90,50)
ORDER BY department_id,
  salary DESC;
```

Query Results:

```
EMPLOYEE_ID    DEPARTMENT_ID  SALARY         RK
121            50             8200           1
120            50             8000           2
122            50             7900           3
```

```
123          50          6500        4
124          50          5800        5
184          50          4200        6
185          50          4100        7
192          50          4000        8
193          50          3900        9
188          50          3800        10
137          50          3600        11
189          50          3600        11
141          50          3500        13
186          50          3400        14
133          50          3300        15
```

Explanation: The `SELECT` statement ranks salary by department, if there is a tie the next ranking skips a number.

Code file: Code7_4_1.sql

DENSE_RANK

`DENSE_RANK` computes the rank of a row in an ordered group of rows. The ranks are consecutive integers beginning with 1. The largest rank value is the number of unique values returned by the query. Rank values are not skipped in the event of ties. Rows with equal values for the ranking criteria receive the same rank.

Syntax:

```
DENSE_RANK ( expr [, expr]... ) WITHIN GROUP
( ORDER BY expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]
```

SQL Example:

```
SELECT employee_id,
  department_id,
  Salary,
  DENSE _RANK() OVER (PARTITION BY department_id ORder by
salary DESC) Rk
FROM employees
WHERE department_id IN (90,50)
ORDER BY department_id,
  salary DESC;
```

Query Results:

```
EMPLOYEE_ID    DEPARTMENT_ID  SALARY       RK
121            50             8200         1
120            50             8000         2
```

```
122           50           7900           3
123           50           6500           4
124           50           5800           5
184           50           4200           6
185           50           4100           7
192           50           4000           8
193           50           3900           9
188           50           3800           10
137           50           3600           11
189           50           3600           11
141           50           3500           12
186           50           3400           13
133           50           3300           14
```

Explanation: The `SELECT` statement ranks salary by department, if there is a tie the next ranking continues with next rank number.

Code file: Code7_4_2.sql

SUM Over

`SUM Over` computes the sum of a column in an ordered group of rows.

Syntax:

```
SUM OVER ( expr [, expr]... ) WITHIN GROUP
( ORDER BY expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]
```

SQL Example:

```
SELECT employee_id,
  department_id,
  Salary,
  SUM(Salary) OVER (PARTITION BY department_id ) Total
FROM employees
WHERE department_id IN (20,30)
ORDER BY department_id,
  salary DESC;
```

Query Results:

```
EMPLOYEE_ID    DEPARTMENT_ID  SALARY         TOTAL
201            20             13000          19000
202            20             6000           19000
114            30             11000          24900
```

```
115          30          3100          24900
116          30          2900          24900
117          30          2800          24900
118          30          2600          24900
119          30          2500          24900
```

Explanation: The `SELECT` statement sum salary by department

Code file: Code7_4_3.sql

Section 8: Display Data from Multiple Tables Using Joins

In this section you will:

- Join Types
- Write SELECT statements to access data from more than one table
- View data that generally does not meet a join condition by using outer joins
- Join a table to itself by using a self-join
- Cross Joins
- Natural joins

Lesson 8.1: Overview of joins

Joins are used to extract information from two or more tables. Join queries are different from regular queries as the FROM clause will contain two or more tables or views and a condition is placed to link the tables.

Joins Types

- Oracle INNER JOIN (or sometimes called simple join)
- Oracle LEFT OUTER JOIN (or sometimes called LEFT JOIN)
- Oracle RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)
- Oracle FULL OUTER JOIN (or sometimes called FULL JOIN)
- Natural joins:
- Cross joins

Ansi 1999 Syntax

```
SELECT table1.column, table2.column
FROM table1
[NATURAL JOIN table2] |
[JOIN table2  USING (column_name)] |
[JOIN table2
ON (table1.column_name  = table2.column_name)]|
[LEFT|RIGHT|FULL OUTER JOIN table2
ON (table1.column_name  = table2.column_name)]|
[CROSS JOIN table2];
```

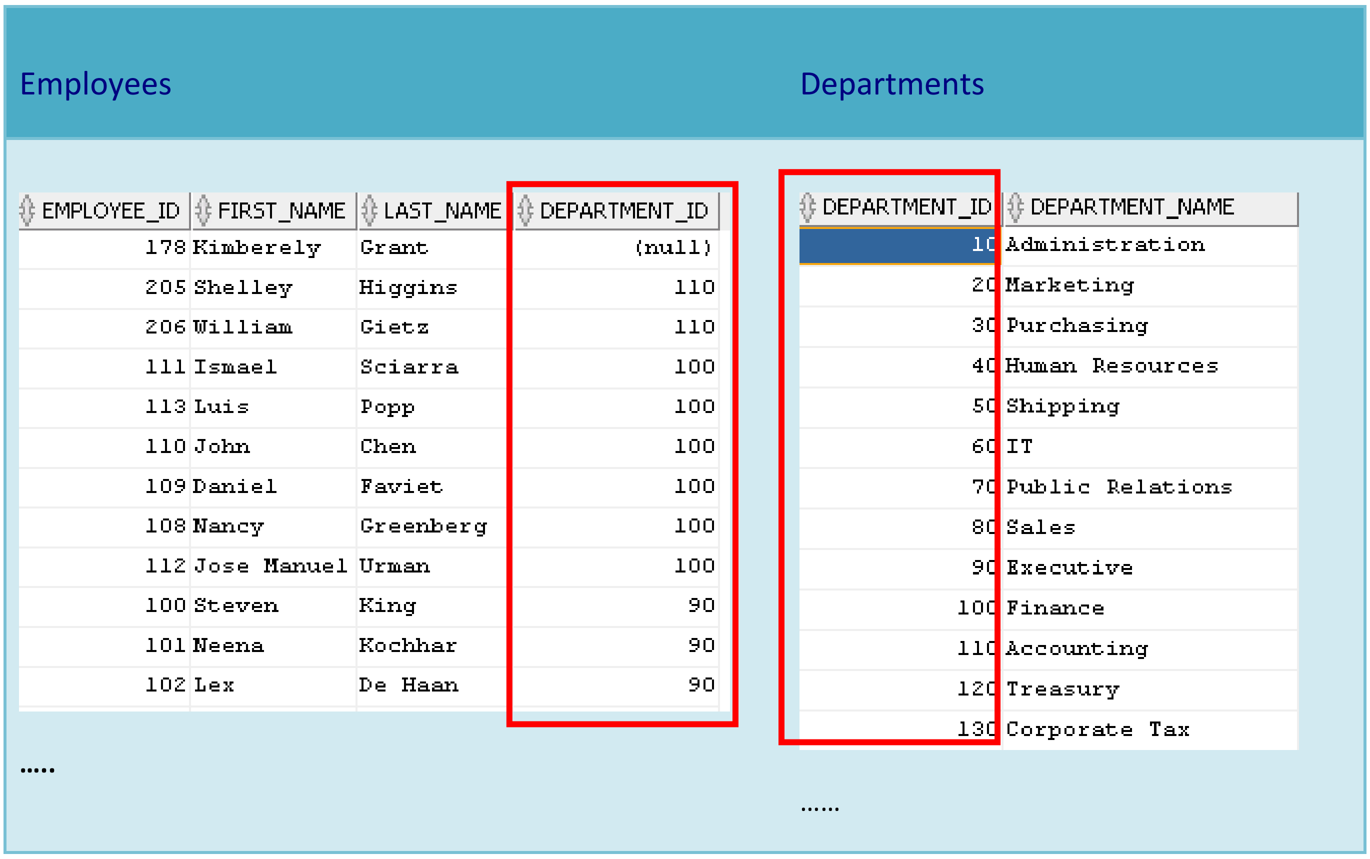
SQL99 syntax is the recommended syntax for all new development. It offers many enhanced features. One of the primary changes is the table joins are no longer specified in the WHERE clause. The joins are performed after the FROM clause.

Table Aliases

It is recommended that table aliases are used when referencing tables in the FROM clause. Whenever there is an ambiguity in the column names, you must use a table alias.

Lesson 8.2: Write SELECT statements to access data from more than one table

The following examples are based on the following tables:



Employees

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
178	Kimberely	Grant	(null)
205	Shelley	Higgins	110
206	William	Gietz	110
111	Ismael	Sciarra	100
113	Luis	Popp	100
110	John	Chen	100
109	Daniel	Faviet	100
108	Nancy	Greenberg	100
112	Jose Manuel	Urman	100
100	Steven	King	90
101	Neena	Kochhar	90
102	Lex	De Haan	90

.....

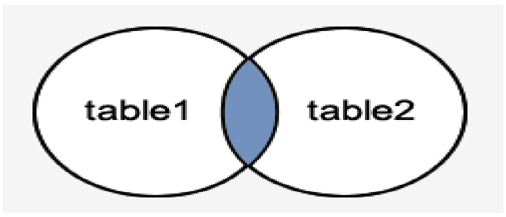
Departments

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
30	Purchasing
40	Human Resources
50	Shipping
60	IT
70	Public Relations
80	Sales
90	Executive
100	Finance
110	Accounting
120	Treasury
130	Corporate Tax

......

Lesson 8.2: Inner Joins

It is the most common type of join. Oracle INNER JOINS return all rows from multiple tables where the join condition is met. In this visual diagram, the Oracle INNER JOIN returns the shaded area.



SQL99 Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

SQL Example:

```
SELECT      Employee_id,first_name, last_name,
           e.Department_id , Department_Name
FROM        EMPLOYEES e
INNER  JOIN Departments d
ON          e.department_id=d.department_id
ORDER BY    department_ID desc
```

*** SQL92 Syntax:**

```
SELECT columns
FROM table1 ,  table2
WHERE  table1.column = table2.column;
```

SQL Example:

```
SELECT       Employee_id,first_name, last_name, e.Department_id
           , Department_Name
FROM        EMPLOYEES e, Departments d
WHERE       e.department_id=d.department_id
ORDER BY    department_ID desc;
```

Query Results:

```
EMPLOYEE_ID FIRST_NAME LAST_NAME  DEPARTMENT_ID    DEPARTMENT_NAME
205         Shelley    Higgins    110              Accounting
206         William    Gietz      110              Accounting
109         Daniel     Faviet     100              Finance
…….
```

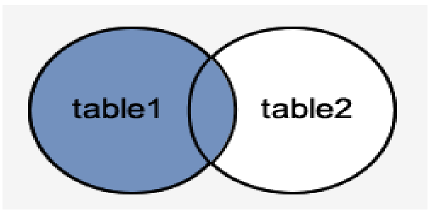
Explanation: To determine an employee's department name, you compare the value in the `DEPARTMENT_ID` column in the `EMPLOYEES` table with the `DEPARTMENT_ID` values in the `DEPARTMENTS` table. The relationship between the `EMPLOYEES` and `DEPARTMENTS` tables is an *`equijoin`*; that is, values in the `DEPARTMENT_ID` column in both the tables must be equal.Note Kimberly Grant is not included in the resultset as department ID null does not exist in the `DEPARTMENTS` table

Code file: Code8_2_1.sql

Lesson 8.3: Outer Joins

LEFT OUTER JOIN

This type of join returns all rows from the LEFT-hand table specified in the ON condition and those rows from the other table where the joined fields are equal (join condition is met).



SQL99 Syntax:

```
SELECT columns
FROM table1
LEFT OUTER JOIN table2
ON table1.column = table2.column;
```

SQL Example:

```
SELECT      Employee_id,first_name, last_name,
           e.Department_id , Department_Name
FROM        EMPLOYEES e
LEFT OUTER  JOIN Departments d
ON          e.department_id=d.department_id
ORDER BY    department_ID desc
```

*** SQL92 Syntax:**

```
SELECT columns
FROM table1 ,  table2
WHERE  table1.column = table2.column (+);
```

SQL Example:

```
SELECT      Employee_id,first_name,  last_name,  e.Department_id
            ,Department_Name
FROM        EMPLOYEES e, Departments d
WHERE       e.department_id=d.department_id (+)
ORDER BY    department_ID desc;
```

Query Results:

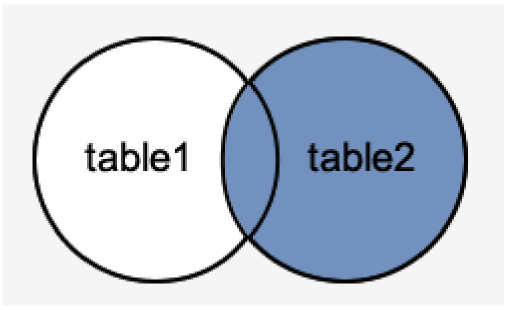
```
EMPLOYEE_ID FIRST_NAME LAST_NAME  DEPARTMENT_ID    DEPARTMENT_NAME
178         Kimberely  Grant
205         Shelley    Higgins    110              Accounting
206         William    Gietz      110              Accounting
111         Ismael     Sciarra    100              Finance
```

Explanation: To determine an employee's department name, you compare the value in the `DEPARTMENT_ID` column in the `EMPLOYEES` table with the `DEPARTMENT_ID` values in the DEPARTMENTS table. The relationship between the EMPLOYEES and DEPARTMENTS tables is left outer join that is, values in the DEPARTMENT_ID column in employees table that are not matched will be included. Note Kimberly Grant is included.

Code file: Code8_2_2.sql

RIGHT OUTER JOIN

This type of join returns all rows from the Right-hand table specified in the ON condition and those rows from the other table where the joined fields are equal (join condition is met).



SQL99 Syntax:

```
SELECT columns
FROM table1
RIGHT OUTER JOIN table2
ON table1.column = table2.column;
```

SQL Example:

```
SELECT      Employee_id,first_name, last_name,
           e.Department_id , Department_Name
FROM        EMPLOYEES e
RIGHT OUTER  JOIN Departments d
ON          e.department_id=d.department_id
ORDER BY    department_ID desc
```

*** SQL92 Syntax:**

```
SELECT columns
FROM table1 ,  table2
WHERE  table1.column(+) = table2.column;
```

SQL Example:

```
SELECT       Employee_id,first_name, last_name, d.Department_id
           , Department_Name
FROM        EMPLOYEES e, Departments d
WHERE       e.department_id(+)=d.department_id
ORDER BY    department_ID desc;
```

Query Results:

```
EMPLOYEE_ID FIRST_NAME  LAST_NAME   DEPARTMENT_ID     DEPARTMENT_NAME
                                    140   Control And Credit
                                    130   Corporate Tax
```

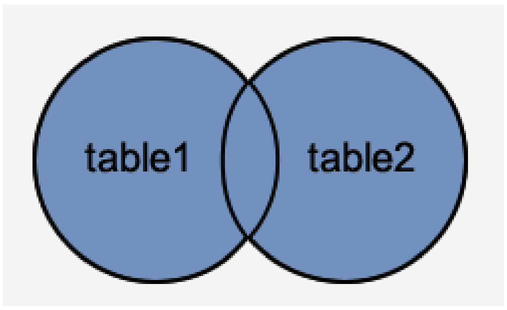
```
                                        120    Treasury
206         William     Gietz           110    Accounting
205         Shelley     Higgins         110    Accounting
```

Explanation: This query retrieves all rows in the `DEPARTMENTS` table, which is the right table, even if there is no match in the `EMPLOYEES` table

Code file: Code8_2_3.sql

FULL OUTER JOIN

This type of join returns all rows from the LEFT-hand table specified in the ON condition and on the RIGHT-hand table and those rows from the other table where the joined fields are equal (join condition is met).



1999 Syntax:

```
SELECT columns
FROM table1
FULL OUTER JOIN table2
ON table1.column = table2.column;
```

SQL Example:

```
SELECT      Employee_id,first_name, last_name, d.Department_id
            ,Department_Name
FROM        EMPLOYEES e
FULL OUTER JOIN Departments d
ON          e.department_id=d.department_id
ORDER BY    department_ID desc;
```

Query Results:

```
EMPLOYEE_ID FIRST_NAME  LAST_NAME   DEPARTMENT_ID     DEPARTMENT_NAME
178         Kimberely   Grant
                                              270    Payroll
                                              260    Recruiting
                                              250    Retail Sales
                                              240    Government
Sales
```

Explanation: This query retrieves all rows in the `DEPARTMENTS` table, which is the right table, even if there is no match in the `EMPLOYEES` table

Code file: Code8_2_4.sql

Lesson 8.4: Natural Joins

You can join tables automatically based on the columns in the two tables that have matching data types and names. You do this by using the `NATURAL JOIN` keywords.

Note 1: The join can happen on only those columns that have the same names and data types in both tables.

Note 2: If the columns have the same name but different data types, then the `NATURAL JOIN` syntax causes an error.

Note 3: If there are no columns with the same name a CROSS `JOIN` is created.

Syntax:

```
SELECT columns
FROM table1
NATURAL JOIN table2
```

SQL Example:

```
SELECT      Employee_id,first_name, last_name, Department_id ,
Department_Name
FROM        EMPLOYEES
NATURAL JOIN Departments
ORDER BY    department_ID desc;
```

Query Results:

```
EMPLOYEE_ID FIRST_NAME  LAST_NAME   DEPARTMENT_ID    DEPARTMENT_NAME
206         William     Gietz       110              Accounting
109         Daniel      Faviet      100              Finance
110         John        Chen        100              Finance
111         Ismael      Sciarra     100              Finance
…..
```

Explanation: The EMPLOYEES table is joined to the `DEPARTMENT` table by the Department `_ID` column, which is the only column of the same name in both tables. If other common

columns were present, the join would have used them all.

Code file: Code8_4_1.sql

Natural Joins using

Natural joins use all columns with matching names and data types to join the tables. The `USING` clause can be used to specify only those columns that should be used for an equijoin.

Syntax:

```
SELECT columns
FROM table1
INNER|LEFT|RIGHT|FULL JOIN table2
USING (column)
```

SQL Example:

```
SELECT       Employee_id,first_name, last_name, Department_id ,
Department_Name
FROM        EMPLOYEES
INNER JOIN  Departments
USING      (department_ID)
ORDER BY    department_ID desc;
```

Query Results:

```
EMPLOYEE_ID FIRST_NAME LAST_NAME  DEPARTMENT_ID     DEPARTMENT_NAME
206         William    Gietz      110               Accounting
109         Daniel     Faviet     100               Finance
110         John       Chen       100               Finance
111         Ismael     Sciarra    100               Finance
…..
```

Explanation: The EMPLOYEES table is joined to the `DEPARTMENT` table by the Department `_ID` column.

.

Code file: Code8_4_2.sql

When joining with the `USING` clause, you cannot qualify a column that is used in the `USING` clause itself. Furthermore, if that column is used anywhere in the SQL statement, you cannot alias it. The columns that are referenced in the `USING` clause should not have a qualifier (table name or alias) anywhere in the SQL statement. Other columns that are common in both the tables, but not used in the `USING` clause, must be prefixed with a table alias otherwise you get the “column ambiguously defined” error.

Natural Joins using ON

Natural joins use all columns with matching names and data types to join the tables. The USING clause can be used to specify only those columns that should be used for an equijoin.

Syntax:

```
SELECT columns
FROM table1
INNER|LEFT|RIGHT|FULL JOIN table2
USING (column)
```

SQL Example:

```
SELECT       Employee_id,first_name, last_name, Department_id ,
Department_Name
FROM         EMPLOYEES
INNER JOIN   Departments
USING       (department_ID)
ORDER BY     department_ID desc;
```

Query Results:

```
EMPLOYEE_ID FIRST_NAME  LAST_NAME   DEPARTMENT_ID      DEPARTMENT_NAME
206         William     Gietz       110                Accounting
109         Daniel      Faviet      100                Finance
110         John        Chen        100                Finance
111         Ismael      Sciarra     100                Finance
…..
```

Explanation: The EMPLOYEES table is joined to the DEPARTMENT table by the Department _ID column.

.

Code file: Code8_2_4.sql

Lesson 8.5: Cross Joins

When a join condition is invalid or omitted completely, the result is a `Cartesian product`, in which all combinations of rows are displayed. All rows in the first table are joined to all rows in the second table.

SQL 92 Syntax:

```
SELECT columns
FROM table1
CROSS JOIN table2
```

SQL Example:

```
SELECT      Employee_id,first_name, last_name, e.Department_id
, Department_Name
FROM        EMPLOYEES e
CROSS JOIN  Departments d;
```

SQL 99 Syntax:

```
SELECT columns
FROM table1 ,table2
```

SQL Example:

```
SELECT      Employee_id,first_name, last_name, e.Department_id
, Department_Name
FROM        EMPLOYEES ,  Departments d;
```

Query Results:

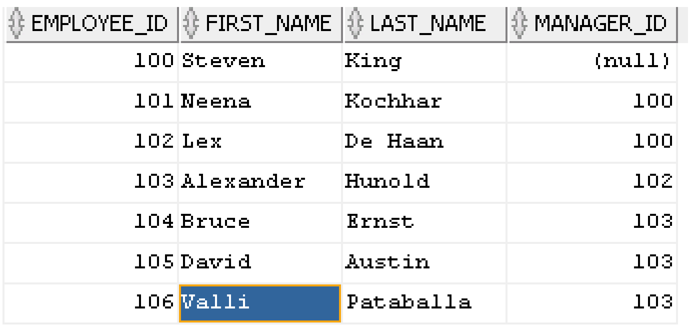
```
EMPLOYEE_ID FIRST_NAME  LAST_NAME   DEPARTMENT_ID      DEPARTMENT_NAME
100         Steven            King        90           Administration
101         Neena             Kochhar     90           Administration
102         Lex               De Haan     90           Administration
103         Alexander         Hunold      60           Administration
104         Bruce             Ernst       60           Administration
…..
```

Explanation: The EMPLOYEES table is cross joined to the `DEPARTMENT` table. Every department is repeated for every employee row

Code file: Code8_5_1.sql

Lesson 8.6: Join a table to itself by using a self-join

Sometimes you need to join a table to itself. To find the name of each employee's manager, you need to join the EMPLOYEES table to itself, or perform a self-join.



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	MANAGER_ID
100	Steven	King	(null)
101	Neena	Kochhar	100
102	Lex	De Haan	100
103	Alexander	Hunold	102
104	Bruce	Ernst	103
105	David	Austin	103
106	Valli	Pataballa	103

In the example above Alexander is Bruce's manager.

1999 Syntax:

```
SELECT columns
FROM table1 a
INNER JOIN table1 b
ON a.column = b.column;
```

SQL Example:

```
SELECT      b.Employee_id, b.first_name , b.last_name,
          a.Employee_id,a.first_name, a.last_name,a.manager_id
FROM        EMPLOYEES a
Inner join    EMPLOYEES b
On            a.employee_id=b.manager_id
```

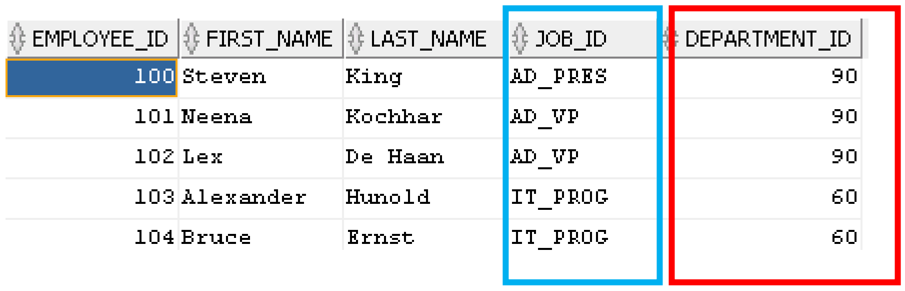
Query Results:

```
EMPLOYEE_ID FIRST_NAME  LAST_NAME   EMPLOYEE_ID_1     FIRST_NAME_1
        LAST_NAME_1 MANAGER_ID
101    Neena Kochhar      100    Steven       King
102    Lex    De Haan      100    Steven       King
103    Alexander    Hunold       102    Lex    De Haan      100
104    Bruce Ernst 103    Alexander    Hunold       102
105    David Austin       103    Alexander    Hunold       102
```

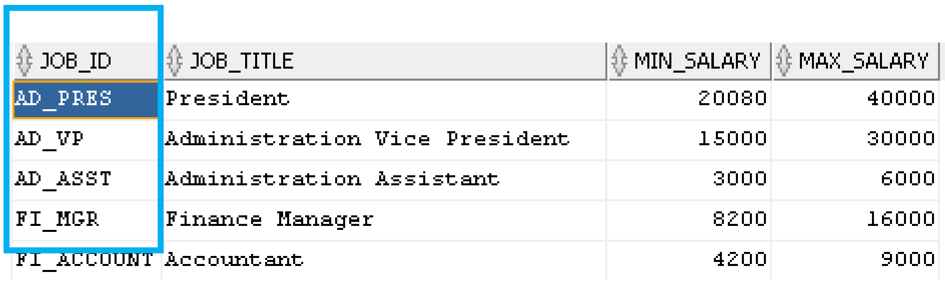
Explanation: This statement is a self-join of the EMPLOYEES table, based on the EMPLOYEE_ID and MANAGER_ID columns

Code file: Code8_6_1.sql

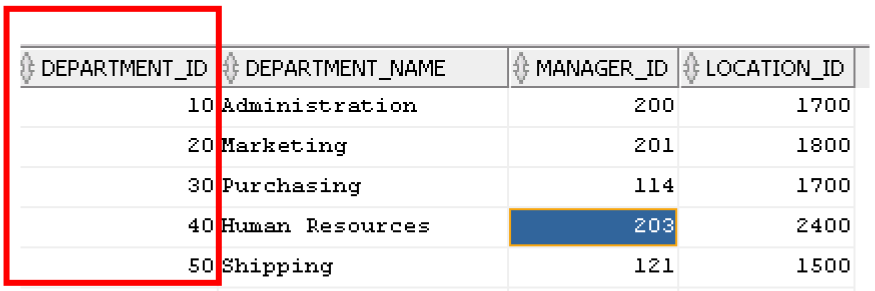
Lesson 8.7: Joining multiple tables



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	Steven	King	AD_PRES	90
101	Neena	Kochhar	AD_VP	90
102	Lex	De Haan	AD_VP	90
103	Alexander	Hunold	IT_PROG	60
104	Bruce	Ernst	IT_PROG	60



JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20080	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000



DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column
INNER JOIN table3
ON table1.column = table3.column
```

SQL Example:

```
SELECT E.EMPLOYEE_ID,
  E.FIRST_NAME,
  E.LAST_NAME,
  J.JOB_TITLE,
  D.DEPARTMENT_NAME
FROM JOBS J
INNER JOIN EMPLOYEES E
```

```
ON J.JOB_ID = E.JOB_ID
INNER JOIN DEPARTMENTS D
ON D.DEPARTMENT_ID = E.DEPARTMENT_ID
```

Query Results:

```
EMPLOYEE_ID      FIRST_NAME LAST_NAME  JOB_TITLE
     DEPARTMENT_NAME
200  Jennifer  Whalen    Administration             Assistant
     Administration
202  Pat   Fay   Marketing Representative   Marketing
201  Michael    Hartstein  Marketing Manager     Marketing
114  Den   Raphaely   Purchasing Manager    Purchasing
119  Karen Colmenares Purchasing Clerk Purchasing
```

Explanation: This statement shows the `EMPLOYEES` table joined to Jobs table and department table

Code file: Code8_7_1.sql

*.

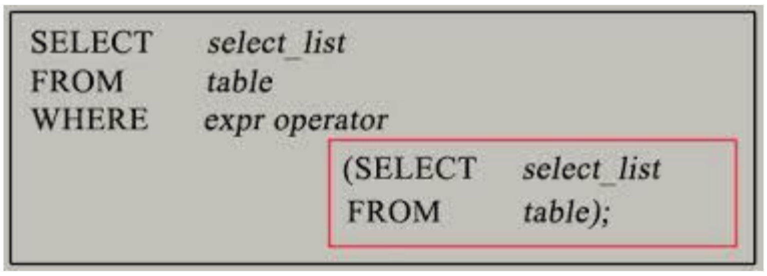
Section 9: Use Sub-queries to Solve Queries

In this section you will:

- Describe the types of problem that sub-queries can solve
- Define sub-queries
- List the types of sub-queries
- Write single-row and multiple-row sub-queries

Lesson 9.1: Overview of sub-queries

Subqueries are used to combining multiple queries into one result table. A sub query can be thought of as a temporary table, it is created when the whole query is run and then discarded when the primary query is finished.



```
SELECT    select_list
FROM      table
WHERE     expr operator
                    (SELECT    select_list
                     FROM      table);
```

Lesson 9.2: Define sub-queries

Sub queries the use of a SELECT statement inside one of the clauses of another SELECT statement. In fact, a subquery can be contained inside another subquery, which is inside another subquery, and so forth. A subquery can also be nested inside INSERT, UPDATE, and DELETE statements. Subqueries must be enclosed within parentheses. A subquery can be used any place where an expression is allowed providing it returns a sing le value. This means that a subquery that returns a sing le value can also be listed as an object in a FROM clause listing. This is termed an inline view because when a subquery is used as part of a FROM clause, it is treated like a virtual table or view. Subquery can be placed either in FROM clause, WHERE clause or HAVING clause of the main query. Oracle allows a maximum nesting of 255 subquery levels in a WHERE clause. There is no limit for nesting subqueries expressed in a FROM clause. In practice, the limit of 255 levels is not really a limit at all because it is rare to encounter subqueries nested beyond three or four levels. A subquery SELECT statement is very similar to the SELECT statement used to beg in a regular or outer query.

The complete syntax of a subquery is:

```
SELECT [DISTINCT] subquery_select_parameter
FROM {table_name | view_name}
{table_name | view_name} ...
[WHERE search_conditions]
[GROUP BY column_name [,column_name ] ...]
[HAVING search_conditions] )
```

Lesson 9.3: List the types of sub-queries

Single Row Sub Query: Sub query which returns sing le row output. They mark the usage of single row comparison operators, when used in WHERE conditions.

Multiple row sub query: Sub query returning multiple row output. They make use of multiple row comparison operators like IN, ANY, ALL. There can be sub queries returning multiple columns also.

Correlated Sub Query: Correlated subqueries depend on data provided by the outer query. This type of subquery also includes subqueries that use the EXIST S operator to test the existence of data rows satisfying specified criteria.

Lesson 9.4: Write single-row and multiple-row sub-queries

Single-row

A sing le-row subquery is used when the outer query's results are based on a sing le, unknown value. Although this query type is formally called "sing le-row," the name implies that the query returns multiple columns-but only one row of results. However, a sing le-row subquery can return only one row of results consisting of only one column to the outer query.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE column = (Subquery) ];
```

SQL Example:

```
SELECT first_name, salary, department_id
FROM employees
WHERE salary = (SELECT MIN (salary)
FROM employees);
```

Query Results:

```
FIRST_NAME SALARY     DEPARTMENT_ID
TJ         2100       50
…
```

Explanation: The inner SELECT query returns only one row i.e. the minimum salary for the company. It in turn uses this value to compare salary of all the employees and displays only those, whose salary is equal to minimum salary.

Code file: Code9_4_1.sql

Multiple-row

Multiple-row subqueries are nested queries that can return more than one row of results to the parent query. Multiple-row subqueries are used most commonly in WHERE and HAVING clauses. Since it returns multiple rows, it must be handled by set comparison operators (IN, ALL, ANY).While IN operator holds the same meaning as discussed in earlier chapter, ANY operator compares a specified value to each value returned by the sub query while ALL compares a value to every value returned by a sub query.

Usage of Multiple Row operators

- [> ALL] More than the highest value returned by the subquery
- [< ALL] Less than the lowest value returned by the subquery
- [< ANY] Less than the highest value returned by the subquery
- [> ANY] More than the lowest value returned by the subquery
- [= ANY] Equal to any value returned by the subquery (same as IN)

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE column >[ALL][ANY] [IN] (Subquery) ];
```

SQL Example:

```
SELECT first_name, department_id
FROM employees
WHERE department_id IN (SELECT department_id
FROM departments
WHERE LOCATION_ID =1700)
```

Query Results:

```
FIRST_NAME DEPARTMENT_ID
Shelli     30
John       100
Karen      30
Lex        90
Daniel     100
…
```

Explanation: IN matches department ids returned from the sub query, compares it with that in the main query and returns employee's name who satisfy the condition

Code file: Code9_4_2.sql

Multiple Column Subqueries

As the name suggests this type of query has multiple columns within the sub query. The condition used to match the sub query must have the same number of columns as the sub query.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE column1, column2(s) operator (SELECT column1, column2 ….];
```

SQL Example:

```
SELECT last_name, first_name, manager_id, department_id
FROM         employees
WHERE (manager_id, department_id) in (SELECT manager_id,
department_id
                                      FROM   employees
                                      WHERE  last_name = 'King');
```

Query Results:

```
LAST_NAME  FIRST_NAME MANAGER_ID DEPARTMENT_ID
King       Janette    146        80
Sully      Patrick    146        80
McEwen     Allan      146        80
Smith      Lindsey    146        80
Doran      Louise     146        80
…
```

Explanation: The `SELECT` statement returns all employees with the same Manager and department as employee King

Code file: Code9_4_3.sql

Lesson 9.5: Correlated Sub queries

Work in the opposite direction to standard subqueries. This subquery depends on data retrieved from the master query. As opposed to a regular subquery, where the outer query depends on values provided by the inner query, a correlated subquery is one where the inner query depends on values provided by the outer query. This means that in a correlated subquery, the inner query is executed repeatedly, once for each row that might be selected by the outer query.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT EMPLOYEE_ID, salary, department_id
FROM employees E
WHERE salary > (SELECT AVG(salary)
FROM employees T
WHERE E.department_id = T.department_id)
```

Query Results:

```
EMPLOYEE_ID      SALARY      DEPARTMENT_ID
100              24000       90
103              9000        60
104              6000        60
108              12008       100
109              9000        100
```

Explanation: The subquery in this SELECT statement cannot be resolved independently of the main query. The outer query specifies that rows are selected from the employee table with an alias name of e. The inner query compares the employee department number column (DepartmentID) of the employee table with alias T to the same column for the alias table name e.

Code file: Code9_4_4.sql

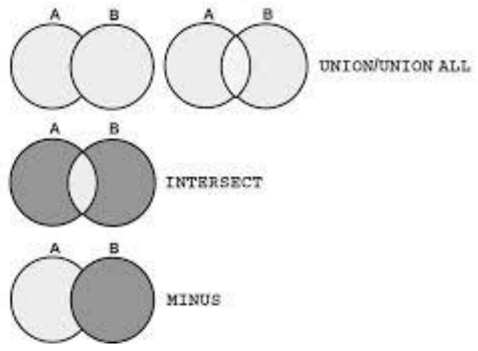
Section 10: The SET Operators

In this section you will:

- Describe the SET operators
- Use a SET operator to combine multiple queries into a single query
- Control the order of rows returned

Lesson 10.1: Describe the SET operators – UNION, INTERSECT, MINUS

Set operators are used to join the results of two (or more) SELECT statements. The SET operators available in Oracle 11g are UNION, UNION ALL, INTERSECT and MINUS.



The UNION set operator returns the combined results of the two SELECT statements. It removes duplicates from the results i.e. only one row will be listed for each duplicated result. To counter this behavior, use the UNION ALL set operator which retains the duplicates in the final result.

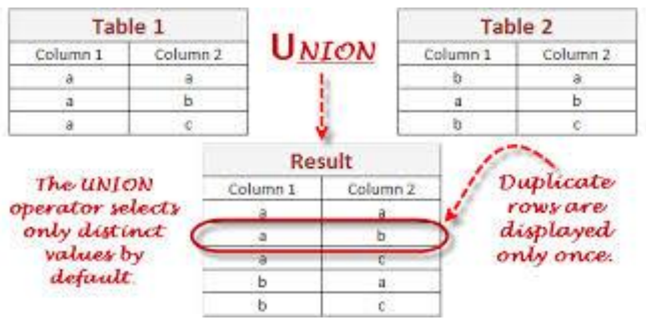

UNION
The UNION operator selects only distinct values by default
Duplicate rows are displayed only once.

Table 1	
Column 1	Column 2
a	a
a	b
a	c

Table 2	
Column 1	Column 2
b	a
a	b
b	c

Result	
Column 1	Column 2
a	a
a	b
a	c
b	a
b	c

INTERSECT lists only records that are common to both the SELECT queries; the MINUS set operator removes the second query's results from the output if they are also found in the first query's results. INTERSECT and MINUS set operations produce unduplicated results.

Lesson 10.2: Use a SET operator to combine multiple queries into a single query

UNION

This operator will combine the results of multiple select statements and eliminates copies of duplicate values.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)]
UNION [ALL]
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT JOB_ID
FROM employees
WHERE DEPARTMENT_ID = 10
UNION
SELECT JOB_ID
FROM employees
WHERE DEPARTMENT_ID = 20;
```

Query Results:

```
JOB_ID
AD_ASST
MK_MAN
MK_REP
```

Explanation: The `SELECT` statement job ids for department 10 and append jobs id from department 20

Code file: Code 10_2_1.sql

INTERSECT

This operator will combine the results of multiple select statements and only return rows that appear in both queries

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)]
INTERSECT
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT salary
FROM employees
WHERE DEPARTMENT_ID = 50
INTERSECT
SELECT salary
FROM employees
WHERE DEPARTMENT_ID = 60;
```

Query Results:

```
SALARY
4200
```

Explanation: SELECT query retrieves the salary which are common in department 50 and 60

Code file: Code 10_2_2.sql

MINUS

This operator will take the rows from first query and removes the rows returned by the second query.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)]
MINUS
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT salary
FROM employees
WHERE DEPARTMENT_ID = 50
MINUS
SELECT salary
FROM employees
WHERE DEPARTMENT_ID = 60;
```

Query Results:

```
SALARY
4200
```

Explanation: SELECT query retrieves the salary which in department 50 and remove the rows in department 60

Code file: Code 10_2_3.sql

Lesson 10.3: Control the order of rows returned

The ORDER BY clause can appear only once at the end of the query containing compound SELECT statements. It implies that individual SELECT statements cannot have ORDER BY clause. Additionally, the sorting can be based on the columns which appear in the first SELECT query only. For this reason, it is recommended to sort the compound query using column positions.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)]
MINUS|UNION |INTERSECT
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
[WHERE condition(s)];
```

SQL Example:

```
SELECT employee_id, first_name, salary
FROM employees
WHERE department_id=10
UNION
SELECT employee_id, first_name, salary
FROM employees
WHERE department_id=20
ORDER BY 3;
```

Query Results:

```
EMPLOYEE_ID      FIRST_NAME SALARY
200              Jennifer   4400
202              Pat        6000
201              Michael    13000
```

Explanation: SELECT query unifies the results from two departments and sorts by the SALARY column

Code file: Code 10_3_1.sql

Section 11: Data Manipulation Statements and ETL

In this section you will:

- Describe each DML statement
- Insert rows into a table
- Change rows in a table by the UPDATE statement
- Delete rows from a table with the DELETE statement
- Save and discard changes with the COMMIT and ROLLBACK statements
- Explain read consistency

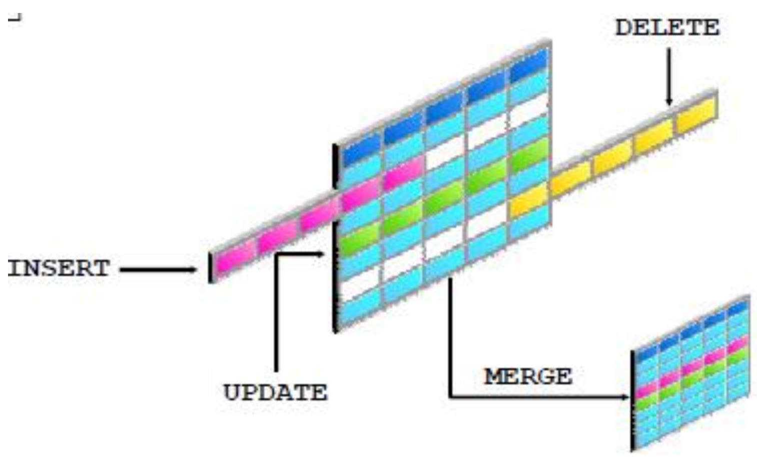
Lesson 11.1: Describe each DML statement

DML (Data Manipulation Language) is the subset of SQL used to access and manipulate data contained within the data structures previously defined via data definition language (DDL).

