## Types of Triggers in Forms

**Block-processing triggers:** - Block processing triggers fire in response to events related to record management in a block. E.g. When-Create-Record, When-Clear-Block, When-Database-Record, When-Remove-Record

**Interface event triggers:** - Interface event triggers fire in response to events that occur in the form interface. Some of these trigger, such as When-Button-Pressed, fire only in response to operator input or manipulation. Others, like When-Window-Activated, can fire in response to both operator input and programmatic control. E.g. When-Button-Pressed, When-Checkbox-Changed, Key- [all], When-Radio-Changed, When-Timer-Expired, When-Window-Activated, When-Window-Resized

**Master-detail triggers:** - Form Builder generates master-detail triggers automatically when you define a master-detail relation between blocks. The default master-detail triggers enforce coordination between records in a detail block and the master record in a master block. Unless you are developing your own custom block-coordination scheme, you do not need to define these triggers yourself. Instead, simply create a relation object, and let Form Builder generate the triggers required to manage coordination between the master and detail blocks in the relation. E.g. On-Check-Delete-Master, On-Clear-Details, On-Populate-Details

**Message-handling triggers:** - Form Builder automatically issues appropriate error and informational messages in response to runtime events. Message handling triggers fire in response to these default messaging events. E.g. On-Error, On-Message

**Navigational triggers**: - Navigational triggers fire in response to navigational events. For instance, when the operator clicks on a text item in another block, navigational events occur as Form Builder moves the input focus from the current item to the target item. Navigational events occur at different levels of the Form Builder object hierarchy (Form, Block, Record, and Item). Navigational triggers can be further sub-divided into two categories: Pre- and Post- triggers, and When-New-Instance triggers. Pre- and Post- triggers fire as Form Builder navigates internally through different levels of the object hierarchy. As you might expect, these triggers fire in response to navigation initiated by an operator, such as pressing the [Next Item] key. However, be aware that these triggers also fire in response to internal navigation that Form Builder performs during default processing. To avoid unexpected results, you must consider such internal navigation when you use these triggers. E.g. Pre-Form, Pre-Block, Pre-Text-Item, Post-Text-Item, Post-Record, Post-Block, Post-Form

When-New-Instance triggers fire at the end of a navigational sequence that places the input focus in a different item. Specifically, these triggers fire just after Form Builder moves the input focus to a different item, when the form returns to a quiet state to wait for operator input. Unlike the Pre- and Post- navigational triggers, the When-New-Instance triggers do not fire in response to internal navigational events that occur during default form processing. E.g. When-New-Form-Instance, When-New-Block-Instance, When-New-Record-Instance, When-New-Item-Instance

**Query-time triggers:** - Query-time triggers fire just before and just after the operator or the application executes a query in a block. E.g. Pre-Query, Post-Query

**Transactional triggers:** - Transactional triggers fire in response to a wide variety of events that occur as a form interacts with the data source. E.g. On-Delete, On-Insert, On-Lock, On-Logon, On-Update, Post-Database-Commit, Post-Delete, Post-Forms-Commit, Post-Insert, Post-Update, Pre-Commit, Pre-Delete, Pre-Insert, Pre-Update

**Validation triggers:** - Validation triggers fire when Form Builder validates data in an item or record. Form Builder performs validation checks during navigation that occurs in response to operator input, programmatic control, or default processing, such as a Commit operation. E.g. When-Validate-Item, When-Validate-Record

## Sequence of Trigger Fire while Committing

- KEY Commit
- Pre Commit
- Pre/On/Post Delete
- Pre/On/Post Update
- Pre/On/Post Insert
- On commit
- Post Database Commit

## Master-Detail Relation (Triggers/Procedures/Properties)

**On-Check-Delete-Master**: - Fires when Form Builder attempts to delete a record in a block that is a master block in a master-detail relation.
**On-Clear-Details**: - Fires when Form Builder needs to clear records in a block that is a detail block in a master-detail relation because those records no longer Correspond to the current record in the master block.
**On-Populate-Details**: - Fires when Form Builder needs to fetch records into a block that is the detail block in a master-detail relation so that detail records are synchronized with the current record in the master block.

**(i) Isolated: - Masters Can be deleted when Child is existing**

| | | |
|---|---|---|
| Triggers: - | On Populate details | Block |
| | On Clear Details | Form |
| Procedure | | |

Check Package Failure
Clear all master Detail
Query Master Detail

**(ii) Non- Isolated: - Masters Cannot be deleted when Child is existing.**

| | |
|---|---|
| Triggers: - On Populate details | Block |
| On Check Delete master | Block |
| On Clear Details | Form |

Procedure

Check Package Failure
Clear all master Detail
Query Master Detail

**(iii) Cascading: - Child Record Automatically Deleted when Masters is deleted.**

| | |
|---|---|
| Triggers: - On Populate details | Block |
| Pre Delete | Block |
| On Clear Details | Form |

Procedure

Check Package Failure
Clear all master Detail
Query Master Detail

## Dynamically create LOV/List Item

You can also add list elements individually at runtime by using the ADD_LIST_ELEMENT built-in subprogram, or you can populate the list from a record group at runtime using the POPULATE_LIST built-in. If you populate the list from a record group, be sure that the record group you are using to populate the list contains the relevant values before you call POPULATE_LIST. If the record group is a static record group, it will already contain the appropriate values. Otherwise, you should populate the group at runtime using one of the record group subprograms.

## Object Libraries (Use/Benefits)

The Object Library provides an easy method of reusing objects and enforcing standards across the entire development organization.
Object Library can be used to:

Create, store, maintain, and distribute standard and reusable objects.
Rapidly create applications by dragging and dropping predefined objects to your form.

There are several advantages to using object libraries to develop applications:

Object libraries are automatically re-opened when you startup Form Builder, making your reusable objects immediately accessible.

You can associate multiple object libraries with an application. For example, you can create an object library specifically for corporate standards, and you can create an object library to satisfy project-specific requirements.

Object libraries feature Smart Classes-- objects that you define as being the standard. You use Smart Classes to convert objects to standard objects.

## Key-next/Post-Text (Difference)

Post-Text–Item: Fires during the Leave the Item process for a text item. Specifically, this trigger fires when the input focus moves from a text item to any other item.

Key-Next-Item: The key-next is fired as a result of the key action. Key next will not fire unless there is a key event.

## Call From/New Form/Open Form (Difference)

**Call Form**: Runs an indicated form while keeping the parent form active. Form Builder runs the called form with the same Runform preferences as the parent form. When the called form is exited Form Builder processing resumes in the calling form at the point from which you initiated the call to CALL_FORM.
**PROCEDURE CALL_FORM (formmodule_name VARCHAR2, display NUMBER, switch_menu NUMBER, query_mode NUMBER, data_mode NUMBER, paramlist_name/id VARCHAR2);**

**New Form**: Exits the current form and enters the indicated form. The calling form is terminated as the parent form. If the calling form had been called by a higher form, Form Builder keeps the higher call active and treats it as a call to the new form. Form Builder releases memory (such as database cursors) that the terminated form was using.
Form Builder runs the new form with the same Runform options as the parent form. If the parent form was a called form, Form Builder runs the new form with the same options as the parent form.
**PROCEDURE NEW_FORM (formmodule_name VARCHAR2, rollback_mode NUMBER, query_mode NUMBER, data_mode NUMBER, paramlist_name/id VARCHAR2);**

**Open Form**: Opens the indicated form. Use OPEN_FORM to create multiple-form applications, that is, applications that open more than one form at the same time.
PROCEDURE OPEN_FORM (form_name VARCHAR2, activate_mode NUMBER, session_mode NUMBER, data_mode NUMBER, paramlist_id/name PARAMLIST);

## Types of Canvases (Stacked/Content Difference)

**Content Canvas** (Default Canvas) [A content canvas is the required on each window you create]

**Stack Canvas** [you can display more then one stack canvas in a window at the same time]

**Tab Type Window** [In Tab canvas that have tab pages and have one or more then tab page]

**Toolbar Canvas** [A toolbar canvas often is used to create Toolbar Windows. There are two type of Toolbar window.

**Horizontal Toolbar Canvas**: - Horizontal Toolbar canvases are displayed at the top of the window, just under the Main Menu Bar.

**Vertical Toolbar Canvas:** - While vertical Toolbar are displayed along the Left Edge of the window.

## Object Groups (Use)

An object group is a container for a group of objects. You define an object group when you want to package related objects so you can copy or subclass them in another module.
Object groups provide a way to bundle objects into higher-level building blocks that can be used in other parts of an application and in subsequent development projects.
**For example**, you might build an appointment scheduler in a form and then decide to make it available from other forms in your applications. The scheduler would probably be built from several types of objects, including a window and canvas, blocks, and items that display dates and appointments, and triggers that contain the logic for scheduling and other functionality. If you packaged these objects into an object group, you could then copy them to any number of other forms in one simple operation.
You can create object groups in form and menu modules. Once you create an object group, you can add and remove objects to it as desired.

## Various Block Co-ordination Properties

The various Block Coordination Properties are
a) Immediate Default Setting. The Detail records are shown when the Master Record are shown.
b) Deffered with Auto Query
Oracle Forms defer fetching the detail records until the operator navigates to the detail block.
c) Deferred with No Auto Query
The operator must navigate to the detail block and explicitly execute a query

## How to attach same LOV to multiple items

We can use the same LOV for 2 columns by passing the return values in global values and using the global values in the code.

## Report Level Triggers (Sequence)

Before parameter form
After parameter form
Before Report
Between Pages
After Report

### Static & Dynamic LOV

The static LOV contains the predetermined values while the dynamic LOV contains values that come at run time

### Format Triggers (What are they)

A format trigger is a PL/SQL function executed before an object is formatted. A trigger can be used to dynamically change the formatting attributes of the object.

### Flex & Confine Mode in Reports

**Confine mode**:

Switched on by default; change via View® View Options® Layout...

It prevents operations, which would cause a report not to work e.g. moving a field outside its parent-repeating frame

**Flex mode**:

Moves the object its enclosing objects and objects in their push path simultaneously to maintain the same overall relationship in the report. E.g. if you try to move a field outside its repeating frame, the Repeating Frame will grow to accommodate the field and so will any objects around the repeating frame.

Only one object can be moved/resized at one time in flex mode - if you try more than one only one whose control point is clicked on will be done, the other objects will be de-selected.

Objects can be moved /resized horizontally or vertically; not diagonally.

## Matrix Reports (Matrix By Groups)

A matrix (cross tab) report contains one row of labels, one column of labels, and information in a grid format that is related to the row and column labels. A distinguishing feature of matrix reports is that the number of columns is not known until the data is fetched from the database.
To create a matrix report, you need at least **four groups**: one group must be a cross-product group, two of the groups must be within the cross-product group to furnish the "labels," and at least one group must provide the information to fill the cells. The groups can belong to a single query or to multiple queries.

A matrix with group report is a group above report with a separate matrix for each value of the master group.

A nested matrix (cross tab) report is a matrix report in which at least one parent/child relationship appears within the matrix grid.
The new Child Dimension property of the nested group enables you to eliminate empty rows and/or columns in your single-query nested matrix.

**Types of Matrix Reports**
**Simple Matrix Report**: Is a matrix with only two dimensions
**Nested Matrix Report**: Has multiple dimensions going across and/or down the page
**Multi-Query Matrix with Break**: Is similar to a nested matrix report in that it has more than two dimensions. Does not display records that do not contain data
**Matrix Break Reports**: Contains a new matrix for each master record

## Lexical & Bind Parameters in Reports

**Lexical Parameters**: Lexical references are placeholders for text that you embed in a SELECT statement. You can use lexical references to replace the clauses appearing after SELECT, FROM, WHERE, GROUP BY, ORDER BY, HAVING, CONNECT BY, and START WITH.
You cannot make lexical references in a PL/SQL statement. You can, however, use a bind reference in PL/SQL to set the value of a parameter that is then referenced lexically in SQL. Look at the example below.

You create a lexical reference by entering an ampersand (&) followed immediately by the column or parameter name. A default definition is not provided for lexical references. Therefore, you must do the following:

- ➢ Before you create your query, define a column or parameter in the data model for each lexical reference in the query. For columns, you must enter Value if Null, and, for parameters, you must enter Initial Value. Report Builder uses these values to validate a query with a lexical reference.
- ➢ Create your query containing lexical references.

**Bind Parameters**: Bind references (or bind variables) are used to replace a single value in SQL or PL/SQL, such as a character string, number, or date. Specifically, bind references may be used to replace expressions in SELECT, WHERE, GROUP BY, ORDER BY, HAVING, CONNECT BY, and START WITH clauses of queries. Bind references may not be referenced in FROM clauses or in place of reserved words or clauses.
You create a bind reference by entering a colon (:) followed immediately by the column or parameter name. If you do not create a column or parameter before making a bind reference to it in a SELECT statement, Report Builder will create a parameter for you by default.

## Column Mode Property in Reports

The Column Mode property controls how Report Builder fetches and formats data for instances of repeating frames. With Column Mode set to **Yes**, the next instance of a repeating frame can begin formatting before the previous instance is completed. With Column Mode set to **No**, the next instance cannot begin formatting before the previous instance is completed. Column Mode is used mainly for master repeating frames or repeating frames that contain fields that may expand vertically or horizontally (i.e., elasticity is Variable or Expand).

## Diff b/w Package Spec & Body

**Packages provide a method of encapsulating and storing related procedures, funtions and other package constructs as a unit in the database. They offer increased functionality (for example, global package variables can be declared and used by any procedure in the package). They also improve performance (for example, all objects of the package are parsed, compiled, and loaded into memory once).**
**Package specification contains declarations of public constructs where as the package body contains definitions of all those public constructs and declarations & definitions of private constructs.**

## P/L SQL Tables / Arrays

**PL/SQL tables are declared in the declaration portion of the block. <u>A table is a composite data type in PL/SQL. PL/SQL tables can have one column and a primary key neither of which can be named.</u> The column can be any scalar type but primary key should be a BINARY_INTEGER data type.**
**Rules for PL/SQL Tables:**
**A loop must be used to insert values into a PL/SQL Table**
**You cannot use the Delete command to delete the contents of PL/SQL Table. You must assign an empty table to the PL/SQL table being deleted.**

## Various Cursor Attributes

**SQL%ROWCOUNT: Number of rows affected by most recent SQL statement.**
**SQL%FOUND: Boolean attribute that evaluates to TRUE if most recent SQL statement affects one or more rows.**
**SQL%NOTFOUND: Boolean attribute that evaluates to TRUE if most recent SQL statement does not affect any row.**
**SQL%ISOPEN: Always evaluates to FALSE because P/L SQL closes implicit cursors immediately after they are executed.**

## Different Database Triggers

**Database triggers are PL/SQL, Java, or C procedures that run implicitly whenever a table or view is modified or when some user actions or database system actions occur. Database triggers can be used in a variety of ways for managing your database. For example, they can automate data generation, audit data modifications, enforce complex integrity constraints, and customize complex security authorizations.**

### Row Triggers

**A row trigger is fired each time the table is affected by the triggering statement.**
**For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.**
**If a triggering statement affects no rows, a row trigger is not executed at all.**
**Row triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected.**

**Statement Triggers**

**A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects (even if no rows are affected).**

**For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once, regardless of how many rows are deleted from the table.**

**Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected.**

**For example, if a trigger makes a complex security check on the current time or user, or if a trigger generates a single audit record based on the type of triggering statement, a statement trigger is used.**

**BEFORE vs. AFTER Triggers**

**When defining a trigger, you can specify the trigger timing.**

**That is, you can specify whether the trigger action is to be executed before or after the triggering statement.**

**BEFORE and AFTER apply to both statement and row triggers.**

**BEFORE Triggers BEFORE triggers, execute the trigger action before the triggering statement. This type of trigger is commonly used in the following situations:**

**BEFORE triggers are used when the trigger action should determine whether the triggering statement should be allowed to complete. By using a BEFORE trigger for this purpose, you can eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the trigger action.**

**BEFORE triggers are used to derive specific column values before completing a triggering INSERT or UPDATE statement**.

**AFTER triggers execute the trigger action after the triggering statement is executed. AFTER triggers are used in the following situations:**

**AFTER triggers are used when you want the triggering statement to complete before executing the trigger action.**

**If a BEFORE trigger is already present, an AFTER trigger can perform different actions on the same triggering statement**.

**Combinations**

Using the options listed in the previous two sections, you can create four types of triggers:

**BEFORE statement trigger** Before executing the triggering statement, the trigger action is executed.
**BEFORE row trigger** Before modifying each row affected by the triggering statement and before checking appropriate integrity constraints, the trigger action is executed provided that the trigger restriction was not violated.
**AFTER statement trigger** After executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.
**AFTER row trigger** After modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is executed for the current row provided the trigger restriction was not violated. Unlike BEFORE row triggers, AFTER row triggers lock rows.

**New Database Triggers**
Startup, Shutdown, Logon, Logoff, Alter, Create, Drop

## List Item Types

**Poplist:** The poplist style list item appears initially as a single field (similar to a text item field). When the end user selects the list icon, a list of available choices appears.
**Tlist**: The Tlist style list item appears as a rectangular box, which displays a fixed number of values. When the Tlist contains values that cannot be displayed (due to the displayable area of the item), a vertical scroll bar appears, allowing the end user to view and select undisplayed values.
**Combo Box:** The combo box style list item combines the features found in poplists and text items. It displays fixed values and can accept a user-entered value.