DML comprises four statements.

SELECT
INSERT
UPDATE
DELETE
MERGE

- The SELECT statement is a limited form of DML statement in that it can only access data in the database. Unlike the other statements it cannot manipulate data in the database.
- The INSERT statement adds new rows to the table.
- The UPDATE statement modifies existing rows in the table.
- The DELETE statement removes rows from the table.
- The MERGE statement selects rows from one table to update or insert into another table.



Lesson 11.2: Insert rows into a table

Inserting Data into One Table

Single table inserts only allowing inserting data into one table at a time. The data to be inserted can be queried from multiple tables. Not all the columns in the table need to be inserted but be aware of columns that require non-NULL values. Values to be inserted must be compatible with the data type of the column. Literals, fixed values and special values like functions, SYSDATE, CURRENT_DATE, SEQ.CURRVAL (NEXTVAL), or USER can be used as column values. Values specified must follow the generic rules. String literals and date values must be enclosed within quotes. Date value can be supplied in DD-MON-RR or D-MON-YYYY format, but YYYY is preferred since it clearly specifies the century and does not depend on internal RR century calculation logic.

Syntax:

```
INSERT INTO table(column1, column2….)
VALUES (column1 value, column2 value, ...);
```

SQL Example:

```
INSERT INTO employees (EMPLOYEE_ID, FIRST_NAME,last_name,
SALARY, DEPARTMENT_ID,email, hire_date,job_id)
VALUES (99, 'Kee','John', 3800, 10,'a@b.ie', '10-JAN-
2014','IT_PROG');
```

Query Results:

```
1 rows inserted.
```

Explanation: Add a new employee record in the EMPLOYEES table. it inserts the values for the primary columns EMPLOYEE_ID, FIRST_NAME,last_name, SALARY, DEPARTMENT_ID,email, hire_date,job_id

Code file: Code11_2_1.sql

Another example of inserting a single row in a table

Syntax:

```
INSERT INTO     table [(column [, column...])]
VALUES          (value [, value...]);
```

SQL Example:

```
INSERT INTO departments
VALUES (300, 'Infrastructure', 100, 1700);


INSERT INTO departments(department_id,
       department_name, manager_id, location_id)
VALUES (310, 'Invoicing', 100, 1700);
```

Explanation:	In the syntax:	
	table	is the name of the table
	column	is the name of the column in the table to populate
	value	is the corresponding value for the column

Both these examples each insert a single row at a time into the DEPARTMENTS table. As you can insert a new row that contains values for each column, the column list is not required in the INSERT clause. If you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column – the first example shows this technique. It is, however, good practice to specify the column listing in the INSERT statement as in the second example.

You will need to enclose character and date values in single quotation marks. Dates will also need to be in the database date default format e.g. DD-MON-YY unless you use conversion functions such as TO_DATE

Code file: Code11_2_2.sql

Inserting nulls

There are two ways of inserting NULL values into a column:

IMPLICT - remove the column from the column list.

EXPLICT - specify the NULL keyword in the VALUES list.

SQL Example: IMPLICIT method:

```
INSERT INTO departments(department_id, department_name)
VALUES (320, 'Auditing');
```

EXPLICIT method:

```
INSERT INTO departments
VALUES (330, 'Banking', NULL, NULL);

A combined method:

INSERT INTO departments(department_id,
       department_name, manager_id, location_id)
VALUES (340, 'Networking', NULL, 1700);
```

Explanation: The first example inserts a row into the DEPARTMENTS table with NULL values in the LOCATION_ID and MANAGER_ID columns by omitting those columns in the column listings. The second example does the same thing but omits all the column listings so the values are put positionally ie 110 in the DEPARTMENT_ID and Public Relations in the DEPARTMENT_NAME. NULLs are put in the remaining columns. The third example just puts a NULL into the MANAGER_ID column.

Code file: Code11_2_3.sql

Inserting Data in Multiple Tables

The INSERT statement allows one INSERT to insert rows into multiple tables. This is particularly useful in data warehouse applications where data may require to be loaded into multiple tables after being extracted from one or more table. Normally this type of action would require multiple insert statement.

ALL – this allow an unconditional insert to occur, i.e. no condition added

WHEN – this allows a conditional insert to occur. This is similar to an IF statement in PL/SQL.

Unconditional Insert

This allows data to be inserted into one or more table base on a select statement.

The syntax required is:

Syntax:

```
INSERT ALL
      INTO table1 [(cols..)]
      VALUES (value1…)
      INTO table2 [(cols..)]
      VALUES (value1)
SELECT ….;
```

SQL Example:

```
INSERT ALL
INTO emp_stage (emp_id, first_name, last_name)
VALUES(emp_id, first_name, last_name)
INTO car_stage (regno, emp_id, make, model)
VALUES(regno, emp_id, make, model)
SELECT e.emp_id,
      e.first_name,
      e.last_name,
      c.regno,
      c.make,
      c.model
FROM employees e, cars c
WHERE e.emp_id = c.emp_id
```

Query Results:

Explanation: In this example we have create 2 empty tables called emp_stage and car_stage which will have data loaded from the cars and employees' tables.

Code file: Code11_2_4.sql

Conditional Insert Statements

It is possible to provide a condition to an insert statement using the WHEN clause. This behaves in a similar way to an IF statement in PL/SQL where based on the evaluation of a condition a specific insert will occur. There are two syntax options available with the WHEN clause of the INSERT statement. The INSERT FIRST will evaluate the WHEN clause for a row if it finds a true the remaining WHEN statements will be ignored for the current row. The INSERT ALL will evaluate each WHEN statement independently, this means that potentially one row could be evaluated in the sub query which would cause multiple inserts for each WHEN statement.

SQL Example:

```
INSERT ALL
   WHEN jobtitle like '%MANAGER%'
     INTO manager_info
     VALUES(emp_id, first_name, last_name)
   WHEN dept_id in (10,20,30,40)
     INTO emp_info
     VALUES(emp_id, first_name, last_name)
SELECT    emp_id,
     first_name,
     last_name,
     jobtitle
from employees;
```

Query Results:

Explanation: When the job title is manager insert into manager info. When the employee is in a specific department insert into emp_info.

Code file: Code11_2_3.sql

Lesson 11.3: Change rows in a table by the UPDATE statement

The UPDATE statement allows data that has already been entered to be updated. UPDATE uses a WHERE clause to specify which rows are to be updated. If more than one column in a table is to be amended then there are two choices of update. First option is to provide a set of columns/value pairs separated by commas or the second option is to provide a set of columns and a subquery.

Syntax:

```
UPDATE tablename  table_alias
SET column 1 = expression 1
WHERE conditions;
```

SQL Example:

```
UPDATE     employees
SET        salary = 1.05 * salary
WHERE      COMMISSION_PCT IS NOT NULL
```

Query Results: …

Explanation: Update statement increase the salary by 5%

Code file: Code11_3_1.sql

Change rows in a table by using a sub-query

You can update columns in the SET clause of an UPDATE statement by writing sub-queries. Sub-queries can also be used in the WHERE clause to define the rows to be updated.

Syntax:

```
UPDATE table
SET    column  = (sub-query)                         (
       [ , column  = (sub-query)]
[WHERE  condition ]
```

SQL Example:

```
UPDATE employees
SET    department_id = (SELECT department_id
                        FROM departments
                        WHERE department_name =
                        'Marketing')

WHERE  salary = (SELECT salary
                 FROM employees
                 WHERE job_id = 'AD_PRES' );
```

Explanation: This example changes the DEPARTMENT_ID to that of the Marketing department for all those employees whose salary is the same as that of the AD_PRES ‘s JOB_ID

Code file: Code11_3_2.sql

Merging Data

When data is being loaded there may be occasions to perform both an insert and an update, this is often referred to as an Upsert.

Syntax:

```
MERGE INTO table_to_merge_to
USING table_to_merge_from
ON (primary_key = primary_key)
WHEN matched THEN
UPDATE SET column = column
WHEN NOT MATCHED THEN
INSERT (columns…..)
VALUES (values……);
```

SQL Example:

```
merge into dw_customers d
using customer_2002 c
on (d.id=c.id)
when matched then
update set d.name=c.name,
           d.post_code=c.post_code
when not matched then
insert (id, name, post_code)
values (c.id, c.name, c.post_code)
```

Explanation: We have a customers table that contains a list of our customer's data which is being constantly added to and amended. We also have a DW_CUSTOMERS table in our data warehousing that contains the id, name and postcode of the each customer. If the Customer exists in DW_CUSTOMERS table then update the fields otherwise insert the record

Code file: Code11_3_3.sql

Lesson 11.4: Delete rows from a table with the DELETE statement

The DELETE statement allows for deleting of specific data held within a table. The DELETE statement has a where clause to allow you to delete specific data or if the WHERE clause is forgotten then all the data from the table will be deleted.

A subquery can be used within the where clause if required. Another option is to specify how many rows to delete.

Syntax:

```
DELETE FROM tablename

WHERE condition(s);
```

SQL Example:

```
DELETE FROM     employees

WHERE           manager_id = 1001;
```

Explanation: Delete employees where the manager is 1001

Code file: Code11_4_1.sql

Removing rows from a table using a sub-query

You can use sub-queries to delete rows from a table based on values from another table.

Syntax:

```
DELETE [FROM] table
[WHERE        sub-query];
```

SQL Example:

```
DELETE FROM employees
WHERE  department_id = (SELECT department_id
FROM departments
WHERE department_name = 'Banking');
```

Explanation: The example deletes all employees from the EMPLOYEES tables who are in the Banking department. The Banking department's id is returned via the sub-query.

Code file: Code11_4_2.sql

Truncating Tables

Truncating tables is a more common method of deleting data from a table. Unlike the DELETE command it is not possible to reverse the deletion by using the ROLLBACK command. However, the TRUNCATE command is more efficient and quicker than using the DELETE command.

Syntax:

```
TRUNCATE table_name;
```

Example:

```
TRUNCATE employees;
```

Lesson 11.5: Save and discard changes with the COMMIT and ROLLBACK statements

A transaction is a logical unit of work done in database. It can either contain:

- Multiple DML commands ending with a TCL command i.e. COMMIT or ROLLBACK
- One DDL command
- One DCL command

Beginning of a transaction is marked with the first DML command. It ends with a TCL, DDL or DCL command. A TCL command i.e. COMMIT or ROLLBACK is issues explicitly to end an active transaction. By virtue of their basic behavior, if any of DDL or DCL commands get executed in a database session, commit the ongoing active transaction in the session. If the database instance crashes abnormally, the transaction is stopped.

COMMIT, ROLLBACK and SAVEPOINT are the transaction control language. COMMIT applies the data changes permanently into the database while ROLLBACK does anti-commit operation. SAVEPOINT controls the series of a transaction by setting markers at different transaction stages. User can roll back the current transaction to the desired save point, which was set earlier.

COMMIT - Commit ends the current active transaction by applying the data changes permanently into the database tables. COMMIT is a TCL command which explicitly ends the transaction. However, the DDL and DCL command implicitly commit the transaction.

SAVEPOINT - Savepoint is used to mark a specific point in the current transaction in the session. Since it is logical marker in the transaction, savepoints cannot be queried in the data dictionaries.

ROLLBACK - The ROLLBACK command is used to end the entire transaction by discarding the data changes. If the transaction contains marked savepoints, ROLLBACK TO SAVEPOINT [name] can be used to roll back the transaction upto the specified savepoint only. As a result, all the data changes upto the specified savepoint will be discarded.

Syntax:

Transactions with end following either:

```
COMMIT; --Confirms changes made
ROLLBACK; --Removed changes made since the last COMMIT
```

Consider the EMPLOYEES table which gets populated with newly hired employee details during first quarter of every year. The clerical staff appends each employee detail with a savepoint, so as to rollback any faulty data at any moment during the data feeding activity. Note that he keeps the savepoint names same as the employee names.

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table
WHERE condition(s)];
```

SQL Example:

```
INSERT   INTO   employees   (employee_id,   first_name,
hire_date, job_id, salary, department_id)
VALUES      (105,      'Allen',TO_DATE      ('15-JAN-
2013','SALES',10000,10);

SAVEPOINT Allen;

INSERT   INTO   employees   (employee_id,   first_name,
hire_date, job_id, salary, department_id)
VALUES      (106,      'Kate',TO_DATE      ('15-JAN-
2013','PROD',10000,20);

SAVEPOINT Kate;

INSERT   INTO   employees   (employee_id,   first_name,
hire_date, job_id, salary, department_id)
VALUES      (107,      'McMan',TO_DATE      ('15-JAN-
2013','ADMIN',12000,30);

SAVEPOINT McMan;


ROLLBACK TO SAVEPOINT Kate;

INSERT   INTO   employees   (employee_id,   first_name,
hire_date, job_id, salary, department_id)
VALUES      (106,      'Kate',TO_DATE      ('15-JAN-
2013','PROD',12500,20);

SAVEPOINT Kate;

INSERT   INTO   employees   (employee_id,   first_name,
hire_date, job_id, salary, department_id)
VALUES      (107,      'McMan',TO_DATE      ('15-JAN-
2013','ADMIN',13200,30);

SAVEPOINT McMan;

Commit
```

Query Results: Rolls back the active transaction to the savepoint Kate and re-enters the employee details for Kate and McMan

.

Code file: Code11_5_1.sql

Lesson 11.6: ETL

ETL comes from Data Warehousing and stands for Extract-Transform-Load. ETL covers a process of how the data are loaded from the source system to the data warehouse. Currently, the ETL encompasses a cleaning step as a separate step. The sequence is then Extract-Clean-Transform-Load.

Extract

The Extract step covers the data extraction from the source system and makes it accessible for further processing. The main objective of the extract step is to retrieve all the required data from the source system with as little resources as possible. The extract step should be designed in a way that it does not negatively affect the source system in terms or performance, response time or any kind of locking.

There are several ways to perform the extract:

- Update notification - if the source system is able to provide a notification that a record has been changed and describe the change, this is the easiest way to get the data.
- Incremental extract - some systems may not be able to provide notification that an update has occurred, but they are able to identify which records have been modified and provide an extract of such records. During further ETL steps, the system needs to identify changes and propagate it down. Note, that by using daily extract, we may not be able to handle deleted records properly.
- Full extract - some systems are not able to identify which data has been changed at all, so a full extract is the only way one can get the data out of the system. The full extract requires keeping a copy of the last extract in the same format in order to be able to identify changes. Full extract handles deletions as well.
- When using Incremental or Full extracts, the extract frequency is extremely important. Particularly for full extracts; the data volumes can be in tens of gigabytes.

Clean

The cleaning step is one of the most important as it ensures the quality of the data in the data warehouse. Cleaning should perform basic data unification rules, such as:

- Making identifiers unique (Man/Woman/Not Available are translated to standard Male/Female/Unknown)
- Convert null values into standardized Not Available/Not Provided value
- Convert phone numbers, ZIP codes to a standardized form
- Validate address fields, convert them into proper naming, e.g. Street/St/St./Str./Str
- Validate address fields against each other (State/Country, City/State, City/ZIP code, City/Street).

Transform

The transform step applies a set of rules to transform the data from the source to the target. This includes converting any measured data to the same dimension (i.e. conformed dimension) using the same units so that they can later be joined. The transformation step also requires joining data from several sources, generating aggregates, generating surrogate keys, sorting, deriving new calculated values, and applying advanced validation rules.

Load

During the load step, it is necessary to ensure that the load is performed correctly and with as little resources as possible. The target of the Load process is often a database. In order to make the load process efficient, it is helpful to disable any constraints and indexes before the load and enable them back only after the load completes. The referential integrity needs to be maintained by ETL tool to ensure consistency. Managing ETL Process

The ETL process seems quite straight forward. As with every application, there is a possibility that the ETL process fails. This can be caused by missing extracts from one of the systems, missing values in one of the reference tables, or simply a connection or power outage. Therefore, it is necessary to design the ETL process keeping fail-recovery in mind.

Staging

It should be possible to restart, at least, some of the phases independently from the others. For example, if the transformation step fails, it should not be necessary to restart the Extract step. We can ensure this by implementing proper staging. Staging means that the data is simply dumped to the location (called the Staging Area) so that it can then be read by the next processing phase. The staging area is also used during ETL process to store intermediate results of processing. This is ok for the ETL process which uses for this purpose. However, tThe staging area should is be accessed by the load ETL process only. It should never be available to anyone else; particularly not to end users as it is not intended for data presentation to the end-user.may contain incomplete or in-the-middle-of-the-processing data.

ETL Tool Implementation

When you are about to use an ETL tool, there is a fundamental decision to be made: will the company build its own data transformation tool or will it use an existing tool?

Building your own data transformation tool (usually a set of shell scripts, sql scripts) is the preferred approach for a small number of data sources which reside in storage of the same type. The reason for that is the effort to implement the necessary transformation is little due to similar data structure and common system architecture.

There are many ready-to-use ETL tools on the market. The main benefit of using off-the-shelf ETL tools is the fact that they are optimized for the ETL process by providing connectors to common data sources like databases, flat files, mainframe systems, xml, etc. They provide a means to implement data transformations easily and consistently across various data sources. This includes filtering, reformatting, sorting, joining, merging, aggregation and other operations ready to use. The tools also support transformation scheduling, version control, monitoring and unified metadata management. Some of the ETL tools are even integrated with BI tools. The most well known commercial tools are Ab Initio, IBM InfoSphere DataStage, Informatica, Oracle Data Integrator and SAP Data Integrator. There are several open source ETL tools, among others Apatar, CloverETL, Pentaho and Talend.

Section 12: Introduction to PL/SQL

In this section you will:

- Overview PL/SQL
- Identify the benefits of PL/SQL Subprograms
- Overview the types of PL/SQL blocks
- Create a Simple Anonymous Block
- Generate output from a PL/SQL Block

Lesson 12.1: Overview of PL/SQL

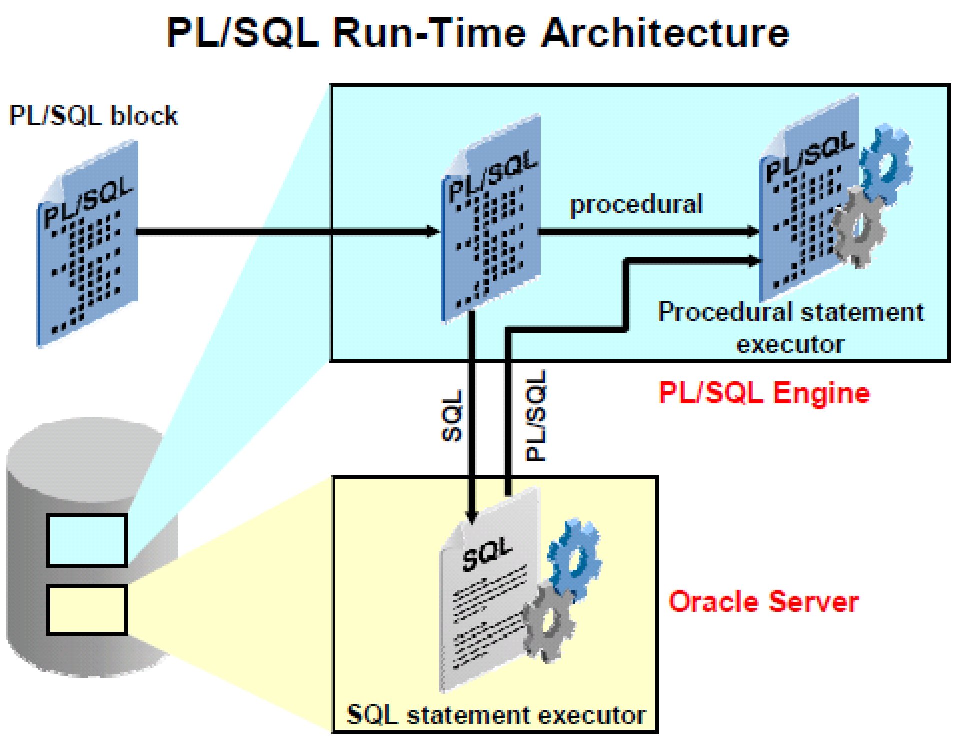
The early versions of the Oracle database were only equipped with the 4th generation language of SQL (Structured Query Language). Although a very powerful language, SQL does not contain the 3rd generation language features required by developers. These are provided by the PL/SQL language which was introduced to the Oracle Database in version 6 and has developed with each new release of the database. The PL/SQL acronym stands for the Procedural Language of SQL and unlike other programming languages it is not a standalone language, i.e. it only exists in Oracle.

PL/SQL defines a block structure for writing code. Maintaining and debugging code is made easier with such a structure because you can easily understand the flow and execution of the program unit.PL/SQL offers data encapsulation, exception handling, information hiding, and object orientation.

There are several versions of the PL/SQL:

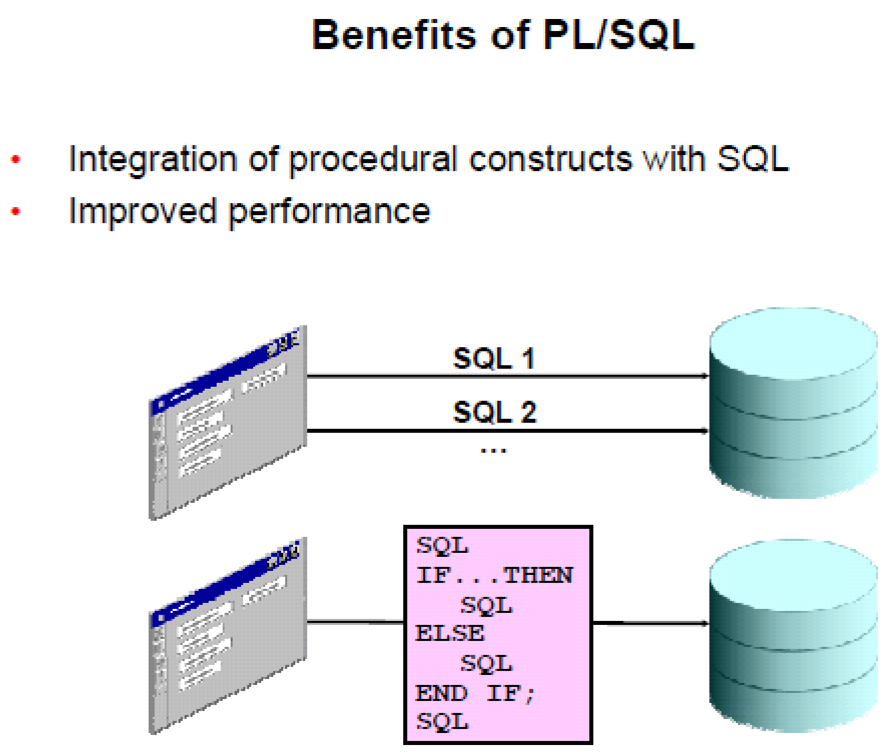
PL/SQL Version	Database Version
Version 2.0 – 2.3	Oracle 7
Version 8.0	Oracle 8
Version 8.1	Oracle 8.1
Version 9.1	Oracle 9.1
Version 10.2	Oracle 10g
Version 11.1	Oracle 11g
Version 12	Oracle 12g

PL/SQL runtime



All PL/SQL statements are processed in the Procedural Statement Executor, and all SQL statements must be sent to the SQL Statement Executor for processing by the Oracle Server processes. The SQL environment may also invoke the PL/SQL environment. For example, the PL/SQL environment is invoked when a PL/SQL function is used in a SELECT statement. The PL/SQL engine is a virtual machine that resides in memory and processes the PL/SQL m-code instructions. When the PL/SQL engine encounters a SQL statement, a context switch is made to pass the SQL statement to the Oracle Server processes. The PL/SQL engine waits for the SQL statement to complete and for the results to be returned before it continues to process subsequent statements in the PL/SQL block.

Lesson 12.2: Identify the benefits of PL/SQL Subprograms



Benefits of PL/SQL

- Integration of procedural constructs with SQL
- Improved performance

```
SQL
IF...THEN
   SQL
ELSE
   SQL
END IF;
SQL
```

Benefits of PL/SQL

Integration of procedural constructs with SQL

PL/SQL integrates procedural constructs with SQL. SQL is a nonprocedural language. PL/SQL integrates control statements and conditional statements with SQL, giving you better control of your SQL statements and their execution

Improved performance:

PL/SQL, you would not be able to logically combine SQL statements as one unit. The application can send the entire block to the database instead of sending the SQL statements one at a time. This significantly reduces the number of database calls.

Modularized program development:

The basic unit in all PL/SQL programs is the block. Blocks can be in a sequence or they can be nested in other blocks.

Maintain and debug code.

In PL/SQL, modularization is implemented using procedures, functions, and packages. These units can be run in debug mode this allowing Step into and other debugging tools.

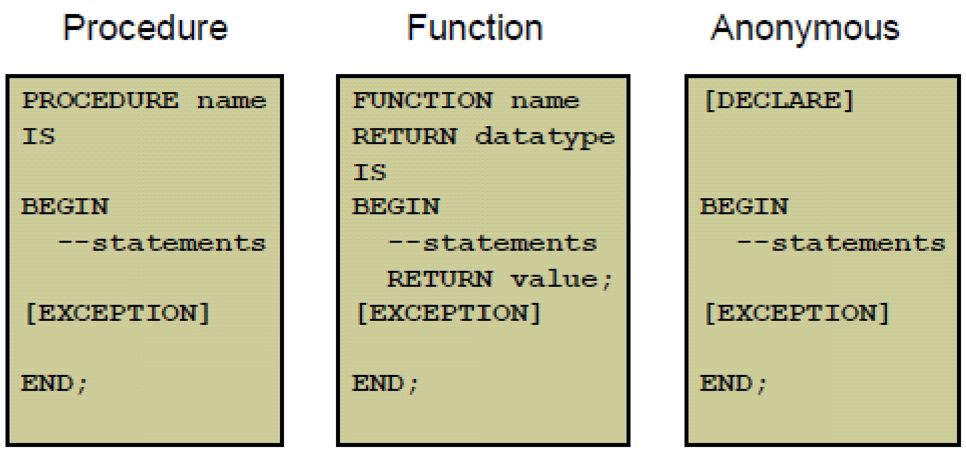
Exception handling:

PL/SQL enables you to handle exceptions efficiently. You can define separate blocks for dealing with exceptions.

Lesson 12.3: Overview of the types of PL/SQL blocks

PL/SQL allows the user to develop blocks of code. There are several different types of block that can be created:

- Anonymous Blocks
- Procedures
- Functions



```
PROCEDURE name
IS

BEGIN
   --statements

[EXCEPTION]

END;
```

```
FUNCTION name
RETURN datatype
IS
BEGIN
   --statements
   RETURN value;
[EXCEPTION]

END;
```

```
[DECLARE]


BEGIN
   --statements

[EXCEPTION]

END;
```

Procedures

Procedures are named objects that contain SQL and/or PL/SQL statements.

Functions

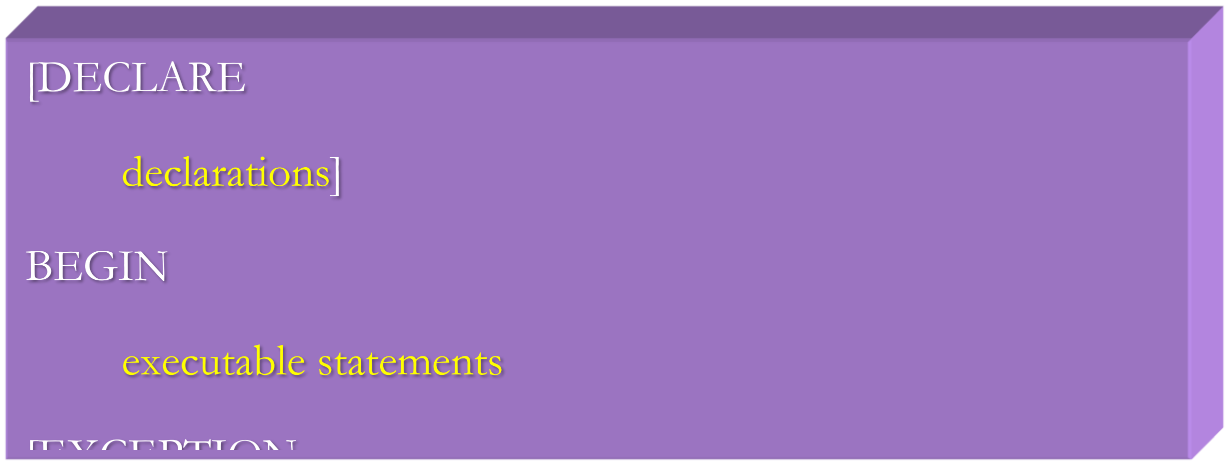
Functions are named objects that contain SQL and/or PL/SQL statements. Unlike a procedure, a function returns a value of a specified data type.

Anonymous blocks

Anonymous blocks are unnamed blocks. They are declared inline at the point in an application where they are to be executed and are compiled each time the application is executed. These blocks are not stored in the database. They are passed to the PL/SQL engine for execution at run time.

Lesson 12.4: Anonymous Block

The diagram below shows the structure of an anonymous block.



This is an optional section, which is used to define items that will be used in the executable section of the code. The main items that will be created in this section are:

Variables – temporary objects created to hold data, which can be changed during the execution of the code. For example, an object to hold the result of a calculation.

Constants – temporary objects created to hold data, which cannot be changed during the execution of the code. For example, a tax rate.

Explicit Cursors – these are used to retrieve data from the database. We will cover Cursors in detail later in this course.

Collections – these are used to group lists of related data together. These are more advanced techniques and will be introduced towards the end of this course.

Begin Section

The Begin section is the executable part of the PL/SQL code. This section can contain both SQL and PL/SQL commands. Therefore, it is possible to enter SQL operations such as:

- DML Commands INSERT, UPDATE, DELETE and a specific type of SELECT statement
- Functions – for example SQL functions like TO_CHAR, UPPER etc.

One type of SQL that by default cannot be incorporated into PL/SQL are the DDL (Data Definition Language) commands:

- CREATE, ALTER or DROP
- GRANT and REVOKE

It is possible using the block structure to nest additional blocks of code within the begin section for example:

```
DECLARE
        ….
BEGIN
        DECLARE
        ….
        BEGIN
        ….
        END;
END;
```

Exception Section

The Exception section is an optional section, which can be added between the Begin and End commands. This section is used to handle errors or exceptions to business logic in the code. Exceptions will be looked at in detail later in this course.

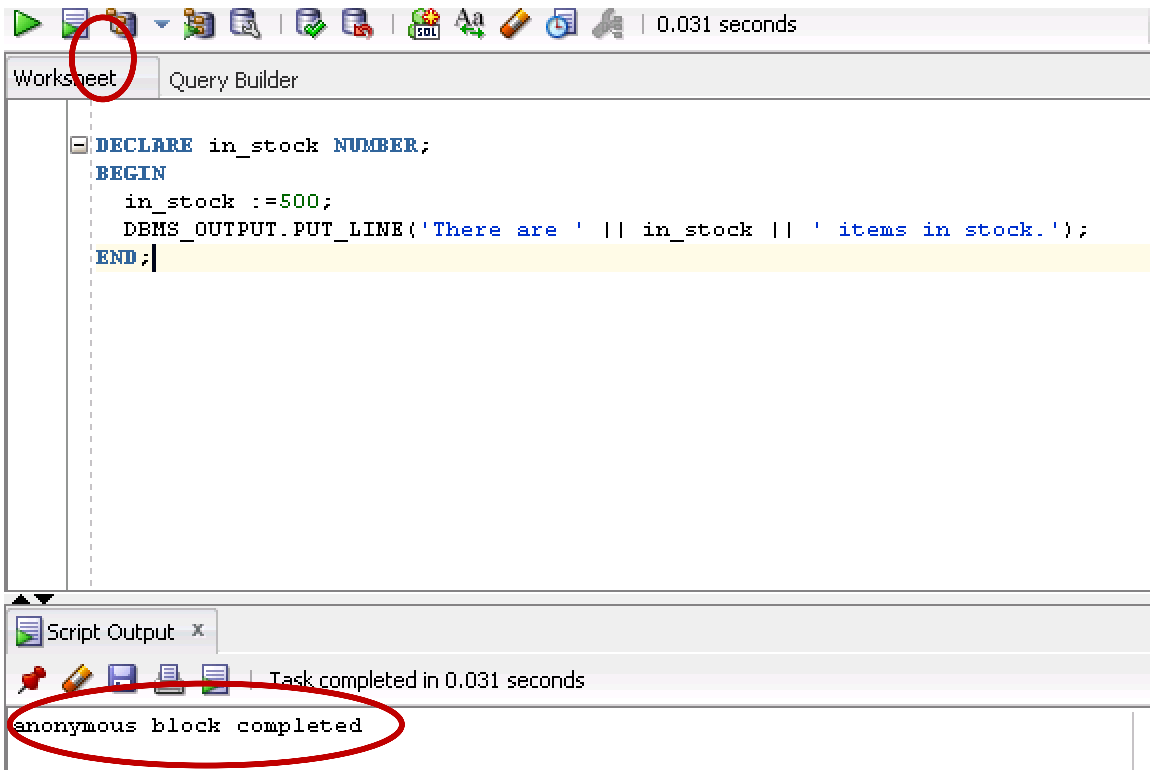
End Command

The End command, which is then followed by a semi-colon, indicates the termination of the code.

Create an anonymous block

To create an anonymous block by using SQL Developer, enter the block in the workspace. To execute an anonymous block, click the Run Script button (or press F5).

Note: The message “anonymous block completed” is displayed in the Script Output window after the block is executed.



```
DECLARE in_stock NUMBER;
BEGIN
  in_stock :=500;
  DBMS_OUTPUT.PUT_LINE('There are ' || in_stock || ' items in stock.');
END;
```

```
anonymous block completed
```

Lesson 12.5: How to generate output from a PL/SQL Block?

PL/SQL does not have built-in input or output functionality. Therefore, you need to use predefined Oracle packages for input and output. Oracle has a package of code called DBMS_OUTPUT. A package is a library of related PL/SQL code, which is stored in the database or client application as a single entity.

DBMS_OUTPUT Package

The DBMS_OUTPUT package consists of many different procedures and functions. However, the most common item used from the package is the PUT_LINE procedure.

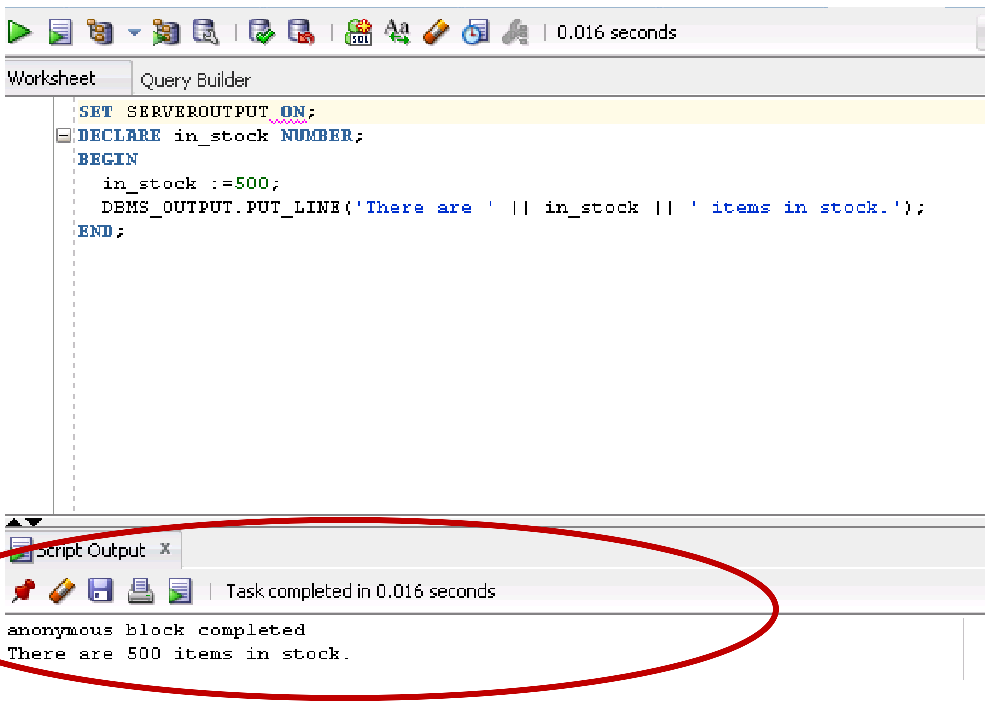
Syntax: SET SERVEROUTPUT ON

```
DBMS_OUTPUT.PUT_LINE([string][number][date]);
```

SQL Example:

```
SET SERVEROUTPUT ON;
DECLARE in_stock NUMBER;
BEGIN
  in_stock :=500;
  DBMS_OUTPUT.PUT_LINE('There are ' || in_stock || '
items in stock.');
END;
```

Query Results:



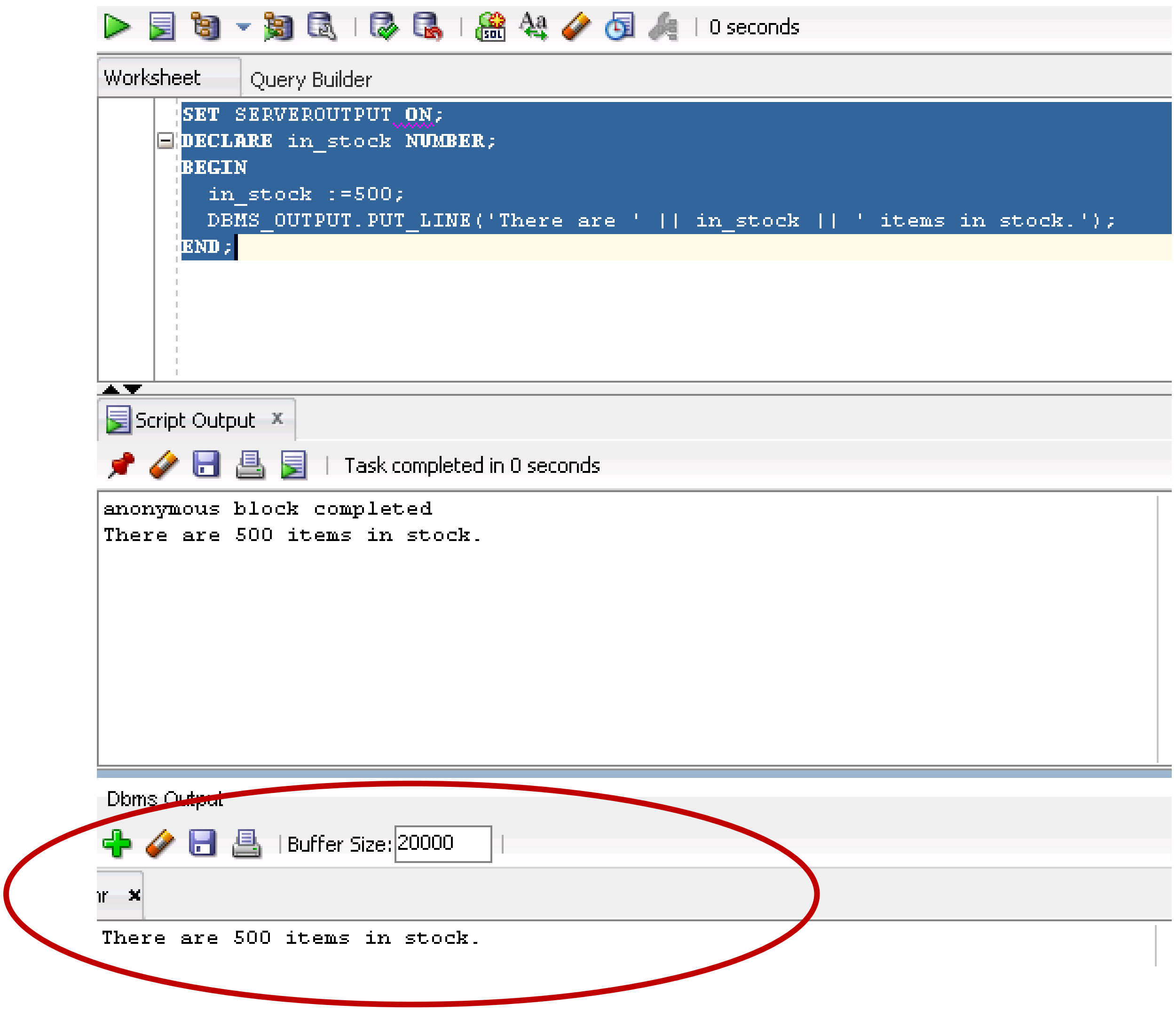
```
SET SERVEROUTPUT ON;
DECLARE in_stock NUMBER;
BEGIN
  in_stock :=500;
  DBMS_OUTPUT.PUT_LINE('There are ' || in_stock || ' items in stock.');
END;
```

```
anonymous block completed
There are 500 items in stock.
```

Explanation: To generate output, you must perform the SET SERVEROUTPUT ON

Code file: Code12.sql

The DBMS_OUTPUT.PUT_LINE result can be also viewed in the DBMS Output window.