## Inline Views & Top N Analysis

**The Inline view**: It is a construct in Oracle SQL where you can place a query in the SQL FROM, clause, just as if the query was a table name.
A common use for in-line views in Oracle SQL is to simplify complex queries by removing join operations and condensing several separate queries into a single query.

**Top N Analysis**: The task of retrieving the top or bottom N rows from a database table. You can do so either by using the ROWNUM pseudocolumn available in several versions of Oracle or by utilizing new analytic functions available in Oracle 8i: RANK () and DENSE_RANK ().

**Using the ROWNUM Pseudocolumn**
One-Way to solve this problem is by using the Oracle pseudocolumn ROWNUM. For each row returned by a query, the ROWNUM pseudocolumn returns a number indicating the order in which Oracle selects the row from a table or set of joined rows.
E.g. To select top 5 rows

```
SELECT Empno, Ename, Job, Mgr, Hiredate, Sal
  FROM
  (SELECT Empno, Ename, Job, Mgr, Hiredate, Sal
    FROM Emp
    ORDER BY NVL (Sal, 0) DESC)
  WHERE ROWNUM < 6;
```

**Utilizing Oracle 8i's Ranking Functions**
Another way to perform a top-N query uses the new Oracle 8i feature called "analytic functions.

```
SELECT Empno, Ename, Job, Mgr, Sal,
  RANK () OVER
    (ORDER BY SAL Desc NULLS LAST) AS Rank,
  DENSE_RANK () OVER
    (ORDER BY SAL Desc NULLS LAST) AS Drank
  FROM Emp
  ORDER BY SAL Desc NULLS LAST;
```

The difference between RANK () and DENSE_RANK () is that RANK () leaves gaps in the ranking sequence when there are ties. In our case, Scott and Ford tie for second place with a $3,000 salary; Jones' $2,975 salary brings him in third place using DENSE_RANK () but only fourth place using RANK (). The NULLS FIRST | NULLS LAST clause determines the position of rows with NULL values in the ordered query.

## Property Class & Visual Attributes (Difference)

**A property class is a named object that contains a list of properties and their settings. Once you create a property class you can base other objects on it. An object based on a property class can inherit the setting of any property in the class that makes sense for that object. Property class inheritance is an instance of subclassing. Conceptually, you can consider a property class as a universal subclassing parent. Property classes are separate objects, and, as such, can be copied between modules as needed. Perhaps more importantly, property classes can be subclassed in any number of modules. Property class inheritance is a powerful feature that allows you to quickly define objects that conform to your own interface and functionality standards. Property classes also allow you to make global changes to applications quickly. By simply**

**changing the definition of a property class, you can change the definition of all objects that inherit properties from that class.**

**Visual attributes are the font, color, and pattern properties that you set for form and menu objects that appear in your application's interface. Visual attributes can include the following properties: Font properties, Color and pattern properties. Every interface object has a Visual Attribute Group property that determines how the object's individual visual attribute settings (Font Size, Foreground Color, etc.) are derived.**

**Named visual attributes define only font, color, and pattern attributes; property classes can contain these and any other properties.**

**You can change the appearance of objects at runtime by changing the named visual attribute programmatically; property class assignment cannot be changed programmatically.**

When an object is inheriting from both a property class and a named visual attribute, the named visual attribute settings take precedence, and any visual attribute properties in the class are ignored.

Property Class has triggers and Visual Attributes don’t have same.

## Model/Modeless Windows (Difference)

A window can be either **modeless** or **modal.** A modal window (often a dialog) requires the end user to respond before continuing to work in the current application. A modeless window requires no such response.

You can display multiple modeless windows at the same time, and end users can navigate freely among them. Modeless windows remain displayed until they are dismissed by the end user or hidden programmatically. You can set the Hide on Exit property for a modeless window to specify whether it should remain displayed when the end user navigates to another window. Modal windows are usually used as dialogs, and have restricted functionality compared to modeless windows. On some platforms, modal windows are "always-on-top" windows that cannot be layered behind modeless windows. The Hide on Exit property does not apply to modal windows. Modal dialog windows cannot have scroll bars

## Alerts Styles

An alert is a modal window that displays a message notifying the operator of some application condition. There are three styles of alerts: Stop, Caution, and Note. Each style denotes a different level of message severity. Message severity is represented visually by a unique icon that displays in the alert window.

## Normalization / De-Normalization

**Normalization**: It's the process of efficiently organizing data in a database. There are two goals of the normalization process**: eliminate redundant data (for example, storing the same data in more than one table) and ensure data dependencies make sense (only storing related data in a table).** Both of these are worthy goals as they reduce the amount of space a database consumes and ensure that data is logically stored.

1. Eliminate Repeating Groups - Make a separate table for each set of related attributes, and give each table a primary key.
2. Eliminate Redundant Data - If an attribute depends on only part of a multi-valued key, remove it to a separate table.
3. Eliminate Columns Not Dependent On Key - If attributes do not contribute to a description of the key, remove them to a separate table.
4. Isolate Independent Multiple Relationships - No table may contain two or more 1: n or n: m relationships that are not directly related.
5. Isolate Semantically Related Multiple Relationships - There may be practical constrains on information that justify separating logically related many-to-many relationships.

**1st Normal Form (1NF)**
Def: A table (relation) is in 1NF if
1. There are no duplicated rows in the table.
2. Each cell is single-valued (i.e., there are no repeating groups or arrays).
3. Entries in a column (attribute, field) are of the same kind.
Note: The order of the rows is immaterial; the order of the columns is immaterial.
Note: The requirement that there be no duplicated rows in the table means that the table has a key (although the key might be made up of more than one column—even, possibly, of all the columns).
**2nd Normal Form (2NF)**
Def: A table is in 2NF if it is in 1NF and if all non-key attributes are dependent on all of the key.
Note: Since a partial dependency occurs when a non-key attribute is dependent on only a part of the (composite) key, the definition of 2NF is sometimes phrased as, "A table is in 2NF if it is in 1NF and if it has no partial dependencies."
**3rd Normal Form (3NF)**
Def: A table is in 3NF if it is in 2NF and if it has no transitive dependencies.
**Boyce-Codd Normal Form (BCNF)**
Def: A table is in BCNF if it is in 3NF and if every determinant is a candidate key.
**4th Normal Form (4NF)**
Def: A table is in 4NF if it is in BCNF and if it has no multi-valued dependencies.
**5th Normal Form (5NF)**

Def: A table is in 5NF, also called "Projection-Join Normal Form" (PJNF), if it is in 4NF and if every join dependency in the table is a consequence of the candidate keys of the table.

**Domain-Key Normal Form (DKNF)**

Def: A table is in DKNF if every constraint on the table is a logical consequence of the definition of keys and domains.

**De-Normalization**:

Denormalization is a technique to move from higher to lower normal forms of database modeling in order to speed up database access. You may apply Denormalization in the process of deriving a physical data model from a logical form.

## Autonomous Transactions

**Autonomous Transaction is a new feature in ORACLE. It allows setting up independent transactions that can be called from within other transactions. It lets you suspend the main transaction (without committing or rolling back), perform some DML operations, commit or roll back those operations (without any effect on the main transaction), and then return to the main transaction.**

**T1 going, it stops t1 half way, do some dml then come back to t1**

**Being independent of the main transaction (almost like a separate session), an autonomous transaction does not see the uncommitted changes from the main transaction. It also does not share locks with the main transaction. As a result, it can get into a deadlock with its parent … something the application developer should watch out for.**

**As expected, changes committed by an autonomous transaction are visible to other sessions/transactions immediately, regardless of whether the main transaction is committed or not. These changes also become visible to the main transaction when it resumes, provided its isolation level is set to READ COMMITTED (which is the default).**

**Any of the routines can be marked as autonomous simply by using the following syntax anywhere in the declarative section of the routine (putting it at the top is recommended for better readability):**

**E.g.**

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

**Here is an example of defining a stored procedure as autonomous:**

```
CREATE PROCEDURE process_ord_line_shipment
     (P_order_no number, p_line_no number) AS
  PRAGMA AUTONOMOUS_TRANSACTION;
  l_char_1     varchar2 (100);
BEGIN
  ...
END;
```

## Bitmap/B-Tree Index (Difference, Advantages)

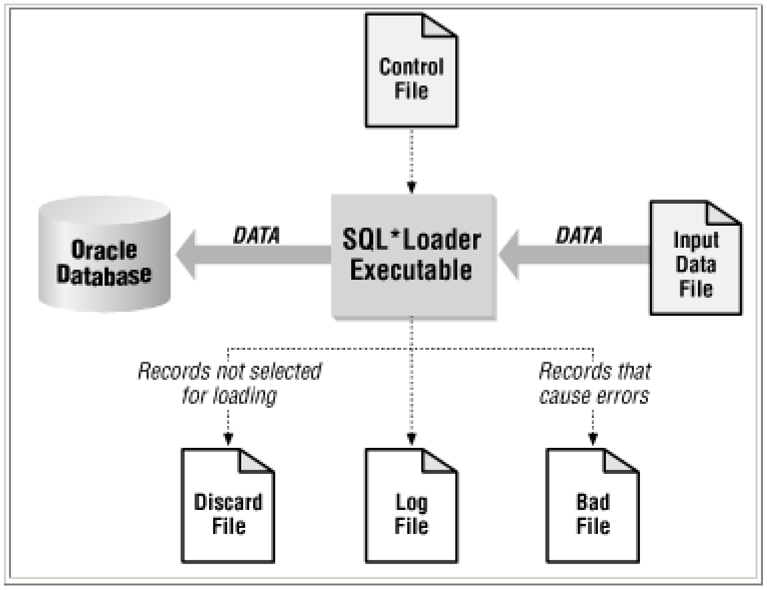
A traditional B-Tree (balanced tree) index stores the key values and pointers in an inverted tree structure. **The key to good B-Tree index performance is to build the index on columns having a lot of different values.** Oracle describes this as "good selectivity" Oracle is able to quickly bypass rows that do not meet the search criteria when searching through indexes built on columns having a high degree of selectivity.
Conversely, bitmapped indexes perform better when the selectivity of an index is poor. **The fewer different values a bitmapped index contains, the better it will perform.**
Bitmap indexes, in certain situations, can provide impressive performance benefits. **Bitmapped indexes are most appropriate for complex and ad-hoc queries that contain lengthy *WHERE* clauses on columns that have a limited number of different values (poor selectivity).**

**Standard B-tree indexes are most effective for columns containing a high number of different values (good selectivity) and bitmapped indexes are most appropriate for columns with a limited number (poor selectivity) of possible values.**

## SQL Loader (Different Types of Files/Contents)

SQL*Loader is a bulk loader utility used for moving data from external files into the Oracle database. Its syntax is similar to that of the DB2 Load utility, but comes with more options. SQL*Loader supports various load formats, selective loading, and multi-table loads.
When we speak of the SQL*Loader environment, we are referring to the database, the SQL*Loader executable, and all the different files that you need to be concerned with when using SQL*Loader. These are shown in Figure

**The SQL*Loader environment**



**The SQL*Loader Control File**

The SQL*Loader control file is the key to any load process. The control file provides the following information to SQL*Loader:

- **The name and location of the input data file**
- **The format of the records in the input data file**
- **The name of the table or tables to be loaded**
- **The correspondence between the fields in the input record and the columns in the database tables being loaded**
- **Selection criteria defining which records from the input file contain data to be inserted into the destination database tables.**
- **The names and locations of the bad file and the discard file**

**The Log File**

The *log file* is a record of SQL*Loader's activities during a load session. It contains information such as the following:

- The names of the control file, log file, bad file, discard file, and data file
- The values of several command-line parameters
- A detailed breakdown of the fields and data types in the data file that was loaded
- Error messages for records that cause errors
- Messages indicating when records have been discarded
- A summary of the load that includes the number of logical records read from the data file, the number of rows rejected because of errors, the number of rows discarded because of selection criteria, and the elapsed time of the load

Always review the log file after a load to be sure that no errors occurred, or at least that no unexpected errors occurred. This type of information is written to the log file, but is not displayed on the terminal screen.

**The Bad File and the Discard File**

**Whenever you insert data into a database, you run the risk of that insert failing because of some type of error. Integrity constraint violations undoubtedly represent the most common type of error. However, other problems, such as the lack of free space in a tablespace, can also cause insert operations to fail. Whenever SQL*Loader encounters a database error while trying to load a record, it writes that record to a file known as the** ***bad file*****.**
**Discard files, on the other hand, are used to hold records that do not meet selection criteria specified in the SQL*Loader control file. By default, SQL*Loader will attempt to load all the records contained in the input file**.

## Procedure/Function (Difference)

**Procedure**

**A procedure is a subprogram that performs a specific action**
**Procedure Does and Does not return the Value.**
**Procedure we can use (In, Out, InOut Parameter)**
**You cannot use the procedure in select Statement.**
**Execute as a PL/SQL statement**
**No RETURN clause in the header**
**Can return none, one, or many values**

**Function**

**A function is a subprogram that computes a value**
**Invoke as part of an expression**
**Must contain a RETURN clause in the header**
**Must return a single value**
**Must contain at least one RETURN statement**
**Always return the Value.**
**Function you can use the (In, Out, InOut Parameter)**
**You can use the Function the in select Statement.**

## Pragma

**A pragma is a directive to the PL/SQL compiler. Pragmas pass information to the compiler; they are processed at compile time but do not execute.** If you include a call to a built-in package in a SQL statement, you must include a RESTRICT

REFERENCES pragma in your code. This pragma tells the compiler the purity level (freedom from side effects) of a packaged program.

**Pragma keyword is used to give instructions to the compiler. There are 4 types of Pragmas:**

**A) Exception_Init: - Tells the compiler to associate the specified error number with an identifier that has been declared an Exception in your current program or an accessible package.**
**B) Restrict_References: - Tells the compiler the purity level of packaged program. The purity level is the degree to which a program does not read/write database tables and/or packaged variables.**
**C) Serially_Reusable: - Tells the runtime engine that package data should not persist between references. This is used to reduce per-user memory requirements when the package data is only needed for duration of call and not the duration of session.**
**D) Autonomous_Transactions: - Tells the compiler that the function, procedure, top-level anonymous P/L SQL block, object method, or database trigger executes in its own transaction space.**

## Purity Levels

Prior to Oracle8*i* Release 8.1, it was necessary to assert the purity level of a packaged procedure or function when using it directly or indirectly in a SQL statement. Beginning with Oracle8*i* Release 8.1, the PL/SQL runtime engine determines a program's purity level automatically if no assertion exists.
**The RESTRICT_REFERENCES pragma asserts a purity level. The syntax for the RESTRICT_REFERENCES pragma is:**
**PRAGMA RESTRICT_REFERENCES (program_name |**
**DEFAULT, purity_level);**
The keyword DEFAULT applies to all methods of an object type or all programs in a package.
There can be from one to five purity levels, in any order, in a comma-delimited list. The purity level describes to what extent the program or method is free of *side effects*. Side effects are listed in the following table with the purity levels they address.

| **Purity Level** | **Description** | **Restriction** |
|---|---|---|
| WNDS | WriteNo Database State | Executes no INSERT, UPDATE, or DELETE statements. |
| RNDS | Read No Database State | Executes no SELECT statements. |

| Purity Level | Description | Restriction |
|---|---|---|
| WNPS | Write No Package State | Does not modify any package variables. |
| RNPS | Read No Package State | Does not read any package variables. |
| TRUST (Oracle8*i*) | | Does not enforce the restrictions declared but allows the compiler to trust they are true. |

The purity level requirements for packaged functions are different depending on where in the SQL statement the stored functions are used:
To be called from SQL, all stored functions must assert WNDS.
All functions not used in a SELECT, VALUES, or SET clause must assert WNPS.
To be executed remotely, the function must assert WNPS and RNPS.
To be executed in parallel, the function must assert all four purity levels or, in Oracle8*i*, use PARALLEL_ENABLED in the declaration.
These functions must not call any other program that does not also assert the minimum purity level.
If a package has an initialization section, it too must assert purity in Oracle7.
If a function is overloaded, each overloading must assert its own purity level, and the levels don't have to be the same. To do this, place the pragma immediately after each overloaded declaration.
Many of the built-in packages, including DBMS_OUTPUT, DBMS_PIPE, and DBMS_SQL, do not assert WNPS or RNPS, so their use in SQL stored functions is necessarily limited.

## Early Binding/Late Binding

**Early and Late Binding**
**When you generate RPCs (remote procedure calls) using SQL*Module, you have a choice of *early binding* or *late binding*. Your choice of early or late binding is controlled by the BINDING option.**
**When you choose early binding, SQL*Module generates a call to the procedure stored in the database, and also uses a *time stamp* that is associated with the call. The time stamp records the date and time (to the nearest second) that the stored procedure was last compiled. The time stamp is created by the Oracle database. If a host application calls the stored procedure through the interface procedure, and the time stamp recorded with the interface procedure is earlier than the time stamp on the stored procedure recorded in the database, an error is returned to the host application in the SQLCODE and/or SQLSTATE status parameter. The SQLCODE error is 4062 "time stamp of *name* has been changed".**

**The late binding option, on the other hand, does not use a time stamp. If your application calls a stored procedure that has been recompiled since SQL*Module generated the interface procedure, no error is returned to the application.**

The advantages of late binding are

Greater flexibility
Changes in the stored procedure(s) are transparent to the user
Gives behavior similar to interactive SQL (for example, SQL*PLus)

The disadvantages of late binding are

There might be additional performance overhead at runtime, due to the necessity of compiling the PL/SQL anonymous block.

It is difficult to detect runtime PL/SQL compilation errors in the host application. For example, if the anonymous block that calls the late-bound procedure fails at runtime, there is no convenient way for the host application to determine the cause of the error.

The lack of time-stamp capability means that changes, perhaps radical changes, in the stored procedure could be made after the host application was built, and the application would have no way of detecting this.

## Hot & Cold Backup

**A cold backup is a physical backup. During a cold backup the database is closed and not available to users. All files of the database are copied (image copy). The datafiles do not change during the copy so the database is in sync upon restore.**

**Used when:**
**Service level allows for some down time for backup**

**A hot backup is a physical backup. In a hot backup the database remains open and available to users. All files of the database are copied (image copy). There may be changes to the database as the copy is made and so all log files of changes being made during the backup must be saved too. Upon a restore, the changes in the log files are reapplied to bring the database in sync.**
**Used when:**
**A full backup of a database is needed Service level allows no down time for the backup**

**A logical backup is an extract of the database. All SQL statements to create the objects and all SQL statements to populate the objects are included in the extract. Oracle provides a utility export, to create the extract. A partner utility, import, is used to bring the data back into the database.**

**A logical backup can be done at the table, schema (or proxy owner), or database level. That is, we can extract only a list of specified tables, a list of specified schemas or the full database.**

**Used to:**

**Move or archive a database**
**Move or archive a table(s)**
**Move or archive a schema(s)**
**Verify the structures in the database**

## Background Processes in Oracle

**Background Process:**

**Database writer (DBWR): Writes modified blocks from database buffer cache to the datafile.**
**Log Writer (LGWR): Writes redo log entries to disk.**
**Check Point: At specific times all modified databases buffers in the SGA are written to the data files by DBWR. This event is called Checkpoint.**
**System Monitor (SMON): Performs instance recovery at instance startup**
**Process Monitor (PMON): Performs process recovery when user process fails**
**Archiver (ARCH): Copies on line redo log files to archival storage when they are full**
**Dispatcher: For multi threaded server for each request one dispatcher process**
**Lock: For parallel server mode for inter instance locking**

## Types of SQL Statements

There are basically 6 types of sql statments. They are
**a) Data Definition Language (DDL): The DDL statements define and maintain objects and drop objects.**
**b) Data Manipulation Language (DML): The DML statements manipulate database data.**
**c) Transaction Control Statements: Manage change by DML**
**d) Session Control: Used to control the properties of current session enabling and disabling roles and changing .e.g: Alter Statements, Set Role**
**e) System Control Statements: Change Properties of Oracle Instance. e.g.: Alter System**
**f) Embedded Sql: Incorporate DDL, DML and T.C.S in Programming Language.e.g: Using the Sql Statements in languages such as 'C', Open, Fetch, execute and close**

## What is Transaction (Its ACID Property)

**A transaction is a Logical unit of work that compromises one or more SQL Statements executed by a single User**. According to ANSI, a transaction begins with first executable statement and ends when it is explicitly committed or rolled back. A transaction is an atomic unit of work that either fails or succeeds. There is no such thing as a partial completion of a transaction. Since a transaction can be made up of many steps, each step in the transaction must succeed for the transaction to be successful. If any one part of the transaction fails, then the entire transaction fails. **When a transaction fails, the system needs to return to the state that it was in before the transaction was started. This is known as rollback**. When a transaction fails, then the changes that had been made are said to be "rolled back." In effect, this is acting similar to the way the Undo command works in most word processors. When you select undo, the change that you just may have made is reversed. The transaction processing system is responsible for carrying out this undo.
When a transaction processing system creates a transaction, it will ensure that the transaction will have certain characteristics. ACID is an acronym for atomicity, consistency, isolation, and durability.

- **An atomic transaction is either fully completed, or is not begun at all.**
- **A transaction enforces consistency in the system state by ensuring that at the end of any transaction the system is in a valid state.**
- **When a transaction runs in isolation, it appears to be the only action that the system is carrying out at one time.**
- **A transaction is durable in that once it has been successfully completed, all of the changes it made to the system are permanent.**

## Exceptions (User-Defined/Oracle Defined)

**Oracle includes about 20 predefined exceptions (errors) - we can allow Oracle to raise these implicitly. For errors that don't fall into the predefined categories - declare in advance and allow oracle to raise an exception. For problems that are not recognized as an error by Oracle - but still cause some difficulty within your application - declare a User Defined Error and raise it explicitly**
**Trap non-predefined errors by declaring them. You can also associate the error no. with a name so that you can write a specific handler. This is done with the PRAGMA EXCEPION_INIT pragma. PRAGMA (pseudo instructions) indicates that an item is a 'compiler directive' running this has no immediate effect but causes all subsequent references to the exception name to be interpreted as the associated Oracle Error. When an exception occurs you can identify the associated error code/message with two supplied functions SQLCODE and SQLERRM.**
**Trapping user-defined exceptions**

**DECLARE the exception**
**RAISE the exception**

**Handle the raised exception**

**Propagation of Exception handling in sub blocks**
**If a sub block does not have a handler for a particular error it will propagate to the enclosing block - where it can be caught by more general exception handlers.**

**RAISE_APPLICATION_ERROR (error_no, message [, {TRUE|FALSE}]);**

**This procedure allows user defined error messages from stored sub programs - call only from stored sub prog.**
**Error_no = a user defined no (between -20000 and -20999)**

**TRUE = stack errors**
**FALSE = keep just last**

**This can either be used in the executable section of code or the exception section**

## SGA (System Global Area)

**The SGA is a shared memory region allocated by the Oracle that contains Data and control information for one Oracle Instance. Oracle allocates the SGA when an instance starts and deallocates it when the instance shuts down. Each instance has its own SGA. Users currently connected to an Oracle server share the data in the SGA. For optimal performance, the entire SGA should be as large as possible (while still fitting in real memory) to store as much data in memory as possible and to minimize disk I/O. It consists of Database Buffer Cache and Redo log Buffer.**
**The PGA (Program Global Area) is a memory buffer that contains data and control information for server process. A PGA is created by Oracle when a server process is started. The information in a PGA depends on the Oracle configuration.**

## Candidate, Artificial (Derived) Primary Key, Unique & Primary Key

**Integrity Constraint**
An **integrity constraint** is a declarative way to define a business rule for a column of a table. An integrity constraint is a statement about a table's data that is always true and that follows these rules:

If an integrity constraint is created for a table and some existing table data does not satisfy the constraint, then the constraint cannot be enforced.

After a constraint is defined, if any of the results of a DML statement violate the integrity constraint, then the statement is rolled back, and an error is returned.

Integrity constraints are defined with a table and are stored as part of the table's definition in the data dictionary, so that all database applications adhere to the same set of rules. When a rule changes, it only needs be changed once at the database level and not many times for each application.

**Candidate key:**
A relation (table) may contain more than one key.
In this case, one of the keys is selected as the primary key and the remaining keys are called candidate keys.
The candidate key constraint indicates that the table cannot contain two different rows with the same values under candidate key columns.

**Artificial (Derived) Primary Key**
A derived key comes from a sequence. Usually it is used when a concatenated key becomes too cumbersome to use as a foreign key.

**UNIQUE Constraints**
You can use UNIQUE constraints to ensure that no duplicate values are entered in specific columns that do not participate in a primary key. Although both a UNIQUE constraint and a PRIMARY KEY constraint enforce uniqueness, use a UNIQUE constraint instead of a PRIMARY KEY constraint when you want to enforce the uniqueness of a column, or combination of columns, that is not the primary key.
**Multiple UNIQUE constraints can be defined on a table, whereas only one PRIMARY KEY constraint can be defined on a table**.

**FOREIGN KEY Constraints**
A foreign key (FK) is a column or combination of columns used to establish and enforce a link between the data in two tables. A link is created between two tables by adding the column or columns that hold one table's primary key values to the other table. This column becomes a foreign key in the second table.

**PRIMARY KEY Constraints**
A table usually has a column or combination of columns whose values uniquely identify each row in the table. This column (or columns) is called the primary key of the table and enforces the entity integrity of the table. **A table can have only one PRIMARY KEY constraint, and a column that participates in the PRIMARY KEY constraint cannot accept null values.**

## PCT Free & PCT Used

**PCT Free:** This parameter tells oracle how much space to leave in each oracle block for future updates. This defaults to 10. If table will have large number of updates, a larger value is needed. If the table will be static, small value can be used.

**PCT Used:** This parameter tells oracle the minimum level of space in a block to maintain. Default is 40. A block becomes a candidate for updates if its storage falls below PCTUSED. PCTUSED + PCTFREE should not exceed 100. A high PCTUSED value results in more efficient space utilization but higher overhead, as oracle must work harder to maintain the free block list.

## 2 Phase Commit

**Two Phase Commit:** Oracle provides the same assurance of data consistency in a distributed environment as in a non-distributed environment.
There is two phases:

- **Prepare Phase –** initiating node called global co-ordinator notifies all sites involved in transaction to be ready to either commit or rollback the transaction.
- **Commit Phase –** If there is no problem in prepare phase, then all sites commit their transactions. If network or node failure occurs then all sites rollback their transactions.

## Delete & Truncate Table

**If you want to delete all the rows in a table, TRUNCATE TABLE is faster than DELETE. DELETE physically removes rows one at a time and records each deleted row in the transaction log. TRUNCATE TABLE deallocates all pages associated with the table. For this reason, TRUNCATE TABLE is faster and requires less transaction log space than DELETE. TRUNCATE TABLE is functionally equivalent to DELETE with no WHERE clause, but TRUNCATE TABLE cannot be used with tables referenced by foreign keys. Both DELETE and TRUNCATE TABLE make the space occupied by the deleted rows available for the storage of new data.**
**Truncate just resets the high-water mark. It does not delete any rows. If you delete rows, then whatever you have specified regarding the referential integrity will be done (ON DELETE SET NULL, or ON DELETE CASCADE, or the default which is to return ORA-02292 if the referential integrity constraint would be violated) whereas TRUNCATE just returns ORA-02266 if you have any enabled foreign key constraints referencing the table, even if the tables are empty.**

| | |
|---|---|
| *schema* | is the schema to contain the trigger. If you omit *schema*, Oracle creates the trigger in your own schema. |
| TABLE | specifies the schema and name of the table to be truncated. You can truncate index-organized tables. This table cannot be part of a cluster. |
| | When you truncate a table, Oracle also automatically deletes all data in the |

| | | |
|---|---|---|
| | table's indexes. | |
| SNAPSHOT LOG | specifies whether a snapshot log defined on the table is to be preserved or purged when the table is truncated. This clause allows snapshot master tables to be reorganized through export/import without affecting the ability of primary-key snapshots defined on the master to be fast refreshed. To support continued fast refresh of primary-key snapshots the snapshot log must record primary-key information. For more information about snapshot logs and the TRUNCATE command, see *Oracle8 Replication*. | |
| | PRESERVE | specifies that any snapshot log should be preserved when the master table is truncated. This is the default. |
| | PURGE | specifies that any snapshot log should be purged when the master table is truncated. |
| CLUSTER | specifies the schema and name of the cluster to be truncated. You can only truncate an indexed cluster, not a hash cluster. | |
| | When you truncate a cluster, Oracle also automatically deletes all data in the cluster's tables' indexes. | |
| DROP STORAGE | deallocates the space from the deleted rows from the table or cluster. This space can subsequently be used by other objects in the tablespace. This is the default. The DROP STORAGE option deallocates all but the space specified by the table's MINEXTENTS parameter. | |
| REUSE STORAGE | retains the space from the deleted rows allocated to the table or cluster. Storage values are not reset to the values when the table or cluster was created. This space can subsequently be used only by new data in the table or cluster resulting from inserts or updates. | |
| | The DROP STORAGE and REUSE STORAGE options also apply to the space freed by the data deleted from associated indexes. | |

Deleting rows with the TRUNCATE command is also more convenient than dropping and re-creating a table because dropping and re-creating:

invalidates the table's dependent objects, while truncating does not

requires you to regrant object privileges on the table, while truncating does not

requires you to re-create the table's indexes, integrity constraints, and triggers and respecify its STORAGE parameters, while truncating does not

**Note:**

When you truncate a table, the storage parameter NEXT is changed to be the size of the last extent deleted from the segment in the process of truncation.

**Restrictions**

When you truncate a table, NEXT is automatically reset to the last extent deleted.

You cannot individually truncate a table that is part of a cluster. You must either truncate the cluster, delete all rows from the table, or drop and re-create the table.
You cannot truncate the parent table of an enabled referential integrity constraint. You must disable the constraint before truncating the table. (An exception is that you may truncate the table if the integrity constraint is self-referential.)
You cannot roll back a TRUNCATE statement.

## Mutating Table/Constraining Table

**"Mutating" means "changing". A mutating table is a table that is currently being modified by an update, delete, or insert statement. When a trigger tries to reference a table that is in state of flux (being changed), it is considered "mutating" and raises an error since Oracle should not return data that has not yet reached its final state.**
Another way this error can occur is if the trigger has statements to change the primary, foreign or unique key columns of the table off which it fires. If you must have triggers on tables that have referential constraints, the workaround is to enforce the referential integrity through triggers as well.
A constraining table is a table that a triggering statement might need to read either directly, for a SQL statement, or indirectly, for a declarative referential integrity constraint.
There are several restrictions in Oracle regarding triggers:
A row-level trigger cannot query or modify a mutating table. (Of course, NEW and OLD still can be accessed by the trigger).
A statement-level trigger cannot query or modify a mutating table if the trigger is fired as the result of a CASCADE delete. Etc.

**Mutating Table error happens with triggers. It occurs because the trigger is trying to update a row it is currently using. The usual fix involves either use of views or temporary tables so the database is selecting from one while updating the other.**
**For all row triggers, or for statement triggers that were fired as the result of a DELETE CASCADE, there are two important restrictions regarding mutating and constraining tables. These restrictions prevent a trigger from seeing an inconsistent set of data.**

**The SQL statements of a trigger cannot read from (query) or modify a mutating table of the triggering statement.**

- **The statements of a trigger cannot change the PRIMARY, FOREIGN, or UNIQUE KEY columns of a constraining table of the triggering statement.**

**If you need to update a mutating or constraining table, you could use a temporary table, a PL/SQL table, or a package variable to bypass these restrictions.**

## Pseudo Columns

A pseudocolumn behaves like a table column, but is not actually stored in the table. You can select from pseudocolumns, but you cannot insert, update, or delete their values. E.g. CURRVAL, NEXTVAL, ROWID, ROWNUM, LEVEL

## Record Groups. Can these be created at run time (How)

A record group is an internal Oracle Forms data structure that has a column/row framework similar to a database table. However, unlike database tables, record groups are separate objects that belong to the form module in which they are defined. A record group can have an unlimited number of columns of type CHAR, LONG, NUMBER, or DATE provided that the total number of columns does not exceed 64K.
Record group column names cannot exceed 30 characters.
Programmatically, record groups can be used whenever the functionality offered by a two-dimensional array of multiple data types is desirable.
TYPES OF RECORD GROUP:
**Query Record Group:** A query record group is a record group that has an associated SELECT statement. The columns in a query record group derive their default names, data types, and lengths from the database columns referenced in the SELECT statement. The records in a query record group are the rows retrieved by the query associated with that record group.
**Non-query Record Group:** A non-query record group is a group that does not have an associated query, but whose structure and values can be modified programmatically at runtime.
**Static Record Group:** A static record group is not associated with a query; rather, you define its structure and row values at design time, and they remain fixed at runtime.

## Key-Mode & Locking Mode Properties

**Key Mode**: Specifies how oracle forms uniquely identifies rows in the database. This is property includes for application that will run against NON-ORACLE datasources. By default, the ORACLE database uses unique ROWID values to identify each row. Non-ORACLE databases do not include the ROWID construct, but instead rely solely on unique primary key values to identify unique rows. If you are creating a form to run against a non-ORACLE data source, you must use primary keys, and set the Key Mode block property accordingly.
**Automatic** (default) Specifies that Form Builder should use ROWID constructs to identify unique rows in the datasource but only if the datasource supports ROWID.
**Non-Updateable** Specifies that Form Builder should not include primary key columns in any UPDATE statements. Use this setting if your database does not allow primary key values to be updated.

**Unique** Instructs Form Builder to use ROWID constructs to identify unique rows in an ORACLE database.
**Updateable:** Specifies that Form Builder should issue UPDATE statements that include primary key values. Use this setting if your database allows primary key columns to be updated and you intend for the application to update primary key values.
**Locking mode**: Specifies when Oracle Forms should attempt to obtain database locks on rows that correspond to queried records in the form.
**Automatic** (default) Identical to Immediate if the datasource is an Oracle database. For other datasources, Form Builder determines the available locking facilities and behaves as much like Immediate as possible.
**Immediate** Form Builder locks the corresponding row as soon as the end user presses a key to enter or edit the value in a text item.
**Delayed** Form Builder locks the row only while it posts the transaction to the database, not while the end user is editing the record. Form Builder prevents the commit action from processing if values of the fields in the block have changed when the user causes a commit action.