```
SET SERVEROUTPUT ON;
DECLARE in_stock NUMBER;
BEGIN
  in_stock :=500;
  DBMS_OUTPUT.PUT_LINE('There are ' || in_stock || ' items in stock.');
END;
```

Task completed in 0 seconds

```
anonymous block completed
There are 500 items in stock.
```

Buffer Size: 20000

```
There are 500 items in stock.
```

Section 13: Declare PL/SQL Identifiers

In this section you will:

- List the different Types of Identifiers in a PL/SQL subprogram
- Usage of the Declarative Section to Define Identifiers
- Use variables to store data
- Identify Scalar Data Types
- The %TYPE Attribute
- What are Bind Variables?
- Sequences in PL/SQL Expressions

Lesson 13.1: Types of Identifiers in a PL/SQL subprogram

You use identifiers to name PL/SQL program items and units, which include constants, variables, exceptions, cursors, cursor variables, subprograms, and packages. Some examples of identifiers follow:

```
X
t2
phone#
credit_limit
LastName
oracle$number
```

An identifier consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs. Other characters such as hyphens, slashes, and spaces are not allowed, as the following examples show:

```
mine&yours    -- not allowed because of ampersand
debit-amount  -- not allowed because of hyphen
on/off        -- not allowed because of slash
user id       -- not allowed because of space
```

The next examples show that adjoining and trailing dollar signs, underscores, and number signs are allowed:

```
money$$$tree
SN##
try_again_
```

You can use upper, lower, or mixed case to write identifiers. PL/SQL is not case sensitive except within string and character literals. So, if the only difference between identifiers is the case of corresponding letters, PL/SQL considers the identifiers to be the same, as the following example shows:

```
last name
LastName  -- same as last name
LASTNAME  -- same as last name and LastName
```

The size of an identifier cannot exceed 30 characters. Identifiers should be descriptive.

Reserved Words

Some identifiers, called *reserved words*, have a special syntactic meaning to PL/SQL and so should not be redefined. For example, the words `BEGIN` and `END`, which bracket the executable part of a block or subprogram, are reserved. As the next example shows, if you try to redefine a reserved word, you get a compilation error:

```
DECLARE
   end BOOLEAN;  -- not allowed; causes compilation error
```

However, you can embed reserved words in an identifier, as the following example shows:

```
DECLARE
   end_of_game BOOLEAN;  -- allowed
```

Predefined Identifiers

Identifiers globally declared in package `STANDARD`, such as the exception `INVALID_NUMBER`, can be redeclared. However, redeclared predefined identifiers is error prone because your local declaration overrides the global declaration.

Quoted Identifiers

For flexibility, PL/SQL lets you enclose identifiers within double quotes. Quoted identifiers are seldom needed, but occasionally they can be useful. They can contain any sequence of printable characters including spaces but excluding double quotes. Thus, the following identifiers are valid:

```
"X+Y"
"last name"
"on/off switch"
"employee(s)"
```

Lesson 13.2: Usage of the Declarative Section to Define Identifiers

Your program stores values in variables and constants. As the program executes, the values of variables can change, but the values of constants cannot.

You can declare variables and constants in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its datatype, and name the storage location so that you can reference it.

Syntax:

```
DECLARE
     ….
BEGIN
     ….
END
```

SQL Example:

```
SET SERVEROUTPUT ON;
DECLARE
     birthday  DATE;
     emp_count SMALLINT := 0;
     credit_limit CONSTANT REAL := 5000.00;
BEGIN
  emp_count:=10;
  DBMS_OUTPUT.PUT_LINE('There are ' || emp_count || ' in
the company');
END;
```

Query Results:

```
anonymous block completed
There are 10 in the company
```

Explanation: The first declaration names a variable of type DATE. The second declaration names a variable of type SMALLINT and uses the assignment operator to assign an initial value of 0 to the variable. The third declaration names a constant of type REAL and assigns an initial (also final) value of 5000 to the constant. . A constant must be initialized in its declaration

Code file: Code13_2_1.sql

Lesson 13.3: Use variables to store data

Variables

A variable in PL/SQL is an item used to store variable data values. Variables are mainly used for storage of data and manipulation of stored values. Variables can store any PL/SQL object such as variables, types, cursors, and subprograms. Reusability is another advantage of declaring variables. After the variables are declared, you can use them repeatedly in an application by referring to them multiple times in various statements.

Rules for Creating Variables and Constants

There are several things to consider when constructing both variables and constants.

- Names given to variables and constants should be unique. They should not be given names used by database objects. For example, if you have a column name of "annual_salary" do not create a variable called "annual_salary". If problems occur the database object will take precedence over the PL/SQL variable or constant.
- Names given are not case sensitive.
- Variables or constants are given assigned values each time the block of code is entered.
- Each variable or constant declared must be terminated with a semi-colon.
- Forward referencing of variables or constants is not allowed in PL/SQL

There are several different types of variable that can be used in PL/SQL.

System variables

System variables are items that hold data the value of which is controlled by Oracle. For example: current date and time is held in the "SYSDATE" system variable the user logged in is held in the "USER" system variable System variables can be referenced at any point within the block of code.

PL/SQL parameters

A PL/SQL Parameter is an explicitly declared variable that is created and used with other PL/SQL objects; for example, Procedures and Functions. PL/SQL parameters will be discussed later in this manual and are used with a key element of the PL/SQL language called Cursors.

PL/SQL variables

PL/SQL variables are the most common type of variables used in PL/SQL. These are variables defined in the declaration section of a block of PL/SQL code. A developer will be normally creating PL/SQL variables to: Store values retrieved from the database that require to be referenced and manipulated in the code Store the result of calculations or expressions; for example, a variable may store the total amount of the tax to be paid by an employee.

Creating a PL/SQL Variable

The syntax required for creating a PL/SQL variable is as follows:

```
variable_name datatype [DEFAULT| := ][value|exp
```

After a unique name has been provided, the variable must be assigned a datatype. This indicates which type of data can be held in the variable i.e. a number, text, date. By default, each time a block of code is executed any variables created are assigned the value NULL. A variable containing a null value could cause problems to occur when it is used in a calculation, i.e. a null value added to a value equals a null result. To avoid this, variables can be assigned a value, this is achieved by using either the:

```
:= (the assignment operator) or DEFAULT keyword
```

Syntax:

```
identifier_name datatype [DEFAULT| := ][value|exp];
```

SQL Example:

```
DECLARE
v_deptno NUMBER(4) NOT NULL := 10;
v_jobid  VARCHAR(10);
BEGIN

  SELECT    department_id, job_id
  INTO      v_deptno,v_jobid
  FROM      employees
  WHERE     employee_id=120;

END;
```

Query Results:

```
anonymous block completed
```

Explanation: All PL/SQL identifiers must be declared in the declaration section before referencing them in the PL/SQL block. You have the option of assigning an initial value to a variable. You do not need to assign a value to a variable in order to declare it. If you refer toother variables in a declaration, be sure that they are already declared separately in a previous statement.

Code file: Code13_3_1.sql

Constants

A constant is a PL/SQL item that is used to store static values. For example, a block of code may require a tax rate in a calculation which will be a static value. Therefore, the value could be assigned to a PL/SQL constant. Creating constants reduces the chance of side effects in the code; if a variable is created, there is the possibility that the value could be changed during the execution of the code.

Creating a PL/SQL Constant

Syntactically constants are very similar to variables:

Syntax:

```
constant_name CONSTANT datatype [DEFAULT|:=] [value|expression];
```

Constants must be assigned a value when defined.

Syntax:

```
constant_name    CONSTANT    datatype    [DEFAULT|:=]
[value|expression];
```

SQL Example:

```
DECLARE
  v_tax CONSTANT NUMBER(4,2):=0.25;
  v_salary  NUMBER(9,2):=10000;
  v_pay  NUMBER(9,2);
BEGIN
  v_pay:=v_salary*v_tax;
  DBMS_OUTPUT.PUT_LINE('Pay is ' || v_pay );
END;
```

Query Results:

```
anonymous block completed
Pay is 2500
```

Explanation: In this example a constant has been created to hold the value of the tax rate to be applied to the salary.

Code file: Code13_3_2.sql

Lesson 13.4: Identify Scalar Data Types

Datatypes

In the last sections we looked at defining variables and constants. Each item must be given a datatype to ensure that only the correct type of data can be assigned.

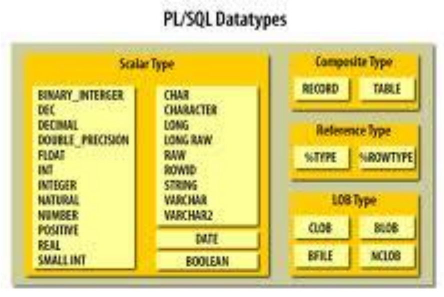
There are two categories of datatype that can be created:

User Defined Datatypes

User defined types in Oracle are more complex data types based on the built-in (standard) data types and can be used in both Oracle PL/SQL and in SQL. In this section of the course we will only consider the Oracle defined datatype as these are the most widely used.

Oracle Defined Datatypes

There are a very large number of different datatypes that could be used in PL/SQL.



In this section we shall consider some the most common data types. The Oracle data type can be split into 4 main groups:

- Scalar
- Composite
- Reference
- Large Object

Scalar Datatypes

PL/SQL provides a variety of predefined data types for example integer. A scalar data type holds a single value and has no internal components. Scalar data types can be classified into four categories: number, character, date, and Boolean.

Datatype	Description
NUMBER	Floating point numeric
NUMBER(p)	Numeric that can accept digits up to a maximum number of fixed points e.g. Number(10) indicates a maximum of 10 point numeric
NUMBER(p,s)	Numeric that can accept digits up to a maximum number of fixed points with a scale e.g. Number(10,2) indicates a numeric with a maximum of 10 points 2 of which are decimals
CHAR[n]	Fixed length characters up to a specified amount of alphanumerics or bytes. These can accept up to a maximum of 32767 bytes/characters.
VARCHAR2(n)	Variable length characters up to a specified amount of alphanumerics or bytes. These can accept up to a maximum of 32767 bytes/characters.
VARCHAR(n)	Same as VARCHAR2.
DATE	Fixed length dates / times.
BOOLEAN	Logical values. Can hold only the values TRUE, FALSE or NULL.
ROWID	Internal identifier of storage row in table
LONG	Variable length character storage up to a maximum of 32760 bytes. (Oracle database columns can store up to 2Mb long columns)
RAW	Variable length binary data storage.
LONG RAW	Variable length binary data storage up to a maximum of 32760 bytes. (Oracle database columns can store up to 2Mb in a LONG RAW column)
BINARY_INTEGER	Base type for integers between –2,147,483,647 and 2,147,483,647
PLS_INTEGER	Base type for signed integers between –2,147,483,647 and 2,147,483,647. PLS_INTEGER values require less storage and are faster than NUMBER values. In Oracle Database 11*g*, the PLS_INTEGER and BINARY_INTEGER data types are identical. The arithmetic operations on PLS_INTEGER and BINARY_INTEGER values are faster than on NUMBER values.

BINARY_FLOAT	Represents floating-point number in IEEE 754 format. It requires 5 bytes to store the value.
BINARY_DOUBLE	Represents floating-point number in IEEE 754 format. It requires 9 bytes to store the value.
TIMESTAMP	The TIMESTAMP data type, which extends the DATE data type, stores the year, month, day, hour, minute, second, and fraction of second. The syntax is TIMESTAMP[(precision)], where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.
TIMESTAMP WITH TIME ZONE	The TIMESTAMP WITH TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is TIMESTAMP[(precision)] WITH TIME ZONE, where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.
TIMESTAMP WITH LOCAL TIME ZONE	The TIMESTAMP WITH LOCAL TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is TIMESTAMP[(precision)] WITH LOCAL TIME ZONE, where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–9. The default is 6. This data type differs from TIMESTAMP WITH TIME ZONE in that when you insert a value into a database column, the value is normalized to the database time zone, and the time-zone displacement is not stored in the column. When you retrieve the value, the Oracle server returns the value in your local session time zone.
INTERVAL YEAR TO MONTH	You use the INTERVAL YEAR TO MONTH data type to store and manipulate intervals of years and months. The syntax is INTERVAL YEAR[(precision)] TO MONTH, where precision specifies the number of digits in the years field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–4. The default is 2.
INTERVAL DAY TO SECOND	You use the INTERVAL DAY TO SECOND data type to store and manipulate intervals of days, hours, minutes, and seconds. The syntax is INTERVAL DAY[(precision1)] TO SECOND[(precision2)], where precision1 and precision2 specify the number of digits in the days field and seconds field, respectively. In both cases, you cannot use a symbolic

	constant or variable to specify the precision; you must use an integer literal in the range 0–9. The defaults are 2 and 6, respectively.

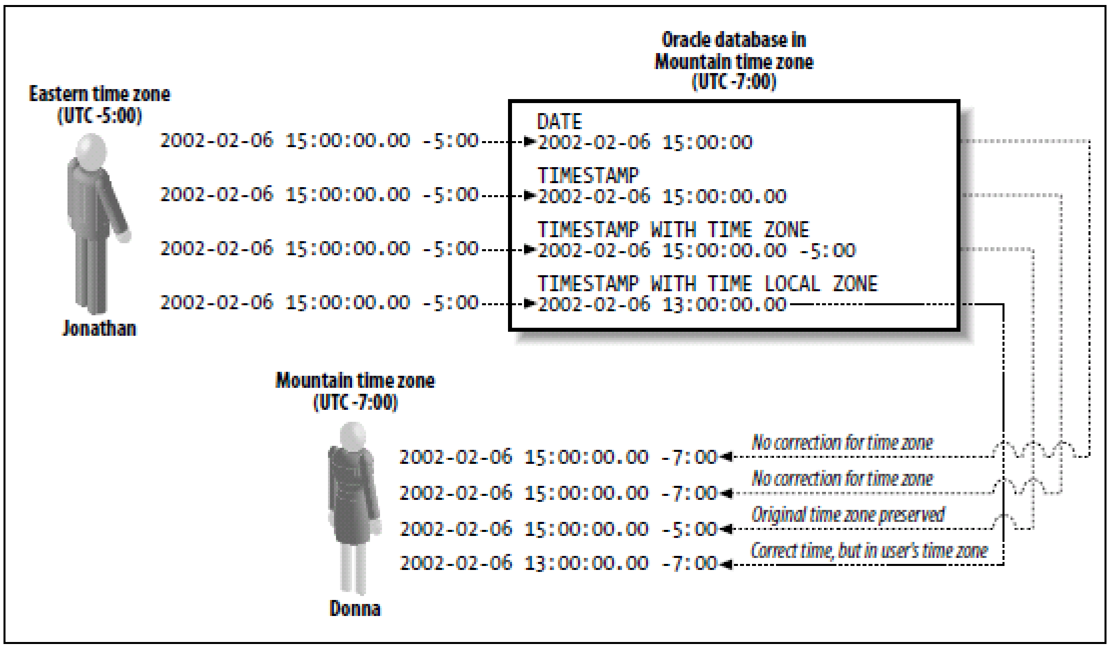
Large Object Datatypes

These were introduced with version 8 of the Oracle Database and will eventually replace LONG and LONG RAW datatypes as the method of working with large amount of text or binary data.

Data Type	Description
BFILE	References files stored on an external file system.
BLOB	Large binary objects stored in the database for example Image files, Word documents, etc.
CLOB	Large character blocks stored in the database.

Time zones

The figure below shows the effects of the different datetime datatypes on a datetime value as it moves from a user in one time zone through the database to another user in a different time zone.



The figure shows the user Jonathan in the Eastern Time Zone, which is five hours behind Coordinated Universal Time (UTC). Jonathan stores the same datetime value in four database fields. The datetime value is represented using ANSI/ISO standard notation, and in this case represents 3:00 PM (15:00:00.00) Eastern Standard Time (-5:00) on Feb 6, 2002 (2002-02-06).

The database in the figure is in the Mountain Standard Time Zone. Notice how the database representation varies with each of the different datetime datatypes. DATE and TIMESTAMP totally ignore the time zone difference between the user and the database. They also don’t preserve the original time zone, so that information is lost along with any knowledge of the time the user really was referring to. The TIMESTAMP WITH TIME ZONE column preserves the time zone information and represents the exact time, from his point of view, that Jonathan entered the values. Compare this with the behavior of the TIMESTAMP WITH LOCAL TIME ZONE column. Here, you see that the time has been converted from Eastern Time into Mountain Time, the database’s local time zone. The correct time has been preserved, but the point of view has been lost; we no longer know the time zone in which the time was entered.

Donna, like the database, is in the Mountain Standard Time Zone. Notice the values she gets back when she queries the database for the four values Jonathan entered. The DATE and TIMESTAMP values are completely misleading. Jonathan entered a value at 3:00 PM Eastern Time, and Donna now sees that as 3:00 PM Mountain

Time. The situation is much better with the other datatypes. Donna sees the TIMESTAMP WITH TIME ZONE value exactly as it was originally entered and can see that Jonathan entered an Eastern Standard Time value. Donna sees the correct time for the TIMESTAMP WITH LOCAL TIME ZONE value (1:00 PM Mountain Time is equivalent to 3:00 PM Eastern Time), but she has no idea what time zone was used to enter the value originally.

Lesson 13.5: The %TYPE Attribute

In many cases variables will be created to hold data retrieved from the database. It is possible that table column definitions could change after a block of code has been created and executed successfully. This could lead to major problems in an application. For example, the LAST_NAME column may be changed to allow a larger name to be inserted, however the variables may be limited to a smaller number of characters. To avoid this a datatype can be anchored to a column in a database table.

Syntax:

```
variable_name [owner].table.column_name%TYPE;
```

Syntax:

```
variable_name [owner].table.column_name%TYPE;
```

SQL Example:

```
DECLARE
v_deptno employees.department_id%type;
v_jobid  employees.job_id%type;
v_lastname employees.last_name%type;
BEGIN

  SELECT    department_id, job_id, last_name
  INTO      v_deptno,v_jobid,v_lastname
  FROM      employees
  WHERE     employee_id=120;

END;
```

Query Results:

```
anonymous block completed
```

Explanation: The datatype is anchored to a column in a database table

Code file: Code13_5_1.sql

Lesson 13.6: Bind Variables

Bind variables are variables that you create in a host environment. For this reason, they are sometimes called *host* variables. Bind variables are created in the environment and not in the declarative section of a PL/SQL block. Therefore, bind variables are accessible even after the block is executed. When created, bind variables can be used and manipulated by multiple subprograms. They can be used in SQL statements and PL/SQL blocks just like any other variable. These variables can be passed as runtime values into or out of PL/SQL subprograms.

Note: A bind variable is an environment variable, but is not a global variable.

Creating Bind Variables

To create a bind variable in SQL Developer, use the VARIABLE command. For example, you declare a variable of type NUMBER and VARCHAR2 as follows:

VARIABLE return_code NUMBER

VARIABLE return_msg VARCHAR2(30)

Syntax:

```
VARIABLE identifier DataType
```

SQL Example:

```
VARIABLE b_result NUMBER;
BEGIN
SELECT (SALARY*12) + NVL(COMMISSION_PCT,0) INTO :b_result
FROM employees WHERE employee_id = 144;
 DBMS_OUTPUT.PUT_LINE('Result is ' || :b_result );
END;
```

Query Results:

```
anonymous block completed
Result is 30000
```

Explanation: Bind variable declaration

Code file: Code13_6_1.sql

Lesson 13.7: Sequences in PL/SQL Expressions

In Oracle Database 11*g*, you can use the NEXTVAL and CURRVAL pseudocolumns in any PL/SQL context, where an expression of the NUMBER data type may legally appear. Although the old style of using a SELECT statement to query a sequence is still valid, it is recommended that you do not use it.

Before Oracle Database 11*g*, you were forced to write a SQL statement in order to use a sequence object value in a PL/SQL subroutine. Typically, you would write a SELECT statement to reference the pseudocolumns of NEXTVAL and CURRVAL to obtain a sequence number. This method created a usability problem.

```
DECLARE
      v_new_id NUMBER;
BEGIN
      SELECT my_seq.NEXTVAL INTO v_new_id FROM Dual;
END;
```

In Oracle Database 11*g*, the limitation of forcing you to write a SQL statement to retrieve a sequence value is eliminated. With the sequence enhancement feature:

- Sequence usability is improved
- The developer has to type less
- The resulting code is clearer

```
DECLARE
      v_new_id NUMBER;
BEGIN
      v_new_id := my_seq.NEXTVAL;
END;
```

Section 14: Write Executable Statements

In this section you will:

- Describe Basic PL/SQL Block Syntax Guidelines
- Learn to Comment the Code
- Deployment of SQL Functions in PL/SQL
- How to convert Data Types?
- Describe Nested Blocks
- Identify the Operators in PL/SQL

Lesson 14.1: Describe Basic PL/SQL Block Syntax Guidelines

Lexical Units in a PL/SQL Block

Lexical units include letters, numerals, special characters, tabs, spaces, returns, and symbols.

Identifiers

Identifiers are the names given to PL/SQL objects. You learned to identify valid and invalid identifiers. Recall that keywords cannot be used as identifiers.

Quoted identifiers

- Make identifiers case-sensitive.
- Include characters such as spaces.
- Use reserved words.

Examples:

"begin date" DATE;

"end date" DATE;

"exception thrown" BOOLEAN DEFAULT TRUE;

All subsequent usage of these variables should have double quotation marks. However, use of quoted identifiers is not recommended.

Delimiters:

Delimiters are symbols that have special meaning. You already learned that the semicolon (;) is used to terminate a SQL or PL/SQL statement. Therefore, ; is an example of a delimiter.

Literals:

Any value that is assigned to a variable is a literal. Any character, numeral, Boolean, or date value that is not an identifier is a literal. All string literals have the data type CHAR or VARCHAR2 and are, therefore, called character literals (for example, John, and 12C).

Lesson 14.2: Learn to Comment the Code

Commenting Code

You should comment code to document each phase and to assist debugging. In PL/SQL code:

- A single-line comment is commonly prefixed with two hyphens (--)
- You can also enclose a comment between the symbols /* and */

It is good programming practice to explain what a piece of code is trying to achieve. However, when you include the explanation in a PL/SQL block, the compiler cannot interpret these instructions. Therefore, there should be a way in which you can indicate that these instructions need not be compiled. Comments are mainly used for this purpose.

Lesson 14.3: Use of SQL Functions in PL/SQL

SQL Functions in PL/SQL

SQL provides several predefined functions that can be used in SQL statements. Most of these functions such as single-row number and character functions, data type conversion functions, and date and time-stamp functions are valid in PL/SQL expressions.

Syntax:

```
Variable:=sqlfunction(,,,);
```

SQL Example:

```
DECLARE
     v_result NUMBER(10,2);
BEGIN
      SELECT  (SALARY*12)  +  NVL(COMMISSION_PCT,0)  INTO
      v_result
      FROM employees WHERE employee_id = 144;
      v_result:=round(v_result,0);
      DBMS_OUTPUT.PUT_LINE('Result is ' || v_result );
END;
```

Query Results:

```
anonymous block completed
Result is 30000
```

Explanation: The Sql round function is used in PLSQL block

Code file: Code14_3_1.sql

The following functions are **not** available in procedural statements:

- DECODE
- Group functions: AVG, MIN, MAX, COUNT, SUM, STDDEV, and VARIANCE

Lesson 14.4: How to convert Data Types

In any programming language, converting one data type to another is a common requirement. PL/SQL can handle such conversions with scalar data types. Data type conversions can be of two types:

Implicit conversions

PL/SQL attempts to convert data types dynamically if they are mixed in a statement.

Implicit conversions can be between:

- Characters and numbers
- Characters and dates

Syntax

SQL Example:

```
DECLARE
v_salary NUMBER(6):=6000;
v_sal_hike VARCHAR2(5):='1000';
v_total_salary v_salary%TYPE;
BEGIN
v_total_salary:=v_salary + v_sal_hike;
END;
```

Query Results: …

Explanation: In this example, the sal_hike variable is of the VARCHAR2 type. When calculating the total salary, PL/SQL first implicitly converts sal_hike to NUMBER, and then performs the operation. The result is a the NUMBER type

Code file: Code14_4_1.sql

Explicit conversions

To convert values from one data type to another, use built-in functions. For example, to convert a CHAR value to a DATE or NUMBER value, use TO_DATE or TO_NUMBER, respectively.

SQL Example:

```
DECLARE

-- implicit data type conversion
v_date1 DATE:= '02-Feb-2000';
-- error in data type conversion
v_date2 DATE:= 'February 02,2000';
-- explicit data type conversion
v_date3  DATE  :=  TO_DATE('February  02,2000','Month  DD,
YYYY');

BEGIN
DBMS_OUTPUT.PUT_LINE(v_date1);

END;
```

Query Results:

```
Error report -
ORA-01858:  a  non-numeric  character  was  found  where  a
numeric was expected
ORA-06512: at line 6
01858. 00000 -  "a non-numeric character was found where
a numeric was expected"
…
```

Explanation: v-date1 successful, v_date2 fails to convert

Code file: Code4_2_14.sql

Lesson 14.5: Describe Nested Blocks

Being procedural gives PL/SQL the ability to nest statements. You can nest blocks wherever an executable statement is allowed, thus making the nested block a statement. If your executable section has code for many logically related functionalities to support multiple business requirements, you can divide the executable section into smaller blocks. The exception section can also contain nested blocks.

When you access this variable in the inner block, PL/SQL first looks for a local variable in the inner block with that name. There is no variable with the same name in the inner block, so PL/SQL looks for the variable in the outer block.

Syntax:

```
DECLARE
     ….
BEGIN
     DECLARE
     ….
     BEGIN
     ….
     END;
END;
```

SQL Example:

```
DECLARE
v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
     DECLARE
          v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
     BEGIN
          DBMS_OUTPUT.PUT_LINE(v_inner_variable);
          DBMS_OUTPUT.PUT_LINE(v_outer_variable);
     END;
DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

Query Results:

```
anonymous block completed
LOCAL VARIABLE
GLOBAL VARIABLE
GLOBAL VARIABLE
```

Explanation: The v_outer_variable variable is declared in the outer block and the v_inner_variable variable is declared in the inner block.v_outer_variable is local to the outer block but global to the inner block. When you access this variable in the inner block, PL/SQL first looks for a local variable in the inner block with that name. There is no variable with the same name in the inner block, so PL/SQL looks for the variable in the outer block. Therefore, v_outer_variable is considered to be the global variable for all the enclosing blocks. You can access this variable in the inner block. Variables declared in a PL/SQL block are considered local to that block and global to all its subblocks. v_inner_variable is local to the inner block and is not global because the inner block does not have any nested blocks. This variable can be accessed only within the inner block. If PL/SQL does not find the variable declared locally, it looks upward in the declarative section of the parent blocks. PL/SQL does not look downward in the child blocks.

Code file: Code14_5_1.sql

Variable Scope and Visibility

- The *scope* of a variable is the portion of the program in which the variable is declared and is accessible.
- The *visibility* of a variable is the portion of the program where the variable can be accessed without using a qualifier.

Syntax:

```
DECLARE
     ….
BEGIN
     DECLARE
     ….
     BEGIN
     ….
     END;
END;
```

SQL Example:

```
DECLARE
  v_name VARCHAR2(20):='Bill';
  v_date DATE:='2-Apr-2014';
BEGIN
  DECLARE
    v_other_name VARCHAR2(20):='Mike';
    v_date DATE:='3-Jan-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Name: '||v_name);
    DBMS_OUTPUT.PUT_LINE('Date: '||v_date);
    DBMS_OUTPUT.PUT_LINE('other Name: '||v_other_name);
  END;
  DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date);
END;
```

Query Results:

```
anonymous block completed
Name: Bill
Date: 03-JAN-02
other Name: Mike
Date of Birth: 02-APR-14…
```

Explanation: The v_name variable and the first occurrence of the v_date variable are declared in the outer block. These variables have the scope of the block in which they are declared.

The v_other_name and v_date variables are declared in the inner block or the nested block. These variables are accessible only within the nested block and are not accessible in the outer block. These variables have the scope of the block in which they are declared.

The v_date variable declared in the outer block has scope even in the inner block. However, this variable is not visible in the inner block because the inner block has a local variable with the same name.

Code file: Code14_5_2.sql

Qualifier

A *qualifier* is a label given to a block. You can use a qualifier to access the variables that have scope but are not visible. Labeling is not limited to the outer block. You can label any block.

Syntax:

```
BEGIN <<qualifier>>
     DECLARE
     ….
        BEGIN
          DECLARE
          ….
          BEGIN
          ….
          END;
     END;
END;
```

SQL Example:

```
BEGIN <<block1>>
  DECLARE
    v_name VARCHAR2(20):='Bill';
    v_date DATE:='2-Apr-2014';
  BEGIN
    DECLARE
      v_other_name VARCHAR2(20):='Mike';
      v_date DATE:='3-Jan-2002';
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Name: '||v_name);
      DBMS_OUTPUT.PUT_LINE('Date: '||block1.v_date);
           DBMS_OUTPUT.PUT_LINE('Date: '||v_date);

      DBMS_OUTPUT.PUT_LINE('other Name: '||v_other_name);

    END;
    DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date);
  END;
END;
```

Query Results:

```
anonymous block completed
Name: Bill
Date: 02-APR-14
Date: 03-JAN-02
other Name: Mike
Date of Birth: 02-APR-14
```

Explanation: The outer v_date is now accessible. Within the inner block, the outer qualifier is used to access the v_date variable that is declared in the outer block.

Code file: Code14_5_3.sql

Lesson 14.6: Identify the Operators in PL/SQL

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators:

Arithmetic Operators

Operator	Description
+	Adds two operands
-	Subtracts second operand from the first
*	Multiplies both operands
/	Divides numerator by de-numerator
**	Exponentiation operator, raises one operand to the power of other

Relational Operators

Operator	Description
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!= <> ~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Comparison Operators

Operator	Description
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.

Logical Operators

Operator	Description
and	Called logical AND operator. If both the operands are true then condition becomes true.
or	Called logical OR Operator. If any of the two operands is true then condition becomes true.
not	Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.

PL/SQL Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated.

Operator	Operation
**	exponentiation
+, -	identity, negation
***, /**	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	comparison
NOT	logical negation
AND	conjunction
OR	inclusion

Programming Guidelines

Follow programming guidelines shown below to produce clear code and reduce maintenance when developing a PL/SQL block. The following table provides guidelines for writing code in uppercase or lowercase characters to help distinguish keywords from named objects.

Statement	Convention
SQL statements	Uppercase SELECT, INSERT
PL/SQL keywords	Uppercase DECLARE, BEGIN, IF
Data types	Uppercase VARCHAR2, BOOLEAN
Identifiers and parameters	Lowercase v_sal, emp_cursor, g_sal, p_empno
Database tables	Lowercase, plural employees, departments
Database columns	Lowercase, singular employee_id,

Indenting Code

For clarity and enhanced readability, indent each level of code. To show structure, you can divide lines by using carriage returns and you can indent lines by using spaces and tabs. Compare the following IF statements for readability:

```
IF x > y THEN
     max := x;
ELSE
     max := y;
END IF;
```

Section 15: Interaction with the Oracle Server

In this section you will:

- Invoke SELECT INTO Statements in PL/SQL
- Retrieve Data in PL/SQL
- SQL Cursor concept
- Data Manipulation in the Server using PL/SQL
- Use SQL Cursor Attributes to Obtain Feedback on DML
- Save and Discard Transactions

Lesson 15.1: Invoke SELECT Statements in PL/SQL

In a PL/SQL block,

- You use SQL statements to retrieve and modify data from the database table. PL/SQL supports data manipulation language (DML) and transaction control commands. You can use DML commands to modify the data in a database table. However, remember the following points while using DML statements and transaction control commands in PL/SQL blocks:
- The END keyword signals the end of a PL/SQL block, not the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.
- PL/SQL does not directly support data definition language (DDL) statements such as CREATE TABLE, ALTER TABLE, or DROP TABLE.
- DDL statements cannot be directly executed. These statements are dynamic SQL statements. Dynamic SQL statements are built as character strings at run time and can contain placeholders for parameters. Therefore, you can use dynamic SQL to execute your DDL statements in PL/SQL.
- PL/SQL does not directly support data control language (DCL) statements such as GRANT or REVOKE. You can use dynamic SQL to execute them.

Lesson 15.2: Retrieve Data in PL/SQL

A SELECT statement in PL/SQL is normally known as a SELECT INTO statement. The reason for this is that data retrieved from the database will require to be placed into variables.

Syntax

```
SELECT     col1|expression1,
           col2|expression2…
INTO       variable1,
           variable2….
FROM  table1|view1,
           Table2|view2….
WHERE      condition
[GROUP BY]
[HAVING]
[ORDER BY];
```

There is a major limitation to the SELECT INTO statement as **it can only retrieve one row of data**. In many cases the developer will require to process multiple rows of data therefore the SELECT INTO will not be suitable. If your requirement is to retrieve multiple rows and operate on the data, you can make use of **explicit cursors**.

- Use the SELECT statement to retrieve data from the database.
- Every value retrieved must be stored in a variable by using the INTO clause.
- The WHERE clause is optional and can be used to specify input variables, constants, literals, and PL/SQL expressions. However, when you use the INTO clause, you should fetch only one row; using the WHERE clause is required in such cases.
- Specify the same number of variables in the INTO clause as the number of database columns in the SELECT clause. Be sure that they correspond positionally and that their data types are compatible.
- The INTO clause is required.
- Queries must return only one row.

Use Select INTO

Syntax:

```
SELECT     col1|expression1,
           col2|expression2…
INTO       variable1,
           variable2….
FROM  table1|view1,
           Table2|view2….
WHERE      condition
[GROUP BY]
[HAVING]
[ORDER BY];
```

SQL Example:

```
DECLARE
v_fname VARCHAR2(25);
v_lname VARCHAR2(25);
BEGIN
  SELECT first_name, last_name
  INTO v_fname,v_lname
  FROM employees WHERE employee_id=100;
  DBMS_OUTPUT.PUT_LINE(' First Name is : '||v_fname);
  DBMS_OUTPUT.PUT_LINE(' Last Name is : '||v_lname);
END;
```

Query Results:

```
anonymous block completed
 First Name is : Steven
 Last Name is : King
```

Explanation:

Code file: Code15_2_1.sql

Lesson 15.3: SQL Cursor concept

A cursor is a pointer to the private memory area allocated by the Oracle Server. It is used to handle the result set of a SELECT statement.

There are two types of cursors: implicit and explicit.

Implicit:

Created and managed internally by the Oracle Server to process SQL statements

Explicit:

Declared explicitly by the programmer covered in section 18

Implicit Cursor Attributes

These attributes will return information about SELECT INTO, INSERT, UPDATE and DELETE statements.

As implicit cursors are not manually assigned a name the syntax is slightly different. The name **SQL** is assigned to implicit cursors

Syntax

```
SQL%attribute_name
```

%NOTFOUND Implicit Attribute

The %NOTFOUND can hold only Boolean values i.e. TRUE, FALSE or NULL. The %NOTFOUND will hold the value TRUE if the DML (INSERT, UPDATE, DELETE or SELECT) statement causes rows to be affected. Otherwise the value held will be FALSE.

%FOUND Cursor Attribute

The %FOUND attribute can hold only Boolean values i.e. TRUE, FALSE or NULL. %FOUND will hold the value FALSE if the DML (INSERT, UPDATE, DELETE or SELECT) statement causes no rows to be affected. Otherwise the value held will be TRUE.

%ISOPEN Cursor Attribute

The %ISOPEN attribute is not applicable to be used. It will always hold the value FALSE as Oracle automatically closes the cursor after it is executed.

%ROWCOUNT Cursor Attribute

The %ROWCOUNT will hold the number of rows processed by DML statements.