## What is Call Form Stack

When successive forms are loaded via the CALL_FORM procedure, the resulting module hierarchy is known as the call form stack.

## Triggers related to Mouse

When-Mouse-Click
When-Mouse-Double-Click
When-Mouse-Down
When-Mouse-Enter
When-Mouse-Leave
When-Mouse-Move
When-Mouse-Up

## How to change Visual Attribute at Runtime

You can programmatically change an object's named visual attribute setting to change the font, color, and pattern of the object at **runtime**. E.g. using SET_CANVAS_PROPERTY, Set_Item_Property ('Blockname.Itemname', VISUAL_ATTRIBUTE, 'VisualAttributeName' );
Where 'VisualAttributeName' is either the name of a visual attribute you've set up in the form (which has foreground, background, font, etc.), or is the name of a logical visual attribute from the resource file.

## Types of Columns in Reports (Summary/Formula/Placeholder)

**Formula columns**: For doing mathematical calculations and returning one value
**Summary Columns**: For doing summary calculations such as summations etc.
**Placeholder Columns**: These columns are useful for storing the value in a variable. Used to assign a value through formula column.

## Types of Columns in Forms (Summary/Formula)

**Summary**—the calculated item's value is the summary of all values of a single item in a single block. A summary calculation for a calculated item consists of a single computation performed on all records of a single item in a single data block.

**Formula**—the calculated item's value is the result of a formula that performs calculations involving one or more values. A formula calculation assigns a value to a calculated item by performing calculations involving one or more bind variables or constants. Bind variables include form items, parameters, global variables, and system variables.

## How many Integrity Rules are there and what are they

There are Three Integrity Rules. They are as follows:
a) **Entity Integrity Rule**: The Entity Integrity Rule enforces that the Primary key cannot be Null
b) **Foreign Key Integrity Rule**: The FKIR denotes that the relationship between the foreign key and the primary key has to be enforced. When there is data in Child Tables the Master tables cannot be deleted.
c) **Business Integrity Rules**: The Third Integrity rule is about the complex business processes which cannot be implemented by the above 2 rules.

## Cascading Triggers

**When a statement in a trigger body causes another trigger to be fired, the triggers are said to be *cascading*.**

## Forward Declaration

P/L SQL does not allow forward declaration. Identifier must be declared before using it. Therefore, subprogram must be declared before calling it. PL/SQL requires that every identifier must be declared before use. There are occasions where such

declaration is not possible. For instance, 2 mutually recursive procedures will need to refer to each other before anyone such procedure can be declared. The solution is by Forward Declaration in which function or procedure's specifications are declared at the declaration. Another way to do forward declaration is to set up a package, which can be separated into specification part and body part.
Forward Declaration can be used to do the following:

- Define Subprograms in logical or alphabetical order.
- Define mutually recursive subprograms.
- Group subprograms in a package.

## Packages provided by Oracle

Oracle provides the DBMS_ series of packages. There are many which developers should be aware of such as DBMS_SQL, DBMS_PIPE, DBMS_TRANSACTION, DBMS_LOCK, DBMS_ALERT, DBMS_OUTPUT, DBMS_JOB, DBMS_UTILITY, DBMS_DDL, UTL_FILE.

## SQL Code & SQLERRM

SQLCODE returns the value of the error number for the last error encountered. The SQLERRM returns the actual error message for the last error encountered. They can be used in exception handling to report, or, store in an error log table, the error that occurred in the code. These are especially useful for the WHEN OTHERS exception.

## How can variables be passed to SQL Runtime

**Using Ampersand (&)**

## Cartesian Product

A Cartesian product is the result of an unrestricted joins of two or more tables. The result set of a three table Cartesian product will have x * y * z number of rows where x, y, z correspond to the number of rows in each table involved in the join.

## How to prevent output coming to screen

The SET option TERMOUT controls output to the screen. Setting TERMOUT OFF turns off screen output. This option can be shortened to TERM.

## How to prevent from giving information messages during SQL Execution

The SET options FEEDBACK and VERIFY can be set to OFF.

## Row Chaining

Row chaining occurs when a VARCHAR2 value is updated and the length of the new value is longer than the old value and won't fit in the remaining block space. This results in the row chaining to another block. Setting the storage parameters on the table to appropriate values can reduce it. It can be corrected by export and import of the effected table.

## ERD (Relations)

An ERD is an Entity-Relationship-Diagram. It is used to show the entities and relationships for a database logical model. The entity-relationship model is a tool for analyzing the semantic features of an application that are independent of events. Entity-relationship modeling helps reduce data redundancy.
Type of Relationship
**One-One Relationship** - For each entity in one class there is at most one associated entity in the other class. For example, for each husband there is at most one current legal wife (in this country at least). A wife has at most one current legal husband.
**Many-One Relationships** - One entity in class E2 is associated with zero or more entities in class E1, but each entity in E1 is associated with at most one entity in E2. For example, a woman may have many children but a child has only one birth mother.
**Many-Many Relationships** - There are no restrictions on how many entities in either class are associated with a single entity in the other. An example of a many-to-many relationship would be students taking classes. Each student takes many classes. Each class has many students.
A relational database is a set of relations. It can be thought of as a set of related tables. Each table has a specific form. Attributes can be thought of as column headings.
**Entity-Relationship Diagram**
Symbols used in entity-relationship diagrams include:
Rectangles represent ENTITY CLASSES
Circles represent ATTRIBUTES
Diamonds represent RELATIONSHIPS
Arcs - Arcs connect entities to relationships. Arcs are also used to connect attributes to entities. Some styles of entity-relationship diagrams use arrows and double arrows to indicate the one and the many in relationships. Some use forks etc.
Underline - Key attributes of entities are underlined.

## Snapshots

Oracle provides an automatic method for table replication and update called snapshots. Snapshots are read-only copies of a master table located on a remote node. A snapshot can only be queried, not updated. A snapshot is periodically refreshed to reflect changes made to the master table.
Snapshots are local; copies (also known as replicas) of remote data, based upon queries. The refreshes of the replicated data can be done automatically by the database, at time intervals you specify.

## Materialized Views

Introduced with Oracle8*i*, a materialized view is designed to improve performance of the database by doing some intensive work in advance of the results of that work being needed. In the case of a materialized view, the data for the view is assembled when the view is created or refreshed. Later queries that need this data automatically use the materialized view, thus saving the overhead of performing the work already done by the view.
The work avoided by a materialized view is essentially twofold:

A materialized view can be used to pre-collect aggregate values.

A materialized view can be used to assemble data that would come from many different tables, which would in turn require many different joins to be performed.

A materialized view is **like** a view in that it represents data that is contained in other database tables and views; yet it is **unlike** a view in that it contains actual data. A materialized view is **like** an index in that the data it contains is derived from the data in database tables and views; yet **unlike** an index in that its data must be explicitly refreshed. Finally, a materialized view is very much like a snapshot in that an administrator can specify when the data is to be refreshed; but it is unlike a snapshot in that a materialized view should either include summary data or data from many different joined tables.

| | |
|---|---|
| CREATE MATERIALIZED VIEW | are required keywords |
| name | is the qualified name of the materialized view |
| Physical attributes clause | allows you to specify the physical attributes, such the tablespace name, for the materialized view |
| BUILD clause | The BUILD clause allows you to specify when you want to build the actual data in the table. Your options are BUILD IMMEDIATE, which calls for the view to be |

| | |
|---|---|
| | immediately built, BUILD DEFERRED, which calls for the view to be built when it is first refreshed (see explanation of REFRESH clause below) or ON PREBUILT TABLE, which indicates that you are identifying a table that is already built as a materialized view. |
| REFRESH clause | Since the materialized view is built on underlying data that is periodically changed, you must specify how and when you want to refresh the data in the view. You can specify that you want a FAST refresh, which will only update the values in the materialized view, assuming that some preconditions are met, COMPLETE, which recreates the view completely, or FORCE, which will do a FAST refresh if possible and a COMPLETE refresh if the preconditions for a FAST refresh are not available.<br><br>The REFRESH clause also contains either the keywords ON COMMIT, which will cause a refresh to occur whenever the underlying data is changed and the change is committed, or ON DEMAND, which will only perform a refresh when it is scheduled or explicitly called. You can also use keywords in the REFRESH clause to create a schedule for recurring refresh operations. |
| *AS subquery* | The last clause of the CREATE MATERIALIZED VIEW command contains the sub-query that will be used to retrieve the data that will compose the materialized view. |

Oracle uses **materialized views** (also known as snapshots in prior releases) to replicate data to non-master sites in a replication environment and to cache expensive queries in a data warehouse environment.

You can use materialized views to achieve one or more of the following goals:

- Ease Network Loads
- Create a Mass Deployment Environment
- Enable Data Subsetting
- Enable Disconnected Computing

A materialized view can be either read-only, updatable, or writeable. Users cannot perform data manipulation language (DML) statements on read-only materialized views, but they can perform DML on updatable and writeable materialized views.

**More Explanation**

A materialized view is a database object that contains the results of a query. They are local copies of data located remotely, or are used to create summary tables based on aggregations of a table's data. Materialized views, which store data based on remote tables are also, know as snapshots.
A materialized view can query tables, views, and other materialized views. Collectively these are called master tables **(a replication term)** or detail tables **(a data warehouse term).**

**For replication purposes**, materialized views allow you to maintain copies of remote data on your local node. These copies are read-only. If you want to update the local copies, you have to use the Advanced Replication feature. You can select data from a materialized view as you would from a table or view.
**For data warehousing purposes**, the materialized views commonly created are aggregate views, single-table aggregate views, and join views.
In replication environments, the materialized views commonly created are primary key, rowid, and subquery materialized views

**Primary Key Materialized Views**
The following statement creates the primary-key materialized view on the table emp located on a remote database.

```
SQL>  CREATE MATERIALIZED VIEW mv_emp_pk
      REFRESH FAST START WITH SYSDATE
      NEXT  SYSDATE + 1/48
      WITH PRIMARY KEY
      AS SELECT * FROM emp@remote_db;

Materialized view created.
```

Note: When you create a materialized view using the FAST option you will need to create a view log on the master tables(s) as shown below:

```
SQL> CREATE MATERIALIZED VIEW LOG ON emp;
Materialized view log created.
```

**Rowid Materialized Views**
The following statement creates the rowid materialized view on table emp located on a remote database:

```
SQL>  CREATE MATERIALIZED VIEW mv_emp_rowid
      REFRESH WITH ROWID
      AS SELECT * FROM emp@remote_db;

Materialized view log created.
```

**Subquery Materialized Views**
The following statement creates a subquery materialized view based on the emp and dept tables located on the remote database:

```
SQL> CREATE MATERIALIZED VIEW  mv_empdept
AS SELECT * FROM emp@remote_db e
WHERE EXISTS
   (SELECT * FROM dept@remote_db d
   WHERE e.dept_no = d.dept_no)
```

**Refresh Clause in Materialized Views**

## REFRESH CLAUSE

```
[refresh [fast|complete|force]
        [on demand | commit]
        [start with date] [next date]
        [with {primary key|rowid}]]
```
The refresh option specifies:
The refresh method used by Oracle to refresh data in materialized view
Whether the view is primary key based or row-id based
The time and interval at which the view is to be refreshed
*Refresh Method* - FAST Clause
The FAST refreshes use the materialized view logs (as seen above) to send the rows that have changed from master tables to the materialized view.
You should create a materialized view log for the master tables if you specify the REFRESH FAST clause.
```
SQL> CREATE MATERIALIZED VIEW LOG ON emp;

Materialized view log created.
```
Materialized views are not eligible for fast refresh if the defined subquery contains an analytic function.
*Refresh Method* - COMPLETE Clause
The complete refresh re-creates the entire materialized view. If you request a complete refresh, Oracle performs a complete refresh even if a fast refresh is possible.
*Refresh Method* - FORCE Clause
When you specify a FORCE clause, Oracle will perform a fast refresh if one is possible or a complete refresh otherwise. If you do not specify a refresh method (FAST, COMPLETE, or FORCE), FORCE is the default.
*PRIMARY KEY and ROWID Clause*
WITH PRIMARY KEY is used to create a primary key materialized view i.e. the materialized view is based on the primary key of the master table instead of ROWID (for ROWID clause). PRIMARY KEY is the default option. To use the PRIMARY KEY clause you should have defined PRIMARY KEY on the master table or else you should use ROWID based materialized views.
Primary key materialized views allow materialized view master tables to be reorganized without affecting the eligibility of the materialized view for fast refresh.
Rowid materialized views should have a single master table and cannot contain any of the following:
Distinct or aggregate functions
GROUP BY Subqueries , Joins & Set operations
Timing the refresh
The START WITH clause tells the database when to perform the first replication from the master table to the local base table. It should evaluate to a future point in time. The NEXT clause specifies the interval between refreshes
```
SQL>  CREATE MATERIALIZED VIEW mv_emp_pk
      REFRESH FAST
```

```
START WITH SYSDATE
NEXT  SYSDATE + 2
WITH PRIMARY KEY
AS SELECT * FROM emp@remote_db;
```

Materialized view created.
In the above example, the first copy of the materialized view is made at SYSDATE and the interval at which the refresh has to be performed is every two days.
Summary Materialized Views thus offer us flexibility of basing a view on Primary key or ROWID, specifying refresh methods and specifying time of automatic refreshes.

- **Scenario for Using Multi-tier Materialized Views**

Consider a multinational company that maintains all employee information at headquarters, which is in the in the United States. The company uses the tables in the hr schema to maintain the employee information. This company has one main office in 14 countries and many regional offices for cities in these countries.
For example, the company has one main office for all of the United Kingdom, but it also has an office in the city of London. The United Kingdom office maintains employee information for all of the employees in the United Kingdom, while the London office only maintains employee information for the employees at the London office. In this scenario, the hr.employees master table is at headquarters in the United States and each regional office has a hr.employees materialized view that only contains the necessary employee information.
The following statement creates the hr.employees materialized view for the United Kingdom office. The statement queries the master table in the database at headquarters, which is orc1.world. Notice that the statement uses subqueries so that the materialized view only contains employees whose country_id is UK.

```
CREATE MATERIALIZED VIEW hr.employees REFRESH FAST FOR UPDATE AS
  SELECT * FROM hr.employees@orc1.world e
    WHERE EXISTS
      (SELECT * FROM hr.departments@orc1.world d
       WHERE e.department_id = d.department_id
       AND EXISTS
         (SELECT * FROM hr.locations@orc1.world l
          WHERE l.country_id = 'UK'
          AND d.location_id = l.location_id));
```

**Note:**
To create this hr.employees materialized view, the following columns must be logged:

- The department_id column must be logged in the materialized view log for the hr.employees master table at orc1.world.
- The country_id must be logged in the materialized view log for the hr.locations master table at orc1.world.

The following statement creates the hr.employees materialized view for the London office based on the level 1 materialized view at the United Kingdom office. The statement queries the materialized view in the database at the United Kingdom office, which is reg_uk.world. Notice that the statement uses subqueries so that the materialized view only contains employees whose city is London.

```
CREATE MATERIALIZED VIEW hr.employees REFRESH FAST FOR UPDATE AS
  SELECT * FROM hr.employees@reg_uk.world e
    WHERE EXISTS
      (SELECT * FROM hr.departments@reg_uk.world d
       WHERE e.department_id = d.department_id
       AND EXISTS
         (SELECT * FROM hr.locations@reg_uk.world l
          WHERE l.city = 'London'
          AND d.location_id = l.location_id));
```

**Note:**

To create this hr.employees materialized view, the following columns must be logged:

- The department_id column must be logged in the materialized view log for the hr.employees master materialized view at reg_uk.world.
- The country_id must be logged in the materialized view log for the hr.locations master materialized view at reg_uk.world

## DDL Statements/Dynamic SQL from P/L Sql (Execute Immediate)

EXECUTE IMMEDIATE is the replacement for DBMS_SQL package from Oracle 8i onwards. It parses and immediately executes a dynamic SQL statement or a PL/SQL block created on the fly. Dynamically created and executed SQL statements are performance overhead; EXECUTE IMMEDIATE aims at reducing the overhead and give better performance. It is also easier to code as compared to earlier means. The error messages generated when using this feature are more user friendly. Though DBMS_SQL is still available, it is advisable to use EXECUTE IMMEDIATE calls because of its benefits over the package.

**Usage tips**

1. EXECUTE IMMEDIATE will not commit a DML transaction carried out and an explicit commit should be done.

If the DML command is processed via EXECUTE IMMEDIATE, one needs to explicitly commit any changes that may have been done before or as part of the EXECUTE IMMEDIATE itself. If the DDL command is processed via EXECUTE IMMEDIATE, it will commit all previously changed data.

2. Multi-row queries are not supported for returning values, the alternative is to use a temporary table to store the records (see example below) or make use of REF cursors.

3. Do not use a semi-colon when executing SQL statements, and use semi-colon at the end when executing a PL/SQL block.
4. This feature is not covered at large in the Oracle Manuals. Below are examples of all possible ways of using Execute immediate. Hope it is handy.

6. For Forms Developers, this feature will not work in Forms 6i front-end as it is on PL/SQL 8.0.6.3.

Example
```
declare
 l_depnam varchar2(20) := 'testing';
 l_loc    varchar2(10) := 'Dubai';
begin
 execute immediate 'insert into dept values (:1, :2, :3)'
   using 50, l_depnam, l_loc;
 commit;
end;
```

EXECUTE IMMEDIATE is a much easier and more efficient method of processing dynamic statements than could have been possible before. As the intention is to execute dynamic statements, proper handling of exceptions becomes all the more important. Care should be taken to trap all possible exceptions.

## % Type & % Row Type

The %TYPE and %ROWTYPE constructs provide data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs.
%ROWTYPE is used to declare a record with the same types as found in the specified database table, view or cursor. Example:

```
DECLARE
  v_EmpRecord  emp%ROWTYPE;
```
%TYPE is used to declare a field with the same type as that of a specified table's column. Example:

```
DECLARE
  v_EmpNo  emp.empno%TYPE;
```

## Instead of Triggers

Views are commonly used to separate the logical database schema from the physical schema. Unfortunately the desired transparency often falls short in the case of UPDATE, DELETE or INSERT operations, since all but the simplest views are not updatable.

Instead Of Trigger execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.
Instead of Trigger can't be written at Statement Level.

## Tables/Views (Difference)

V**iews** 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.
This is the whole list of reasons to use views:

1) to provide an additional level of table security by restricting access to a predetermined set of rows and/or columns of a table

2) to hide data complexity
For example, a single view might be defined with a join, which is a collection of related columns or rows in multiple tables. However, the view hides the fact that this information actually originates from several tables.
to simplify commands for the user
For example, views allow users to select information from multiple tables without actually knowing how to perform a join.

3) To present the data in a different perspective from that of the base table
For example, the columns of a view can be renamed without affecting the tables on which the view is based.

4) to isolate applications from changes in definitions of base tables
For example, if a view's defining query references three columns of a four column table and a fifth column is added to the table, the view's definition is not affected and all applications using the view are not affected.

5) To express a query that cannot be expressed without using a view
For example, a view can be defined that joins a GROUP BY view with a table, or a view can be defined that joins a UNION view with a table. For information about GROUP BY or UNION, see the Oracle8 Server SQL Reference.

6) To save complex queries
For example, a query could perform extensive calculations with table information. By saving this query as a view, the calculations can be performed each time the view is queried.

7) To achieve improvements in availability and performance.
For example, a database administrator can divide a large table into smaller tables (partitions) for many reasons, including partition level load, purge, backup, restore, reorganization, and index building. Once partition views are defined, users can query partitions, rather than very large tables. This ability to prune unneeded partitions from queries increases performance and availability.

## Dbms_SQL

You can write stored procedures and anonymous PL/SQL blocks that use dynamic SQL. Dynamic SQL statements are not embedded in your source program; rather, they are stored in character strings that are input to, or built by, the program at runtime.
This permits you to create procedures that are more general purpose. For example, using dynamic SQL allows you to create a procedure that operates on a table whose name is not known until runtime.
Additionally, you can parse any data manipulation language (DML) or data definition language (DDL) statement using the DBMS_SQL package. This helps solve the problem of not being able to parse data definition language statements directly using PL/SQL. For example, you might now choose to issue a DROP TABLE statement from within a stored procedure by using the PARSE procedure supplied with the DBMS_SQL package.
Table provides a brief description of each of the procedures and functions associated with the DBMS_SQL package, which are described in detail later in this chapter. An example of how these functions can be used begins.

**DBMS_SQL Package Functions and Procedures**

| **Function/Procedure** | **Description** |
|---|---|
| OPEN_CURSOR | Return cursor ID number of new cursor. |
| PARSE | Parse given statement. |
| BIND_VARIABLE | Bind a given value to a given variable. |
| DEFINE_COLUMN | Define a column to be selected from the given cursor, used only with SELECT statements. |
| DEFINE_COLUMN_LONG | Define a LONG column to be selected from the given cursor, used only with SELECT statements. |
| EXECUTE | Execute a given cursor. |

| | |
|---|---|
| EXECUTE_AND_FETCH | Execute a given cursor and fetch rows. |
| FETCH_ROWS | Fetch a row from a given cursor. |
| COLUMN_VALUE | Returns value of the cursor element for a given position in a cursor. |
| COLUMN_VALUE_LONG | Returns a selected part of a LONG column, that has been defined using DEFINE_COLUMN_LONG. |
| VARIABLE_VALUE | Returns value of named variable for given cursor. |
| IS_OPEN | Returns TRUE if given cursor is open. |
| CLOSE_CURSOR | Closes given cursor and frees memory. |
| LAST_ERROR_POSITION | Returns byte offset in the SQL statement text where the error occurred. |
| LAST_ROW_COUNT | Returns cumulative count of the number of rows fetched. |
| LAST_ROW_ID | Returns ROWID of last row processed. |
| LAST_SQL_ FUNCTION_CODE | Returns SQL function code for statement. |

## Eliminate duplicate rows from table

```
Delete from EMP a
Where a.rowid <> (select min (b.rowid) from emp b where a.empno =
b.empno);
```

**OR**

```
Delete from EMP a
Where a.rowid <> (select max (b.rowid) from emp b where a.empno =
b.empno);
```

## Tree-Structured Queries

Tree-structured queries are definitely non-relational (enough to kill Codd and make him roll in his grave). Also, this feature is not often found in other database offerings.
The SCOTT/TIGER database schema contains a table EMP with a self-referencing relation (EMPNO and MGR columns). This table is perfect for tesing and demonstrating tree-structured queries as the MGR column contains the employee number of the "current" employee's boss.

The LEVEL pseudo-column is an indication of how deep in the tree one is. Oracle can handle queries with a depth of up to 255 levels. Look at this example:

```
Select LEVEL, EMPNO, ENAME, MGR
  From EMP
Connect by prior EMPNO = MGR
  Start with MGR is NULL;
```

One can produce an indented report by using the level number to substring or lpad() a series of spaces, and concatenate that to the string. Look at this example:

```
Select lpad(' ', LEVEL * 2) || ENAME ........
```

One uses the "start with" clause to specify the start of the tree. More than one record can match the starting condition. One disadvantage of having a "connect by prior" clause is that you cannot perform a join to other tables. The "connect by prior" clause is rarely implemented in the other database offerings. Trying to do this programmatically is difficult, as one has to do the top-level query first, then, for each of the records open a cursor to look for child nodes.
One way of working around this is to use PL/SQL, open the driving cursor with the "connect by prior" statement, and the select matching records from other tables on a row-by-row basis, inserting the results into a temporary table for later retrieval.

## Locking in Forms. Possible to defer locking of records until commit

Yes, there is a block property called 'Locking Mode', which can be either 'Immediate' or 'Delayed'. If you set it to 'Delayed', then a record in a block is not locked until commit-time, specifically until changes are 'posted'.

At that time, the current forms value is compared to the DB value before issuing the update statement. The user will get an error if the record has been updated since it was fetched.

'Locking Mode' can be changed at runtime in a trigger with the Set_Block_Property Built-in.

## Locking in Oracle

Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource - either user objects (such as tables and rows) or system objects not visible to users (such as shared data structures in memory and data dictionary rows). In all cases, Oracle automatically obtains necessary locks when executing SQL statements, so users need not be concerned with such details. Oracle automatically uses the lowest applicable level of restrictiveness to provide the

highest degree of data concurrency yet also provide fail-safe data integrity. Oracle also allows the user to lock data manually.
Oracle uses two modes of locking in a multi-user database:
**Exclusive lock mode:** Prevents the associates resource from being shared. This lock mode is obtained to modify data. The first transaction to lock a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.
**Share lock mode:** Allows the associated resource to be shared, depending on the operations involved. Multiple users reading data can share the data, holding share locks to prevent concurrent access by a writer (who needs an exclusive lock). Several transactions can acquire share locks on the same resource.
**Types of Locks**
Oracle automatically uses different types of locks to control concurrent access to data and to prevent destructive interaction between users. Oracle automatically locks a resource on behalf of a transaction to prevent other transactions from doing something also requiring exclusive access to the same resource. The lock is released automatically when some event occurs so that the transaction no longer requires the resource.
**DML locks (data locks):** DML locks protect data. For example, table locks lock entire tables, row locks lock-selected rows.
**DDL locks (dictionary locks):** DDL locks protect the structure of schema objects - for example, the definitions of tables and views.
**Internal locks and latches:** Internal locks and latches protect internal database structures such as datafiles. Internal locks and latches are entirely automatic.
**Distributed locks:** Distributed locks ensure that the data and other resources distributed among the various instances of an Oracle Parallel Server remain consistent. Distributed locks are held by instances rather than transactions. They communicate the current status of a resource among the instances of an Oracle Parallel Server.
P**arallel cache management (PCM) locks:** Parallel cache management locks are distributed locks that cover one or more data blocks (table or index blocks) in the buffer cache. PCM locks do not lock any rows on behalf of transactions.

## Joins

Join is the process of combining data from two or more tables using matching columns. This relational computing feature consolidates multiple data tables for use in a single report.
The SQL JOIN statement is used to combine the data contained in two relational database tables based upon a common attribute.
**Different Joins**
**Equi-Join**
A join statement that uses an equivalency operation (i.e.: colA = colB). The converse of an equijoin is a nonequijoin operation. In the Equi-Join two (or more)

tables are linked via a common domain attribute. This is the most common form of joining used in relational data manipulation.

**Outer Join**

The Outer-Join is the opposite of an Equi-Join. It searches for records that exist outside of the specified Join condition. The (+) symbol is the Outer Join operator which causes the join condition to be inverted.

**Self Join**

A join in which a table is joined with itself. Sometimes we need to join a table to itself in order to search for the correct data.

**Nonequijoin**

A join statement that does not use an equality operation (i.e: colA <> colB). The converse of a nonequijoin is a equijoin.

## Types of Cursors

There are two types of cursors

**Implicit**

Implicit cursor will made automatically by oracle system it open fetch and close it automatically

**Explicit**

In explicit cursors you have make it forcefully like: -

```
Cursor C1 is select ename from emp;
```

## When 'Where Current of' clause is used.

This clause refers to the latest row processed by the FETCH statement associated with the cursor identified by cursor_name. The cursor must be FOR UPDATE and must be open and positioned on a row.

If the cursor is not open, the CURRENT OF clause causes an error. If the cursor is open, but no rows have been fetched or the last fetch returned no rows, PL/SQL raises the predefined exception NO_DATA_FOUND.

WHERE CURRENT is used as a reference to the current row when using a cursor to UPDATE or DELETE the current row.

## Different ways in which block can be built

Table
Views
Procedure
Ref cursors
Transactional triggers.
From clause of a query

## Difference in Parameter Form & Report Parameter Form.

In Report Parameter form all values are lost when report is closed. Also we can't use Global Variables through Report Parameter Form.

## 'POST' in forms. Different from Commit

Writes data in the form to the database, but does not perform a database commit. Form Builder first validates the form. If there are changes to post to the database, for each block in the form Form Builder writes deletes, inserts, and updates to the database.
Any data that you post to the database is committed to the database by the next COMMIT_FORM that executes during the current Runform session. Alternatively, this data can be rolled back by the next CLEAR_FORM
During a default commit operation, Form Builder issues the SQL statements necessary to update, delete, or insert records that have been marked in the form as changed, deleted, or inserted. Form Builder then issues the COMMIT statement to commit these transactions in the database.
Posting consists of writing updates, deletions, and insertions in the form to the database, but not committing these transactions to the database. You can explicitly cause Form Builder to post without committing by executing the POST built-in procedure.
When an application executes the POST procedure, Form Builder does all of the default validation and commits processing, but does not issue the COMMIT statement to finalize these transactions.
Since posted transactions have been written to the database, Form Builder does not have to maintain the status of the affected records across called forms. More importantly, because these transactions have not been committed, they can be rolled back programmatically. You can take advantage of this functionality to manage transactions across called forms.

## How to use 2 different tables at runtime In my query or Main Block

Set_block_Property built in Query_Data_source_name, but you have to give alias to the items of block and Select statement you have to do same.

## Types of Views can block be based on

Updateable view
INSTEAD OF trigger view
Non-updateable view

## Default Validations done by forms

Datatype
Range
Lov for validation
Required
Format mask

## Copying/Referencing (Difference)

When you copy an object, you create a new and separate instance of the object, and any objects owned by the object being copied are always copied automatically.
Referencing items indirectly allows you to write more generic, reusable code. By using variables in place of actual item names, you can write a subprogram that can operate on any item whose name has been assigned to the indicated variable. Also, using indirect reference is mandatory when you refer to the value of a form bind variable (item, parameter, global variable) in PL/SQL that you write in a library or a menu module. Because libraries, menus, and forms are separate application modules, you cannot refer directly to the value of a form item in a menu-item command or library procedure. You can reference items indirectly with the NAME_IN and COPY built-in subprograms.

## Name-In/Copy (Difference)

You can reference items indirectly with the NAME_IN and COPY built-in subprograms.
The NAME_IN function returns the contents of an indicated variable or item. Use the NAME_IN function to get the value of an item without referring to the item directly. The return value is always a character string. To use NAME_IN for a DATE or NUMBER item, convert the string to the desired data type with the appropriate conversion function The NAME_IN function cannot return the contents of a global or local variable. PL/SQL triggers that will be executed in enter-query mode, you must use NAME_IN rather than normal bind-variable notation to access values in the data-block. (This is because the end-user might type relational operators into the item, producing a value, which is not in a form that can be processed by PL/SQL.) If you nest the NAME_IN function, Form Builder evaluates the individual NAME_IN functions from the innermost one to the outermost one.
Copy copies a value from one item or variable into another item or global variable. Use specifically to write a value into an item that is referenced through the NAME_IN built-in. COPY exists for two reasons:

- You cannot use standard PL/SQL syntax to set a referenced item equal to a value.
- You might intend to programmatically place characters such as relational operators in NUMBER and DATE fields while a form is in Enter Query mode.

The COPY procedure assigns an indicated value to an indicated variable or item. Unlike standard PL/SQL assignment, however, using the COPY procedure allows you to indirectly reference the item whose value is being set.
COPY can be used with the NAME_IN function to assign a value to an item whose name is stored in a reference variable or item.
No validation is performed on a value copied to a text item. However, for all other types of items, standard validation checks are performed on the copied value.
COPY (source VARCHAR2, destination VARCHAR2);
NAME_IN (variable_name VARCHAR2);

Kinds of variables in Oracle Forms

## What is Concurrency

Concurrency is allowing simultaneous access of same data by different users. Locks useful for accessing the database are
a) Exclusive
The exclusive lock is useful for locking the row when an insert, update or delete is being done. This lock should not be applied when we do only select from the row.
b) Share lock
We can do the table as Share_Lock as many share_locks can be put on the same resource.

## Deadlock

A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function. A deadlock can arise when two or more users wait for data locked by each other.
The oracle server automatically detects and resolves deadlocks by rolling back the statement that detected the deadlock. Deadlocks most often occur when transactions explicitly override the default locking of the oracle server. A deadlock situation is recorded in a trace file in the USER_DUMP_DEST directory. One has to monitor the trace files for deadlock errors to determine if there are problems with the application. The trace file contains the row IDs of the locking rows.

## Varray/Nested Table (Difference)

**Varrays**

An array is an ordered set of data elements.

All elements of a given array are of the same datatype.

Each element has an index, which is a number corresponding to the element's position in the array.

The number of elements in an array is the size of the array.

Oracle allows arrays to be of variable size, which is why they are called VARRAYs. You must specify a maximum size when you declare the array type.

Creating an array type does not allocate space. It defines a datatype, which you can use as

The datatype of a column of a relational table.

An object type attribute

A PL/SQL variable, parameter, or function return type.

A VARRAY is normally stored in line, that is, in the same tablespace as the other data in its row.

**Nested Tables**

A nested table is an unordered set of data elements, all of the same datatype. It has a single column, and the type of that column is a built-in type or an object type. If an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type.

A table type definition does not allocate space. It defines a type, which you can use as

The datatype of a column of a relational table.

An object type attribute.

A PL/SQL variable, parameter, or function return type

When a table type appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table.

**Differences between Nested Tables and Varrays**

Nested tables differ from varrays in the following ways:

Varrays have a maximum size, but nested tables do not.

Varrays are always dense, but nested tables can be sparse. So, you can delete individual elements from a nested table but not from a varray.

Oracle stores varray data in-line (in the same tablespace). But, Oracle stores nested table data out-of-line in a store table, which is a system-generated database table associated with the nested table.

When stored in the database, varrays retain their ordering and subscripts, but nested tables do not.

## 2 modes of isolation for form module

Specifies whether or not transactions in a session will be serializable. If Isolation Mode has the value Serializable, the end user sees a consistent view of the database for the entire length of the transaction, regardless of updates committed by other users from other sessions. If the end user queries and changes a row, and a second user updates and commits the same row from another session, the first user sees Oracle error (ORA-08177: Cannot serialize access.).

Read committed (default)
Serialize

The serialize works in conjunction with the block's locking property set as delayed. The basic idea behind serialize is that the user sees a consistent set of rows throughout the entire runtime session, inspite of any row being changed by another user of different session.
This is mainly useful when the application access a very large databases and very few users use the appln so that the chances of users waiting for locked records is less.