Syntax:

```
SELECT     col1|expression1,
           col2|expression2…
INTO       variable1,
           variable2….
FROM  table1|view1,
           Table2|view2….
WHERE      condition
[GROUP BY]
[HAVING]
[ORDER BY];


SQL%attribute_name
```

SQL Example:

```
DECLARE
v_fname VARCHAR2(25);
v_lname VARCHAR2(25);
BEGIN
  SELECT first_name, last_name
  INTO v_fname,v_lname
  FROM employees WHERE employee_id=100;
  IF sql%found THEN
    DBMS_OUTPUT.PUT_LINE(' First Name is : '||v_fname);
    DBMS_OUTPUT.PUT_LINE(' Last Name is : '||v_lname);
    DBMS_OUTPUT.PUT_LINE(' Count is : '||sql%ROWCOUNT);
  END IF;
END;
```

Query Results:

```
anonymous block completed
 First Name is : Steven
 Last Name is : King
 Count is : 1
```

Explanation: The SELECT INTO statement returns the first and last name into variables and uses the SQL attributes to determine row count

Code file: Code15_3_1.sql

Lesson 15.4: Data Manipulation in the Server using PL/SQL

Inserting Data

An INSERT statement is used within a PL/SQL block to insert a record into a table.

Syntax:

```
BEGIN
INSERT…
END;
```

SQL Example:

```
BEGIN
INSERT INTO employees
(employee_id, first_name, last_name, email,
hire_date, job_id, salary)
VALUES(employees_seq.NEXTVAL, 'Jim', 'Bell',
'RCORES',CURRENT_DATE, 'AD_ASST', 1500);
END;
```

Query Results:

```
anonymous block completed
```

Explanation: The PL/SQL block inserts a new employee

Code file: Code15_4_1.sql

Updating Data

Use the UPDATE statement to change data

Syntax:

```
DECLARE

BEGIN
UPDATE …
END;
```

SQL Example:

```
DECLARE
sal_increase employees.salary%TYPE := 800;
BEGIN
UPDATE employees
SET salary = salary + sal_increase
WHERE job_id = 'ST_CLERK';
END;
```

Query Results:

```
anonymous block completed
```

Explanation: The PL/SQL block increases the salary of all employees who are stock clerks.

Code file: Code15_4_2.sql

Deleting Data

Use the UPDATE statement to change data:

Syntax:

```
DECLARE

BEGIN
DELETE  …
END;
```

SQL Example:

```
DECLARE
v_empid employees.department_id%TYPE := 1010;
BEGIN
DELETE FROM employees
WHERE employee_id = v_empid;
IF SQL%NOTFOUND THEN
  DBMS_OUTPUT.PUT_LINE('Employee ' || v_empid || ' not
found');
END IF;

END;
```

Query Results:

```
anonymous block completed
```

Explanation: Delete rows that belong to employee 1010 from the employees table.

Code file: Code15_4_3.sql

Merging Rows

The MERGE statement inserts or updates rows in one table by using data from another table. Each row is inserted or updated in the target table depending on an equijoin condition.

Syntax:

```
DECLARE
BEGIN
MERGE…
END;
```

SQL Example:

```
BEGIN
MERGE INTO copy_emp c
USING employees e
ON (e.employee_id = c.empno)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
c.email = e.email,
c.phone_number = e.phone_number,
c.hire_date = e.hire_date,
c.job_id = e.job_id,
c.salary = e.salary,
c.commission_pct = e.commission_pct,
c.manager_id = e.manager_id,
c.department_id = e.department_id
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
END;
```

Query Results:

Explanation: The example shown matches the empno column in the copy_emp table to the employee_id column in the employees table. If a match is found, the row is updated to match the rowin the employees table. If the row is not found, it is inserted into the copy_emp table..

Code file: N/A

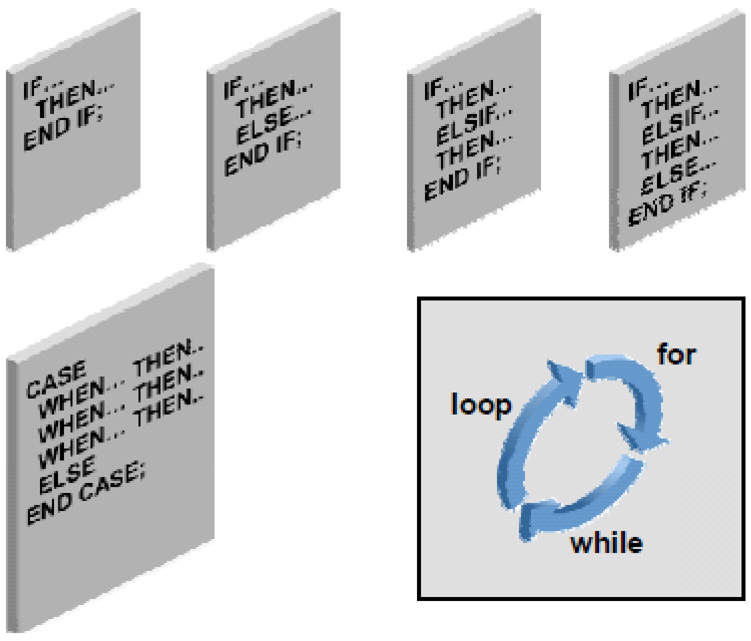
Note: Transactions with end following either:

```
COMMIT; --Confirms changes made
ROLLBACK; --Removed changes made since the last COMMIT
```

Section 16: Control Structures

In this section you will:

- Conditional processing using IF Statements
- Conditional processing using CASE Statements
- Describe simple Loop Statement
- Describe While Loop Statement
- Describe For Loop Statement
- Use the Continue Statement


IF... THEN... END IF;
IF... THEN... ELSE... END IF;
IF... THEN... ELSIF... THEN... END IF;
IF... THEN... ELSIF... THEN... ELSE... END IF;
CASE WHEN... THEN.. WHEN... THEN.. WHEN... THEN.. ELSE END CASE;
loop
for
while

The PL/SQL language has the many of the standard techniques found in other programming languages to allow movement and control flow through a block of code. This chapter will consider these features in detail.

To control the flow through a block of PL/SQL code the developer has the following features available:

Loops – these allow the same code to be executed a specific number of times. For example, our code may require checking and processing a batch of 1000 data records.

Conditional Statements – these are normally referred to as IF statements. They allow data to be evaluated and specific actions performed based on the results of the evaluation.

Lesson 16.1: Conditional processing using IF Statements

The IF conditional statement is one of the most common features of the PL/SQL language. It allows the developer to conditionally control the transition through the code.

For example, a block of code may require the following:

A check to find out the number of years the person has worked in the company and based on it:

IF the person has worked less than 2 years their holiday entitlement should be 20 days.

IF the person has worked between 2 years and 5 years their entitlement should be 22 days.

IF the person has worked more than 5 years their holiday entitlement should be 25 days.

The syntax required is for an IF statement is as follows:

```
IF condition THEN
      Statement;
[ELSIF condition THEN
      Statement;]
[ELSE
      Statement;]
END IF;
```

- The ELSIF section is optional and it is possible to have multiple ELSIF commands. The ELSE is also an optional command.
- The syntax of the IF statement is very prone to typing errors.

The most common errors are:

- END IF must be 2 words. In other languages it is one-word ENDIF
- ELSIF has only one "E". A common mistake is to add another i.e. ELSEIF.
- Ensure that the THEN keyword is used with IF and ELSIF but not included with the ELSE keyword.

IF statement

Syntax:

```
IF condition THEN
      Statement;
[ELSIF condition THEN
      Statement;]
[ELSE
      Statement;]
END IF;
```

SQL Example:

```
DECLARE
      v_mydept number:=31;
BEGIN
IF v_mydept < 11 THEN
      UPDATE departments
      set manager_id=100
      where department_id <11;
ELSIF v_mydept < 20 THEN
      DBMS_OUTPUT.PUT_LINE(' For review ');
ELSE
      DBMS_OUTPUT.PUT_LINE(' OK ');
END IF;
END;
```

Query Results:

```
anonymous block completed
 OK
```

Explanation: The example below evaluates a variable and updates records if department is <11

Code file: Code16_1_1.sql

Lesson 16.2: Conditional processing using CASE Statements

We have two types of CASE functionality:

- CASE Expressions
- CASE statements

PL/SQL Case Expressions

Return a value to a variable based on the value of a single variable or expression. The value is determined based on a selector which is a comparison operator 2 types of CASE expressions

- Simple
- Searched

Simple Case Expressions

Evaluated against a single expression

Syntax:

```
Variable:= CASE selector
WHEN expression1 THEN result1
WHEN expression2 THEN result2 ...
[ELSE result]
END;
```

Syntax:

```
Variable:= CASE selector
     WHEN expression1 THEN result1
     WHEN expression2 THEN result2 ...
     [ELSE result]
END;
```

SQL Example:

```
DECLARE
v_deptid number:=4;
v_deptname varchar(50);
BEGIN
v_deptname := case v_deptid
  when 1 then 'Sales'
  when 2 then 'IT'
  when 3 then 'IS'
  when 4 then 'TRAINING'
  else
  'Unknown'
end;

DBMS_OUTPUT.PUT_LINE(v_deptname);
```

```
END;
```

Query Results:

```
anonymous block completed
TRAINING
```

Explanation: Uses the simple case to evaluate a variable

Code file: Code16_2_1.sql

Searched Case Expressions

Evaluated against a single expression

Syntax:

```
Variable:= CASE
WHEN search_expression1 THEN result1
WHEN search_expression2 THEN result2 ...
[ELSE result]
END;
```

Syntax:

```
Variable:= CASE
WHEN search_expression1 THEN result1
WHEN search_expression2 THEN result2 ...
[ELSE result]
END;
```

SQL Example:

```
DECLARE
v_deptid number:=4;
v_deptname varchar(50);
BEGIN
v_deptname := case
  when v_deptid = 1 then 'Sales'
  when v_deptid = 2 then 'IT'
  when v_deptid = 3 then 'IS'
  when v_deptid = 4 then 'TRAINING'
  else
  'Unknown'
end;

DBMS_OUTPUT.PUT_LINE(v_deptname);

END;
```

Query Results:

```
anonymous block completed
TRAINING
```

Explanation: Uses the simple case to evaluate a variable

Code file: Code16_2_1.sql

Lesson 16.3: Describe simple Loop Statement

The Simple loop is often referred to as an infinite loop as the basic syntax if would repeat the execution of the code infinitely, and therefore crash the code.

Syntax:

```
LOOP
      --statement
END LOOP;
```

To avoid an infinite loop the EXIT command is required to ensure the loop is terminated.

Syntax:

```
EXIT WHEN condition;
```

Or

```
EXIT;
```

The condition can use the standard SQL logical operators ie. <, >,=, <> etc.

For example

```
EXIT WHEN employee_id =1000;
```

Syntax:

```
LOOP
      --statement
      EXIT WHEN condition;
END LOOP;
```

SQL Example:

```
DECLARE
v_deptid NUMBER(10):=1;

BEGIN

LOOP
  INSERT      INTO      departments      (department_id,
department_name)
  Values (v_deptid,'Dept' ||v_deptid);
  DBMS_OUTPUT.PUT_LINE('Dept' ||v_deptid );
  v_deptid:=v_deptid+1;
  EXIT WHEN v_deptid =10;
END LOOP ;
END;
```

Query Results:

```
Dept1
Dept2
Dept3
Dept4
Dept5
Dept6
Dept7
Dept8
Dept9
```

Explanation: The example on the next page uses a simple loop to populate a table with an incremented value

Code file: Code16_3_1.sql

Lesson 16.4: Describe While Loop Statement

The main difference between the Simple loop and the While loop is the While loop will evaluate the continuation condition at the beginning of the loop (in the loop header) rather than in a statement in the main body of the loop.

Syntax:

```
WHILE condition LOOP
     Statements…
END LOOP;
```

The example below uses a while loop to populate a table with an incremented value.

Syntax:

```
WHILE condition LOOP
           Statements…
END LOOP;
```

SQL Example:

```
DECLARE
v_deptid NUMBER(10):=1;

BEGIN
 WHILE v_deptid =10 LOOP
  INSERT      INTO      departments      (department_id,
department_name)
  Values (v_deptid,'Dept' ||v_deptid);
  DBMS_OUTPUT.PUT_LINE('Dept' ||v_deptid );
  v_deptid:=v_deptid+1;
END LOOP ;
END;
```

Query Results:

```
Dept1
Dept2
Dept3
Dept4
Dept5
Dept6
Dept7
Dept8
Dept9
```

Explanation: The example on the next page uses a simple loop to populate a table with an incremented value

Code file: Code16_4_1.sql

Lesson 16.5: Describe For Loop Statement

There are two types of For Loop:

- A numeric For loop
- A Cursor For loop (covered in Section 18)
-

The numeric FOR loop moves through the loop for a specific number of iterations. The loop has a start and end point and a numeric index controls the movements through the loop.

Syntax:

```
FOR variable IN [REVERSE] low_value ..high_value LOOP
      Executable_statements…
END LOOP;
```

The variable used in the FOR LOOP does not require to be defined in the declaration section of the PL/SQL code. It will be implicitly declared in the loop and will only exist for the duration of the loop. The low value and high value must be integers. When the loop is started the low value is assigned to the variable and each transition of the loop will increment the variable by 1. The loop will terminate after the variable has reached the high value and completed the final transition. If the REVERSE keyword is used the variable will start at the high value and decrement the variable by 1 until the low value is reached and completed.

- Reference the counter only within the loop; it is undefined outside the loop.
- Do not reference the counter as the target of an assignment.
- Neither loop bound should be **NULL**.

Syntax:

```
FOR  variable  IN  [REVERSE]  low_value  ..high_value
LOOP
      Executable_statements…
END LOOP;
```

SQL Example:

```
BEGIN
 FOR v_deptid IN 1 .. 10 LOOP
  INSERT      INTO      departments      (department_id,
department_name)
  Values (v_deptid,'Dept' ||v_deptid);
  DBMS_OUTPUT.PUT_LINE('Dept' ||v_deptid );

END LOOP ;
```

```
END;
```

Query Results:

```
Dept1
Dept2
Dept3
Dept4
Dept5
Dept6
Dept7
Dept8
Dept9
```

Explanation: The example uses a for loop to populate a table with an incremented value

Code file: Code16_5_1.sql

Lesson 16.6: Use the Continue Statement

The CONTINUE statement enables you to transfer control within a loop back to a new iteration or to leave the loop. Many other programming languages have this functionality. With the Oracle Database 11*g* release, PL/SQL also offers this functionality. Before the Oracle Database 11*g* release, you could code a workaround by using Boolean variables and conditional statements to simulate the CONTINUE programmatic functionality. In some cases, the workarounds are less efficient.

The CONTINUE statement offers you a simplified means to control loop iterations. It may be more efficient than the previous coding workarounds. The CONTINUE statement is commonly used to filter data within a loop body before the main processing begins.

Syntax:

```
FOR variable IN [REVERSE] low_value ..high_value LOOP
     Executable_statements…
END LOOP;
```

SQL Example:

```
BEGIN
 FOR v_deptid IN 1 .. 10 LOOP
  if v_deptid = 3 then
    CONTINUE;
  END IF;

  INSERT INTO departments (department_id, department_name)
  Values (v_deptid,'Dept' ||v_deptid);

  DBMS_OUTPUT.PUT_LINE('Dept' ||v_deptid );

END LOOP ;
END;
```

Query Results:

```
Dept1
Dept2
Dept4
Dept5
Dept6
Dept7
Dept8
Dept9
```

Explanation: The example on the next page uses a simple loop to populate a table with an incremented value

Code file: Code16_6_1.sql

Section 17: Composite Data Types

In this section you will:

- Use PL/SQL Records
- The %ROWTYPE Attribute
- Insert and Update with PL/SQL Records
- INDEX BY Tables
- Examine INDEX BY Table Methods
- Use INDEX BY Table of Records

Composite Data Types

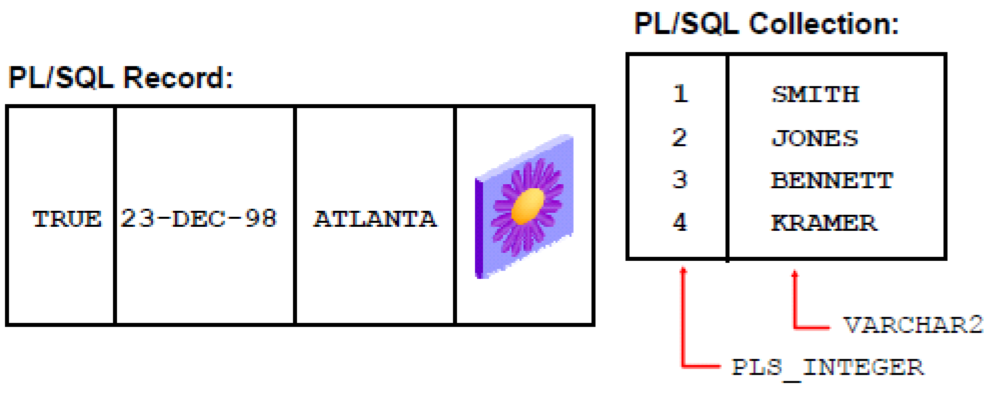
You learned that variables of the scalar data type can hold only one value, whereas a variable of the composite data type can hold multiple values of the scalar data type or the composite data type. There are two types of composite data types:

PL/SQL records

Records are used to treat related but dissimilar data as a logical unit. A PL/SQL record can have variables of different types. For example, you can define a record to hold employee details. This involves storing an employee number as **NUMBER**, a first name and last name as **VARCHAR2**, and so on. By creating a record to store employee details, you create a logical collective unit. This makes data access and manipulation easier.

PL/SQL collections: Collections are used to treat data as a single unit. Collections are of three types:

- Associative array
- Nested table
- **VARRAY**



Use PL/SQL records when you want to store values of different data types that are logically related. For example, you can create a PL/SQL record to hold employee details and indicate that all the values stored are related because they provide information about a particular employee.

Use PL/SQL collections when you want to store values of the same data type. Note that this data type can also be of the composite type (such as records). You can define a collection to hold the first names of all employees.

Lesson 17.1: Use PL/SQL Records

A record is a group of related data items stored in one data structure. The developer can place or retrieve information from fields within a record, which are equivalent of columns in a table. A record can only hold one row (a record) at any time.

Records can be either:

Oracle defined structures

Or

User defined structures

Oracle Defined Records

These are records, which are created and controlled by Oracle. They allow a record to be created based on either the columns in a table row or the items retrieved from a cursor.

All that is required by the developer is to define a name for the record and associated it to an Oracle defined type.

The %TYPE is used to declare a variable of the column type. The variable has the same data type and size as the table column. The benefit of %TYPE is that you do not have to change the variable if the column is altered.

The %ROWTYPE attribute is used to declare a record that can hold an entire row of a table or view. The fields in the record take their names and data types from the columns of the table or view. The record can also store an entire row of data fetched from a cursor or cursor variable.

Syntax

```
record_name table_name%ROWTYPE;
record_name cursor_name%ROWTYPE;
```

To reference a field with a record, use the following syntax:

record_name.column_name_or_alias

Syntax:

```
record_name table_name%ROWTYPE;
```

SQL Example:

```
DECLARE
     t_emprec  employees%ROWTYPE;

BEGIN
     t_emprec.first_name:='Bob';
     DBMS_OUTPUT.PUT_LINE(t_emprec.first_name );
END;
```

Query Results:

```
anonymous block completed
Bob
```

Explanation:

Code file: Code17_1_1.sql

User Defined Records

There are 2 steps to creating records.

Define the record type – this is the structure of the record

Declare the record – this involves giving the record a name and assigning the record type to it

The syntax to define a record is:

```
TYPE type_name IS RECORD
(field_1 datatype,
[,field_2 datatype….];
```

Example:

```
TYPE car_emp_rec IS RECORD
(emp_id number,
last_name employee.last_name%type,
reg_number cars.regno%type);
```

Syntax:

```
TYPE type_name IS RECORD
(field_1 datatype,
[,field_2 datatype….];
```

SQL Example:

```
DECLARE
     TYPE car_rec IS RECORD
     (
     carname varchar(50),
     reg_number varchar(50)
     );
     v_mycar  car_rec;
BEGIN
     v_mycar.carname:='Mini';
     DBMS_OUTPUT.PUT_LINE(v_mycar.carname );
END;
```

Query Results:

```
anonymous block completed
Mini
```

Explanation: The example below shows a basic example of defining, declaring and assigning values to fields in a record.

Code file: Code17_1_2.sql

Lesson 17.2: Insert and Update with PL/SQL Records

Prior to Oracle 9i if we wished to insert the row of data in a record into a table, we would require to reference each value during the insert statement.

For example:

```
INSERT INTO table_name
VALUES (record.value1, record.value2);
```

In Oracle 9i it is possible to insert an entire record into a table using the insert statement.

Syntax:

```
INSERT INTO table_name
VALUES record_name;
```

Syntax:

```
INSERT INTO table_name
VALUES record_name;
```

SQL Example:

```
DECLARE
  t_deptrec  departments%rowtype;
BEGIN
t_deptrec.department_id:=0;
t_deptrec.department_name:='Support';
t_deptrec.manager_id:=200;
t_deptrec.location_id:=1700;
INSERT INTO Departments
VALUES t_deptrec;
END;
```

Query Results:

```
anonymous block completed
```

Explanation: In this example we are inserting the entire record into the department table.

Code file: Code17_2_1.sql

PL/SQL record can be used to update multiple columns in table as one entity without specifying each value.

Syntax:

```
UPDATE table
SET ROW = record_name
WHERE…;
```

Example:

Syntax:

```
UPDATE table
SET ROW = record_name
WHERE…;
```

SQL Example:

```
DECLARE
  t_deptrec  departments%rowtype;
BEGIN
t_deptrec.department_id:=0;
t_deptrec.department_name:='Support';
t_deptrec.manager_id:=200;
t_deptrec.location_id:=1700;
UPDATE departments
SET   ROW=t_deptrec
WHERE department_id=0;
END;
```

Query Results:

```
anonymous block completed
```

Explanation: In this example we are updating the entire record in the department table.

Code file: Code17_2_1.sql

Lesson 17.3: INDEX BY Tables

Collections

These are items that can be created to hold groups of related data items. Unlike records, collections can hold multiple rows of data. They are most commonly used to pass large amounts of data between sub programs. For example, you may populate a collection with the information from the employees table, use it in subprogram1 and then pass it as a parameter to subprogram2. This can be a very efficient way to operate.

There are 3 types of collection that can be used in PL/SQL:

Index by Tables (Associative Array's)

Varray's

Nested Tables

Index by Tables

Index by tables are memory-based structures that consist of 2 columns.

The main column is the actual data structure, which can be a standard data type i.e. a number, varchar2, etc., or a record. The other column is a binary index value, which must be unique for each row in the table.

The data held within the index by table is unconstrained, in other words there is no defined limit to the number of rows entered. The index column can be sparse, which means the index value does not have to be in sequential order. For example, the first row entered in the table can have the index value 2000 and the second could have the index value 2.

There are 2 parts to creating an Index by Tables:

1. **Define the Index by table**

   Syntax:

```
TYPE type_name IS TABLE OF data_type
INDEX BY BINARY_INTEGER;
```

   For example:

```
TYPE emp_tab_type  IS TABLE OF varchar2
INDEX BY BINARY_INTEGER;
```

2. Declare a table of that the defined type

Syntax:

```
table_name      index_table_type;
```

For example:

```
emp_list        emp_tab_type;
```

The syntax to populate an index by table is:

```
table_name(index):=value;
```

For example:

```
emp_list(1) :=’SMITH’;
```

Syntax:

```
TYPE type_name IS TABLE OF data_type
INDEX BY BINARY_INTEGER;

table_name      index_table_type;
```

SQL Example:

```
DECLARE
  TYPE products IS TABLE OF VARCHAR2(30)
    INDEX BY BINARY_INTEGER;
  t_products products;
BEGIN
  t_products(1) := 'HP PC';
  t_products(2) := 'CD Writer';
  t_products(3) := 'DVD';
DBMS_OUTPUT.PUT_LINE('Product  Code  "B001"  =  '  ||
t_products(1));
END;
```

Query Results:

```
anonymous block completed
Product Code 1 = HP PC
```

Explanation:

Code file: Code17_3_1.sql

The example above creates an index by table and populates it with an index from the emp_id column and the table value from the last_name column as shown below.

Index	Value
1	HP PC
2	CD Writer
3	DVD

Associative Arrays

New name for PL/SQL Index By Tables. Do not have to index by binary integer, can index by string

Syntax:

```
TYPE type_name IS TABLE OF datatype
INDEX BY varchar2(n);
```

Index value must be unique

Syntax:

```
TYPE type_name IS TABLE OF data_type
INDEX BY BINARY_INTEGER;

table_name      index_table_type;
```

SQL Example:

```
DECLARE
  TYPE products IS TABLE OF VARCHAR2(30)
    INDEX BY VARCHAR2(4);
  t_products products;
BEGIN
  t_products('A001') := 'HP PC';
  t_products('B001') := 'CD Writer';
  t_products('B002') := 'DVD';
DBMS_OUTPUT.PUT_LINE('Product   Code   "B001"   =   '   ||
t_products('B001'));
END;
```

Query Results:

```
anonymous block completed
Product Code "B001" = HP PC
```

Explanation:

Code file: Code17_3_2.sql

Index	Value
A001	HP PC
B001	CD Writer
B002	DVD

Lesson 17.5: Examine INDEX BY Table Methods

Collection methods can be used to operate and manipulate collections.

Method Name	Description
Exists(n)	Used to check if the entry is already used. It will return false if the row does not exist or true if it does. e.g. `IF car_nested.exists(2) =false then car_nested(2):='FORD';…`
COUNT	This will return the number of elements in the collection. e.g. `DBMS_OUTPUT.PUT_LINE(car_nested.count);`
FIRST and LAST	Return the first and last index values in a collection. Often used with a for loop. e.g. `FOR cnt IN car_nested.first..car_nested.last LOOP`
NEXT and PRIOR	Return the next and prior index values in a collection
DELETE	Removes either single or multiple entries from a collection. `E.g. car_nest.delete`

Lesson 17.6: Use INDEX BY Table of Records

As previously discussed, an associative array that is declared as a table of scalar data type can store the details of only one column in a database table. However, there is often a need to store all the columns retrieved by a query. The INDEX BY table of records option enables one array definition to hold information about all the fields of a database table.

Creating and Referencing a Table of Records

- Use the %ROWTYPE attribute to declare a record that represents a row in a database table
- Refer to fields within the dept_table array because each element of the array is a record

The differences between the %ROWTYPE attribute and the composite data type PL/SQL record are as follows:

- PL/SQL record types can be user-defined, whereas %ROWTYPE implicitly defines the record.
- PL/SQL records enable you to specify the fields and their data types while declaring them. When you use %ROWTYPE, you cannot specify the fields. The %ROWTYPE attribute represents a table row with all the fields based on the definition of that table.
- User-defined records are static, but %ROWTYPE records are dynamic—they are based on a table structure. If the table structure changes, the record structure also picks up the change

Index by table of records using RowType

Syntax:

SQL Example:

```
DECLARE
TYPE dept_table_type IS TABLE OF
departments%ROWTYPE INDEX BY PLS_INTEGER;
dept_table dept_table_type;
-- Each element of dept_table is a record
Begin
SELECT * INTO dept_table(1) FROM departments
WHERE department_id = 10;
DBMS_OUTPUT.PUT_LINE(dept_table(1).department_id      ||
dept_table(1).department_name
||dept_table(1).manager_id);
END;
```

Query Results:

```
anonymous block completed
10Administration200
```

Explanation: Index by table of records using RowType

Code file: Code17_6_1.sql

Syntax:

SQL Example:

```
DECLARE
TYPE emp_table_type IS TABLE OF
employees%ROWTYPE INDEX BY PLS_INTEGER;
my_emp_table emp_table_type;
max_count NUMBER(3):= 104;
BEGIN
FOR i IN 100..max_count
LOOP
SELECT * INTO my_emp_table(i) FROM employees
WHERE employee_id = i;
END LOOP;
FOR i IN my_emp_table.FIRST..my_emp_table.LAST
LOOP
DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
END LOOP;
END;
```

Query Results:

```
anonymous block completed
10Administration200
```

Explanation: Index by table of records using RowType

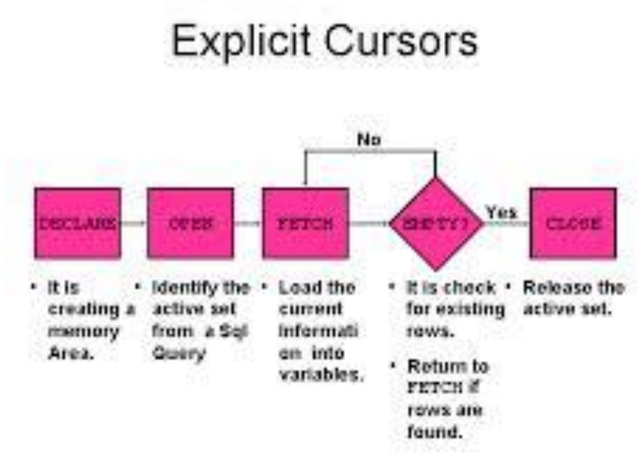
Code file: Code17_6_2.sql

Section 18: Explicit Cursors

In this section you will learn:

- What are Explicit Cursors?
- Declare the Cursor
- Open the Cursor
- Fetch data from the Cursor
- Close the Cursor
- Cursor FOR loop
- The %NOTFOUND and %ROWCOUNT Attributes
- Describe the FOR UPDATE Clause and WHERE CURRENT Clause

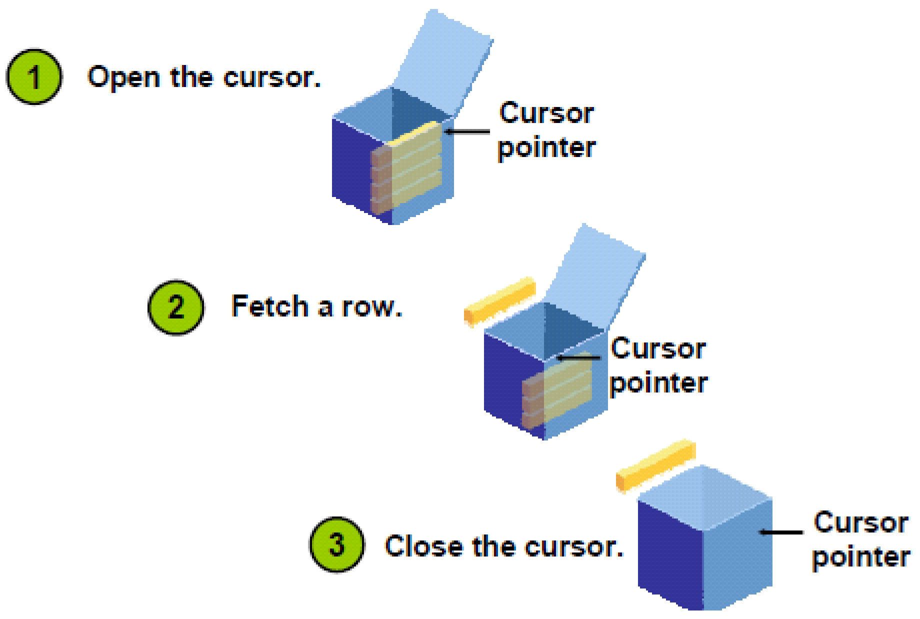
Lesson 18.1: What are Explicit Cursors?


Explicit Cursors
No
DECLARE
OPEN
FETCH
EMPTY?
Yes
CLOSE
• It is creating a memory Area.
• Identify the active set from a Sql Query
• Load the current informati on into variables.
• It is check for existing rows.
• Return to FETCH if rows are found.
• Release the active set.

Explicit Cursors are cursors that are explicitly defined in the declaration section of the PL/SQL code. Unlike implicit cursors the developer is responsible for manually performing the required actions. Explicit cursors can only be created for SELECT statements and therefore cannot be used with INSERT, UPDATE and DELETE commands.

An explicit cursor could be viewed as a virtual temporary table that exists and can be referenced from memory.

Controlling Explicit Cursors



There are several manual steps that require to be performed with explicit cursors:

PL/SQL program opens a cursor, processes rows returned by a query, and then closes the cursor. The cursor marks the current position in the active set.

1. The OPEN statement executes the query associated with the cursor, identifies the active set, and positions the cursor at the first row.

2. The FETCH statement retrieves the current row and advances the cursor to the next row until there are no more rows or a specified condition is met.

3. The CLOSE statement releases the cursor.

Lesson 18.2: Declare the Cursor

The first step is to define the cursor in the declaration section of the code. This involves providing a name for the cursor and defining the SELECT statement to be used.

The basic syntax for creating a cursor is as follows:

```
CURSOR cursor_name IS
select statement;
```

The name of the cursor must be unique and up to 30 characters in length. Many cursors can be defined in the declaration section.

```
DECLARE
CURSOR c_emp_cursor IS
SELECT employee_id, last_name FROM employees
WHERE department_id =30;
```

Do not include the INTO clause in the cursor declaration because it appears later in the FETCH statement.

- If you want the rows to be processed in a specific sequence, use the ORDER BY clause in the query.
- The cursor can be any valid SELECT statement, including joins, subqueries, and so on.

Declaring Variables

The next step is to define variables to hold the data that is to be retrieved from the cursor.

```
DECLARE
CURSOR c_emp_cursor IS
SELECT employee_id, last_name FROM employees
WHERE department_id =30;



V_employee_id employees.employee_id%type;
V_last_name employees.last_name%type;
```

Lesson 18.3: Open the Cursor

Although defined in the declaration section the select statement is not executed until the cursor is opened. Each cursor requires to be opened with the OPEN command.

Syntax:

```
OPEN cursor_name;
```

The open command only executes the query; the data is not retrieved until the next step.

```
OPEN c_emp_cursor;
```

Lesson 18.4: Fetch data from the Cursor

To retrieve the data from an open cursor the FETCH command is used. It places the data into the variables that have been defined.

Syntax:

```
        FETCH cursor_name INTO variable1, variable2..;
```

The variable list in the FETCH command must match the order of the columns in the SELECT statement.

The FETCH command will only return one row of data, which will be first row of the subset of data retrieved. Therefore a FETCH command would require to be repeated so each row of data in the select statement can be retrieved.

The fetch command moves the pointer to the first row of the subset and places the data into the referenced variables, as shown below.

FETCH c_emp_cursor INTO V_employee_id, V_last_name;

LAST_NAME	ANNUAL_SALARY	ANNUAL_BONUS
MACDONALD	15500	
WELSH	12500	
CALDERWOOD	15000	

Each time a fetch command is found for an open cursor the pointer is moved to the next row within the subset. For example, if we add an additional fetch the pointer will move to the data for WELSH, as shown below.

```
FETCH c_emp_cursor INTO V_employee_id, V_last_name;
```

LAST_NAME	ANNUAL_SALARY	ANNUAL_BONUS
MACDONALD	15500	
WELSH ➡	12500	
CALDERWOOD	15000	

The following code iterates through all the records.

```
LOOP
     FETCH c_emp_cursor INTO V_employee_id, V_last_name;
     EXIT WHEN c_emp_cursor%NOTFOUND;
     DBMS_OUTPUT.PUT_LINE(V_employee_id ||' '|| V_last_name);
END LOOP;
```

Lesson 18.5: Close the Cursor

Explicit cursors should always be closed once they are no longer required. Although in most cases the code could be successfully executed without closing the cursor it is good practice to always close. If the cursor is left open it is taking memory that no longer is required and could lead to problems of locked data.

Syntax:

```
CLOSE cursor_name;
```

Syntax:

```
DECLARE
CURSOR cursor_name  IS ….
OPEN cursor_name;
LOOP
FETCH cursor_name INTO variable1, variable2..;
EXIT WHEN cursor_name %NOTFOUND;
END LOOP;
CLOSE cursor_name;
```

SQL Example:

```
DECLARE
  CURSOR c_emp_cursor
  IS
    SELECT  employee_id,  last_name  FROM  employees  WHERE
department_id =30;
  V_employee_id employees.employee_id%type;
  V_last_name employees.last_name%type;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO V_employee_id, V_last_name;
    EXIT
  WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(    V_employee_id    ||'    '||
V_last_name);
  END LOOP;
  CLOSE c_emp_cursor;
END;
```

Query Results:

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

Explanation: Explicit cursor – to access more than 1 row of data

Code file: Code18_2_1.sql

Lesson 18.6: Cursor FOR loop

The *explicit cursor* for loop can save development time, as it does not require all the steps considered earlier in this chapter for explicit cursors.

The steps detailed earlier for explicit cursors were as follows:

1. Declare the cursor
2. Create storage variables for the data
3. Open the cursor
4. Fetch the data into variables
5. Close the cursor

If a **Cursor for loop** is used the only step required is:

Declare the cursor

The rest of the steps are automatically maintained by the system.

- A storage record is implicitly created for the data
- The system will automatically execute the query as part of the loop
- The data will be fetched row by row into the storage record
- Once there is no more data to retrieve the loop will close

Syntax:

```
FOR record_name IN cursor_name LOOP
      Statements…;
END LOOP;
```

Syntax:

```
FOR record_name IN cursor_name LOOP
      Statements…;
END LOOP;
```

SQL Example:

```
DECLARE
  CURSOR c_emp_cursor
  IS
    SELECT employee_id, last_name FROM employees WHERE
department_id =30;
BEGIN
  FOR c_rec IN c_emp_cursor
  LOOP
```

```
    DBMS_OUTPUT.PUT_LINE(   c_rec.employee_id   ||'   '||
c_rec.last_name);
  END LOOP;
END;
```

Query Results:

```
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares…
```

Explanation: If a **Cursor for loop** is used to iterate through all the records

Code file: Code18_6_1.sql

The cursor for loop can save development time but will not always be suitable. It should only be used if all the rows in the cursor require to be processed. In more complicated examples many cursors will be used, which will require multiple open, fetch and close command to be performed manually. The cursor for loop will not be suitable for these more complicated scenarios

Lesson 18.7: The %NOTFOUND and %ROWCOUNT Attributes

Cursor attributes can be applied and used with both implicit and explicit cursors. However, they tend to be more commonly found with explicit cursors.

Explicit Cursor Attributes

The syntax for explicit cursors is as follows:

```
cursor_name%attribute_name
```

For example

```
c_low_pay%notfound
```

%NOTFOUND Attribute

%FOUND Cursor Attribute

%ISOPEN Cursor Attribute

%ROWCOUNT Cursor Attribute

Syntax:

SQL Example:

```
DECLARE
  CURSOR c_emp_cursor
  IS
    SELECT employee_id, last_name FROM employees WHERE
department_id =30;
  V_employee_id employees.employee_id%type;
  V_last_name employees.last_name%type;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO V_employee_id, V_last_name;
    EXIT
  WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(   V_employee_id   ||'   '||
V_last_name || ' ' ||c_emp_cursor%rowcount);
  END LOOP;
  CLOSE c_emp_cursor;
END;
```

Query Results:

```
114 Raphaely 1
115 Khoo 2
116 Baida 3
```

```
117 Tobias 4
118 Himuro 5
119 Colmenares 6 …
```

Explanation: Row count attribute increments for each iteration

Code file: Code18_7_1.sql

Lesson 18.8: Describe the FOR UPDATE Clause and WHERE CURRENT Clause

When declaring an explicit cursor it is possible to assign an exclusive lock on the rows of data. This is achieved by using the FOR UPDATE and CURRENT OF clauses.