## What is Ref Cursor

A Ref Cursor is a pointer to a server side cursor variable. The stored procedure returns a reference to a cursor that is open and populated by a select statement to be used as a block Datasource
REF cursors hold cursors in the same way that VARCHAR2 variables hold strings.
This is an added feature that comes with PL/SQL v2.2. A REF cursor allows a cursor to be opened on the server and passed to the client as a unit rather than one row at a time. One can use a Ref cursor as a target of assignments and can be passed as parameters to the Program Units. Ref cursors are opened with an OPEN FOR statement and in all other ways, they are the same as regular cursors.

## Different States/Attributes of Constraints

A constraint is **deferred** if the system checks that it is satisfied only on commit. If a deferred constraint is violated, then commit causes the transaction to roll back.
If a constraint is **immediate** (not deferred), then it is checked at the end of each statement. If it is violated, the statement is rolled back immediately.
Constraints can be defined as either **deferrable** or **not deferrable**, and either **initially deferred** or **initially immediate**. These attributes can be different for each

constraint. Constraints can be added, dropped, enabled, disabled, or validated. You can also modify a constraint's attributes.

**Constraint States**

ENABLE ensures that all incoming data conforms to the constraint

DISABLE allows incoming data, regardless of whether it conforms to the constraint

VALIDATE ensures that existing data conforms to the constraint

NOVALIDATE means that some existing data may not conform to the constraint

ENABLE VALIDATE is the same as ENABLE. The constraint is checked and is guaranteed to hold for all rows. If a constraint is in this state, then all data in the table is guaranteed to adhere to the constraint. In addition, this state prevents any invalid data from being entered. This is the normal state of operation of a constraint for online transaction processing.

ENABLE NOVALIDATE means that the constraint is checked, but it does not have to be true for all rows. This allows existing rows to violate the constraint, while ensuring that all new or modified rows are valid. If a constraint is in this state, the new data that violates the constraint cannot be entered. However the table can contain data that is invalid - that is, data that violates the constraint. This is usually an intermediate stage that ensures that all new data is checked before being accepted into the table.

DISABLE NOVALIDATE is the same as DISABLE. The constraint is not checked and is not necessarily true. A constraint that is disabled novalidate is not checked, even though the constraint definition is still stored in the data dictionary. Data in the table, as well as the new data that is entered or updated, may not conform to the rules defined by the constraint. This is the normal state of operation of a constraint for online transaction processing.

DISABLE VALIDATE disables the constraint, drops the index on the constraint, and disallows any modification of the constrained columns. If the constraint is in this state, them any modification of the constrained columns is not allowed. In addition, the index on the constraint is dropped and the constraint is disabled.

## Different Actions of Constraints

**Update and Delete No Action**
The No Action (default) option specifies that referenced key values cannot be updated or deleted if the resulting data would violate a referential integrity

constraint. For example, if a primary key value is referenced by a value in the foreign key, then the referenced primary key value cannot be deleted because of the dependent data.

**Delete Cascade**

A **delete cascades** when rows containing referenced key values are deleted, causing all rows in child tables with dependent foreign key values to also be deleted. For example, if a row in a parent table is deleted, and this row's primary key value is referenced by one or more foreign key values in a child table, then the rows in the child table that reference the primary key value are also deleted from the child table.

**Delete Set Null**

A delete **sets null** when rows containing referenced key values are deleted, causing all rows in child tables with dependent foreign key values to set those values to null. For example, if employee_id references manager_id in the TMP table, then deleting a manager causes the rows for all employees working for that manager to have their manager_id value set to null.

## What is Savepoint

**Save Point:** For long transactions that contain many SQL statements, intermediate markers or save points can be declared. Save points can be used to divide a transaction into smaller parts. You can rollback or commit up to that save point.

## When to use Procedure/Function/Package/PLL

These different forms of procedures will produce different performance results depending on what you are trying to achieve. The database procedure will be invoked on the server side and will perform all of the processing there.

This may help reduce network traffic, but may overload the server if many clients are doing this.

Local form and library procedures are quite similar in that they are both stored and run on the client with any SQL statements being passed to the server.

The local procedures are typically faster on execution because they are actually part of the .fmx file, but may use more memory and have a longer startup time when the .fmx file is initially invoked.

Library procedures may be better in terms of overall memory as procedures are only loaded into memory in 4K chunks when they are required. However, once loaded they are read from memory.

They have the additional advantage that the library can be attached to a number of forms and the code within the library is then available to the forms.

If the code within the library procedures is altered, the form does not require re-generation. That can be a *big* advantage.

## Anchoring in Reports

Anchors are used to determine the vertical and horizontal positioning of a child object relative to its parent. The end of the anchor with a symbol on it is attached to the parent object. Since the size of some layout objects may change when the report runs (and data is actually fetched), you need anchors to define where you want objects to appear relative to one another. An anchor defines the relative position of an object to the object to which it is anchored. Positioning is based on the size of the objects after the data has been fetched rather than on their size in the editor. It should also be noted that the position of the object in the Layout editor effects the final position in the report output. Any physical offset in the layout is incorporated into the percentage position specified in the Anchor property sheet.

## What is Check option/Force View.

**FORCE**: creates the view regardless of whether the view's base tables exist or the owner of the schema containing the view has privileges on them. Note that both of these conditions must be true before any SELECT, INSERT, UPDATE, or DELETE statements can be issued against the view.
**WITH CHECK OPTION**: specifies that inserts and updates performed through the view must result in rows that the view query can select. The CHECK OPTION cannot make this guarantee if there is a subquery in the query of this view or any view on which this view is based.

## Different Optimization Techniques

**Optimizer:**
Optimization is the process of choosing the most efficient way to execute an SQL statement. When tuning SQL, it's very important to understand WHY and WHEN a particular type of optimization should be used.
When a SQL Select, Update, Insert or Delete statement is processed, Oracle must access the data referenced by the statement. The Optimizer portion of Oracle is used to determine an efficient path to reference data so that minimal I/O and Processing time is required. For statement execution the Optimizer formulates execution plans and chooses the most efficient plan before executing a statement.
The Optimizer uses one of two techniques:
**Rule Based Approach:** Using the rule-based approach, the optimizer chooses an execution plan based on the access paths available and the ranks of these access

paths. If there is more than one way to execute an SQL statement, the rule-based approach always uses the operation with the lower rank. Usually, operations of lower rank execute faster than those associated with constructs of higher rank. The rule-based optimizer is no longer being enhanced by Oracle. The rule-based optimizer (RBO) Uses a fixed ranking, therefore, it's essential to list the correct Table Order in the FROM clause. Normally, the (LAST or Right Most) Table listed in the FROM clause is the driving table. The data distribution and number of records is not taken into account. The RBO execution plan can't be changed by hints.

**Cost Based Approach:** Using the cost-based approach, the optimizer determines which execution plan is most efficient by considering available access paths and factoring in information based on statistics in the data dictionary for the schema objects (tables, clusters or indexes) accessed by the statement. The cost-based approach also considers hints, or optimization suggestions placed in a Comment in the statement. By default, the goal of the cost-based approach is the best throughput, or minimal resource use necessary to process all rows accessed by the statement. The cost-based approach uses statistics to estimate the cost of each execution plan. These statistics quantify the data distribution and storage characteristics of tables, columns, indexes, and partitions. These statistics are generated by using the ANALYZE command. The cost-based optimizer was built to make tuning easier by choosing the best paths for poorly written queries. Normally, the (FIRST or Left Most) Table listed in the FROM clause is the driving table. Just opposite of RBO. The CBO decisions are only as good as the type and frequency of analyzes made on the Table and associated Indexes. To Tune SQL Queries using CBO you must also have an understanding of Table Access Methods, Table Join Methods and Hints.

**The DRIVING Table**

In Oracle the cost-based approach uses various factors in determining which table should be the Driving Table (the table that has the most "restricted" rows). It is the table that drives the query in a multi-table join query. The Driving Table should be listed First in Cost Based and listed Last in Rule Based Optimization. The best thing to remember is that you have control over which table will drive a query through the effective use of the ORDERED hint when using the CBO. The easiest way to determine if your query is using the correct Driving Table is to set the SQL Select to find Zero records. If the return is slow you know the Driving Table is incorrect!

## Difference b/w Dbms_Alert & Dbms_Pipe.

The DBMS_ALERT package provides support for the asynchronous notification of database events (alerts). By appropriate use of this package and database triggers, an application can cause itself to be notified whenever values of interest in the database are changed. Alerts are transaction-based. This means that the waiting session does not get alerted until the transaction signalling the alert commits.There can be any number of concurrent signallers of a given alert, and there can be any number of concurrent waiters on a given alert.

The DBMS_PIPE package lets two or more sessions in the same instance communicate. Oracle pipes are similar in concept to the pipes used in UNIX, but Oracle pipes are not implemented using the operating system pipe mechanisms.
Information sent through Oracle pipes is buffered in the system global area (SGA). All information in pipes is lost when the instance is shut down.
Depending upon your security requirements, you may choose to use either a *public* or a *private* pipe.

## Different types of Data types. Composite Data type

Each literal or column value has a data type. The value's data types associate a fixed set of properties with the value.
Varchar2 (size), number (p, s), char (size), long, raw, long raw, Boolean, smallint, binary_integer, double, decimal, dec, etc.,
The data types are scalar (all the above)
**Composite**: tables and record

### 'HINTS'

Hints are **suggestions** that you give the optimizer for optimizing a SQL statement. Hints allow you to make decisions usually made by the optimizer. You can use **hints** to specify:

Optimization approach for a SQL statement
Goal of the cost-based approach for a SQL statement
Access path for a table accessed by the statement
Join order for a join statement
A join operation in a join statement

If you incorrectly specify a hint in any way, it will become a comment, and will be ignored. You can also use more than one hint at a time, although this may cause some or all of the hints to be ignored. When you use an alias on a table that you want to use in a hint, you must specify the **alias -** NOT the **table name** in the hint.
**Two MOST Used Hints**
**The Ordered Hint –** The ORDERED hint causes tables to be accessed in a particular order, based on the order of the tables in the FROM clause. The ORDERED hint guarantees the Driving Order of the Table Joins. The ORDERED hint is one of the most **powerful** hints available. When a multi-table join is slow and you don't know what to do … this should be the **first hint** you should try!
**The RULE Hint –**The RULE Hint will force the optimizer to use Rule-Based Optimization.
**Hints NOT to Use**
Don't HINT to a specific **INDEX-**Index names can and do change frequently. DBA's restructure Indexes and change index names.

Never use the **CACHE** Hint- Tables and Indexes have been assigned to specific storage areas. Please don't cache anything - leave that up to the DBA's.
Never use the **APPEND** Hint- unless, the database uses parallel processing. This Hint will cause the table to be appended in Parallel Write Mode. If your Database is not in Parallel Mode - It will Trash the table

## Privileges/Grants

Previleges are the right to execute a particular type of SQL statements.
e.g.: Right to Connect, Right to create, Right to resource
Grants are given to the objects so that the object might be accessed accordingly. The grant has to be given by the owner of the object.

## Clusters

**Clusters** are an optional method of storing table data. **Clusters** are groups of one or more tables physically stored together because they share common columns and are often used together. Because related rows are physically stored together, disk access time improves. Like indexes, clusters do not affect application design. Whether or not a table is part of a cluster is transparent to users and to applications. Data stored in a clustered table is accessed by SQL in the same way as data stored in a non-clustered table.

**Benefits:**

- Disk I/O is reduced for joins of clustered tables.
- Access time improves for joins of clustered tables.
- In a cluster, a **cluster key value** is the value of the cluster key columns for a particular row. Each cluster key value is stored only once each in the cluster and the cluster index, no matter how many rows of different tables contain the value. Therefore, less storage is required to store related table and index data in a cluster than is necessary in non-clustered table format.

## Do_Key

Executes the key trigger that corresponds to the specified built-in subprogram. If no such key trigger exists, then the specified subprogram executes. This behavior is analogous to pressing the corresponding function key. PROCEDURE DO_KEY (built-in_subprogram_name VARCHAR2);

DO_KEY accepts built-in names only, not key names: DO_KEY (ENTER_QUERY). To accept a specific key name, use the EXECUTE_TRIGGER built-in: EXECUTE_TRIGGER ('KEY_F11').

## Parallel Query

Without the parallel query feature, the processing of a SQL statement is always performed by a single server process. With the parallel query feature, multiple processes can work together simultaneously to process a single SQL statement. This capability is called *parallel query processing.* By dividing the work necessary to process a statement among multiple server processes, the Oracle Server can process the statement more quickly than if only a single server process processed it.
The parallel query feature can dramatically improve performance for data-intensive operations associated with decision support applications or very large database environments. Symmetric multiprocessing (SMP), clustered, or massively parallel systems gain the largest performance benefits from the parallel query feature because query processing can be effectively split up among many CPUs on a single system.
It is important to note that the query is parallelized dynamically at execution time. Thus, if the distribution or location of the data changes, Oracle automatically adapts to optimize the parallelization for each execution of a SQL statement.
The parallel query feature helps systems scale in performance when adding hardware resources. If your system's CPUs and disk controllers are already heavily loaded, you need to alleviate the system's load before using the parallel query feature to improve performance. Chapter 18, "Parallel Query Tuning" describes how your system can achieve the best performance with the parallel query feature.
The Oracle Server can use parallel query processing for any of these statements:

SELECT statements

Subqueries in UPDATE, INSERT, and DELETE statements

CREATE TABLE ... AS SELECT statements

CREATE INDEX

When a statement is parsed, the optimizer determines the execution plan of a statement. After the optimizer determines the execution plan of a statement, the query coordinator process determines the parallelization method of the statement. *Parallelization* is the process by which the query coordinator determines which operations can be performed in parallel and then enlists query server processes to execute the statement. This section tells you how the query coordinator process decides to parallelize a statement and how the user can specify how many query server processes can be assigned to each operation in an execution plan (that is, the *degree of parallelism*).

## 3 Tier Architecture

**Two –tier Architecture**: A two-tier architecture is where a client talks directly to a server with no intervening server. It is typically used in small environments. Here you can add more users to the server. This approach will usually result in an ineffective system, as the server becomes overwhelmed to properly scale the users.

Eg. Oracle Products SQL*NET is the middleware between the client/server systems.

**Three – tier Architecture**: A three – tier architecture introduces a server or an agent between the client and server. The role of the agent is manifolds. It can provide translation services (as in adapting a legacy application on a mainframe to a client/server environment), metering services (as in acting as a transaction monitor to limit the number of simultaneous requests to a given server), or intelligent agent services (as in mapping a request to a number of different servers, collating the results and returning a single response to the client.

E.g. The Application Programming Interface (API) like ODBC is the middleware between the client and server development in this architecture.
The primary advantage of using API in client application is that the resulting application can then use any backend database servers rather than specific server.
The disadvantage of using API is the "Least common denominators" of all tools and database servers that support the standard. Therefore, when you develop a client application, you might not be able to take advantage of all the unique and special features that any one database servers offers.

## Indexes

An Index is database structure that is used by the server to find a row in a table quickly. It is composed of a key value (the column in a row) and the rowid. Indexes are used to improve the performance of queries. It provides a faster access route to table data. Properly used, indexes are the primary means to reduce disk I/O.

- Index a column only if you frequently want to retrieve less than 15 percent of the rows in a large table.
- Do not index small tables with few rows.
- If the column data contains many nulls and you frequently search for these nulls, do not create an index on this column.
- Likewise if the column contains very similar data, don't create an index on that column.

P/L Sql Library Vs Object Library

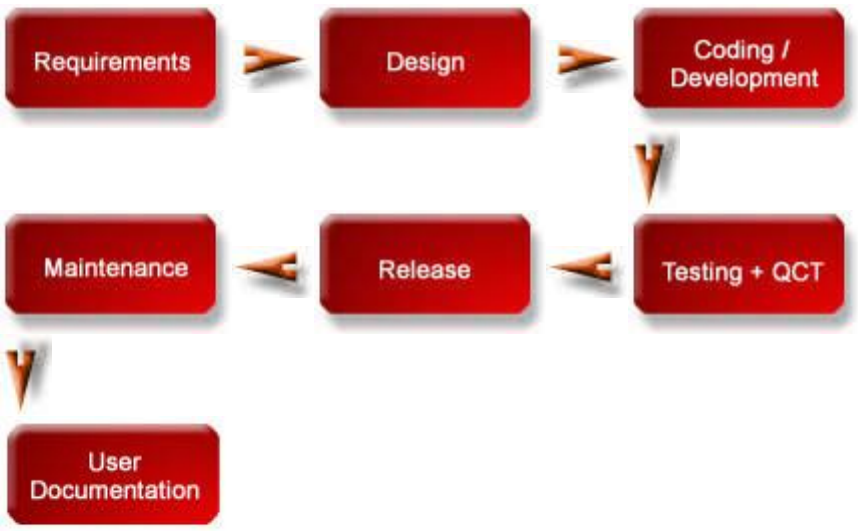
Types of Parameters passing option to procedure & function

## Partitioning

Partitioning addresses the key problem of supporting very large tables and indexes by allowing users to decompose them into smaller and more manageable pieces called partitions. All partitions of a table or index have the same logical attributes, although their physical attributes can be different. For example, all partitions in a table share the same column and constraint definitions; and all partitions in an index share the same index columns. However, storage specifications and other physical attributes such as PCTFREE, PCTUSED, INITRANS, and MAXTRANS can vary for different partitions of the same table or index. Each partition is stored in a separate segment. Optionally, you can store each partition in a separate tablespace, which has the following advantages:

You can contain the impact of damaged data.
You can back up and recover each partition independently.
You can balance I/O load by mapping partitions to disk drives.