The FOR UPDATE clause can be added to the select statement within the cursor. If added, a lock will be place on the rows retrieved by the select statement. It is possible to specify column names in the clause however a lock will be placed on an entire record not just the individual columns.

Syntax for the FOR UPDATE clause:

```
CURSOR cursor_name IS
SELECT …
FOR UPDATE [OF column_name];
```

Syntax for the CURRENT OF:

```
UPDATE table_name
SET col_name = value or expression
WHERE CURRENT OF cursor_name;

DELETE FROM table_name
WHERE CURRENT OF cursor_name;
```

The CURRENT OF clause can be used with UPDATE and DELETE statement, if the FOR UPDATE clause has been applied to the cursor from where data is retrieved.

Syntax:

```
CURSOR cursor_name IS
SELECT …
FOR UPDATE [OF column_name];
```

SQL Example:

```
DECLARE
CURSOR c_emp_cursor
```

```
  IS
    SELECT  employee_id,  last_name  FROM  employees  WHERE
department_id =30
    FOR UPDATE;
  V_employee_id employees.employee_id%type;
  V_last_name employees.last_name%type;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO V_employee_id, V_last_name;
    EXIT
  WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(    V_employee_id    ||'    '||
V_last_name || ' ' ||c_emp_cursor%rowcount);
    UPDATE employees
    set Salary=salary*1.02
    Where current of c_emp_cursor;
  END LOOP;
  CLOSE c_emp_cursor;
END;
```

Query Results:

```
114 Raphaely 1
115 Khoo 2
116 Baida 3
117 Tobias 4
118 Himuro 5
119 Colmenares 6…
```

Explanation: Updates the salary field as it iterates through the records

Code file: Code18_8_1.sql

Section 19: Exception Handling

In this section you will:

- Understand Exceptions
- Handle Exceptions with PL/SQL
- Trap Predefined Oracle Server Errors
- Trap Non-Predefined Oracle Server Errors
- Trap User-Defined Exceptions
- Propagate Exceptions
- RAISE_APPLICATION_ERROR Procedure

Lesson 19.1: Understand Exceptions

An exception is an error in PL/SQL that is raised during the execution of a block. A block always terminates when PL/SQL raises an exception, but you can specify an exception handler to perform final actions before the block ends.

Exception are raised when

- An Oracle error occurs, and the associated exception is raised automatically.
- Depending on the business functionality your program implements, you may have to explicitly raise an exception

There are two types of exceptions in PL/SQL:

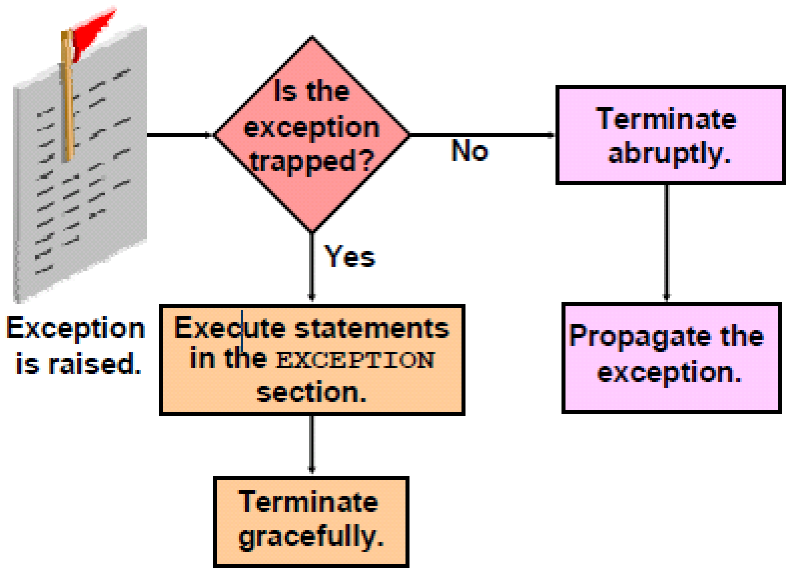
Oracle Defined

There are a large number of pre-defined Oracle exceptions that will automatically produce error messages. However the developer can redefine these error messages and error numbers.

User Defined

These are exceptions that are created and handled by the developer.

Lesson 19.2: Handle Exceptions with PL/SQL



An exception is an error or warning produced by PL/SQL. Exceptions can be handled within the optional exception section. Unlike other programming languages PL/SQL provides a specific section to handle the error or exception code. However, once the execution has moved to the exception section it is not possible to move back to the main executable block of code. The exception section appears between the BEGIN and END commands as shown in the diagram below.

```
[DECLARE

      declarations]

BEGIN

      executable statements

[EXCEPTION
```

You can trap any error by including a corresponding handler within the exception-handling section of the PL/SQL block. Each handler consists of a WHEN clause, which specifies an exception name, followed by a sequence of statements to be executed when that exception is raised

Lesson 19.3: Trap Predefined Oracle Server Errors

You can trap any error by including a corresponding handler within the exception-handling section of the PL/SQL block. Each handler consists of a WHEN clause, which specifies an exception name, followed by a sequence of statements to be executed when that exception is raised.

You can include any number of handlers within an EXCEPTION section to handle specific exceptions. However, you cannot have multiple handlers for a single exception. Exception trapping syntax includes the following elements:

exception Is the standard name of a predefined exception or the name of a user defined exception declared within the declarative section

statement Is one or more PL/SQL or SQL statements OTHERS Is an optional exception-handling clause that traps any exceptions that have not been explicitly handled

```
EXCEPTION
WHEN exception1 [OR exception2 . . .] THEN
statement1;
statement2;
. . .
[WHEN exception3 [OR exception4 . . .] THEN
statement1;
statement2;
. . .]
[WHEN OTHERS THEN
statement1;
statement2;
. . .]
```

Many of the key errors, which will occur with PL/SQL, will have exception names. The errors, which have these exception names, are listed below.

Exception Name	Oracle Error
ACCESS_INTO_NULL	**ORA-06530**
CASE_NOT_FOUND	**ORA-06592**
COLLECTION_IS_NULL	**ORA-06531**
CURSOR_ALREADY_OPEN	**ORA-06511**
DUP_VAL_ON_INDEX	**ORA-00001**
INVALID_CURSOR	**ORA-01001**
INVALID_NUMBER	**ORA-01722**

LOGIN_DENIED	**ORA-01017**
NO_DATA_FOUND	**ORA-01403**
NOT_LOGGED_ON	**ORA-01012**
PROGRAM_ERROR	**ORA-06501**
ROWTYPE_MISMATCH	**ORA-06504**
STORAGE_ERROR	**ORA-06500**
SUBSRIPT_BEYOND_COUNT	**ORA-06533**
SUBSCRIPT_OUTSIDE_LIMIT	**ORA-06532**
SYS_INVALID_ROWID	**ORA-01410**
TIMEOUT_ON_RESOURCE	**ORA-00051**
TOO_MANY_ROWS	**ORA-01422**
VALUE_ERROR	**ORA-06502**
ZERO_DIVIDE	**ORA-01476**

Syntax:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
INTO
FROM table
[WHERE condition(s)];
```

SQL Example:

```
DECLARE
v_fname VARCHAR2(25);
v_lname VARCHAR2(25);
BEGIN
  SELECT first_name, last_name
  INTO v_fname,v_lname
  FROM employees WHERE employee_id=99;

END;
```

Query Results:

```
Error report -
ORA-01403: no data found
ORA-06512: at line 5
01403. 00000 -  "no data found"
```

Explanation: Generates an error as there is no employee 99

Code file: Code19_3_1.sql

Handling the error

Syntax:

```
EXCEPTION
WHEN exception1 [OR exception2 . . .] THEN
statement1;
statement2;
. . .
[WHEN exception3 [OR exception4 . . .] THEN
statement1;
statement2;
. . .]
[WHEN OTHERS THEN
statement1;
statement2;
. . .]
```

SQL Example:

```
DECLARE
v_fname VARCHAR2(25);
v_lname VARCHAR2(25);
BEGIN
  SELECT first_name, last_name
  INTO v_fname,v_lname
  FROM employees WHERE employee_id=99;
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('There are too many rows');
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No data found');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Others');
END;
```

Query Results:

```
anonymous block completed
No data found
```

Explanation: The error is handled

Code file: Code19_3_2.sql

The example below will cause an exception to occur due to the Select Into statement returning multiple rows.

Syntax:

```
EXCEPTION
WHEN exception1 [OR exception2 . . .] THEN
statement1;
statement2;
. . .
[WHEN exception3 [OR exception4 . . .] THEN
statement1;
statement2;
. . .]
[WHEN OTHERS THEN
statement1;
statement2;
. . .]
```

SQL Example:

```
DECLARE
v_fname VARCHAR2(25);
v_lname VARCHAR2(25);
BEGIN
  SELECT first_name, last_name
  INTO v_fname,v_lname
  FROM employees WHERE department_id=20;
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('There are too many rows');
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No data found');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Others');
END;
```

Query Results:

```
anonymous block completed
No data found
```

Explanation: The example will cause an exception to occur due to the Select Into statement returning multiple rows. The Exception block will catch it.

Code file: Code19_3_3.sql

Lesson 19.4: Trap Non-Predefined Oracle Server Errors

Non-predefined exceptions are similar to predefined exceptions; however, they are not defined as PL/SQL exceptions in the Oracle Server. They are standard Oracle errors. You create exceptions with standard Oracle errors by using the PRAGMA EXCEPTION_INIT function. Such exceptions are called non-predefined exceptions.

You can trap a non-predefined Oracle Server error by declaring it first. The declared exception is raised implicitly. In PL/SQL, PRAGMA EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. This enables you to refer to any internal exception by name and to write a specific handler for it.

Syntax:

```
PRAGMA EXCEPTION_INIT(exception_name, error_number);
```

SQL Example:

```
DECLARE
e_insert_excep EXCEPTION;
PRAGMA EXCEPTION_INIT(e_insert_excep, -01400);
BEGIN
INSERT INTO departments
(department_id, department_name) VALUES (280, NULL);
EXCEPTION
WHEN e_insert_excep THEN
DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
```

Query Results:

```
INSERT OPERATION FAILED
ORA-01400:      cannot      insert      NULL      into
("HR"."DEPARTMENTS"."DEPARTMENT_NAME")
```

Explanation: PRAGMA EXCEPTION_INIT tells the compiler to associate an exception name e_insert_excep with an Oracle error number -01400

Code file: Code19_4_1.sql

SQLCODE and SQLERRM

The SQLCODE system variable will hold the ORA number of the exception. For example SQLCODE will hold the value -6502 for the VALUE_ERROR exception.

The SQLERRM system variable will hold the full error number and message. For example SQLERRM will hold the value “ORA-01422: exact fetch returns more than requested number of rows”.

The SQLCODE or SQLERRM variable are most commonly used with the WHEN OTHER exception handler.

```
DECLARE
error_code NUMBER;
error_message VARCHAR2(255);
BEGIN
...
EXCEPTION
...
WHEN OTHERS THEN
ROLLBACK;
error_code := SQLCODE ;
error_message := SQLERRM ;
INSERT INTO errors (e_user, e_date, error_code,
error_message) VALUES(USER,SYSDATE,error_code,
error_message);
END;
```

Lesson 19.5: Trap User-Defined Exceptions

In the creation of effective code, the developer will want to provide error handling to ensure the execution of the code is successful and without side effects.

There are 3 steps involved in creating user-defined exceptions:

1. Define the exception. This involves declaring an exception and providing a name that can be referred to in the code.
2. Raise the exception. This is where the exception is called in the main executable section.
3. Handle the exception. This involves writing the PL/SQL code in the exception section of the block.

Defining the Exception

The first step is to declare the exception. Exceptions are declared in the similar way as variables and constants. The syntax is shown below:

```
      exception_name EXCEPTION;
```

For example

```
      declare
      err_late       EXCEPTION;
      begin
…..
```

Raising the Exception

The exception name is called from the main executable code using the RAISE command. This command is normally used as part of a conditional section of the code i.e. an IF statement. Once the exception is raised it will move the execution of the code to the EXCEPTION section. The code can never return to main execution statement once it has navigated into the EXCEPTION section.

The syntax for raising the exception is as follows:

```
RAISE exception_name;
```

For example

```
IF …. THEN
raise err_late;
…
```

Handling the Exception

Once raised the execution of the code has moved to the Exception section. This is where the actions to be performed are created and therefore the handling code is entered. For example, we may require an error message to be displayed or information entered into an auditing table. The EXCEPTION is handled by a WHEN clause that operates in a similar way to an IF statement.

Syntax:

```
WHEN exception_name THEN
PL/SQL statement;
[WHEN exception_name OR exception_name THEN
PL/SQL statement;]
[WHEN OTHERS THEN
PL/SQL statement;]
```

For example:

```
EXCEPTION
WHEN err_late THEN
INSERT INTO…..;
END;
```

Syntax:

```
exception_name EXCEPTION;
RAISE exception_name;
     EXCEPTION
     WHEN err_late THEN
     INSERT INTO…..;
     END;
```

SQL Example:

```
DECLARE
  v_deptno             NUMBER       := 500;
  v_name               VARCHAR2(20) := 'Testing';
  e_invalid_department EXCEPTION;
BEGIN
  UPDATE departments
  SET department_name = v_name
  WHERE department_id = v_deptno;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_department;
  END IF;
  COMMIT;
EXCEPTION
```

```
WHEN e_invalid_department THEN
  DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
```

Query Results:

```
anonymous block completed
No such department id.…
```

Explanation: Creating a user-defined exception

Code file: Code19_5_1.sql

Lesson 19.6: Propagate Exceptions

When a subblock handles an exception, it terminates normally. Control resumes in the enclosing block immediately after the subblock's END statement. However, if a PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates to successive enclosing blocks until it finds a handler. If none of these blocks handles the exception, an unhandled exception in the host environment results. When the exception propagates to an enclosing block, the remaining executable actions in that block are bypassed. One advantage of this behavior is that you can enclose statements that require their own exclusive error handling in their own block, while leaving more general exception handling to the enclosing block. Note in the example that the exceptions (no_rows and integrity) are declared in the outer block. In the inner block, when the no_rows exception is raised, PL/SQL looks for the exception to be handled in the subblock. Because the exception is not handled in the subblock, the exception propagates to the outer block, where PL/SQL finds the handler.

```
DECLARE
. . .
     e_no_rows exception;
     e_integrity exception;
     PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
     FOR c_record IN emp_cursor LOOP
          BEGIN
               SELECT ...
               UPDATE ...
                    IF SQL%NOTFOUND THEN
                         RAISE e_no_rows;
                    END IF;
               END;
          END LOOP;
     EXCEPTION
          WHEN e_integrity THEN ...
          WHEN e_no_rows THEN ...
END;
```

Lesson 19.7: RAISE_APPLICATION_ERROR Procedure

You can use this procedure to issue user-defined error messages from stored subprograms. You can report errors to your application and avoid returning unhandled exceptions.

The Syntax is as follows:

```
RAISE_APPLICATION_ERROR (error_number, ‘message string’);
```

RAISE_APPLICATION_ERROR causes a *fatal* error; therefore the code that completed successfully prior to the exception will be rolled back thus ensuring data integrity.

Oracle has reserved specific error numbers for this built-in. The developer can use numbers from -20001 to -20999. When developing an application it is good practice to assign numbers to specific standard error messages. Each Oracle exception has a name and a number in addition to the displayed message. The “exact fetch returns more than requested number of rows” message has an ORA number of -01422 and is called TOO_MANY_ROWS. It is possible to intercept the Oracle exception and provide alternative exception handling. For example the message “exact fetch returns more than requested number of rows” may be changed to a more meaningful message e.g. “More than one individuals details are returned”.

To provide alternative exception handling for Oracle predefined exceptions there are fewer steps to perform.

The developer does not require to:

Name the exception as a name has been provided by Oracle.

Raise the exception. The event, which causes the exception to occur, is defined by Oracle. Therefore, the only step required by the developer is:

Syntax:

```
RAISE_APPLICATION_ERROR (errnumber,errromessage)
```

SQL Example:

```
DECLARE
v_fname VARCHAR2(25);
v_lname VARCHAR2(25);
BEGIN
```

```
  SELECT first_name, last_name
  INTO v_fname,v_lname
  FROM employees WHERE department_id=20;
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
   -- DBMS_OUTPUT.PUT_LINE('There are too many rows');
   RAISE_APPLICATION_ERROR(-20000, 'More than one employee in
this department');
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No data found');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Others');
END;
```

Query Results:

```
Error report -
ORA-20000: More than one employee in this department
ORA-06512: at line 11
20000. 00000 -  "%s"
*Cause:    The stored procedure 'raise_application_error'
           was called which causes this error to be generated.
*Action:   Correct the problem as described in the error message
or contact
           the application administrator or DBA for more
information.
```

Explanation: We have redefined the previous example to display a different error message.

Code file: Code19_3_4.sql

Section 20: Stored Procedures and functions

In this section you will:

- Create a Modularized and Layered Subprogram Design
- Modularize Development With PL/SQL Blocks
- Understand the PL/SQL Execution Environment
- List the benefits of using PL/SQL Subprograms
- List the differences between Anonymous Blocks and Subprograms
- Create, Call, and Remove Stored Procedures
- Implement Procedures Parameters and Parameters Modes
- View Procedure Information

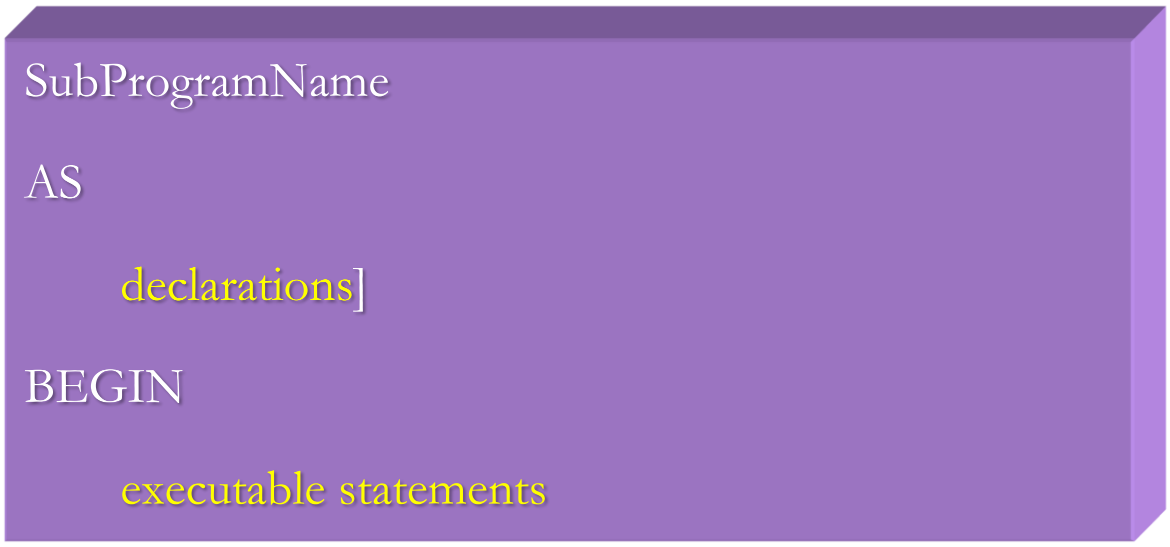
Lesson 20.1: Create a Modularized and Layered Subprogram Design

A sub program is a block of PL/SQL code that is provided with a name and is both stored and executed from the database.

In this chapter we shall consider the following 2 types of PL/SQL subprogram:

- Procedures
- Functions

Syntactically the 2 types of sub program are very similar to the anonymous blocks detailed over the earlier chapters of this manual. The user will find it fairly easy to adapt to using these structures.



```
SubProgramName
AS
        declarations]
BEGIN
        executable statements
```

Anonymous blocks are unnamed executable PL/SQL blocks. Because they are unnamed, they can be neither reused nor stored for later use.

Procedures and functions are named PL/SQL blocks that are also known as *subprograms*. These subprograms are compiled and stored in the database. The block structure of the subprograms is similar to the structure of anonymous blocks. Subprograms can be declared not only at the schema level but also within any other PL/SQL block. A subprogram contains the following sections:

Declarative section: Subprograms can have an optional declarative section. However, unlike anonymous blocks, the declarative section of a subprogram does not start with the DECLARE keyword. The optional declarative section follows the IS or AS keyword in the subprogram declaration.

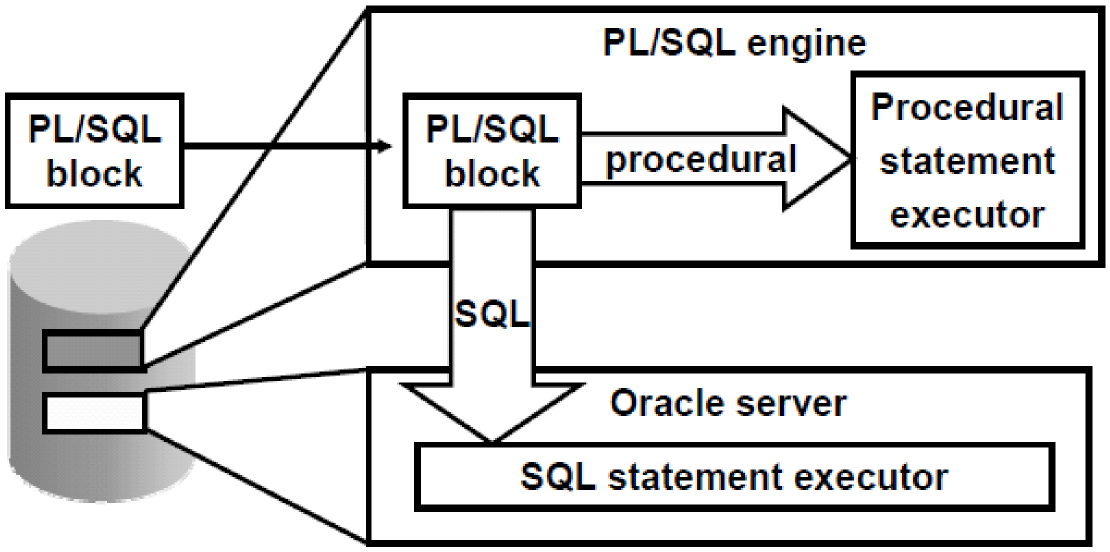
Executable section: This is the mandatory section of the subprogram, which contains the implementation of the business logic. Looking at the code in this section, you can easily determine the business functionality of the subprogram. This section begins and ends with the **BEGIN** and **END** keywords, respectively.

Exception section: This is an optional section that is included to handle exceptions.

Lesson 20.2: Understand the PL/SQL Execution Environment

PL/SQL Execution Environment

The PL/SQL run-time architecture:



All PL/SQL statements are processed in the Procedural Statement Executor, and all SQL statements must be sent to the SQL Statement Executor for processing by the Oracle Server processes. The SQL environment may also invoke the PL/SQL environment. For example, the PL/SQL environment is invoked when a PL/SQL function is used in a SELECT statement. The PL/SQL engine is a virtual machine that resides in memory and processes the PL/SQL m-code instructions. When the PL/SQL engine encounters a SQL statement, a context switch is made to pass the SQL statement to the Oracle Server processes. The PL/SQL engine waits for the SQL statement to complete and for the results to be returned before it continues to process subsequent statements in the PL/SQL block.

Lesson 20.3: List the benefits of using PL/SQL Subprograms

There are several advantages that can be gained by using subprograms.

Increased performance

The anonymous block considered earlier can lead to increased performance as they/ allow blocks of SQL statements to be sent to the database rather being sent as individual statements. This reduces the traffic between the client and database. However, subprograms take this a step further as once compiled and stored on the database they are executed from the database without the syntax requiring to be rechecked.

Reduce Memory Usage

Once executed the PL/SQL code from a subprogram is automatically held in the shared area of memory (SGA). Therefore, this cached code could be reused and shared by other users. By default, the code will be timed out of the SGA depending on the space allocated.

Reusable Code

A key feature of writing good code is its ability to be reused. It is highly likely that the same piece of code may be required by several applications. As subprograms are stored as database objects many users can share them. Therefore, one subprogram may be referenced by many other sub programs.

Security Tool

As previously mentioned, it is possible to share code. A developer can give others access to execute a subprogram in a similar way to giving others access to a database table. By default, those given access will inherit the owner's permissions to database objects referenced by the code, for only the execution duration. Therefore, even if a user does not have direct permission to update the employees table they will inherit it during the execution of the code. In Oracle 8I a new feature was added to allow this behavior to be overwritten. This new feature is known as INVOKER's rights and will be discussed later.

Lesson 20.4: List the differences between Anonymous Blocks and Subprograms

Anonymous	SubPrograms
Cannot take parameters	Can take parameters
Do not return values	If functions, must return values
Cannot be invoked by other applications	can be invoked
Not stored in the database	Stored in the database
Compiled every time	Compiled only once
Unnamed PL/SQL blocks	Named PL/SQL blocks

Lesson 20.5: Create a Stored Procedures

Procedures are blocks of PL/SQL code stored and executed from the database. They are normally used to perform actions on the database, for example update rows in a table.

Basic Procedure Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
IS|AS
[declarations]
BEGIN
      executable statements…
[EXCEPTION]
END;
```

If OR REPLACE syntax is used it will automatically overwrite a procedure with the same name. Procedure names must be unique to the user schema. Therefore, if we have a table called EMPLOYEES we cannot create a procedure called EMPLOYEES.

The commands "IS" or "AS" must be used to identify the start of the declaration section. Unlike anonymous blocks, subprograms do not require the DECLARE command. After the "AS" and "IS" the basic structure of the Procedure is the same as an anonymous block of PL/SQL code.

Procedures are often used to perform actions on the database, for example update rows in a table.

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
IS|AS
[declarations]
BEGIN
     executable statements…
[EXCEPTION]
END;
```

SQL Example:

```
CREATE or REPLACE PROCEDURE upd_emp IS
v_emp_id employees.employee_id%TYPE;

BEGIN
v_emp_id:=100;
Update employees
set salary=salary*1.1
where employee_id=v_emp_id;
DBMS_OUTPUT.PUT_LINE(' Updated '|| SQL%ROWCOUNT
||' row ');
END;
```

Query Results:

```
PROCEDURE UPD_EMP compiled
```

Explanation:

Code file: Code20_5_1.sql

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
IS|AS
[declarations]
BEGIN
     executable statements…
[EXCEPTION]
END;
```

SQL Example:

```
CREATE or REPLACE PROCEDURE add_dept IS
v_dept_id departments.department_id%TYPE;
v_dept_name departments.department_name%TYPE;
BEGIN
v_dept_id:=280;
v_dept_name:='ST-Curriculum';
INSERT INTO departments(department_id,department_name)
VALUES(v_dept_id,v_dept_name);
DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT
||' row ');
END;
```

Query Results:

```
PROCEDURE ADD_DEPT compiled
```

Explanation: The above procedure will update the annual_bonus column for all trainers. They will be provided with an annual bonus of 6% of their annual salary

Code file: Code20_5_1.sql

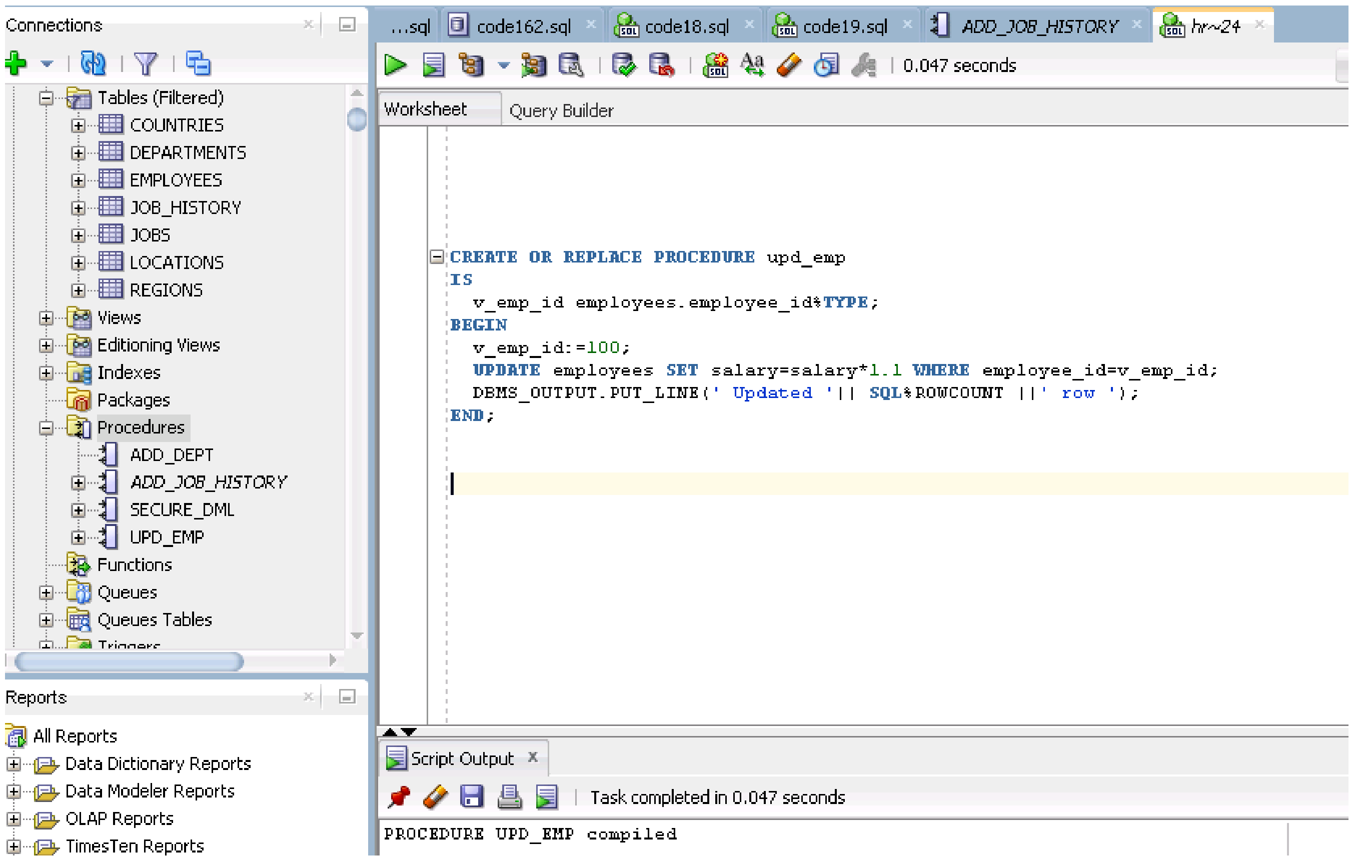
Lesson 20.6: Building and calling a stored procedure

There are 2 steps in building sub programs:

Step 1 -Compile the PL/SQL code

This involves running the file that contains the PL/SQL code to check the syntax and create the sub program as an object on the database.

The picture below shows the procedure being compiled. When it compiles successfully the message PROCEDURE CREATED will be displayed. However, the update statement will not have been executed at this stage.



```
CREATE OR REPLACE PROCEDURE upd_emp
IS
  v_emp_id employees.employee_id%TYPE;
BEGIN
  v_emp_id:=100;
  UPDATE employees SET salary=salary*1.1 WHERE employee_id=v_emp_id;
  DBMS_OUTPUT.PUT_LINE(' Updated '|| SQL%ROWCOUNT ||' row ');
END;
```

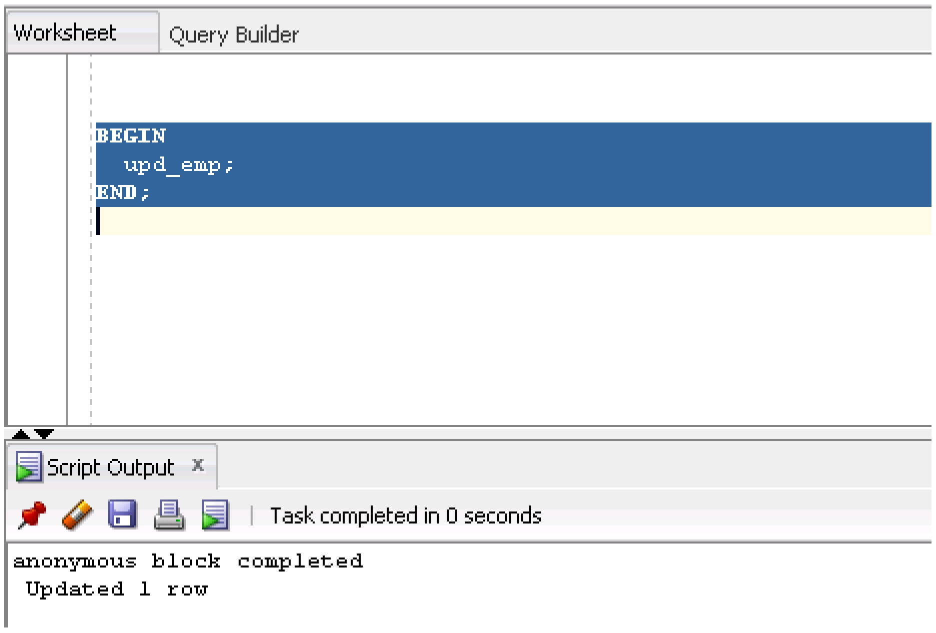
```
PROCEDURE UPD_EMP compiled
```

Step 2 – Executing the Procedure

This involves running the procedure. In the example above this would be when the update statement is performed. A compiled procedure can be executed in several ways:

The syntax is:

```
BEGIN
      [owner.]procedure_name;
END;
```



```
BEGIN
  upd_emp;
END;
```

```
anonymous block completed
 Updated 1 row
```

Lesson 20.7: Create a function

Functions are very similar syntactically to the procedures covered in the last section. However, functions are normally used to produce a result. For example, a function could be created to calculate the amount of tax to be paid or the cost of a sale.

Functions must return a value for example a number, text or date.

Basic Function Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
RETURN datatype
IS|AS
[Declaration]
BEGIN
     Executable statements…
[EXCEPTION]
END;
```

If OR REPLACE syntax is used it will automatically overwrite a function with the same name.

Function names must be unique to the user schema. Therefore if we have a table called EMPLOYEES we cannot create a function called EMPLOYEES

The command "IS" or "AS" must be used to identify the start of the declaration section. Unlike anonymous blocks subprograms do not require the DECLARE command.

After the AS or IS the basic structure of the function is the same as an anonymous block of PL/SQL code except for the fact it must return a value of the same type specified in the function header.

Syntax:

```
CREATE [OR REPLACE] FUNCTION function_name
RETURN datatype
IS|AS
[Declaration]
BEGIN
     Executable statements…
[EXCEPTION]
END
```

SQL Example:

```
CREATE OR REPLACE FUNCTION get_empSalary
  RETURN NUMBER
```

```
IS
  v_emp_id employees.employee_id%TYPE;
  v_Salary employees.salary%type;
BEGIN
  v_emp_id:=100;
  SELECT  Salary  INTO  v_Salary  FROM  employees  WHERE
employee_id=v_emp_id;
  RETURN v_Salary;
END;
```

Query Results:

```
PROCEDURE UPD_EMP compiled
```

Explanation: Function procedure

Code file: Code20_7_1.sql

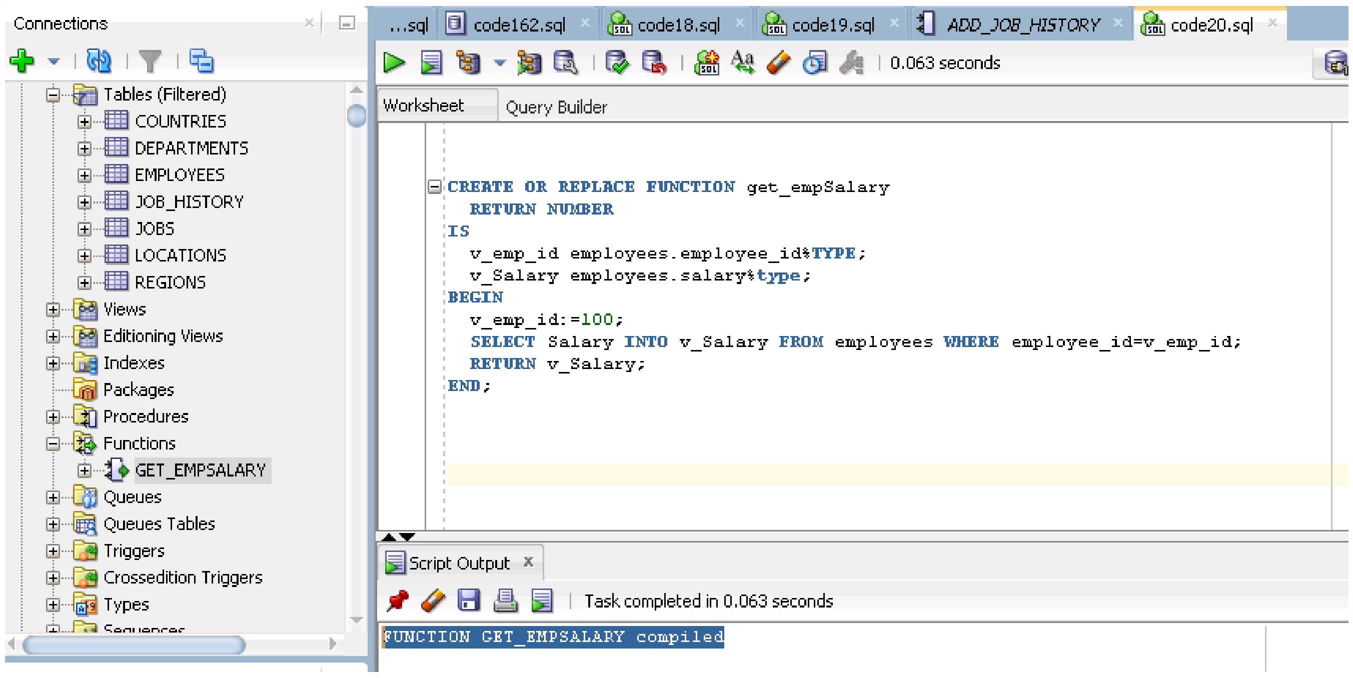
Lesson 20.8: Building and calling a function

There are 2 steps in building sub programs:

Step 1 -Compile the PL/SQL code

This involves running the file that contains the PL/SQL code to check the syntax and create the sub program as an object on the database.

The picture below shows the procedure being compiled. When it compiles successfully the message Function CREATED will be displayed. However, the update statement will not have been executed at this stage.