## SDLC



| | |
|---|---|
| **Requirements:** | Functional Specification Document |
| | Gap Analysis Document. |
| **Design:** | Architectural Design Document |
| | Combined Design Document |
| | Detailed Design Document |
| | Impact Analysis Document |
| | Program Specifications |

|  |  |
|---|---|
| **Coding/Development**: | Program Submission |
|  | Code Header |
| **Testing/QCT:** | QCT Acceptance |
|  | Project Analysis Report |
|  | QCT Weekly Progress Report |
|  | Quality Control Test Plan |
|  | System Test Plan |
|  | Unit Test Plan |
|  | Senior Management Review (QCT) |
| **Release:** | Configuration Item Release |
|  | Customer Release Note |
| **Maintenance** | Annual Technical Support |
|  | Maintenance Plan |
|  | Progress Report |
|  | Project Board Review (Support) |
|  | Project Schedule (Support Projects) |
| **Documentation:** | User Manuals |

**Various lifecycle models descriptions**

- Waterfall
- Spiral
- Prototyping (Evolutionary Prototyping)
- Incremental

## Database Procedure/Form Procedure/Libraries

These different forms of procedures will produce different performance results depending on what you are trying to achieve. The database procedure will be invoked on the server side and will perform all of the processing there. This may help reduce network traffic, but may overload the server if many clients are doing this.

Local form and library procedures are quite similar in that they are both stored and run on the client with any SQL statements being passed to the server.

The local procedures are typically faster on execution because they are actually part of the .fmx file, but may use more memory and have a longer startup time when the .fmx file is initially invoked.

Library procedures may be better in terms of overall memory as procedures are only loaded into memory in 4K chunks when they are required. However, once loaded they are read from memory.

They have the additional advantage that the library can be attached to a number of forms and the code within the library is then available to the forms.

If the code within the library procedures is altered, the form does not require re-generation. That can be a *big* advantage.

## Overloading

Enables you to use same name for different subprograms inside p/l sql block, a subprogram or a package.
Requires formal parameters of subprogram to differ in number, order or datatype
Enables you to build more flexibility because a user or application is not restricted by specific datatype or number of formal parameters.
Only local or packaged subprograms can be overloaded. You cannot overload stand-alone subprograms.

## Types of Menus

Menus are lists of items that end users use to select specific functions or operations. Each item on a menu represents a command or a submenu.
**Form Menus**: A form menu typically includes standard form-level commands for navigation, editing, and database interaction. Every form runs with one of the following form menus: default, custom, none.
**Menu Toolbars**: If you create any type of custom form menu module for your form, you also can derive a menu toolbar from that form menu. A menu toolbar displays selected menu items (from the current form menu) as icons on a horizontal or vertical ribbon.
**Popup Menus**: In addition to one form menu, each form can include any number of popup menus. Popup menus are context-sensitive menus attached to individual canvases and items within a form. You can attach a single popup menu to one or more canvases or items. E ach canvas or item can support one and only one popup menu
**Tear-off**: A tear-off menu is a menu that end users can drag away from the menu bar and reposition elsewhere on the screen. You can enable tear-off functionality for any menu, provided your window-manager supports this feature.
**Debugger menu**: The Debugger menu includes three menus: View, Debug, and Navigator.

## Tablespace, Datafiles

**Table Space:** Tablespace is a logical area of data storage. It is made up of one or more datafiles.
A table space is an area of disk, comprised of one or more disk files. A tablespace can contain many tables, indexes, or clusters. Because a table space has a fixed size, it can get full as rows are added to its tables. When this happens, the table space can

be expanded by someone who has DBA authority. When a database is created two table spaces are created.
a) System Table space: This data file stores all the tables related to the system and dba tables
b) User Table space: This data file stores all the user related tables
We should have separate table spaces for storing the tables and indexes so that the access is fast.
A tablespace can be **online** (accessible) or **offline** (not accessible). A tablespace is generally online, so that users can access the information in the tablespace. However, sometimes a tablespace is taken offline to make a portion of the database unavailable while allowing normal access to the remainder of the database. This makes many administrative tasks easier to perform.
**Data file:** Oracle database has one or more physical datafiles (a datafile is a unit of physical data storage). A database data i.e. tables, indexes, etc. are stored in datafiles.
A datafile is associated with only one tablespace.
Once created, a datafile cannot change in size, (in release 7.1 datafiles can be appended but existing datafile size cannot be altered, however, later versions support change in existing datafile size).
Datafiles can have certain characteristics set to let them automatically extend when the database runs out of space.
One or more datafiles form a logical unit of database storage called a tablespace

Rollback Segments

## Use of Control File

**Control file:** A control file records the physical structure of the database. It records database name, names and locations of a databases datafiles and redo log files, time stamps of database creation.
Every time an **instance** of an Oracle database is started, its control file identifies the database and redo log files that must be opened for database operation to proceed. If the physical makeup of the database is altered (for example, if a new datafile or redo log file is created), then the control file is automatically modified by Oracle to reflect the change. A control file is also used in database recovery.

SQL Vs P/L SQL

## Pre-Query/Pre-Select in Forms (Difference)

PRE-QUERY fires just before Forms begins to prepare the SQL statement that will be required to query the appropriate records from the database in response to the query criteria that have been entered into the example record.

It is the designer’s last chance to "deposit" query criteria into the items in the block as if they had been entered into the example record by the operator.

PRE-SELECT fires after the PRE-QUERY trigger has fired, and just before the actual, physical construction of the SELECT statement begins. It would typically be used in conjunction with an ON-SELECT trigger in which the
designer would be supplanting the normal construction of the SELECT statement with a user exit of his own to construct and prepare the SELECT statement for execution (perhaps against a foreign datasource).

## Sequence of trigger when Form is opened & closed

**Run Form**

| | |
|---|---|
| 1. Pre-Logon | Form |
| 2. On-Logon | Form |
| 3. Post-Logon | Form |
| 4. Pre-Form | Form |
| 5. When-Create-Record | Block |
| 6. Pre-Block | Block |
| 7. Pre-Record | Block |
| 8. Pre-Text-Item | Item |
| 9. When-New-Form-Instance | Form |
| 10. When-New-Block-Instance | Block |
| 11. When-New-Record-Instance | Block |
| 12. When-New-Item-Instance | Item |

**Exit**

| | |
|---|---|
| 1. Post-Text-Item | Item |
| 2. Post-Record | Block |
| 3. Post-Block | Block |
| 4. Post-Form | Form |
| 5. On-Rollback | Form |
| 6. Pre-Logout | Form |
| 7. On-Logout | Form |
| 8. Post-Logout | Form |

## Sequence of trigger when Navigation from one item to another

Key-next -1
Post Change –1
When validate -1
Post text -1
Pre – text - 2
When new item instance - 2

## Database Link

A database link is a named object that describes a "path" from one database to another. Types are Private Database Link, Public Database Link & Network Database Link.

**Private database link** is created on behalf of a specific user. A private database link can be used only when the owner of the link specifies a global object name in a SQL statement or in the definition of the owner's views or procedures.

**Public database link** is created for the special user group PUBLIC. A public database link can be used when any user in the associated database specifies a global object name in a SQL statement or object definition.

**Network database link** is created and managed by a network domain service. A network database link can be used when any user of any database in the network specifies a global object name in a SQL statement or object definition.

## High Water Mark

High-water mark is an indicator or pointer up to which table or index has ever contained data.

## Bind Variables & Host Variables

Bind variables are used in SQL and PL/SQL statements for holding data or result sets. They are commonly used in SQL statements to optimize statement performance. A statement with a bind variable may be re-executed multiple times without needing to be re-parsed. Their values can be set and referenced in PL/SQL blocks. They can be referenced in SQL statements e.g. SELECT. Except in the VARIABLE and PRINT commands, bind variable references should be prefixed with a colon.

Host variables are the key to communication between your host program and Oracle. Typically, a precompiler program inputs data from a host variable to Oracle, and Oracle outputs data to a host variable in the program. Oracle stores input data in database columns, and stores output data in program host variables.

A host variable can be any arbitrary C expression that resolves to a scalar type. But, a host variable must also be an *lvalue*. Host arrays of most host variables are also supported.

## Define & Accept Command

Use the DEFINE command to explicitly create substitution variables. The DEFINE command can also be used to display the value of a known variable. It shows the variable name, value and type. Using DEFINE with no arguments lists all defined

substitution variables. Any variable that DEFINE lists is said to be defined. A variable may be redefined by repeating the DEFINE command with a different value. The DEFINE command only ever creates variables with type CHAR.
The ACCEPT command always prompts for a variable's value, creating a new variable or replacing an existing one. ACCEPT has advantages over a double ampersand (&&) variable reference that causes a prompt. ACCEPT allows the prompting text to be customized and allows a default value to be specified. ACCEPT does type and format checking. The ACCEPT command understands numbers, strings and dates. If a FORMAT clause is used, SQL*Plus validates the input against the given format. If the input is not valid, you are re-prompted for a value.

## Merge Command

Merge Command can dynamically decide whether to insert data or update current data during a load, depending on whether a corresponding row already exists in a table.

### What is Cursor

A Cursor is a handle (a name or pointer) for the memory associated with a specific statement.

### Index by table methods

## Object Type

Object types are abstractions of the real-world entities
An object type has a name, which serves to identify the object type uniquely within that schema. It has attributes, which model the structure and state of the real world entity. Attributes are built-in types or other user-defined types.
It has methods, which are functions or procedures written in PL/SQL and stored in the database, or written in a language like C and stored externally. Methods implement operations the application can perform on the real world entity.
An object type is a template. A structured data unit that matches the template is called an object. An object type has attributes, which reflect the entity's structure, and methods, which implement the operations on the entity. Attributes are defined using built-in types or other object types. Methods are functions or procedures written in PL/SQL or an external language like C and stored in the database. A typical use for an object type is to impose structure on some part of the data kept in the database. For example, an object type named *DataStream* could be used by a cartridge to store large amounts of data in a character LOB (a data type for large objects). This object type has attributes such as an identifier, a name, a date, and so on. A method is a procedure or function that is part of the object type definition and that can operate on the object type data attributes. Such methods are called **member**

**methods**, and they take the keyword MEMBER when you specify them as a component of the object type. The DataStream type definition declares three methods. The first two, DataStreamMin and DataStreamMax, calculate the minimum and maximum values, respectively, in the data stream stored inside the character LOB.

The system implicitly defines a **constructor method** for each object type that you define. The name of the constructor method is the same as the name of the object type. The parameters of the constructor method are exactly the data attributes of the object type, and they occur in the same order as the attribute definition for the object type. At present, only one constructor method can be defined, and thus you cannot define other constructor methods.

## Drill down report

A drill-down report is actually two or more reports working together. The top-level report launches a secondary report that provides more details about the data in the current record. To add flexibility in the selection of details, you can create a button in the top-level report that executes PL/SQL to call the detail report and provide conditions for its execution. The detail report can provide details for a single record, a group of records, or the report as a whole. A report can include buttons with PL/SQL actions attached to them. For example, you could use the packaged procedure SRW.RUN_REPORT to define an action for a button that calls another report. This second (or detail) report appears in its own, modal Runtime Previewer. There is no limit to the number of levels you can drill down in this way.

## Post-Query/Pre-Query (Difference/Usage)

**Pre-Query**

Fires during Execute Query or Count Query processing, just before Form Builder constructs and issues the SELECT statement to identify rows that match the query criteria.

Use a Pre-Query trigger to modify the example record that determines which rows will be identified by the query.

**Post-Query**

When a query is open in the block, the Post-Query trigger fires each time Form Builder fetches a record into a block. The trigger fires once for each record placed on the block's list of records.

Use a Post-Query trigger to perform the following tasks:

- Populate control items or items in other blocks
- Calculate statistics about the records retrieved by a query
- Calculate a running total

When you use a Post-Query trigger to SELECT non-base table values into control items, Form Builder marks each record as CHANGED, and so fires the When-Validate-Item trigger by default.
You can avoid the execution of the When-Validate-Item trigger by explicitly setting the Status property of each record to QUERY in the Post-Query trigger. To set record status programmatically, use SET_RECORD_PROPERTY.

## Methods for passing parameters

**Named** Lists value in order in which parameter is declared
**Positional** List value in arbitrarily order by associating each one with parameter name using =>
**Combination** List first value positionally and remainder using special syntax of named method

## Difference in different version of forms (4.5/5/6i)

Forms 6i New Features
Forms runtime diagnostics
Enhanced PL/SQL editing capabilities and environment
An LOV wizard
A hierarchical tree control
Support for pluggable Java components
Web Enabled

Difference in different version of reports (2.5/3/6i)

## Difference in different version of oracle (7.3/8/8i)

Some new pl/sql enhancements in oracle 8i and 8

EXECUTE IMMEDIATE Command
New Pragmas-serially reusable, autonomous transactions
New Database Triggers
startup,shutdown,logon,logoff,alter,create,drop

Expanded Oracle Supplied Packages

dbms_profiles,dbms_trace,dbms_debug

Extended Security Methodology
Password expire
Password reuse max
Password grace time
Locking user accounts, and unlocking

## What is synonym. Different types of synonyms.

A synonym is a name assigned to a table or a view may thereafter be used to refer it. Creating a synonym does not grant any privileges on referenced objects it just provides at alternative name for objects.
Synonyms can be either private or public. A public synonym is owned by the special user group PUBLIC; every user of a database can access it. A private synonym is in the schema of a specific user, who has control over its availability to others. Individual users can see only the public synonyms and their private synonyms.
A view is a look on a table's data with restrictions or specific data - a subset of the table's data.
A synonym is a way to share any object from your scheme with other schemes.
A synonym is created to avoid using the user.object standard while calling to other scheme objects.
Any privilege you have on the object, will affect the synonym if you grant select only - no insert or update is allowed using synonym (a synonym is a pointer to an object).
A view is just a stored query. So any user with the appropriate privileges can query, insert, update, delete the data. The synonym is a pointer to the base table itself. So any user with the appropriate privileges can query, insert, update, delete the data, and also perform DDL on the base table, such as adding/dropping columns. Of course, all that can be controlled by properly managing privileges anyway.

## When-New-Record-Instance/When-Validate-Record

**When-New-Record-Instance**
Perform an action immediately after the input focus moves to an item in a different record. If the new record is in a different block, fires after the When-New-Block-Instance trigger, but before the When-New-Item-Instance trigger.
Specifically, it fires after navigation to an item in a record, when Form Builder is ready to accept input in a record that is different than the record that previously had input focus.
Use a When-New-Record-Instance trigger to perform an action every time

Form Builder instantiates a new record. For example, when an operator presses [Down] to scroll through a set of records, Form Builder fires this trigger each time the input focus moves to the next record, in other words, each time Form Builder instantiates a new record in the block.

**When-Validate-Record**

It Augments default validation of a record. Fires during the Validate the Record process. Specifically, it fires as the last part of record validation for records with the New or Changed validation status.

Use a When-Validate-Record trigger to supplement Form Builder default record validation processing.

Note that it is possible to write a When-Validate-Record trigger that changes the value of an item in the record that Form Builder is validating. If validation succeeds, Form Builder marks the record and all of the fields as Valid and does not re-validate. While this behavior is necessary to avoid validation loops, it does make it possible for your application to commit an invalid value to the database.

On Failure

If fired as part of validation initiated by navigation, navigation fails, and the focus remains on the original item.

## Difference in Tabular & Form types of reports

Tabular Means that labels are above fields, and are outside the repeating frame containing the fields.

Form Means that labels are to the left of fields, and are inside the repeating frame containing the fields.

## Post-Text-Item/When-Validate-Item

**Post-Text-Item**

It manipulates an item when Form Builder leaves a text item and navigates to the record level. Fires during the Leave the Item process for a text item. Specifically, this trigger fires when the input focus moves from a text item to any other item.

Use a Post-Text-Item trigger to calculate or change item values. It Fires every time On Failure Navigation fails and focus remains in the text item.

**When-Validate-Item**

It augments default validation of an item.

Fires during the Validate the Item process. Specifically, it fires as the last part of item validation for items with the New or Changed validation status.

Use a When-Validate-Item trigger to supplement Form Builder default item validation processing.

It is possible to write a When-Validate-Item trigger that changes the value of an item that Form Builder is validating. If validation succeeds, Form Builder marks the changed item as Valid and does not re-validate it. While this behavior is necessary to avoid validation loops, it does make it possible for your application to commit an invalid value to the database. On failure if fired as part of validation initiated by navigation, navigation fails, and the focus remains on the original item.