```
CREATE OR REPLACE FUNCTION get_empSalary
  RETURN NUMBER
IS
  v_emp_id employees.employee_id%TYPE;
  v_Salary employees.salary%type;
BEGIN
  v_emp_id:=100;
  SELECT Salary INTO v_Salary FROM employees WHERE employee_id=v_emp_id;
  RETURN v_Salary;
END;
```

```
FUNCTION GET_EMPSALARY compiled
```

Step 2 – Executing a function

This involves running the procedure. In the example above this would be when the update statement is performed. A compiled FUNCTION can be executed in several ways:

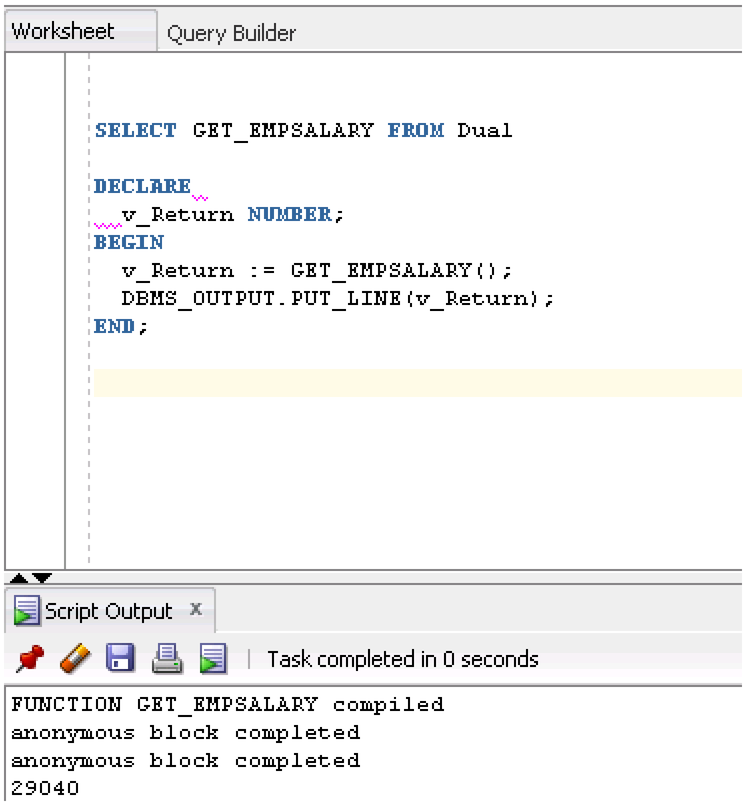
The syntax is:

```
          BEGIN
               VARIABLE :=[owner.]FUNCTION_name;
          ;
          END;
```

```
DECLARE
  v_Return NUMBER;
BEGIN
  v_Return := GET_EMPSALARY();
  DBMS_OUTPUT.PUT_LINE(v_Return);
END;
```

OR

```
SELECT GET_EMPSALARY FROM Dual
```



```
SELECT GET_EMPSALARY FROM Dual

DECLARE
  v_Return NUMBER;
BEGIN
  v_Return := GET_EMPSALARY();
  DBMS_OUTPUT.PUT_LINE(v_Return);
END;
```

```
FUNCTION GET_EMPSALARY compiled
anonymous block completed
anonymous block completed
29040
```

Lesson 20.9: Implement Subprogram Parameters and Parameters Modes

The 2 previous examples shown over the last few pages are a little unrealistic, as we have used hard coded values, which can make them pretty useless and not reusable. Therefore, to create more dynamic sub programs we can define parameters which can allow values to be passed in when the sub program is executed.

Sub program parameters are always defined directly after the name of the sub program, as shown below.

```
CREATE OR REPLACE SUB_PROGRAM_TYPE sub_program_name [(parameter_list)]
AS
….
```

It is possible to create several parameters in a parameter list. The syntax for defining a parameter is as follows:

```
param_name[param_type]datatype[DEFAULT|:=value]
```

Each parameter defined in a list must be separated with a comma.

Note the following when defining parameters:

- Parameter names should be unique and should not be given the same name as existing database objects
- Parameter types are optional to specify but are used to define the behaviour of the parameter. We shall consider these later in this chapter.
- The parameter datatype cannot be constrained i.e. A scale, or maximum characters or numbers cannot be provided e.g. VARCHAR2 must be used rather that VARCHAR2(20)
- Parameters can be assigned a default value by using either the := or "DEFAULT" assignment operators.

Procedure with parameters:

Syntax:

```
CREATE   OR   REPLACE   SUB_PROGRAM_TYPE   sub_program_name
[(parameter_list)]
AS
```

SQL Example:

```
CREATE OR REPLACE PROCEDURE add_deptWithParameters(
    p_dept_id departments.department_id%TYPE,
    p_dept_name departments.department_name%TYPE )
AS
  v_dept_name departments.department_id%TYPE;
BEGIN
  v_dept_name:=Upper(p_dept_name);
  INSERT
  INTO departments
    (
      department_id,
      department_name
    )
    VALUES
    (
      p_dept_id,
      v_dept_name
    );
  DBMS_OUTPUT.PUT_LINE('  Inserted  '||  SQL%ROWCOUNT  ||'
row ');
END
```

Query Results:

```
PROCEDURE ADD_DEPTWITHPARAMETERS compiled
```

Explanation: Procedure with parameters

Code file: Code20_9_1.sql

Syntax:

```
CREATE OR REPLACE SUB_PROGRAM_TYPE sub_program_name
[(parameter_list)]
AS
```

SQL Example:

```
CREATE OR REPLACE PROCEDURE upd_empWithParameters(
    p_emp_id employees.employee_id%TYPE )
IS
BEGIN
  UPDATE employees SET salary=salary*1.1 WHERE
employee_id=p_emp_id;
  DBMS_OUTPUT.PUT_LINE(' Updated '|| SQL%ROWCOUNT ||' row
');
END;
```

Query Results:

```
PROCEDURE UPD_EMPWITHPARAMETERS compiled
```

Explanation:

Code file: Code20_9_2.sql

Execute a stored procedure

```
BEGIN
  UPD_EMPWITHPARAMETERS(120);
END;
```

Function with parameters

Syntax:

```
CREATE   OR   REPLACE   SUB_PROGRAM_TYPE   sub_program_name
[(parameter_list)]
AS
```

SQL Example:

```
CREATE OR REPLACE FUNCTION get_empSalaryWithParameters(
    p_emp_id employees.employee_id%TYPE)
  RETURN NUMBER
IS
  v_Salary employees.salary%type;
BEGIN
  SELECT  Salary  INTO  v_Salary  FROM  employees  WHERE
employee_id=p_emp_id;
  RETURN v_Salary;
END;
```

Query Results:

```
FUNCTION GET_EMPSALARYWITHPARAMETERS compiled
```

Explanation:

Code file: Code20_9_3.sql

Parameter Type (Modes)

When defining a parameter the developer has the option of providing a parameter type or mode. This controls the behaviour of the parameter during the execution of the code. There are 3 parameter types that can be applied:

IN (Default)

If a parameter type is not defined it will be automatically become an IN parameter. This means that the value passed into the parameter at execution is a constant value and therefore is read only.

OUT

If a parameter is defined as OUT it will be assigned a value during the execution of the code and is therefore similar in behaviour to a variable. OUT parameters are write only parameters and are not given a value when the sub program is initially executed.

IN OUT

An IN OUT parameter can have a value passed to it at initial execution in the same way as an IN parameter, but the values can be reassigned in the code in the same way as an OUT parameter. Therefore it can have the behaviour of both IN and OUT parameters.

The types are defined using the following syntax:

```
param_name[IN|OUT|IN OUT] datatype[DEFAULT|:=value]
```

Parameter Type Examples:

This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

Syntax:

```
param_name[IN|OUT|IN OUT] datatype[DEFAULT|:=value]
```

SQL Example:

```
DECLARE
   a number;
   b number;
   c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number)
IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;

BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
```

Query Results:

```
Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.
```

Explanation: This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

Code file: Code20_9_4.sql

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use same parameter to accept a value and then return another result.

SQL Example:

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
```

Query Results:

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

Code file: Code20_9_5.sql

Methods for Passing Parameters

Actual parameters could be passed in three ways:

- Positional notation
- Named notation
- Mixed notation

POSITIONAL NOTATION

In positional notation, you can call the procedure as:

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, a is substituted for x, b is substituted for y, c is substituted for z and d is substituted for m.

NAMED NOTATION

In named notation, the actual parameter is associated with the formal parameter using the arrow symbol (=>). So the procedure call would look like:

```
findMin(x=>a, y=>b, z=>c, m=>d);
```

Lesson 20.10: How to debug Functions and Procedures?

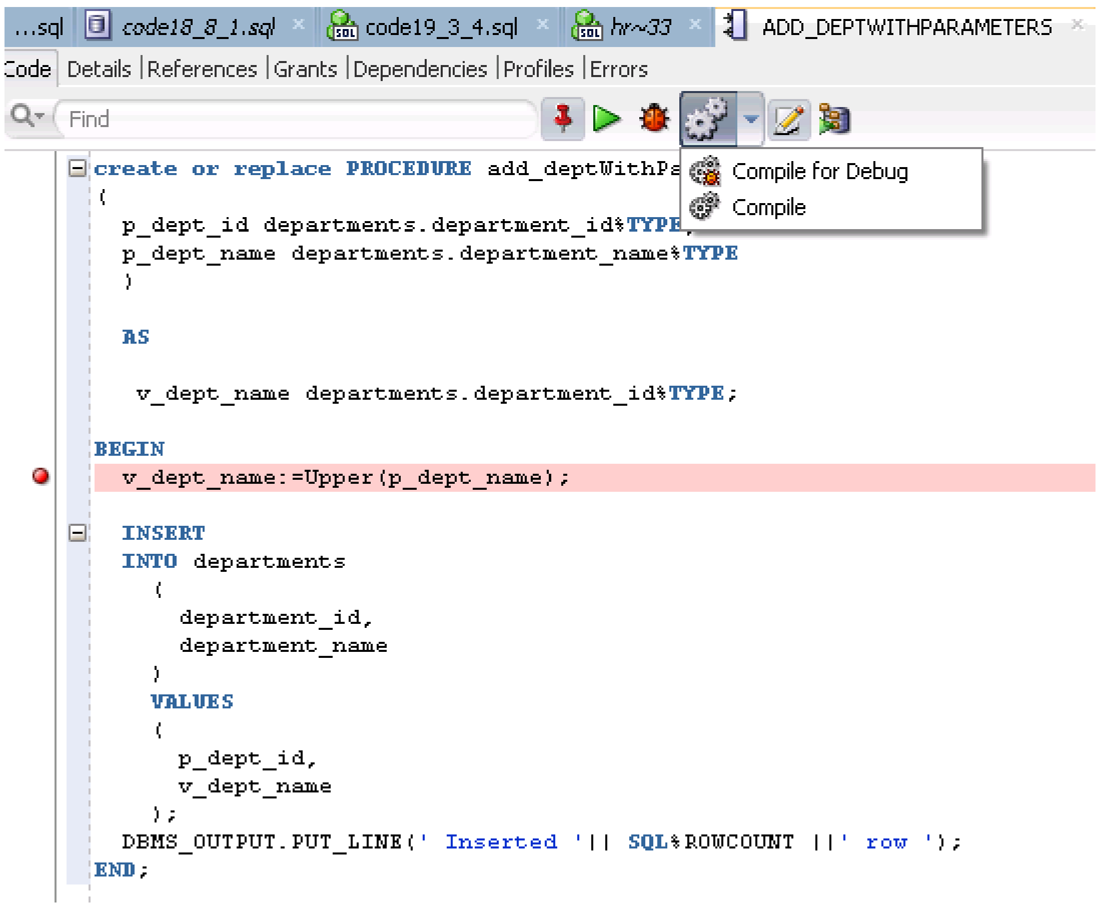
Oracle PL/SQL Debugger is a reliable tool that offers step-by-step code execution, breakpoints, watches, a call stack, a variables evaluation mechanism to automate debugging of Oracle stored functions and procedures.

Features

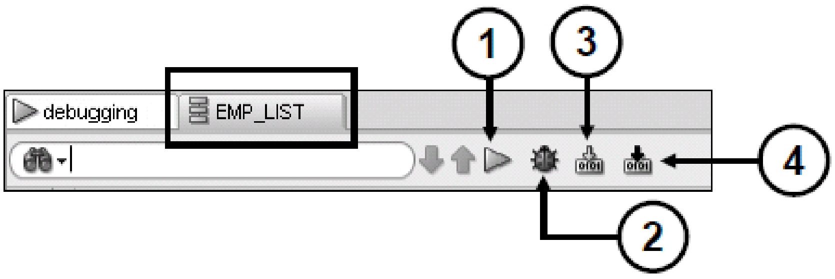
- PL/SQL code, script debugging
- Step Into, Step Over, and Step Out commands for step-by-step execution
- Breakpoints support for procedures, functions, triggers, and scripts
- Breakpoints, Call Stack, Watches windows

To start debugging:

- Open procedure
- Select Compile for Debug
- Click the on line of code that you want to break in
- Select Debug

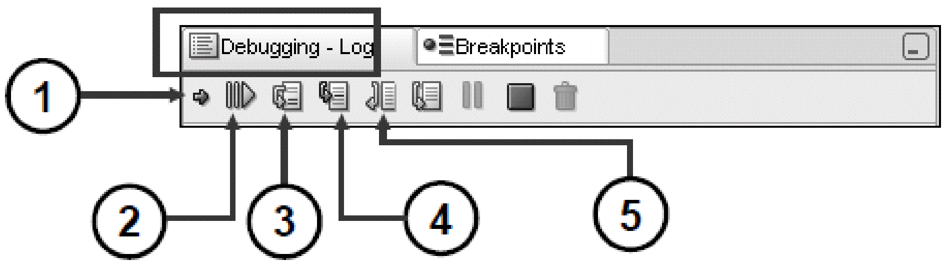


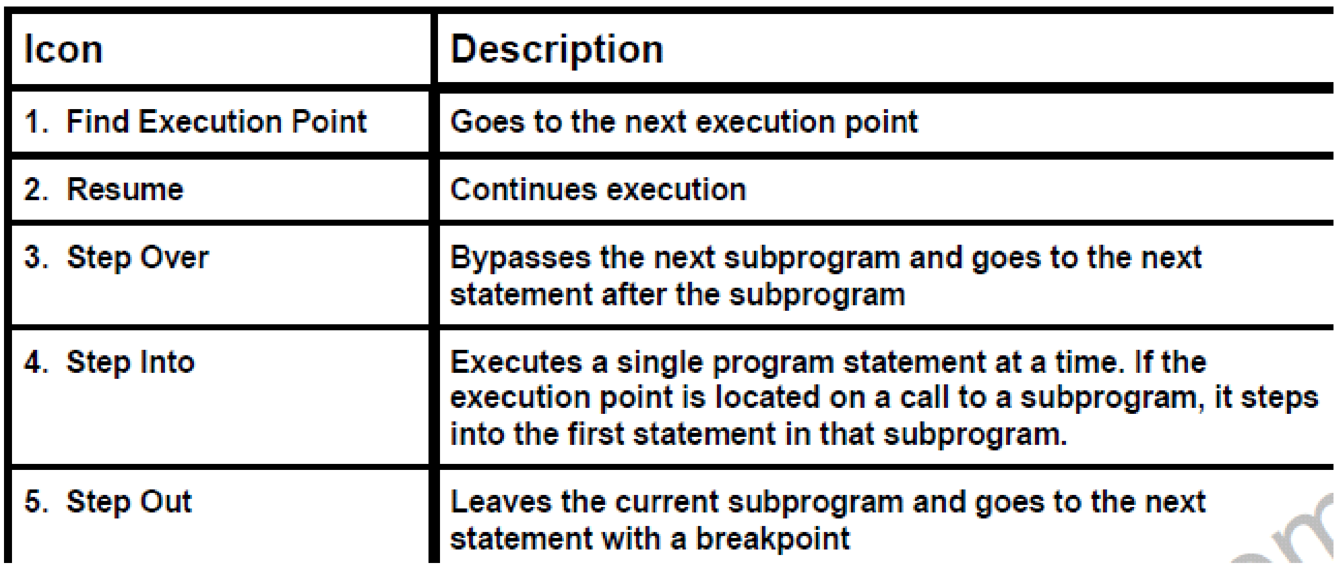
```
create or replace PROCEDURE add_deptWithPa
(
  p_dept_id departments.department_id%TYPE,
  p_dept_name departments.department_name%TYPE
  )

  AS

   v_dept_name departments.department_id%TYPE;

BEGIN
  v_dept_name:=Upper(p_dept_name);

  INSERT
  INTO departments
    (
      department_id,
      department_name
    )
    VALUES
    (
      p_dept_id,
      v_dept_name
    );
  DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT ||' row ');
END;
```



Icon	Description
1. Run	Starts normal execution of the function or procedure, and displays the results in the Running - Log tab
2. Debug	Starts execution of the subprogram in debug mode, and displays the Debugging - Log tab, which includes the debugging toolbar for controlling the execution
3. Compile	Performs a PL/SQL compilation of the subprogram
4. Compile for Debug	Performs a PL/SQL compilation of the subprogram so that it can be debugged

Step Commands





Icon	Description
1. Find Execution Point	Goes to the next execution point
2. Resume	Continues execution
3. Step Over	Bypasses the next subprogram and goes to the next statement after the subprogram
4. Step Into	Executes a single program statement at a time. If the execution point is located on a call to a subprogram, it steps into the first statement in that subprogram.
5. Step Out	Leaves the current subprogram and goes to the next statement with a breakpoint

Section 21: Packages

Packages are database objects that allow different types of PL/SQL objects to be grouped into a single entity. They can consist of items such as:

- Procedures
- Functions
- Variables and Constants
- Cursors
- Collections

There are two types of PL/SQL packages used by developers:

User Defined Packages

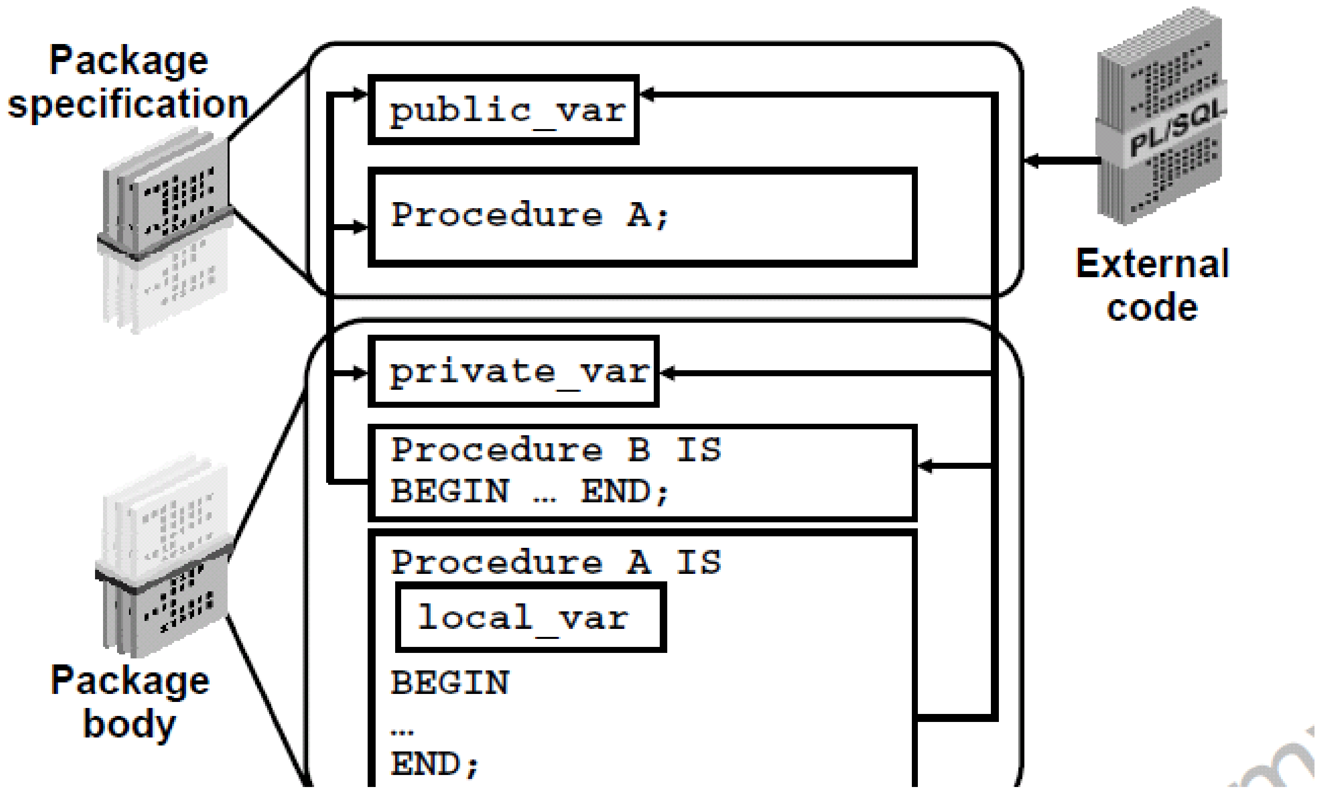
These are PL/SQL libraries of code and other items that can be used to when constructing applications.

Oracle Supplied Packages

These are libraries of PL/SQL code and objects that Oracle has created and developed. They provide additional features to help the developer provide specific actions. We have already encountered a package during this course called DBMS_OUTPUT. Initially the PL/SQL language had no method of displaying information to the screen, which caused problems to the developer. Therefore, the DBMS_OUTPUT package was introduced to enable screen output.

Lesson 21.1: Package Specifications and body

User Defined Packages



There are 2 parts that must be created before a package can be used.

The package specification

This is the public part of the package where the entities to be created are declared. The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

Syntax:

```
CREATE OR REPLACE PACKAGE package_name
AS
….
END;
```

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
```

The package body

This is known as the private section of the package where the entities defined in the specification are fully developed.

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package. `The CREATE PACKAGE BODY Statement is used for creating the package body.`

Syntax:

```
     CREATE OR REPLACE PACKAGE BODY package_name
     AS
     ….
     END;
```

The following code snippet shows the package body declaration for the *cust_sal* package created above.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
      SELECT salary INTO c_sal
      FROM customers
      WHERE id = c_id;
      dbms_output.put_line('Salary: '|| c_sal);
   END find_sal;
END cust_sal;
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax:

```
package_name.element_name;
```

Consider, we already have created above package in our database schema, the following program uses the *find_sal* method of the *cust_sal* package:

```
DECLARE
   code customers.id%type := &cc_id;
BEGIN
   cust_sal.find_sal(code);
END;
/
```

When the above code is executed at SQL prompt, it prompt to enter customer ID and when you enter an ID, it displays corresponding salary as follows:

Package syntax

```
CREATE [OR REPLACE] PACKAGE package_name
   [AUTHID {CURRENT_USER | DEFINER}]
   {IS | AS}
   [PRAGMA SERIALLY_REUSABLE;]
   [collection_type_definition ...]
   [record_type_definition ...]
   [subtype_definition ...]
   [collection_declaration ...]
   [constant_declaration ...]
   [exception_declaration ...]
   [object_declaration ...]
   [record_declaration ...]
   [variable_declaration ...]
   [cursor_spec ...]
   [function_spec ...]
   [procedure_spec ...]
   [call_spec ...]
   [PRAGMA RESTRICT_REFERENCES(assertions) ...]
END [package_name];

[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
   [PRAGMA SERIALLY_REUSABLE;]
   [collection_type_definition ...]
   [record_type_definition ...]
   [subtype_definition ...]
   [collection_declaration ...]
   [constant_declaration ...]
   [exception_declaration ...]
   [object_declaration ...]
   [record_declaration ...]
   [variable_declaration ...]
   [cursor_body ...]
   [function_spec ...]
   [procedure_spec ...]
   [call_spec ...]
[BEGIN
   sequence_of_statements]
END [package_name];]
```

Lesson 21.2: Advantages of Packages

There are many advantages that can be gained by building packages.

Improved Performance

When you execute a sub program from a package for the first time in a session the entire package is loaded into memory. This means that subsequent calls to the package code will be accessed directly from memory. Therefore, performance should be improved if code is regularly executed from the package.

Organising Code

When building an application, packages allow related items such as procedures and functions to be grouped into single entities. This avoids having many sub programs stored in different user schemas. This ensures effective application development.

Security

Rather than having to provide permissions on each sub program, a developer can grant *execute* permissions to a package. This would give access to all the procedures and functions defined in the package.

Reusing Code

This is same advantage that is gained by using sub programs. Developers can reference the code from a package thus provide reusable code and saving development time.

Top Down Design

As the PL/SQL package is split into the specification and body it is possible to initially define the PL/SQL objects in the specification and then add the code to the body at a later date.

Hide Code

Because only the specification is the public part of the code it is possible to hide complex coding in the body.

Lesson 21.3: Creating Packages

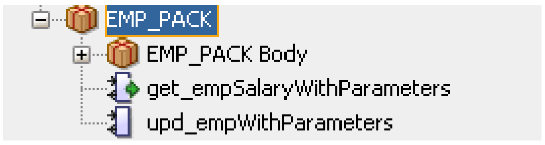
Syntax:

```
CREATE OR REPLACE PACKAGE
```

SQL Example:

```
CREATE OR REPLACE PACKAGE emp_pack
AS
  PROCEDURE upd_empWithParameters(
      p_emp_id employees.employee_id%TYPE);
  FUNCTION get_empSalaryWithParameters(
      p_emp_id employees.employee_id%TYPE)
    RETURN NUMBER;
END;
CREATE PACKAGE BODY emp_pack
AS
  PROCEDURE upd_empWithParameters(
      p_emp_id employees.employee_id%TYPE )
  IS
  BEGIN
    UPDATE   employees   SET   salary=salary*1.1   WHERE
employee_id=p_emp_id;
    DBMS_OUTPUT.PUT_LINE(' Updated '|| SQL%ROWCOUNT ||'
row ');
  END;
  FUNCTION get_empSalaryWithParameters(
      p_emp_id employees.employee_id%TYPE)
    RETURN NUMBER
  IS
    v_Salary employees.salary%type;
  BEGIN
    SELECT  Salary  INTO  v_Salary  FROM  employees  WHERE
employee_id=p_emp_id;
    RETURN v_Salary;
  END;
END;
```

Query Results:


```
EMP_PACK
  EMP_PACK Body
  get_empSalaryWithParameters
  upd_empWithParameters
```

Explanation: The above example shows creating a package specification, which defines a procedure and function

Code file: Code21.sql

Example of a Package

```
CREATE or replace PACKAGE emp_actions AS
   /* Declare externally visible types, cursor, exception. */
   TYPE EmpRecTyp IS RECORD (emp_id INT, salary REAL);
   TYPE DeptRecTyp IS RECORD (dept_id INT, location VARCHAR2(50));
   CURSOR desc_salary RETURN EmpRecTyp;
   invalid_salary EXCEPTION;

   /* Declare externally callable subprograms. */
   FUNCTION hire_employee (
      ename  VARCHAR2,
      job    VARCHAR2,
      mgr    REAL,
      sal    REAL,
      comm   REAL,
      deptno REAL) RETURN INT;
   PROCEDURE fire_employee (emp_id INT);
   PROCEDURE raise_salary (emp_id INT, grade INT, amount REAL);
   FUNCTION nth_highest_salary (n INT) RETURN EmpRecTyp;
END emp_actions;

CREATE PACKAGE BODY emp_actions AS
   number_hired INT;  -- visible only in this package

   /* Fully define cursor specified in package. */
   CURSOR desc_salary RETURN EmpRecTyp IS
      SELECT empno, sal FROM emp ORDER BY sal DESC;

   /* Fully define subprograms specified in package. */
   FUNCTION hire_employee (
      ename  VARCHAR2,
      job    VARCHAR2,
      mgr    REAL,
      sal    REAL,
      comm   REAL,
      deptno REAL) RETURN INT IS
      new_empno INT;
   BEGIN
      SELECT empno_seq.NEXTVAL INTO new_empno FROM dual;
      INSERT INTO emp VALUES (new_empno, ename, job,
         mgr, SYSDATE, sal, comm, deptno);
      number_hired := number_hired + 1;
      RETURN new_empno;
   END hire_employee;

   PROCEDURE fire_employee (emp_id INT) IS
   BEGIN
      DELETE FROM emp WHERE empno = emp_id;
   END fire_employee;

   /* Define local function, available only inside package. */
   FUNCTION sal_ok (rank INT, salary REAL) RETURN BOOLEAN IS
      min_sal REAL;
      max_sal REAL;
   BEGIN
      SELECT losal, hisal INTO min_sal, max_sal FROM salgrade
         WHERE grade = rank;
      RETURN (salary >= min_sal) AND (salary <= max_sal);
   END sal_ok;

   PROCEDURE raise_salary (emp_id INT, grade INT, amount REAL) IS
      salary REAL;
   BEGIN
      SELECT sal INTO salary FROM emp WHERE empno = emp_id;
      IF sal_ok(grade, salary + amount) THEN
         UPDATE emp SET sal = sal + amount WHERE empno = emp_id;
      ELSE
         RAISE invalid_salary;
      END IF;
   END raise_salary;

   FUNCTION nth_highest_salary (n INT) RETURN EmpRecTyp IS
      emp_rec EmpRecTyp;
   BEGIN
      OPEN desc_salary;
      FOR i IN 1..n LOOP
         FETCH desc_salary INTO emp_rec;
      END LOOP;
      CLOSE desc_salary;
      RETURN emp_rec;
   END nth_highest_salary;

BEGIN  -- initialization part starts here
   INSERT INTO emp_audit VALUES (SYSDATE, USER, 'EMP_ACTIONS');
   number_hired := 0;
END emp_actions;
```

Referencing Items from a Package

It is possible to reference items from a package by using the following syntax:

```
[OWNER.]PACKAGE_NAME.OBJECT_NAME
```

The example below shows executing a procedure from a package.

Lesson 21.4: Scope of Packaged Items

When a session references a package item, Oracle Database instantiates the package for that session. Every session that references a package has its own instantiation of that package.

When Oracle Database instantiates a package, it initializes it. Initialization includes whichever of the following are applicable:

- Assigning initial values to public constants
- Assigning initial values to public variables whose declarations specify them
- Executing the initialization part of the package body

The scope of where a packaged item can be used is dependent on its location in the package. Remember we can place items in both the specification and the body.

If items are defined in the specification, they are publicly available to any user with execute permission on the package. Once created cursors or variables can be referenced either from the package body or from an external block of PL/SQL code.

If the variables, constants or cursors are defined in the body of the package they are available to other procedures and functions defined in the body. As the scope of these is in the body of the package they cannot be referenced from out with the package.

The advantage of creating variables, constants and cursors in the package body is that they can be reused by the functions and procedures created in the package body.

The example below shows using variables and cursors defined in the package body and then referenced in a procedure.

```
CREATE OR REPLACE PACKAGE emp_actions AS  -- spec
   TYPE EmpRecTyp IS RECORD (emp_id INT, salary REAL);
   CURSOR desc_salary RETURN EmpRecTyp;
   PROCEDURE hire_employee (
      ename  VARCHAR2,
      job    VARCHAR2,
      mgr    NUMBER,
      sal    NUMBER,
      comm   NUMBER,
      deptno NUMBER);
   PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

CREATE OR REPLACE PACKAGE BODY emp_actions AS  -- body
   CURSOR desc_salary RETURN EmpRecTyp IS
      SELECT empno, sal FROM emp ORDER BY sal DESC;
   PROCEDURE hire_employee (
      ename  VARCHAR2,
      job    VARCHAR2,
      mgr    NUMBER,
      sal    NUMBER,
      comm   NUMBER,
      deptno NUMBER) IS
   BEGIN
      INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename, job,
         mgr, SYSDATE, sal, comm, deptno);
   END hire_employee;

   PROCEDURE fire_employee (emp_id NUMBER) IS
   BEGIN
      DELETE FROM emp WHERE empno = emp_id;
   END fire_employee;
END emp_actions;
```

Package state

The values of the variables, constants, and cursors that a package declares (in either its specification or body) comprise its **package state**. If a PL/SQL package declares at least one variable, constant, or cursor, then the package is **stateful**; otherwise, it is **stateless**.

Each session that references a package item has its own instantiation of that package. If the package is stateful, the instantiation includes its state.

The package state persists for the life of a session, except in these situations:

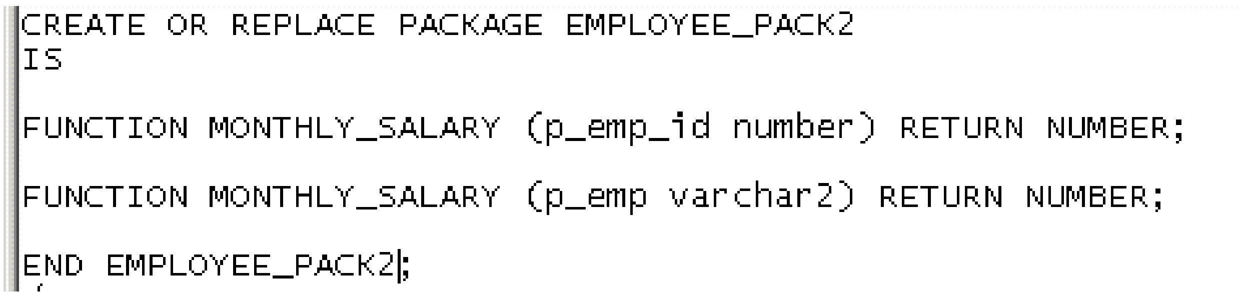
- The package is SERIALLY_REUSABLE.
- The package body is recompiled.
- After PL/SQL raises the exception, a reference to the package causes Oracle Database to re-instantiate the package, which re-initializes it.
- Any of the session's instantiated packages are invalidated and revalidated.

Lesson 21.5: Overloading in Packages

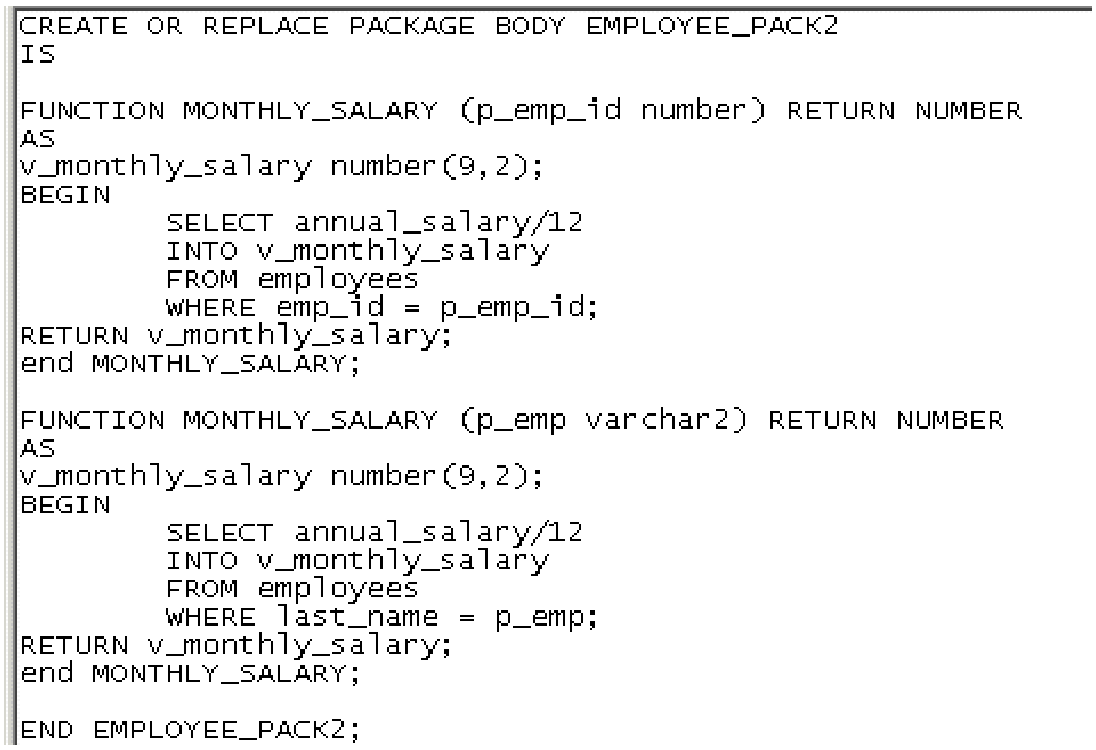
Overloading is very useful feature of packages. It allows procedure and functions to be given the same name but have either different type of data entered as parameters or a different number of parameters defined.

Overloading Example:

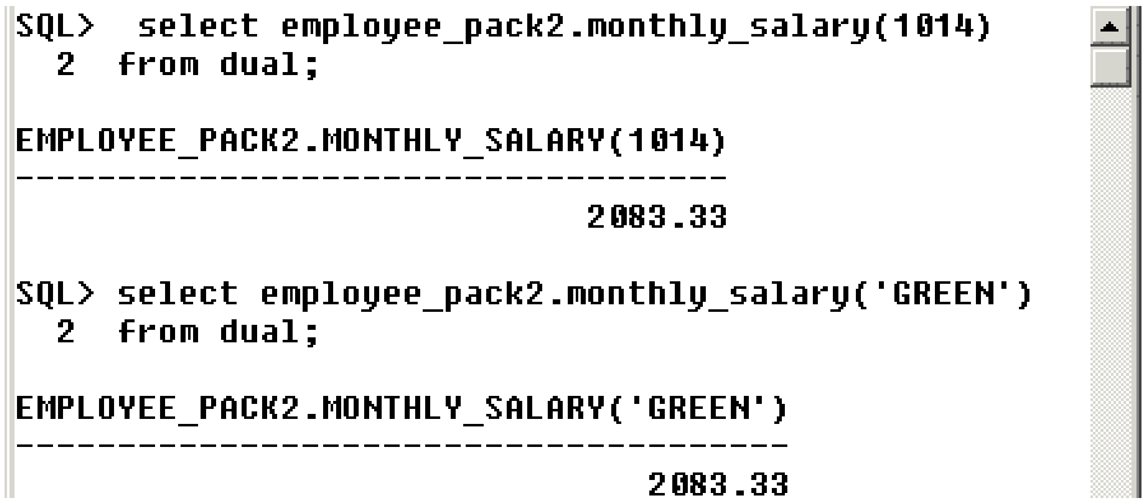
In the example below we have created a package with 2 functions both with the same name. One of the functions will return a monthly salary based on the employee id while the second function will return a monthly salary based on the last name of the employee.



```
CREATE OR REPLACE PACKAGE EMPLOYEE_PACK2
IS

FUNCTION MONTHLY_SALARY (p_emp_id number) RETURN NUMBER;

FUNCTION MONTHLY_SALARY (p_emp varchar2) RETURN NUMBER;

END EMPLOYEE_PACK2;
```



```
CREATE OR REPLACE PACKAGE BODY EMPLOYEE_PACK2
IS

FUNCTION MONTHLY_SALARY (p_emp_id number) RETURN NUMBER
AS
v_monthly_salary number(9,2);
BEGIN
        SELECT annual_salary/12
        INTO v_monthly_salary
        FROM employees
        WHERE emp_id = p_emp_id;
RETURN v_monthly_salary;
end MONTHLY_SALARY;

FUNCTION MONTHLY_SALARY (p_emp varchar2) RETURN NUMBER
AS
v_monthly_salary number(9,2);
BEGIN
        SELECT annual_salary/12
        INTO v_monthly_salary
        FROM employees
        WHERE last_name = p_emp;
RETURN v_monthly_salary;
end MONTHLY_SALARY;

END EMPLOYEE_PACK2;
```



```
SQL>  select employee_pack2.monthly_salary(1014)
  2  from dual;

EMPLOYEE_PACK2.MONTHLY_SALARY(1014)
-----------------------------------
                            2083.33

SQL> select employee_pack2.monthly_salary('GREEN')
  2  from dual;

EMPLOYEE_PACK2.MONTHLY_SALARY('GREEN')
--------------------------------------
                               2083.33
```

The above example shows executing an overloaded function. At execution Oracle decides which function to use based on the parameter values provided.

Lesson 21.6: Oracle Packages

Oracle supplies many PL/SQL packages with the Oracle server to extend database functionality. You can use the supplied packages when creating your applications.

To view a list of the Oracle supplier packages run the following

```
select object_name
 from dba_objects
 where owner = 'SYS'
 and object_type = 'PACKAGE';
```

There are over 300 packages available. Even though not all of them are intended for developers, the ones that are for developers are very well documented in the standard Oracle documentation

Below are a few delivered package solutions

The FORMAT_CALL_STACK function within the delivered dbms_utility package returns the call stack as a character string.

```
CREATE OR REPLACE PROCEDURE p1
  IS
   BEGIN
     DBMS_OUTPUT.put_line (DBMS_UTILITY.format_call_stack);
  END;

 CREATE OR REPLACE PACKAGE pkg1
 IS
     PROCEDURE p1;
  END pkg1;

CREATE OR REPLACE PACKAGE BODY pkg1
 IS
     PROCEDURE p2
    IS
     BEGIN
       P1;
     END;
 END pkg1;

CREATE OR REPLACE PROCEDURE p3
  IS
  BEGIN
     FOR indx IN 1 .. 500
     LOOP
        NULL;
    END LOOP;

    pkg1.p2;
  END;

 BEGIN
     p3;
  END;

——————— PL/SQL Call Stack ———————
   object handle    line number   object name
000007FF7EA83240              4   procedure HR.p1
```

```
000007FF7E9CC3B0             6   package body HR.PKG1
000007FF7EA0A3B0             9   procedure HR.p3
000007FF7EA07C00             2   anonymous block
```

DBMS_UTILITY.FORMAT_ERROR_STACK built-in function, like SQLERRM, returns the message associated with the current error (the value returned by SQLCODE). The DBMS_UTILITY.FORMAT_ERROR_STACK function differs from SQLERRM as it can return an error message as long as 1,899 characters truncation issues when the error stack gets long. (SQLERRM truncates at only 510 characters.)

DBMS_UTILITY.FORMAT_ERROR_BACKTRACE built-in function returns a formatted string that displays a stack of programs and line numbers tracing back to the line on which the error was originally raised.

UTL_CALL_STACK Package

The UTL_CALL_STACK package in Oracle 12c provides information about currently executing subprograms. the execution call stack, the error stack *and* error backtrace data.

Lesson 21.7: WhiteLists

Most PL/SQL-based applications are made up of many packages, some of which are the “top level” API to be used by programmers to implement user requirements and others of which are “helper” packages that are to be used only by certain other packages. In Oracle Database 12c you can limit down to the package/procedure or function level what bits of code may invoke other bits of code in the database. This process is called "white listing" and can be used to implement the concept of least privileges in your database.

Prior to 12c PL/SQL could not prevent a session from using any and all subprograms in packages to which that session’s schema had been granted EXECUTE authority This could have implications in the area of SQL Injection.

In Oracle 12c with the white list approach, the only way to execute a given piece of code would be to run it from a specific set of compiled units. You cannot execute a white listed unit from the top level, it must be called by some specific set of units. This is all accomplished with the new "accessible by" clause. The ACCESSIBLE BY clause can be added to packages, procedures, functions and types to specify which objects are able to reference the PL/SQL object directly

```
CREATE OR REPLACE FUNCTION  HelloWorld
      ( pv_message VARCHAR2 ) RETURN VARCHAR2
            ACCESSIBLE BY
            ( FUNCTION mypkg.f1
            , PROCEDURE mypkg.p1
            , PACKAGE mypkg.api
            , TYPE mypkg.myType )
            IS
            lv_message VARCHAR2(20) := 'Hello ';
            BEGIN
           lv_message := lv_message || pv_message || '!';
           RETURN lv_message;
           END;
```

White List Callers Invalidates or disallows compilation with dependencies as shown below

```
CREATE OR REPLACE FUNCTION CallerFn
       ( pv_message VARCHAR2 ) RETURN VARCHAR2 IS
           BEGIN
            RETURN HelloWorld (pv_message);
            END

      PL/SQL: Statement ignored PLS-00904: insufficient privilege to access
      object CallerFn
```

As the “PLS” error indicates, this issue is caught at compilation time. There is no runtime performance hit for using this feature.

Whitelisting in Package Specification

Consider the following example, create a public package with one procedure

```
CREATE OR REPLACE PACKAGE public_pkg
IS
   PROCEDURE do_work;
END;
/
```

Create a package with two procedures. The package is private in the sense that it can be invoked only from within the public package (public_pkg). The ACCESSIBLE_BY clause is added to the package specification.

```
CREATE OR REPLACE PACKAGE private_pkg
   ACCESSIBLE BY (public_pkg)
IS
   PROCEDURE do_this;

   PROCEDURE do_that;
END;
/
```

The package bodies are as follows

```
CREATE OR REPLACE PACKAGE BODY public_pkg
IS
   PROCEDURE do_work
   IS
   BEGIN
      private_pkg.do_this;
      private_pkg.do_that;
   END;
END;


CREATE OR REPLACE PACKAGE BODY
private_pkg
IS
   PROCEDURE do_this
   IS
   BEGIN
      DBMS_OUTPUT.put_line ('THIS');
   END;

   PROCEDURE do_that
   IS
   BEGIN
      DBMS_OUTPUT.put_line ('THAT');
   END;
END;
/
```

Running the public package's proceduresn are follows run successfully

```
BEGIN
   public_pkg.do_work;
END;
```

Running the following anonymous block fails

```
BEGIN
```

```
   private_pkg.do_this;
END;
```

Consider the following procedure using "definer rights" which is the default in Oracle.

```
CREATE PROCEDURE DEL_EMP AS
BEGIN
   DELETE FROM EMPLOYEES;
END;
```

Another user who calls this procedure only needs Execute privilege for this procedure, it is not required that such user has Delete privilege on table Employees.

Procedure runs under permission of the procedure owner or user who *defined* is, thus it is called "definer" rights.

```
CREATE PROCEDURE DEL_EMP AS
authid current_user
BEGIN
   DELETE FROM EMPLOYEES;
END;
```

A user who runs this procedure successfully must have Execute privilege for this procedure and delete privilege for table Employees.

Lesson 21.8: Invoker and Definer rights

When designing and creating PL/SQL applications, a thorough understanding of the rights models used in the Oracle database is vital for an effective security design.

The **definer rights** mode: This is the default mode of operation, whereby Oracle uses the security privileges and object resolution from the creator and thus "definer" of the procedure. This has been the traditional model for application development over the years.

The **invoker rights** mode: This mode works the same as definer rights during program compilation. However, at execution time, the database uses the privileges and object resolution of the **invoker** of the procedure.

Oracle Modes for Executing Stored Procedures

	DEFINER RIGHTS		INVOKER RIGHTS	
	Compilation	**Execution**	**Compilation**	**Execution**
Object Resolution	Definer	Definer	Definer	Invoker
Privileges	Definer	Definer	Definer	Invoker
Roles	Disabled	Disabled	Disabled	Enabled

Using definer rights, at both compilation and execution, the database will use the privilege set of, and the objects belonging to, the definer of the procedure. An important point to note is that database roles are disabled. When using invoker rights, at execution, the database uses the privilege set and objects belonging to the invoker of the procedure (the schema whose privileges are currently in effect during a particular session). Unlike definer rights, roles are enabled at execution time.

One of the most common problems experienced by PL/SQL developers is illustrated in the following example. We want to create a function that returns the module name (or program) for the current users' session. The module name is obtained from joining the V$SESSION and V$PROCESS views. To ensure access to these objects, we will build the function as the SYSTEM user (this is not a recommended approach, but it is common for people to do this). SYSTEM has been granted the DBA role that allows access to the V$ views. The first thing we'll do is to create an anonymous block to test our logic. An anonymous block, unlike a definer rights program, runs with roles enabled.

```
declare
l_module varchar2(48);
begin
select b.module into l_module
from v$process a, v$session b
where a.addr = b.paddr
and b.audsid = sys_context('userenv','sessionid');
dbms_output.put_line('Current Program is ' || l_module);
end;
```

```
Current Program is SQL*Plus
PL/SQL procedure successfully completed.
```

Place the same logic into a function:

```
create or replace function get_my_program
return varchar2
as
l_module varchar2(48);
begin
select b.module into l_module
from v$process a, v$session b
where a.addr = b.paddr
and b.audsid = sys_context('userenv','sessionid');
return l_module;
end;
```

```
6/3 PL/SQL: SQL Statement ignored
7/23 PL/SQL: ORA-00942: table or view does not exist
```

The function fails to compile because the DBA role is disabled. As such, the V$ views are not accessible inside the (named) PL/SQL program.

Section 22: REF Cursors

Lesson 22:1 Overview of REF Cursors

In PL/SQL, a pointer has the data type REF X, where REF is short for REFERENCE and X stands for a class of objects. A cursor variable has the REF CURSOR data type. Like a cursor, a cursor variable points to the current row in the result set of a multirow query. However, cursors differ from cursor variables the way constants differ from variables. A cursor is static, but a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query.

Using Cursor Variables

- You use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the result set is stored.
- A query work area remains accessible as long as any cursor variable points to it. Therefore, you can pass the value of a cursor variable freely from one scope to another.

Lesson 22.2: Strong and weak cursors

A ***REF CURSOR*** is basically a data type. A variable created based on such a data type is generally called a cursor variable. A cursor variable can be associated with different queries at run-time. The primary advantage of using cursor variables is their capability to pass result sets between sub programs (like stored procedures, functions, packages etc.). `With the REF_CURSOR you can return a recordset/cursor from a procedure or function.`

There are 2 basic types:

Strong ref cursor

A strong ref cursor the returning columns with datatype and length need to be known at compile time.

Weak ref cursor

A weak ref cursor the structure does not need to be known at compile time.

Declaring a SYS_REFCURSOR Cursor Variable

The following is the syntax for declaring a **SYS_REFCURSOR** cursor variable:

```
name SYS_REFCURSOR;
```

name is an identifier assigned to the cursor variable.

The following is an example of a **SYS_REFCURSOR** variable declaration.

```
DECLARE
rc_emp SYS_REFCURSOR;
```

Opening a Cursor Variable

Once a cursor variable is declared, it must be opened with an associated SELECT command. The OPEN FOR statement specifies the SELECT command to be used to create the result set.

```
Syntax:
OPEN name FOR query;
```

name is the identifier of a previously declared cursor variable. Query is a SELECT command that determines the result set when the statement is executed. The value of the cursor variable after the OPEN FOR statement is executed identifies the result set.

Declaring a User Defined REF CURSOR Type Variable

You must perform two distinct declaration steps in order to use a user defined REF CURSOR variable:

Create a referenced cursor TYPE

Declare the actual cursor variable based on that TYPE

The syntax for creating a user defined REF CURSOR type is as follows:

Syntax:

```
TYPE cursor_type_name IS REF CURSOR [RETURN return_type];
```

The following is an example of a cursor variable declaration.

```
DECLARE
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    rc_emp emp_cur_type;
```

Closing a Cursor Variable

Unlike static cursors, a cursor variable does not have to be closed before it can be re-opened again. The result set from the previous open will be lost. The example is completed with the addition of the CLOSE statement.

```
Syntax:
CLOSE cursor_name;
```

Lesson 22.3: Creating REF Cursors

Strongly Typed

```
CREATE OR REPLACEPACKAGE strongly_typed
IS
  TYPEreturn_cur
IS
  REF CURSORRETURNall_tables%ROWTYPE;
  PROCEDUREchild(p_return_rec OUT return_cur);
PROCEDURE parent(
    p_NumRecs PLS_INTEGER);
END strongly_typed;
/
PACKAGE Body
CREATE OR REPLACEPACKAGEBODY strongly_typed
IS
  PROCEDUREchild(p_return_rec OUT return_cur)
IS
BEGIN
  OPEN p_return_rec FOR SELECT * FROM all_tables;
END child;

PROCEDURE parent(
    p_NumRecs PLS_INTEGER)
IS
  p_retcur return_cur;
  at_rec all_tables%ROWTYPE;
BEGIN
  child(p_retcur);
  FOR i IN 1 .. p_NumRecs
  LOOP
    FETCH p_retcur INTO at_rec;
    dbms_output.put_line(at_rec.table_name     ||     '     -     '     ||
at_rec.tablespace_name || ' - ' || TO_CHAR(at_rec.initial_extent) || ' - '
|| TO_CHAR(at_rec.next_extent));
    ENDLOOP;
  END parent;
END strongly_typed;
```

To Run the Demo:

```
set serveroutput on

 exec strongly_typed.parent(1)
 exec strongly_typed.parent(8)
```

Weakly Typed

Note: A REF CURSOR that does not specify the return type such as SYS_REFCURSOR.

```
CREATE OR REPLACE PROCEDUREchild ( p_NumRecs INPLS_INTEGER, p_return_cur
OUT SYS_REFCURSOR)
IS
BEGIN
  OPEN p_return_cur FOR 'SELECT * FROM all_tables WHERE rownum <= ' ||
p_NumRecs ;
END child;
CREATE OR REPLACEPROCEDURE parent (pNumRecs VARCHAR2)
IS
  p_retcur SYS_REFCURSOR;
  at_rec all_tables%ROWTYPE;
BEGIN
  child(pNumRecs, p_retcur);
  FOR i IN 1 .. pNumRecs
  LOOP
    FETCH p_retcur INTO at_rec;
    dbms_output.put_line(at_rec.table_name    ||    '    -    '    ||
at_rec.tablespace_name || ' - ' || TO_CHAR(at_rec.initial_extent) || ' - '
|| TO_CHAR(at_rec.next_extent));
    ENDLOOP;
  END parent;
```

To Run the Demo:

```
set serveroutput on

 exec parent(1)
 exec parent(17)
```

Passing Ref Cursors

```
CREATETABLE employees ( empid NUMBER(5), empname VARCHAR2(30));
INSERTINTO employees (empid, empname) VALUES
(
  1, 'Dan Morgan'
)
;
INSERTINTO employees (empid, empname) VALUES
(
  2, 'Hans Forbrich'
)
;
INSERTINTO employees (empid, empname) VALUES
(
  3, 'Caleb Small'
)
;
COMMIT;
CREATE OR REPLACEPROCEDURE pass_ref_cur(p_cursor SYS_REFCURSOR)
IS
TYPE array_t ISTABLE OF VARCHAR2(4000) INDEX BY BINARY_INTEGER;
rec_array array_t;
BEGIN
  FETCH p_cursor BULK COLLECT INTO rec_array;
  FOR i IN rec_array.FIRST .. rec_array.LAST
  LOOP
    dbms_output.put_line(rec_array(i));
    ENDLOOP;
  END pass_ref_cur;
  /
```

```
  SET serveroutput ON
  DECLARE
    rec_array SYS_REFCURSOR;
  BEGIN
    OPEN rec_array FOR 'SELECT empname FROM employees';
    pass_ref_cur(rec_array);
    CLOSE rec_array;
  END;
```

Section 23: Introduction to SQL Tuning

In this section you will cover the following topics:

- How a SQL statement is executed
- How to gather and interpret execution plans
- Different access paths
- Different join methods and join orders
- Join types
- Autotrace
- Overview of hints

Lesson 23.1: Overview of SQL Statement execution

There are two (three if it is a SELECT) steps that every SQL statement has to go through before you see the results from the statements. All SQL statements must first of all be PARSED. Then, they can be EXECUTED any number of times and if it is a SELECT statement, the results need to be FETCHED for each execution.

Parse

The main aim of the parse phase is to generate an execution plan. To be able to do this a number of steps have to be followed:

1. **Syntax check**
   This step checks if the syntax of the statement is correct.
2. **Semantic check**
   A statement might be invalid even if the syntax is correct. One of the tables or a column referenced may not exist or the user trying to execute the query does not have the necessary object privileges. These checks are carried out on our behalf through something called Recursive SQL – this is SQL that is written by the user SYS.
3. **Private SQL Area**
   Each session that issues an SQL statement has a private SQL area associated with this statement. The first step for Oracle when it executes an SQL statement is to establish a run time area (within the private SQL area) for the statement.

4. **Library Cache**
   If the statement is syntactically and semantically correct, it is placed into the library cache (which is part of the Shared Pool).
5. **Opening the cursor**
   A cursor (an area of memory) for the statement is opened. The statement is hashed and compared with the hashed values in the library cache area. If it is already in the library cache area then a soft parse occurs otherwise, it's a hard parse.
   - Soft parse – it reuses the execution plan already in the library cache
   - Hard Parse - the statement undergoes the following steps:
     - View merging : If the query contains views, the query might be rewritten to join the view's base tables instead of the views.
     - Statement Transformation: Transforms complex statements into simpler ones through sub-query unnesting or in/or transformations
     - Optimization: The CBO uses statistics to minimize the cost to execute the query. The result of the optimization is the evaluation plan. If bind variable are used in the statement then their value is checked. The execution plan is stored in the cursor

Execute

Memory for bind variables is allocated and filled with the actual bind-values and the execution plan is executed.

Oracle checks if the data it needs for the query are already in the buffer cache. If not, it reads the data off the disk into the buffer cache.

If the statement is DML then the row(s) that are changed are locked. No other session will be able to change the row whilst it is being updated. Before and after images describing the changes are written to the redo log buffer and the rollback segments. The original block receives a pointer to the rollback segment. Then, the data is changed.

Fetch (for SELECTs only)

The data is fetched from database blocks. Rows that don't match the predicate are removed. If needed (for example in an order by statement), the data is sorted. The data is then returned to the application.

The Optimizer

The optimizer determines the most efficient way to execute a SQL statement after considering many factors related to the objects referenced and the conditions specified in the query. Oracle uses cost based optimization. You can influence the optimizer's choices by setting the optimizer mode, and by gathering representative statistics for the CBO.

Oracle has two optimizer modes:

- ALL_ROWS: has a goal of best throughput and is aimed at batch processing or report writing applications
- FIRST_ROWS_n: has a goal of best response time to return the first n rows (where n = 1, 10, 100, 1000). This approach is useful for user interfaces type applications.

These modes can be set at instance, session or statement level.

Oracle also has two types of statistics that are usually gathered automatically by jobs running overnight:

- Object statistics: information on objects such as tables, indexes etc
- System statistics: information on CPU and I/O usage

These statistics are used within the optimizer to come up with the best execution plan. Statistics are gathered via the DBMS_STATS package.

Cost, Cardinality and Selectivity

The cost represents the units of work or resource used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work. The lower the cost the better the execution plan

The selectivity is the estimated proportion of rows in the row set that the query selects, with 0 meaning no rows and 1 meaning all rows. Selectivity is tied to a query condition or a combination of conditions. A condition becomes more selective as the selectivity value approaches 0 and less selective as the value approaches 1.

Selectivity = number of rows satisfying condition/total number of rows

The cardinality is the estimated number of rows returned by each operation in an execution plan. Cardinality can be derived from the table statistics.

Cardinality = Total number of rows * Selectivity

For example, let us assume we have 110 rows in our EMPLOYEES table and the number of distinct values in the JOB_ID column is 10 and in the DEPARTMENT_ID column is 20

```
SELECT                                                             *
FROM customers
WHERE job_id = 'AD_PRES';
```

Then the estimated selectivity => 1/10 = 0.1

and the estimated cardinality => (1/10) * 110 = 11

With the following statement

```
SELECT                                                             *
FROM customers
WHERE job_id  IN ('AD_PRES', 'HR_REP');
```

Then the estimated selectivity => 2/10 = 0.2

and the estimated cardinality => (2/10) * 110 = 22

With the following statement

```
SELECT                                                                         *
FROM customers
WHERE job_id  = ‘AD_PRES’
AND department_id = 20;
```

Then the estimated selectivity => (1/10) * (1/20) = 0.005

and the estimated cardinality => ((1/10) * (1/20)) * 110 = 0.55

Lesson 23.2: Execution Plans

An execution plan shows the detailed steps necessary to execute a SQL statement. These steps are expressed as a set of database operators that consume and produce rows. The order of the operators and their implementations is decided by the query optimizer using a combination of query transformations and physical optimization techniques. While the display is commonly shown in a tabular format, the plan is in fact tree-shaped.

Displaying the Execution plan

The two most common methods used to display the execution plan of a SQL statement are:

1. EXPLAIN PLAN command
2. V$SQL_PLAN

Using the EXPLAIN PLAN command and the DBMS_XPLAN.DISPLAY function

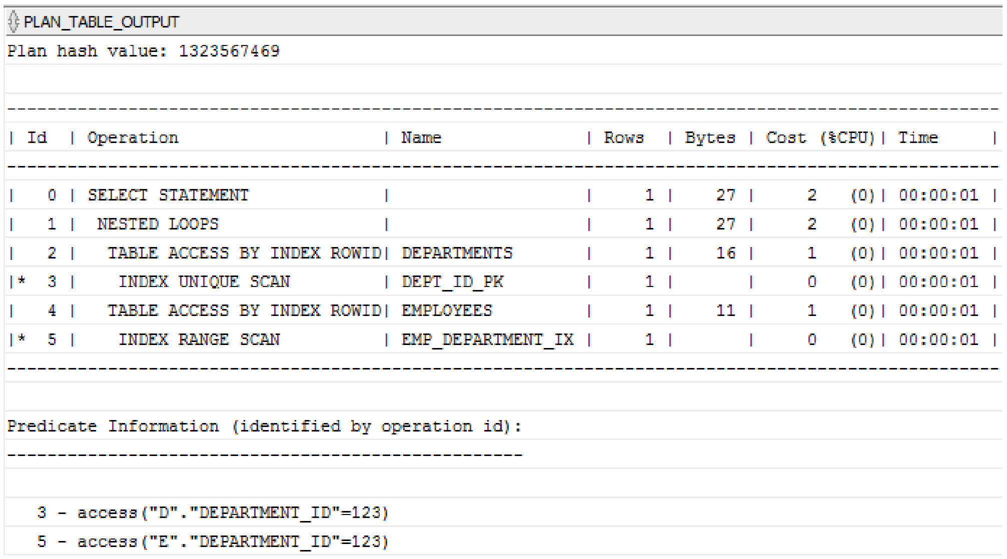
The EXPLAIN PLAN command calculates the execution plan for a SQL statement without actually executing the statement – the theoretical plan - **and outputs its result into a table called PLAN_TABLE. PLAN_TABLE is automatically created as a global temporary table and is visible to all users.**

Syntax:

```
EXPLAIN PLAN
FOR
SELECT…;

SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY());

SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY(null,null,'ALL'));
```

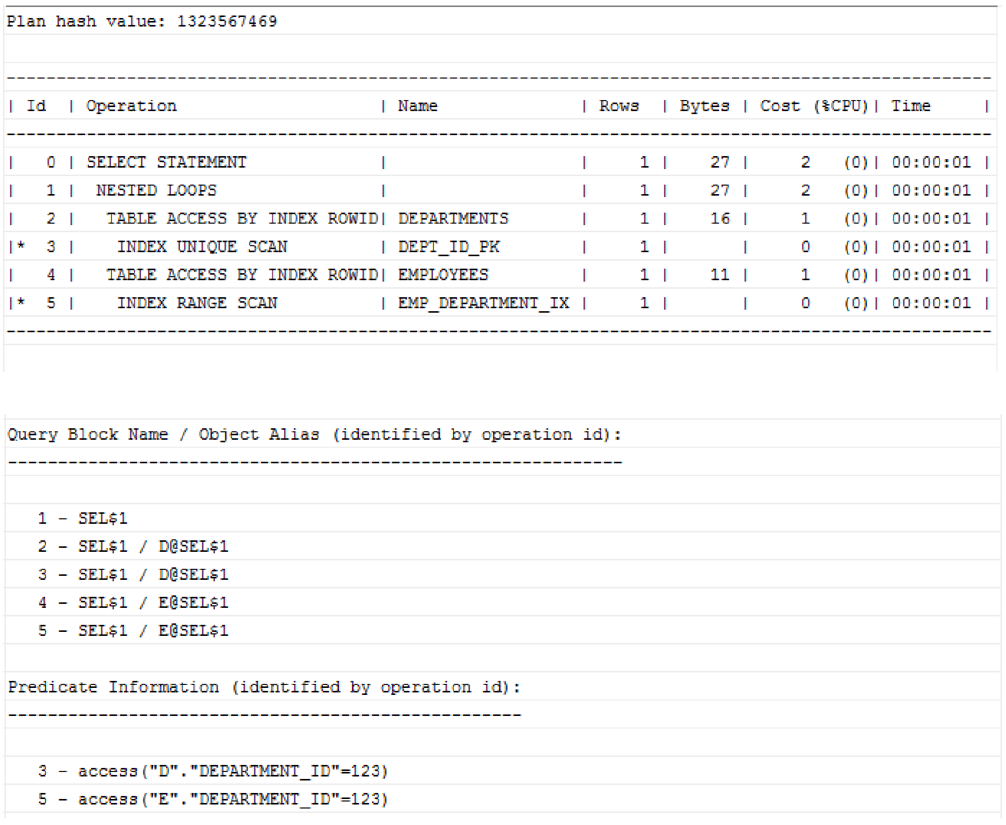
SQL Example:

```
EXPLAIN PLAN
FOR
SELECT e.last_name, d.department_name
FROM employees e, departments d
WHERE e.department_id = d.department_id
AND e.department_id = 123;

SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY());
```

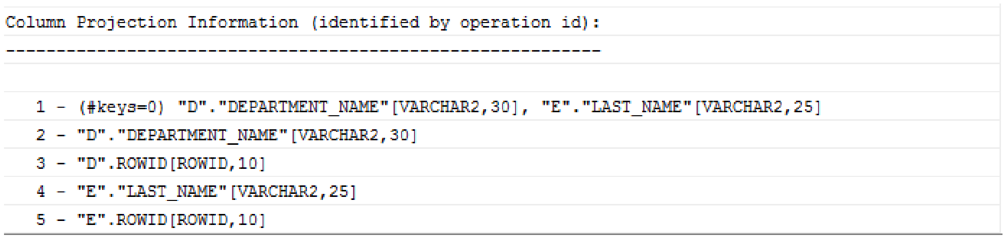


```
PLAN_TABLE_OUTPUT
Plan hash value: 1323567469

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |     1 |    27 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                   |     1 |    27 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |     1 |    16 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |     1 |       |     0   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |     1 |    11 |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("D"."DEPARTMENT_ID"=123)
   5 - access("E"."DEPARTMENT_ID"=123)
```

```
SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY(null,null,'ALL'));
```



```
Plan hash value: 1323567469

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |     1 |    27 |     2   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                   |     1 |    27 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS       |     1 |    16 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | DEPT_ID_PK        |     1 |       |     0   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |     1 |    11 |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   2 - SEL$1 / D@SEL$1
   3 - SEL$1 / D@SEL$1
   4 - SEL$1 / E@SEL$1
   5 - SEL$1 / E@SEL$1

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("D"."DEPARTMENT_ID"=123)
   5 - access("E"."DEPARTMENT_ID"=123)
```



```
Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=0) "D"."DEPARTMENT_NAME"[VARCHAR2,30], "E"."LAST_NAME"[VARCHAR2,25]
   2 - "D"."DEPARTMENT_NAME"[VARCHAR2,30]
   3 - "D".ROWID[ROWID,10]
   4 - "E"."LAST_NAME"[VARCHAR2,25]
   5 - "E".ROWID[ROWID,10]
```

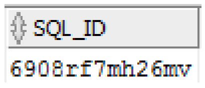
Using V$SQL_PLAN and the DBMS_XPLAN.DISPLAY_CURSOR function

V$SQL_PLAN provides a way of examining the execution plan for cursors that are still in the library cache.

V$SQL_PLAN is a dynamic performance view that has a similar structure to PLAN_TABLE. To query V$SQLPLAN you need to know the SQL_ID for the statement you are interested in.

You can get the SQL_ID from V$SQL e.g.

```
SELECT sql_id
FROM v$sql
WHERE sql_text LIKE '%AND e.department_id = 123';
```



```
SQL_ID
6908rf7mh26mv
```

This can then be entered into the DBMS_XPLAN.DISPLAY_CURSOR to obtain the actual execution plan as follows:

```
SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR('6908rf7mh26mv'));
```

Using the hint GATHER_PLAN_STATISTICS along with the DBMS_XPLAN.DISPLAY_CURSOR function with the ALLSTATS format option you can see how the optimizer's estimates compare with what really happens. By comparing the A-Rows column (actual rows) with the E-Rows column (estimated rows) you should be able to tell where the optimizer is having problems. Where there is a big difference then there is a potential problem. Unfortunately it is not quite as straightforward as that as the A-Rows are cumulative while the E-Rows are not. So you will have to multiply the E-Row by Starts (or divide A-Rows by the number of executions) in order to compare like with like.

```
SELECT /*+ GATHER_PLAN_STATISTICS */ EMPLOYEE_ID, last_name,department_id
FROM employees
WHERE department_id= 10;

SELECT plan_table_output
FROM table(DBMS_XPLAN.DISPLAY_CURSOR (FORMAT=>'ALLSTATS LAST'));
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------

SQL_ID  a1m35brp2h61g, child number 0
-------------------------------------
SELECT /*+ GATHER_PLAN_STATISTICS */ employee_id,
last_name,department_id FROM employees WHERE department_id= 10

Plan hash value: 2056577954

------------------------------------------------------------------------------------------
| Id  | Operation                   | Name              | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                   |      1 |        |      5 |00:00:00.01 |       5 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |      1 |      5 |      5 |00:00:00.01 |       5 |
|*  2 |   INDEX RANGE SCAN          | EMP_DEPARTMENT_IX |      1 |      5 |      5 |00:00:00.01 |       2 |
------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DEPARTMENT_ID"=10)
```

Understanding the execution plan

Execution plan output is a tabular representation of a set of row sources. Each step (line in the execution plan) represents a row source. DBMS_XPLAN utility indents nodes to indicate that they are the children of the parent above it. The order of the nodes under the parent indicates the order of execution of the nodes within that level. SQL execution plans are interpreted using a pre-order traversal (reverse transversal) algorithm. That is:

1. Look for the inner-most indented statement. That is ***usually*** the first statement executed but NOT always!

2. In most cases, if there are two statements at the same level, the first statement is executed first.

Execution plans are read inside-out.

Execution Plan Example – 1

```
SELECT *
FROM   emp
WHERE  empno = 7839;

0           SELECT STATEMENT ()
1       0     TABLE ACCESS BY INDEX ROWID          EMP
2       1         INDEX UNIQUE SCAN                EMP_EMPNO_PK
```

The order of operations is 2,1

Execution Plan Example – 2

```
SELECT e.ename,d.dname
FROM   emp e, dept d
WHERE  e.deptno = d.deptno;


0           SELECT STATEMENT ()
1     0      HASH JOIN
2     1           TABLE ACCESS FULL         DEPT
3     1           TABLE ACCESS FULL         EMP
```

The order of operations is 2,3,1

Execution Plan Example – 3

```
SELECT e.ename,d.dname
FROM   emp e, dept d
WHERE  e.deptno = d.deptno
```

```
AND    e.empno = 7839;


0          SELECT STATEMENT ()
1     0      NESTED LOOPS
2     1          TABLE ACCESS BY INDEX ROWID      EMP
3     2                 INDEX  UNIQUE SCAN        EMP_EMPNO_PK
4     1          TABLE ACCESS BY INDEX ROWID      DEPT
5     2                 INDEX  UNIQUE SCAN        DEPT_DEPTNO_PK
```

The order of operations is 3,5,2,4,1

Execution Plan Example – 4

```
SELECT e.ename,d.dname,c.bonus_amount
FROM   emp e, dept d, bonus c
WHERE  e.deptno = d.deptno
AND   e.empno = c.empno;

0       SELECT STATEMENT
1      0    TABLE ACCESS BY INDEX ROWID EMP
2      1          NESTED LOOPS
3      2                MERGE JOIN CARTESIAN
4      3                      TABLE ACCESS FULL                 DEPT
5      3                      BUFFER SORT
6      5                            TABLE ACCESS FULL           BONUS
7      2                INDEX RANGE SCAN                        EMP_EMPNO_PK
```

The order of operations is 4,6,5,3,7,2,1

Execution Plan Example – 5

```
   0       SELECT STATEMENT
   1     0   UNION-ALL
   2     1     SORT (GROUP BY)
   3     2       FILTER
   4     3         HASH JOIN
   5     4           INDEX  FAST FULL SCAN   IDX1
   6     4           INDEX  RANGE SCAN       IDX2
   7     1     SORT (GROUP BY)
   8     7       FILTER
   9     8         NESTED LOOPS
  10     9           HASH JOIN
  11   10             INDEX FAST FULL SCAN  IDX3
  12   10             INDEX FAST FULL SCAN  IDX1
  13    9           INDEX RANGE SCAN        IDX2
```

The order of operations is 5, 6, 4, 3, 2, 11, 12, 10, 13, 9, 8, 7, 1

In order to determine if you are looking at a good execution plan or not, you need to understand how the Optimizer determined the plan in the first place. You should also be able to look at the execution plan and assess if the Optimizer has made any mistake in its estimations or calculations, leading to a suboptimal plan. The components to assess are:

- Cardinality– Estimate of the number of rows coming out of each of the operations.
- Access method – The way in which the data is being accessed, via either a table scan or index access.
- Join method – The method (e.g., hash, sort-merge, etc.) used to join tables with each other.
- Join type – The type of join (e.g., outer, anti, semi, etc.).
- Join order – The order in which the tables are joined to each other.

Lesson 23.3: Access Paths

The access method - or access path - shows how the data will be accessed from each table (or index). The access method is shown in the operation field of the explain plan.

Oracle supports a number of common access methods:

1. Full table scan - Reads all rows from a table and filters out those that do not meet the where clause predicates. A full table scan will use multi block IO (typically 1MB IOs). A full table scan is selected if a large portion of the rows in the table must be accessed, no indexes exist or the ones present cannot be used or if the cost is the lowest.

2. Table access by ROWID - Oracle first obtains the ROWIDs either from a WHERE clause predicate or through an index scan of one or more of the table's indexes. Oracle then locates each selected row in the table based on its ROWID and does a row-by-row access.

3. Index unique scan – Only one row will be returned from the scan of a unique index. It will be used when there is an equality predicate on a unique (B-tree) index or an index created as a result of a primary key constraint.

4. Index range scan – Oracle accesses adjacent index entries and then uses the ROWID values in the index to retrieve the corresponding rows from the table. An index range scan can be bounded or unbounded. It will be used when a statement has an equality predicate on a non-unique index key, or a non-equality or range predicate on a unique index key. (=, <, >, LIKE). Data is returned in the ascending order of index columns.

5. Index range scan descending – Conceptually the same access as an index range scan, but it is used when an ORDER BY .. DESC clause can be satisfied by an index.

6. Index skip scan - Normally, in order for an index to be used, the leading column of the index would be referenced in the query. However, if all the other columns in the index are referenced in the statement except the first column, Oracle can do an index skip scan, to skip the first column of the index and use the rest of it. This can be advantageous if there are few distinct values in the leading column of a concatenated index and many distinct values in the non-leading key of the index.

7. Full Index scan - A full index scan does not read every block in the index structure, contrary to what its name suggests. An index full scan processes all of the leaf blocks of an index, but only enough of the branch blocks to find the first leaf block. It is used when all of the columns necessary to satisfy the statement are in the index and it is cheaper than scanning the table. It uses single block IOs. It may be used in any of the following situations:

    - An ORDER BY clause has all of the index columns in it and the order is the same as in the index (can also contain a subset of the columns in the index).
    - The query requires a sort merge join and all of the columns referenced in the query are in the index.

- Order of the columns referenced in the query matches the order of the leading index columns.
- A GROUP BY clause is present in the query, and the columns in the GROUP BY clause are present in the index.

8. Fast full index scan - This is an alternative to a full table scan when the index contains all the columns that are needed for the query, and at least one column in the index key has the NOT NULL constraint. It cannot be used to eliminate a sort operation, because the data access does not follow the index key. It will also read all of the blocks in the index using multi-block reads, unlike a full index scan.

9. Index join – This is a join of several indexes on the same table that collectively contain all of the columns that are referenced in the query from that table. If an index join is used, then no table access is needed, because all the relevant column values can be retrieved from the joined indexes. An index join cannot be used to eliminate a sort operation.

Lesson 23.4: Join methods

The join method describes how data from two data producing operators will be joined together. You can identify the join methods used in a SQL statement by looking in the operations column in the execution plan.

Nested Loops Joins

These joins are useful when small subsets of data are being joined and if there is an efficient way of accessing the second table (for example an index look up). For every row in the first table (the outer table), Oracle accesses all the rows in the second table (the inner table). Consider it like two embedded FOR loops. In Oracle Database 11g the internal implementation for nested loop joins changed to reduce overall latency for physical I/O so it is possible you will see two NESTED LOOPS joins in the operations column of the plan, where you previously only saw one on earlier versions of Oracle.

Sort Merge Joins

Sort merge joins are useful when the join condition between two tables is an inequality condition such as, <, <=, >, or >=. Sort merge joins can perform better than nested loop joins for large data sets. The join consists of two steps:

1. Sort join operation: Both the inputs are sorted on the join key.
2. Merge join operation: The sorted lists are merged together.

A sort merge join is more likely to be chosen if there is an index on one of the tables that will eliminate one of the sorts.

Hash Joins

Hash joins are used for joining large data sets. The optimizer uses the smaller of the two tables or data sources to build a hash table, based on the join key, in memory. It then scans the larger table and performs the same hashing algorithm on the join column(s). It then probes the previously built hash table for each value and if they match, it returns a row.

Cartesian Join

With a Cartesian join the optimizer joins every row from one data source with every row from the other data source, creating a Cartesian product of the two sets. Typically, this is only chosen if the tables involved are small or if one or more of the tables does not have a join conditions to any other table in the statement.

Lesson 23.5: Join Types

Oracle offers several join types: inner join, (left) outer join, full outer join, anti join, semi join, etc. As an inner join is the most common type of join, the execution plan does not specify the key word “INNER’. For all other types of joins specific keywords are used.

Outer Join - An outer join returns all rows that satisfy the join condition and also all of the rows from the table without the (+) for which no rows from the other table satisfy the join condition.

Anti Join - An anti-join between two tables returns rows from the first table where no matches are found in the second table. An anti-join is essentially the opposite of a semi-join. While a semi-join returns one copy of each row in the first table for which at least one match is found, an anti-join returns one copy of each row in the first table for which no match is found. Anti-joins are written using the NOT EXISTS or NOT IN syntax.

Semi Join – A semi-join between two tables returns the rows from the first table where one or more matches are found in the second table. The difference between a semi-join and a conventional join is that rows in the first table will be returned at most once. Even if the second table contains two matches for a row in the first table, only one copy of the row will be returned. Semi-joins are written using the EXISTS or IN syntax.

Lesson 23.6: Join Order

The join order is the order in which the tables are joined together in a multi-table SQL statement. To determine the join order in an execution plan, look at the indentation of the tables in the operation column.

In a more complex SQL statement, it may not be so easy to determine the join order by looking at the indentations of the tables in the operations column. In these cases, it might be easier to use the FORMAT parameter in the DBMS_XPLAN procedures to display the outline information for the plan, which will contain the join order.

Lesson 23.7: Autotrace

Autotrace is a SQLPlus command although a simplified version of it is available in SQL Developer. It allows you to see execution plans and also some statistics for a statement. To be able to run AUTOTRACE you need to be granted the PLUSTRACE role.

Autotrace has a number of options that can be set:

SET AUTOTRACE ON – Enables all options.

SET AUTOTRACE ON EXPLAIN – Displays returned rows and the explain plan.

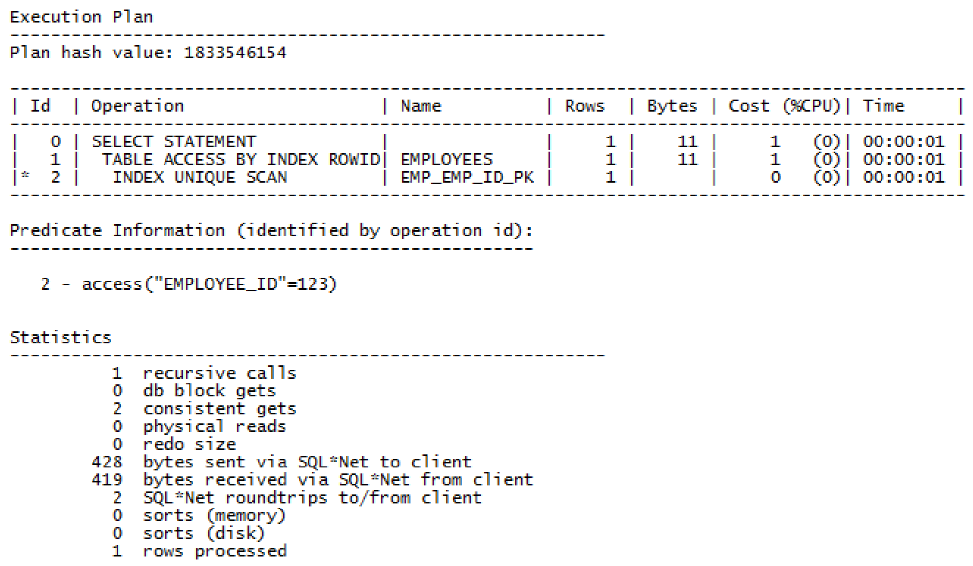
SET AUTOTRACE ON STATISTICS – Displays returned rows and statistics.

SET AUTOTRACE TRACE EXPLAIN – Displays the execution plan for a select statement without actually executing it.

SET AUTOTRACE TRACEONLY – Displays execution plan and statistics without displaying the returned rows. This option should be used when a large result set is expected.

SQL Example:

```
SET AUTOTRACE TRACEONLY
SELECT last_name
FROM employees
WHERE employee_id = 123;
```



```
Execution Plan
----------------------------------------------------------
Plan hash value: 1833546154

---------------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               |     1 |    11 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     1 |    11 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |     1 |       |     0   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPLOYEE_ID"=123)

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          2  consistent gets
          0  physical reads
          0  redo size
        428  bytes sent via SQL*Net to client
        419  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Explanation: In the example you can see we have used the AUTOTRACE TRACEONLY so we only see the execution plan and the statistics – we do not see all the data from the query.

The execution plan is similar in structure to those we have seen previously however the statistics are new. Here is a description of what they all mean.

Name	**Description**
recursive calls	Number of recursive calls generated at both the user and system level. Oracle maintains tables used for internal processing. When Oracle needs to make a change to these tables, it internally generates an internal SQL statement, which in turn generates a recursive call.
db block gets	Number of times a CURRENT block was requested.
consistent gets	Number of times a consistent read was requested for a block.
physical reads	Total number of data blocks read from disk. This number equals the value of "physical reads direct" plus all reads into buffer cache.
redo size	Total amount of redo generated in bytes.
bytes sent via SQL*Net to client	Total number of bytes sent to the client from the foreground processes.
bytes received via SQL*Net from client	Total number of bytes received from the client over Oracle Net.
SQL*Net roundtrips to/from client	Total number of Oracle Net messages sent to and received from the client.
sorts (memory)	Number of sort operations that were performed completely in memory and did not require any disk writes.
sorts (disk)	Number of sort operations that required at least one disk write.
rows processed	Number of rows processed during the operation.

If you add the db block gets and the consistent gets together that will give you the number of blocks read to satisfy the statement – so 2 in our example.

The number of physical reads will depend on what is currently in the Database Buffer Cache – in our example it is 0 as the data had already been requested earlier.

Code file: Code23_7.sql

Lesson 23.8: Overview of Hints

A hint is an instruction to the optimizer. There are hints for most of the operations you will see in an execution plan. There are hints that cover the following and more:

- Optimizer mode
- Query transformation
- Access path
- Join orders
- Join methods

When writing SQL, you may know information about the data unknown to the optimizer. Hints enable you to make decisions normally made by the optimizer, sometimes causing the optimizer to select a plan that it sees as higher cost.

In a development environment, hints are useful for testing the performance of a specific access path. However in the database and host environment can make hints obsolete or have negative consequences. So in 99% of cases hints should be avoided. With good statistics, you usually never need to include them in your statements.

Syntax:

```
[MERGE|INSERT|UPDATE|DELETE|SELECT] /*+ hint(s) */
or
[MERGE|INSERT|UPDATE|DELETE|SELECT] --+ hint(s)
```

SQL Example:

```
SELECT /*+ INDEX (e emp_idx) */ EMPLOYEE_ID,last_name
FROM employees e
WHERE employee_id = 123;
```

Explanation: In the example we are specifying the INDEX hint – the first parameter is the table alias and the second parameter relates to the name of the index. So we are asking the optimizer to create an execution plan that uses the EMP_IDX from the EMPLOYEES table

When including hints, you must code them immediately after the first SQL keyword ie MERGE, INSERT, UPDATE, DELETE or SELECT of the statement block. Each statement block can have only one hint comment, but that can contain multiple hints. Hints apply to only the statement block in which they appear – so if a hint appeared in a sub-query it would only affect the sub-query and not the outer query. It is important to remember that if your statement uses table aliases then your hints must reference the aliases rather than the table names. If your hint is incorrect then the optimizer will ignore it without raising errors.

Here are some common hints:

Optimization Modes:

ALL_ROWS

FIRST_ROWS(n)

Access Path Hints

FULL(table)

INDEX(table index)

NO_INDEX

INDEX_ASC (table index)

INDEX_DESC (table index)

INDEX_COMBINE (table index)

INDEX_JOIN (table index)

INDEX_FFS (table index)

INDEX_SS (table index)

INDEX_SS_ASC (table index)

INDEX_SS_DESC (table index)

NO_INDEX_FFS (table index)

NO_INDEX_SS (table index)

Join Operation

USE_HASH (table)

NO_USE_HASH (table)

USE_MERGE (table)

NO_USE_MERGE (table)

USE_NL (table)

USE_NL_WITH_INDEX (table)

NO_USE_NL (table)

Join Order

ORDERED

LEADING (table)

Other

APPEND

NOAPPEND

CACHE

NOCACHE

CARDINALITY

CURSOR_SHARING_EXACT

DRIVING_SITE

DYNAMIC_SAMPLING

Section 24: Unit Testing

Lesson 24:1 What is Unit Testing

Unit testing refers to the practice of testing certain functions and areas – or units – of our code. This gives us the ability to verify that our functions work as expected. That is to say that for any function and given a set of inputs, we can determine if the function is returning the proper values and will gracefully handle failures during the course of execution should invalid input be provided.

A second advantage to approaching development from a unit testing perspective is that you'll likely be writing code that is easy to test. Since unit testing requires that your code be easily testable, it means that your code must support this particular type of evaluation. As such, you're more likely to have a higher number of smaller, more focused functions that provide a single operation on a set of data rather than large functions performing a number of different operations.

A third advantage for writing solid unit tests and well-tested code is that you can prevent future changes from breaking functionality. Since you're testing your code as you introduce your functionality, you're going to begin developing a suite of test cases that can be run each time you work on your logic. When a failure happens, you know that you have something to address.

As PL/SQL developers, you should see unit tests as part of the development cycle, part of the set of deliverables. Write your unit tests while writing the PL/SQL code and store the tests with the program units. You should be abbe able to run and rerun tests at any point to verify the code still works as required and desired.

Why unit test

Suppose you want to ensure that a block of PL/SQL code is working properly, but don’t want to take the time to write a unit test. Wrap the code in DBMS_OUTPUT statements that display or print the results of intermediate and final computations and the results of complex conditional steps and branches. This will enable you to see the computations and the results of complex conditional steps and branches. The following example demonstrates this tactic for placing comments into strategic locations within a PL/SQL code block in order to help determine if code is functioning as expected.

Syntax: Function syntax

SQL Example:

```
CREATE OR REPLACE FUNCTION factorial (fact INTEGER) RETURN
INTEGER is
BEGIN
IF fact < 0 THEN
  DBMS_OUTPUT.Put_LINE('Value is < 0');
  RETURN NULL;
```

```
ELSIF fact = 0 THEN
  DBMS_OUTPUT.Put_LINE('Value is = 0');
  RETURN 1;
ELSIF fact = 1 THEN
  DBMS_OUTPUT.Put_LINE('Value is = 1');
  RETURN fact;
ELSE
  DBMS_OUTPUT.Put_LINE('Value is >=1');
  RETURN fact * factorial (fact-1);
END IF;
END factorial;

DECLARE
  v_Val NUMBER;
BEGIN
  SELECT FACTORIAL(0) INTO v_Val FROM dual;
END;
```

Query Results:

```
The value =0
```

Explanation: The following example demonstrates this tactic for placing comments into strategic locations within a PL/SQL code block in order to help determine if code is functioning as expected.

Code file: Code24_1_1.sql

The use of DBMS_OUTPUT statements within PL/SQL code for displaying data or information pertaining to the functionality of the code has been a great tactic for testing code in any language. In order to use DBMS_OUTPUT statements for testing your code, you must place them in strategic locations. In the example, comments have been placed within each of the IF-ELSE blocks to display a bit of text that will tell the developer how the values are being processed within the function. Although using DBMS_OUTPUT statements in code can be very useful for determining where code is functioning properly, it can cause clutter, and can also create its own issues as DBMS_OUTPUT is removed before code is released into production. This can take some time, which could be better spent on development. As a means for testing small units of code, using DBMS_OUTPUT statements works quite well. However, if you wish to develop entire test suites and automate it better to develop entire test suites and automated unit using a testing tool.

Lesson 24:2 Tools for building Unit Tests

There are a limited number of tools on the market for writing PL/SQL unit tests; as a result, many users have written their own testing mechanisms.

utPLSQL

utPLSQL is an open source PL/SQL testing framework for building unit tests. You install utPLSQL by running a script that installs all necessary tables, packages, procedures, and other objects required for the tests. You create and build all your tests in SQL*Plus or from the command line, so there is no tool or additional client required. utPLSQL is hosted on Sourceforge, http://utplsql.sourceforge.net, and has a wide variety of resources, documentation, and examples on how to install the software, build tests, and make good use of the framework. The utPLSQL unit-testing framework can alleviate some of the pain of unit testing. The framework is easy to use and performs nicely for testing code. We will focus on utPLSQL on this course.

Quest Code Tester for Oracle

Quest Code Tester for Oracle is a commercial product for defining and running tests. You can build tests for single programs or packages, and you can build individual tests or suites. The great advantage you gain by using a product like this is that you can build tests and then rerun them whenever needed, thus supporting the argument that you need to build up a full regression suite of tests for your project.

Oracle SQL Developer

Oracle SQL Developer is a free product that supports PL/SQL unit testing. Oracle SQL Developer unit testing was first introduced in SQL Developer 2.0 and already provides much of what the other tools on the market offer; supported by an Oracle Repository, it allows users to build and save tests.

Installation of utPL/SQL

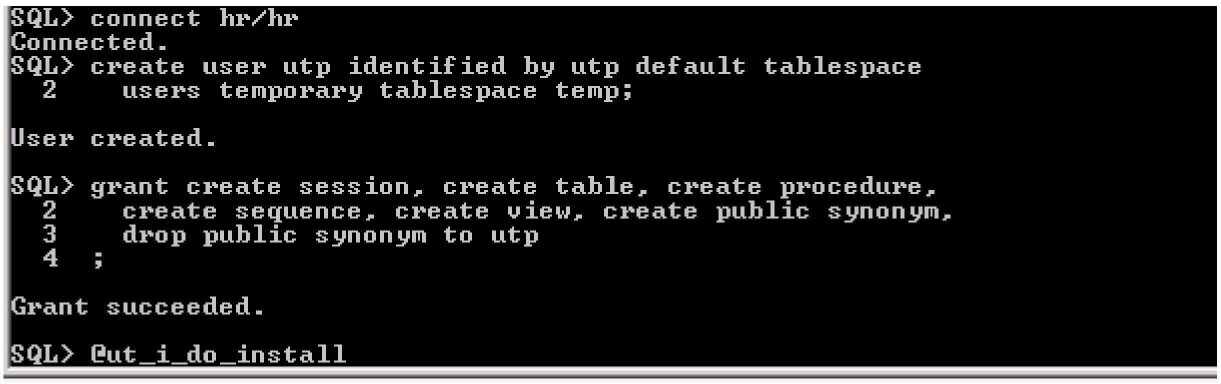
First, download the utPLSQL sources from http://utplsql.sourceforge.net/. Once you have obtained the sources, use the following steps to install the utPLSQL package into the database for which you wish to write unit tests, and make it available for all schemas. Create a user to host the utPLSQL tables, packages, and other objects. In this example, the user will be named UTP, and the default permanent and temporary tablespaces will be used.

```
create user utp identified by abc123;
```

Grant privileges to the newly created UTP user using the GRANT privilege_name TO user_name statement, replacing values with the appropriate privilege and username. The user will require the following privileges:

- Create session
- Create procedure
- Create table
- Create view
- Create sequence
- Create public synonym
- Drop public synonym

Install the objects by running the ut_i_do.sql script. Once these steps have been completed then you will have the ability to run unit tests on packages that are loaded into different schemas within the database.



```
SQL> connect hr/hr
Connected.
SQL> create user utp identified by utp default tablespace
  2    users temporary tablespace temp;

User created.

SQL> grant create session, create table, create procedure,
  2    create sequence, create view, create public synonym,
  3    drop public synonym to utp
  4  ;

Grant succeeded.

SQL> @ut_i_do_install
```

Before you can begin to write and run unit tests within the utPLSQL framework for the PL/SQL contained within your database, you must install the utPLSQL package into a database schema. While the utPLSQL framework can be loaded into the SYSTEM schema, it is better to separate the framework into its own schema by creating a separate user and installing the packages, tables, and other objects into it

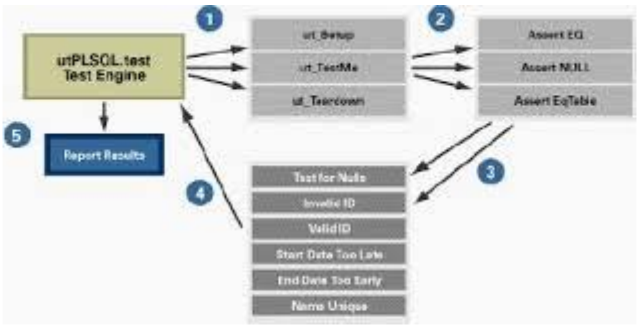
Once you have created a user schema in which to install the utPLSQL framework objects, you must grant it the appropriate privileges. The majority of the privileges are used to create the objects that are required to make the framework functional. Public synonyms are created for many of the framework objects, and this allows them to be accessible to other database user accounts. After all privileges have been granted, running the ut_i_do.sql script and passing the install parameter will complete the installation of the framework. After completion, you can begin to build unit test packages and install them into different schemas within the database, depending on which PL/SQL objects that you wish to test.

Note

Unit tests will be executed from the same schema in which the PL/SQL object that is being tested resides, not from the schema that contains the utPLSQL framework objects.

Lesson 23:3 Building Unit Tests

Structure of Unit Test



A test package consists of two separate files,

a package header

a package body

Create a header for the test package and save it in a file with the same name you have given the header and with a .pks suffix. A header file contains three procedures:

ut_setup,

ut_teardown,

and the procedure that performs the unit tests of the target object in your database.

The package must also contain an implementation for your unit test procedures. The unit test procedure names should begin with the ut_ prefix followed by the name of the PL/SQL object that you are testing.

For example, suppose you want to create a unit test package to test the code for the factorial function. This package header should be stored into a file named ut_factorial.pks and loaded into the database whose objects you are testing.

Syntax: `CREATE OR REPLACE PACKAGE….`

SQL Example:

```
CREATE OR REPLACE PACKAGE ut_ factorial
IS
PROCEDURE ut_setup;
PROCEDURE ut_teardown;
PROCEDURE ut_ factorial;
END ut_calc_quarter_hour;


CREATE OR REPLACE PACKAGE BODY ut_factorial
IS
  PROCEDURE ut_setup
  IS
  BEGIN
    NULL;
  END;
  PROCEDURE ut_teardown
  IS
  BEGIN
    NULL;
  END;
  PROCEDURE ut_factorial
  IS
  BEGIN
    -- Perform unit tests here
    NULL;
  END ut_factorial;
END ut_factorial;
```

Query Results:

Explanation: This package header should be stored into a file named ut_ factorial.pks and loaded into the database whose objects you are testing.Create the package body that implements the procedures specified by the unit test package header and save it as a file with the same name as the header, but this time with a .pkb suffix. The following package body should be stored into a file named factorial.pkb and loaded into the database. The package body in this example conforms to the format that must be used for testing packages using the utPLSQL framework.

Code file: Code24_1_2.sql

Note: The .pks and .pkb suffixes could be changed to something different, like .sql, if you wish. You could also store both the package header and body in the same file. However, utPLSQL framework will look for the .pks and .pkb suffixes in order to automatically recompile your test packages before each test. It is best to follow the utPLSQL convention to ensure that your test packages are always valid.

Writing a unit test

You have a PL/SQL object that you'd like to test to verify it returns the expected values. Create a utPLSQL test package to test every code branch and computation within your function. Use utPLSQL assertion statements to test every foreseeable use case for the function. For example, suppose you wish to test a simple factorial function that contains four code branches, each of which returns a value. Here's the target function:

Step 1 Target function or procedure

```
CREATE OR REPLACE FUNCTION factorial(
    fact INTEGER)
  RETURN INTEGER
IS
BEGIN
  IF fact < 0 THEN
    RETURN NULL;
  ELSIF fact = 0 THEN
    RETURN 1;
  ELSIF fact = 1 THEN
    RETURN fact;
  ELSE
    RETURN fact * factorial (fact-1);
  END IF;
END factorial;
```

Step 2 Create unit test package

Next, create the unit test package to test the factorial function. Name the package using the same name as the function to be tested and adding the prefix ut_ to it In this example, you'll name the package ut_factorial. Create the three required procedures within the package for setup, teardown, and testing. Remember to save the file as a PKS file (i.e., one with a .pks file extension).

```
CREATE OR REPLACE PACKAGE ut_factorial
IS
  PROCEDURE ut_setup;
  PROCEDURE ut_teardown;
  PROCEDURE ut_factorial;
END ut_factorial;
```

Step 3 Create unit test package body

Create the unit testing package body. No code is required for the ut_setup or the ut_teardown procedures as these are usually reserved for code that updates the database prior to or after running the tests. For example, the setup procedure may insert records that are required only by the unit test, which means that the teardown routine must clean up any data the test leaves behind. The ut_factorial procedure is built with a series of assert statements that test each code branch in the factorial function.

```
CREATE OR REPLACE PACKAGE BODY ut_factorial
IS
  PROCEDURE ut_setup
  IS
  BEGIN
    NULL;
  END ut_setup;
  PROCEDURE ut_teardown
  IS
  BEGIN
    NULL;
  END ut_teardown;
  PROCEDURE ut_factorial
  IS
  BEGIN
    utAssert.isnull ('is NULL test', factorial(-1));
    utAssert.eqQuery ('0! Test', 'select factorial(0) from dual', 'select 1
from dual');
    utAssert.eqQuery ('1! Test', 'select factorial(1) from dual', 'select 1
from dual');
    utAssert.eqQuery ('N! Test', 'select FACTORIAL(5) from dual', 'select
120 from dual');
  END ut_factorial;
END ut_factorial;
```

The utPLSQL package contains a number of tests that can be used to ensure that your code is working properly. Each of these tests is an *assertion*, which is a statement that evaluates to either true or false depending on whether its conditions are met. The solution to this recipe uses four tests to determine whether the function returns an appropriate result for each scenario. The utAssert.isnull procedure verifies the second parameter returns a null value when executed. The utAssert.eqQuery procedure uses the select statements in parameter positions two and three to determine if the unit test succeeds or fails. Each select statement must return the same value when executed to succeed. The three calls to utAssert.eqQuery procedure in the ut_factorial procedure tests one branch (if statement) within the factorial function. The expected return value from the factorial is used in the select statement of the third parameter to retrieve the value from dual. If the factorial is updated in such a way that any code branch no longer returns the expected value, the unit test will fail. This test should be performed after modifying the factorial function to test for bugs introduced by the update.

Lesson 23:4 Startup and Teardown Processes

How It Works

A unit test package for the utPLSQL framework consists of a package header and a body. The package header declares a setup procedure, a teardown procedure, and a unit testing procedure. The package body consists of the PL/SQL code that implements the unit test. When you create a ut_PLSQL package, its name must be prefixed with ut_, followed by the procedure or function name for which you are writing the unit test. The unit test prefix can be changed, but ut_ is the default.

The test package body must contain both a setup and teardown procedure. These procedures must also be given names that use the same prefix you have chosen for your unit testing. The package header declares ut_setup and ut_teardown procedures. The ut_setup procedure is to initialize the variables or data structures the unit test procedure uses. When a unit test is executed, ut_setup is always the first procedure to execute. The ut_teardown procedure is used to clean up after all of the tests have been run. You should use this procedure to destroy all of the data structures and variables created to support your unit tests. The ut_teardown procedure is always executed last, after all unit tests have been run.

For example, the ut_setup could add a record to the database and the teardown would roll it back

```
PROCEDURE ut_setup
  AS
  BEGIN
INSERT
INTO EMPLOYEES
  (EMPLOYEE_ID,FIRST_NAME,LAST_NAME,EMAIL,PHONE_NUMBER,HIRE_DATE,JOB_ID,
SALARY,COMMISSION_PCT,MANAGER_ID,DEPARTMENT_ID)
  VALUES(199,'Bob','Joine','bob@mail.ie','012 45896',
  '25-Feb-2014','IT_PROG',20000,.3,100,90);
```

```
END;

  PROCEDURE ut_teardown
  AS
  BEGIN
  rollback;
  END;
```

Lesson 23:5 Assertion Tests

The table below lists the different assertion tests that are part of the utAssert package.

Assertion Name	**Description**
utAssert.eq	Checks equality of scalar values
utAssert.eq_refc_query	Checks equality of RefCursor and Query
utAssert.eq_refc_table	Checks equality of RefCursor and Database Tables
utAssert.eqcoll	Checks equality of collections
utAssert.eqcollapi	Checks equality of collections
utAssert.eqfile	Checks equality of files
utAssert.eqoutput	Checks equality of DBMS_OUTPUT values
utAssert.eqpipe	Checks equality of database pipes
utAssert.eqquery	Checks equality of different queries
utAssert.eqqueryvalue	Checks equality of query against a value
utAssert.eqtabcount	Checks equality of table counts
utAssert.eqtable	Checks equality of different database tables
UTASSERT.isnotnull	Checks for NOT NULL values
utAssert.isnull	Checks for NULL values
utAssert.objexists	Checks for the existence of database objects
utAssert.objnotexists	Checks for the existence of database objects
utAssert.previous_failed	Checks if the previous assertion failed
utAssert.previous_passed	Checks if the previous assertion passed
utAssert.this	Generic “this” procedure
utAssert.throws	Checks if a procedure or function throws an exception

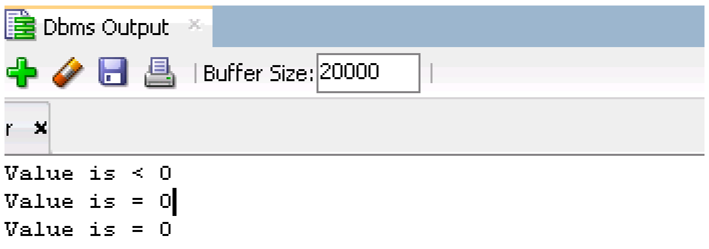
Lesson 24:6 Running Test

With a unit test package defined, you want to run it to verify that a function returns the values you expect under a variety of scenarios.

Use the utPLSQL.test procedure to run your test package. For example, suppose you want to run the unit test you built for factorial

```
Begin
 utPLSQL.test('factorial', recompile_in => FALSE);
end
```

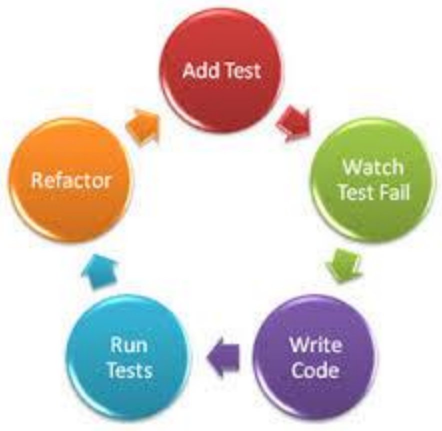
The utPLSQL framework makes it easy to execute all of the tests that you have setup within a unit test package; you need only to enter a utPLSQL.test command. The call to theutPLSQL.test procedure passes two parameters, the first is the name of the unit test to run. Notice thatyou do not specify the name of the package built for the unit test. Instead, you pass the name of the function being tested. The second parameter tells the utPLSQL.test procedure not to recompile any of the code before running the test.



```
Value is < 0
Value is = 0
Value is = 0
```

Lesson 24:7 Test driven development

Test Driven Development (TDD) is a software engineering methodology that states that all code should have a set of clear, repeatable unit tests written for it. Furthermore, it states that these tests should be written before the corresponding code is written, thus driving the development of the code base. The ultimate goal of TDD is to produce code that meets the exact need expressed by the test and providing for a safety net in the event that the code base needs to be modified later. TDD is one aspect of a much larger development methodology known as Agile Development.



Unit Testing refers to what you are testing, TDD to when you are testing.

Unit Testing means, well, testing individual units of behavior. An individual unit of behavior is the smallest possible unit of behavior that can be individually tested in isolation. You can write unit tests before you write your code, after you write your code or while you write your code. TDD means letting your tests drive your development (and your design). You can do that with unit tests, functional tests and acceptance tests. The most important part of TDD is the middle D. You let the tests drive you. The tests tell you what to do, what to do next, when you are done.

Writing a Test Driven Development procedure or function – example 1

For this example, write a function that shows the commission each employee in the company made last year.

```
SELECT first_name, Last_name, Salary, COMMISSION_PCT
FROM employees;
```

FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT
Trenna	Rajs	3500	
Curtis	Davies	3100	
Randall	Matos	2600	
Peter	Vargas	2500	
John	Russell	14000	0.4
Karen	Partners	13500	0.3
Alberto	Errazuriz	12000	0.3
…			

Step 1: Write test that does not pass.

```
CREATE OR REPLACE PACKAGE BODY ut_TEST_calc_comm AS

PROCEDURE ut_calc_comm IS
-- we write our test logic in the package body
BEGIN
  -- our call to assert equals is taking 3 parameters.
  -- The first is the value we expect, the second is the actual value-
--(our function result)
  -- the third is the optional error message to supply if the test
fails.
  utAssert.eqQuery(400, calc_comm(1000, .4), 'Commission is wrong');
END;
  PROCEDURE ut_setup
  AS
  BEGIN
  return;
  END;
  PROCEDURE ut_teardown
  AS
  BEGIN
  return;
  END;
END;
```

Step 2: Write the target function that fails the test

```
CREATE OR REPLACE FUNCTION calc_comm(p_salary IN number, p_comm IN number)
RETURN number IS

BEGIN

  RETURN 0;

END;
```

Step 3: Write the target function that passes the test

Now our task is to make the test pass. Switch back to the function and write the code to calculate percentage correctly:

```
CREATE OR REPLACE FUNCTION calc_comm(p_salary IN number, p_comm IN number)
RETURN number IS
```

```
BEGIN
  RETURN p_comm * p_salary;
END;
```

Step 4: Repeat the process for all test cases

```
CREATE OR REPLACE PACKAGE ut_TEST_calc_comm
IS
  -- unit tests are public procedures that have no parameters
  PROCEDURE ut_setup;
  PROCEDURE ut_teardown;
  PROCEDURE ut_calc_comm;
  PROCEDURE t_zero_salary;
  PROCEDURE t_zero_comm;
  PROCEDURE t_zero_both;
END ;


CREATE OR REPLACE PACKAGE BODY ut_TEST_calc_comm AS

PROCEDURE ut_calc_comm IS
-- we write our test logic in the package body
BEGIN  utAssert.eqQuery(400, calc_comm(1000, .4), 'Commission is wrong');
END;
  PROCEDURE ut_setup
  AS
  BEGIN

INSERT
INTO EMPLOYEES
  (
    EMPLOYEE_ID,
    FIRST_NAME,
    LAST_NAME,
    EMAIL,
    PHONE_NUMBER,
    HIRE_DATE,
    JOB_ID,
    SALARY,
    COMMISSION_PCT,
    MANAGER_ID,
    DEPARTMENT_ID
  )
  VALUES
  (
    199,
    'Bob',
    'Joine',
    'bob@mail.ie',
    '012 45896',
    '25-Feb-2014',
    'IT_PROG',
    20000,
    .3,
    100,
    90
  );
END;
  PROCEDURE ut_teardown
  AS
  BEGIN
  rollback;
  END;


PROCEDURE t_zero_salary IS
BEGIN
 utAssert.eqQuery(1, calc_comm(0, .1), 'Commission  is wrong');
END;

PROCEDURE t_zero_comm IS
BEGIN
  -- with no commission, the amount should be zero
  utAssert.eqQuery(0, calc_comm(1000, 0), 'Commission percent is wrong');
END;


PROCEDURE t_zero_both IS
BEGIN
  -- with no commission and no salary, the amount should be zero
  utAssert.eqQuery(0, calc_comm(0, 0), 'Commission percent is wrong');
END;

END;

CREATE OR REPLACE FUNCTION calc_comm(
    p_salary IN NUMBER,
    p_comm   IN NUMBER)
  RETURN NUMBER
IS
BEGIN
```

```
 IF p_comm = 0 THEN
    RETURN 0;
  ELSE
    RETURN p_comm*p_salary;
  END IF;

END;
```

Writing a Test-Driven Development procedure or function –example 2

For this example, write a function that checks whether a password is considered strong according to the following five criteria:

- contains at least one-digit character (0-9)
- contains at least one lowercase character (a-z)
- contains at least one uppercase character (A-Z)
- contains at least one special character (@#$%)
- length between 6 and 20 characters

Step 1 – Write the test cases

Define what are valid parameters values and invalid parameters for example the ValidatePassword function to return true if ‘ABCdef123#’ is passed as a parameter. Passing Abcdef# to the the ValidatePassword function should return FALSE. The more cases you define, the higher the reliability will be of your unit test.

Syntax:

```
CREATE OR REPLACE PACKAGE….
```

SQL Example:

```
create or replace
package ut_UserLogin as

  procedure ut_setup;
  procedure ut_teardown;

  procedure ut_ValidatePassword;

end ut_UserLogin;

create or replace
package body ut_UserLogin as

  procedure ut_setup as
  begin
```

```
    null;
  end ut_setup;

  procedure ut_teardown as
  begin
    null;
  end ut_teardown;

  procedure ut_ValidatePassword as
  begin
    utassert.eq(
      msg_in => 'ABCdef123# is a strong password',
      check_this_in                                    =>
UserLogin.ValidatePassword('ABCdef123#'),
      against_this_in => true
    );
    utassert.eq(
      msg_in => '%a1B2CD is a strong password',
      check_this_in                                    =>
UserLogin.ValidatePassword('%a1B2CD'),
      against_this_in => true
    );
    utassert.eq(
      msg_in          => 'Abcde1@ is a strong password',
      check_this_in                                    =>
UserLogin.ValidatePassword('Abcde1@'),
      against_this_in => true
    );

    utassert.eq(
      msg_in           =>   'Abcdef#   misses   a   digit
character',
      check_this_in                                    =>
UserLogin.ValidatePassword('Abcdef#'),
      against_this_in => false
    );
    utassert.eq(
      msg_in          =>  'ABCD1234$  misses  a  lowercase
character',
      check_this_in                                    =>
UserLogin.ValidatePassword('ABCD1234$'),
      against_this_in => false
    );
    utassert.eq(
      msg_in          => 'abcd1234@ misses an uppercase
character',
      check_this_in                                    =>
UserLogin.ValidatePassword('abcd1234@'),
      against_this_in => false
    );
    utassert.eq(
      msg_in          =>  'ABcd1234  misses  a  special
character',
      check_this_in                                    =>
UserLogin.ValidatePassword('ABcd1234'),
      against_this_in => false
    );
    utassert.eq(
      msg_in         => 'Abc1% is too short',
      check_this_in                                    =>
UserLogin.ValidatePassword('Abc1%'),
      against_this_in => false
    );
    utassert.eq(
      msg_in          =>  'Abcdefghijk123456789@  is  too
long',
      check_this_in                                    =>
UserLogin.ValidatePassword('Abcdefghijk123456789@'),
      against_this_in => false
    );
    utassert.eq(
      msg_in          => 'An empty string should return
false',
      check_this_in   => UserLogin.ValidatePassword(''),
      against_this_in => false
    );
  end ut_ValidatePassword;

end ut_UserLogin;
```

Query Results:

Explanation: Place unit test code in a separate test package. The name of the test package equals the name of the package to be tested, prefixed with *ut_*. By following this naming convention, it is possible for utPLSQL to automatically recompile your package before each test.

A test package must contain a *ut_setup* and *ut_teardown* procedure. These two procedures offer the possibility to respectively initialize and remove temporary database objects that are available to use in your unit test procedures.

Code file: Code24_7_1.sql

Step 2 – Create the function

Syntax: `CREATE OR REPLACE PACKAGE….`

SQL Example:

```
create or replace
package body UserLogin as

  function ValidatePassword(in_password in varchar2)
  return boolean is
  begin
    if not regexp_like(in_password, '[[:digit:]]') then
      return false;
    end if;

    if not regexp_like(in_password, '[[:lower:]]') then
      return false;
    end if;

    if not regexp_like(in_password, '[[:upper:]]') then
      return false;
    end if;

    if not regexp_like(in_password, '[@#$%]') then
      return false;
    end if;

    if length(in_password) not between 6 and 20 then
      return false;
    end if;

    return true;
  end ValidatePassword;

end UserLogin;
```

Query Results:

Explanation: The function uses RegEx to verify the strength of the parameter

- contains at least one digit character (0-9)
- contains at least one lowercase character (a-z)
- contains at least one uppercase character (A-Z)
- contains at least one special character (@#$%)
- length between 6 and 20 characters

Code file: Code24_7_2.sql

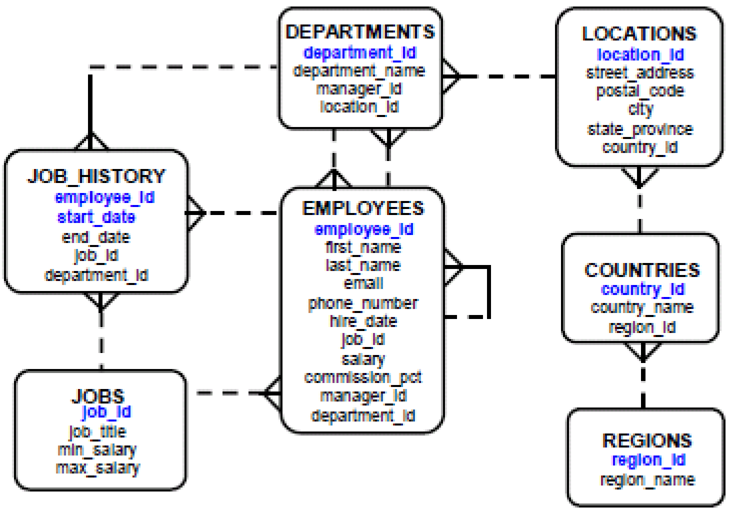
Step 3 – run the unit test

Run the unit tests for UserLogin package by invoking the utplsql.test procedure:

```
set serveroutput on;
begin
utplsql.test(
package_in   => 'UserLogin',
recompile_in => false
);
end;
```

Appendix

HR SCHEMA



Human Resources (HR) Schema for This Course

The Human Resources (HR) schema is part of the Oracle Sample Schemas that can be installed in an Oracle database. The practice sessions in this course use data from the HR schema.

Table	Description
REGIONS	Contains rows that represent a region such as the Americas or Asia.
COUNTRIES	Contains rows for countries, each of which is associated with a region.
LOCATIONS	Contains the specific address of a specific office, warehouse, or production site of a company in a particular country.
DEPARTMENTS	Shows details about the departments in which employees work. Each department may have a relationship representing the department manager in the EMPLOYEES table.
EMPLOYEES	Contains details about each employee working for a department. Some employees may not be assigned to any department.
JOBS	Contains the job types that can be held by each employee.
JOB_HISTORY	Contains the job history of the employees. If an employee changes department within a job or changes jobs within a department, a new row is inserted into this table with the old job information of the employee.