On particular condition disable whole record. How
Multiple Values in one item

## WHEN/CALL Clause in Triggers

The WHEN *trigger_condition* specifies the conditions that must be met for the trigger to fire. Stored functions and object methods are not allowed in the trigger condition.
Optionally, a trigger restriction can be included in the definition of a row trigger by specifying a Boolean SQL expression in a WHEN clause (a WHEN clause cannot be included in the definition of a statement trigger).
If included, the expression in the WHEN clause is evaluated for each row that the trigger affects.
If the expression evaluates to TRUE for a row, the trigger body is fired on behalf of that row.
However, if the expression evaluates to FALSE or NOT TRUE (that is, unknown, as with nulls) for a row, the trigger body is not fired for that row.
The evaluation of the WHEN clause does not have an effect on the execution of the triggering SQL statement (that is, the triggering statement is not rolled back if the expression in a WHEN clause evaluates to FALSE).
For example, in the PRINT_SALARY_CHANGES trigger, the trigger body would not be executed if the new value of EMPNO is zero, NULL, or negative.
In more realistic examples, you might test if one column value is less than another.
The expression in a WHEN clause of a row trigger can include correlation names, which are explained below.
The expression in a WHEN clause must be a SQL expression and cannot include a subquery.
You cannot use a PL/SQL expression (including user-defined functions) in the WHEN clause.

Call Clause is used for calling a Procedure in a Trigger Body

## Data Dependencies

When you create a stored procedure or function, Oracle verifies that the operations it performs are possible based on the schema objects accessed. For example, if a stored procedure contains a SELECT statement that selects columns from a table, Oracle verifies that the table exists and contains the specified columns. If the table is subsequently redefined so that one of its columns does not exist, the stored procedure may not work properly. For this reason, the stored procedure is said to *depend* on the table.

In cases such as this, Oracle automatically manages dependencies among schema objects. After a schema object is redefined, Oracle automatically recompiles all stored procedures and functions in your database that depend on the redefined object the next time they are called. This recompilation allows Oracle to verify that the procedures and functions can still execute properly based on the newly defined object.

Runtime recompilation reduces runtime performance and the possible resulting runtime compilation errors can halt your applications. Follow these measures to avoid runtime recompilation:

Do not redefine schema objects (such as tables, views, and stored procedures and functions) while your production applications are running. Redefining objects causes Oracle to recompile stored procedures and functions that depend on them.

After redefining a schema object, manually recompile dependent procedures, functions, and packages. This measure not only eliminates the performance impact of runtime recompilation, but it also notifies you immediately of compilation errors, allowing you to fix them before production use.

You can manually recompile a procedure, stored function, or package with the COMPILE option of the ALTER PROCEDURE, ALTER FUNCTION, or ALTER PACKAGE command.

Store procedures and functions in packages whenever possible. If a procedure or function is stored in a package, you can modify its definition without causing Oracle to recompile other procedures and functions that call it.

Packages are the most effective way of preventing unnecessary dependency checks from being performed.

The %TYPE attribute provides the datatype of a variable, constant, or column. This attribute is particularly useful when declaring a variable or procedure argument that refers to a column in a database table. The %ROWTYPE attribute is useful if you want to declare a variable to be a record that has the same structure as a row in a table or view, or a row that is returned by a fetch from a cursor.

Dependencies among PL/SQL library units (packages, stored procedures, and stored functions) can be handled either with timestamps or with signatures.

- In the timestamp method, the server sets a timestamp when each library unit is created or recompiled, and the compiled states of its dependent library units contain records of its timestamp. If the parent unit or a relevant schema object is altered, all of its dependent units are marked as invalid and must be recompiled before they can be executed.

- In the signature method, each compiled stored library unit is associated with a signature that identifies its name, the types and modes of its parameters, the number of parameters, and (for a function) the type of the return value. A dependent unit is marked as invalid if it calls a parent unit whose signature has been changed in an incompatible manner.

Oracle dynamically recompiles an invalid view or PL/SQL program unit the next time it is used. Alternatively, you can force the compilation of an invalid view or program unit using the appropriate SQL command with the COMPILE parameter.
Forced compilations are most often used to test for errors when it is known that a dependent view or program unit is invalid, but is not currently being used; therefore, automatic recompilation would not otherwise occur until the view or program unit is executed.

## Top Nth Salary

```
Select min (sal) from EMP a
Where &N > (select count (distinct sal) from emp b where a.sal < b.sal)
Order by sal desc;

Select deptno, min (sal) from EMP a
 Where &n > (select count (distinct sal) from EMP b
          Where a.sal < b.sal and a.deptno = b.deptno)
 Group by deptno
 Order by deptno;

Select empno,sal from (select empno,sal from emp order by sal desc) where
rownum<4

Select empno,sal from EMP a
Where 3 > (select count (distinct sal) from emp b where a.sal < b.sal)
Order by sal desc

Query to return Nth highest salary for each department
----------------------------------------------------
Select deptno, sal from emp a
Where &N > (select count (sal) from emp b where a.sal < b.sal and a.deptno =
b.deptno)
Minus
Select deptno, sal
From emp a
Where &(N-1) > (select count (sal) from emp b where a.sal < b.sal and
a.deptno = b.deptno)
```

Order by deptno, sal desc;

## Logical and Physical Page

A physical page (or panel) is the size of a page that will be output by your printer. A logical page is the size of one page of your actual report (it can be any number of physical pages wide or long). The Runtime Previewer displays the logical pages of your report output, one at a time.

## Bulk Binds

The assigning of values to PL/SQL variables in SQL statements is called *binding*. The binding of an entire collection at once is called *bulk binding*. Bulk binds improve performance by minimizing the number of context switches between the PL/SQL and SQL engines. With bulk binds, entire collections, not just individual elements, are passed back and forth.
The keyword FORALL instructs the PL/SQL engine to bulk-bind input collections before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is *not* a FOR loop. In a FORALL statement, if any execution of the SQL statement raises an unhandled exception, all database changes made during previous executions are rolled back. However, if a raised exception is caught and handled, changes are rolled back to an implicit savepoint marked before each execution of the SQL statement. Changes made during previous executions are *not* rolled back.

The following restrictions apply to the FORALL statement:

You can use the FORALL statement only in server-side programs (not in client-side programs). Otherwise, you get the error *this feature is not supported in client-side programs*.

The INSERT, UPDATE, or DELETE statement must reference at least one collection

The keywords BULK COLLECT tell the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. You can use these keywords in the SELECT INTO, FETCH INTO, and RETURNING INTO clauses.

The FORALL statement is specifically used for processing DML (INSERT, DELETE, and UPDATE) statements to improve performance by reducing the overhead of SQL processing. The PL/SQL interpreter executes all procedural statements. However, all SQL statements in the PL/SQL block are sent to the SQL engine, which parses and executes them. The PL/SQL-to-SQL context switch adds some overhead, which

could become significant when SQL statements are nested in loops. For instance, if it's repeated in a loop—say, 1,000,000 times—that could slow down code execution substantially.
The FORALL statement has a structure similar to FOR LOOP with a range. However, it doesn't contain an END LOOP statement and it cannot contain any statements other than the index, lower and upper bound, and actual SQL statement (which refers to the index). The range specified by lower and upper bounds (in my example, it's 1 to 10,000) must be contiguous and all the elements within that range must exist, otherwise an ORA-22160 exception will be raised.
The FORALL clause is used for DML statements. The equivalent statement for a bulk fetch is the BULK COLLECT clause, which can be used as a part of SELECT INTO, FETCH INTO, or RETURNING INTO clauses
The BULK COLLECT clause can be used for both explicit (FETCH INTO) and implicit (SELECT INTO) cursors. It fetches the data into the collection (PL/SQL table, varray) starting with element 1 and overwrites all consequent elements until it retrieves all the rows. If the collection is varray, it has to be declared large enough to accommodate all fetched rows, otherwise an ORA-22160 exception will be raised.

The bulk binds features allow users to increase the performance and reduce the overhead of SQL processing by operating on multiple rows in a single DML statement. The entire collection-not just one collection element at a time-is passed back and forth between the PL/SQL and SQL engines. According to Oracle, during internal benchmark tests there was a 30 percent performance improvement as a result of using these new features.

## Index-Organized Table, Difference b/w Ordinary Table & Index-Organized Table

An **index-organized table** has a storage organization that is a variant of a primary B-tree. Unlike an ordinary (heap-organized) table whose data is stored as an unordered collection (heap), data for an index-organized table is stored in a B-tree index structure in a primary key sorted manner. Besides storing the primary key column values of an index-organized table row, each index entry in the B-tree stores the nonkey column values as well.

| Ordinary Table | Index-Organized Table |
|---|---|
| Rowid uniquely identifies a row. Primary key can be optionally specified | Primary key uniquely identifies a row. Primary key must be specified |
| Physical rowid in ROWID pseudocolumn allows building secondary indexes | Logical rowid in ROWID pseudocolumn allows building secondary indexes |
| Access is based on rowid | Access is based on logical rowid |
| Sequential scan returns all rows | Full-index scan returns all rows |
| Can be stored in a cluster with other tables | Cannot be stored in a cluster |
| Can contain a column of the LONG datatype and columns of LOB datatypes | Can contain LOB columns but not LONG columns |

Index-organized tables provide faster access to table rows by the primary key or any key that is a valid prefix of the primary key. Presence of nonkey columns of a row in the B-tree leaf block itself avoids an additional block access. Also, because rows are stored in primary key order, range access by the primary key (or a valid prefix) involves minimum block accesses. Because rows are stored in primary key order, a significant amount of additional storage space savings can be obtained through the use of key compression. Use of primary-key based logical rowids, as opposed to physical rowids, in secondary indexes on index-organized tables allows high availability. This is because, due to the logical nature of the rowids, secondary indexes do not become unusable even after a table reorganization operation that causes movement of the base table rows. At the same time, through the use of physical guess in the logical rowid, it is possible to get secondary index based index-organized table access performance that is comparable to performance for secondary index based access to an ordinary table.

## Package/Stored Procedure (Difference)

A Package is a database object that groups logically related PL/SQL types, objects and subprograms. Packages usually have 2 parts - specification & body. The specification is the interface to your applications; it declares the types, variables, constants, exceptions, cursors and subprograms available for use. The Body fully defines cursors and subprograms, and so implements the Specification.
Unlike subprograms, packages cannot be called, passed parameters, or nested.

A Package might include a collection of procedures. Once written, your general-purpose package is compiled, and then stored in an ORACLE database, where, like a library unit, its contents can be shared by many applications.

**Advantages of Packages:**

- Modularity
- Easier Application design: Initially once the Specification has been compiled, the Body need not be fully defined until one is ready to complete the application.
- Information hiding: Certain procedures can be made as Public (visible and accessible) and others as Private (hidden and inaccessible).
- Added Functionality: Packaged public variables and cursors persist for the duration of a session. So, they can be shared by all procedures that execute in the environment. Also, they allow you to maintain data across transactions without having to store it in the database.
- Better Performance: When you call a package for the first time, the whole package is loaded into memory. Therefore, subsequent calls to related procedures in the package require no disk I/O.

Procedure:
A Procedure is a subprogram that performs a specific action.
It has 2 parts: Specification & Body.

## Global Temporary Table

In Oracle 8i, the CREATE GLOBAL TEMPORARY TABLE command creates a temporary table, which can be transaction specific or session specific. For transaction-specific temporary tables, data exists for the duration of the transaction while for session-specific temporary tables, data exists for the duration of the session. Data in a temporary table is private to the session. Each session can only see and modify its own data. The table definition itself is not temporary.
You can create indexes for temporary tables using the CREATE INDEX command. Indexes created on temporary tables are also temporary and the data in the index has the same session or transaction scope as the data in the temporary table.
You can perform DDL commands (ALTER TABLE, DROP TABLE, CREATE INDEX, and so on) on a temporary table only when no session is currently bound to it. A session gets bound to a temporary table when an INSERT is performed on it. The session gets unbound by a TRUNCATE, at session termination, or by doing a COMMIT or ABORT for a transaction-specific temporary table.
Temporary tables use temporary segments. Temporary segments are de-allocated at the end of the transaction for transaction-specific temporary tables. For session-specific temporary tables, they are de-allocated at the end of the session.

Multi-session scenarios:

1-Two different users try to create a global temporary table with the same name
Both tables get created. Each user sees his/her own table.
2-Two sessions by a single user try to create a global temporary table with the same name.
Once the table has been created in one of the sessions, the table exists and an error is given to the second session when trying to create the table. This behavior occurs whether or not the table structure is defined to be the same in both sessions.
3-Two sessions by a single user insert rows into a global temporary table.
Each session can insert rows into the table, no matter which session created the table.
4-Two sessions by the same user select from a global temporary table.
A given session will only see the rows that it has inserted, as if the table was private to that session.
5-A session tries to alter a temporary table that another session is also using.
Columns cannot be added/deleted/modified as long as the table is bound to a session. Renaming, however, can be done even when another session has rows in the table.
6-A session attempts to rename a global temporary table in which another session has inserted some rows
The table gets renamed and rows are preserved. After the rename has taken place, both sessions must use the new name to refer to that table.

## Surrogate Keys

A unique primary key generated by the RDBMS that is not derived from any data in the database and whose only significance is to act as the primary key. A surrogate key is frequently a sequential number (e.g. a Sybase "identity column") but doesn't have to be. Having the key independent of all other columns insulates the database relationships from changes in data values or database design and guarantees uniqueness.

Some database designers use surrogate keys religiously regardless of the suitability of other candidate keys. However, if a good key already exists, the addition of a surrogate key will merely slow down access, particularly if it is indexed.

## Magic Item Property in Menu

Specifies one of the following predefined menu items for custom menus: Cut, Copy, Paste, Clear, Undo, About, Help, Quit, or Window. Magic menu items are automatically displayed in the native style for the platform on which the form is being executed, with the appropriate accelerator key assigned.
Cut, Copy, Paste, Clear, Window and Quit have built-in functionality supplied by Form Builder, while the other magic menu items can have commands associated with them.

## DUAL table

A tiny table with only one row and one column

## Implicit commit

Using any of the following actions will force a commit to occur: quit, exit, create table, create view, drop table, drop view, grant, revoke, connect, disconnect, alert, audit and no audit.

## What is Rollback

If a series of instructions are completed, but have not yet explicitly or implicitly committed them, and if a serious difficulty such as computer failure is experienced, Oracle automatically rollbacks any uncommitted work.

## Database Cache

Database Cache is a caching service that resides on the middle-tier and provides caching services for data from the origin database. It provides caching of the data from the origin database so that the application of the middle-tier can fetch data. It can be created on either the application server or the web server.

## Explain Plain

The EXPLAIN PLAN command displays the execution plan chosen by the Oracle optimizer for SELECT, UPDATE, INSERT, and DELETE statements at the current time, with the current set of initialization and session parameters. A statement's execution plan is the sequence of operations that Oracle performs to execute the SQL statement. By examining the execution plan, you can see exactly how Oracle executes the SQL statement. Use EXPLAIN PLAN to determine the Table Access Order, Table Join Types, Index Access to test modifications to improve SQL performance. It's best if you don't blindly evaluate the plan for a statement, and decide to tune it based **only** on the **execution plan**. EXPLAIN PLAN results alone cannot tell you which statements will perform well, and which won't. For example, just because EXPLAIN PLAN indicates that a statement will use an Index doesn't mean that the statement will run quickly. The index might be v**ery inefficient**. Instead, you should examine the statement's actual RESOURCE CONSUMPTION and the statements ACTUAL RUN TIME. REMEMBER - A low COST doesn't GUARANTEE a FASTER RUN TIME
The AUTOTRACE Command in SQLPlus can be used to generate EXPLAIN PLANS and Run Time Statistics. The EXPLAIN PLAN is generated after the SQL Statement is executed.

**Explain Plan (Optimization):** To analyze for tuning the SQL statement Explain Plan is used. For that run the UTLXPLAN.SQL script to create table named PLAN_TABLE, for receiving the output of an Explain Plan statement.

```
Eg.  Explain Plan
        Set Statement_id 'My Customer'
        Into Plan_table
        For Select lastname, firstname
             From customers
             Where state = 'NY'
```

Output: Select statement_id,operation,options,cost from plan_table;

```
Statement_id       Operation              Options          Cost
My Customer             Select Statement                          9
My Customer             Table Access           Full Access        16
```

The Explain Plan command of oracle7 uses the current statistics in the data dictionary to do its best to return an accurate execution plan and corresponding cost to executing a statement. However, if statistics are not up-to-date then of course the Explain Plan command might not return an accurate execution plan and cost.

## Parsing

**Parsing:** Parsing a statement means breaking down the statement, so that it can be executed. When oracle server parses a SQL statement, it first checks for semantic and syntactical errors in statements. Oracle uses shared the SQL area for parsing (**Parse tree).**

**Programming standards:**

- Using the standard template form.
- Naming conventions for blocks, buttons, record groups, etc.
- Proper Indentation.
- Naming and declaration of identifiers (local variables, global variables, cursors, etc.)
- Always use %Type and %RecType
- Exception handling and error messages
- Proper commenting
- Using performance standard code in Validation and data fetching

## 9I Features

Features:

1. One can create or rebuild online index
2. Buffer cache & shared pool is resized dynamically
3. Data recovery is very fast due to 2 pass recovery algorithm
4. Flashback Query feature of oracle 9I
5. 2 million of users performing millions of transaction/H is possible (Scalability)
6. Multilingual server side debugging has been added
7. Rollback segments are managed own by Oracle
8. Oracle 9i introduces resumable statements, which allow to temporarily suspending batch process or data load.
9. Databases created with Multiple Block size
10. Execution history is maintained
11. Better integration with Windows 2000
12. Feature of intermedia text indexing for fast searches has been introduced
13. Long data types can be converted to LOB using alteration
14. Self tuning memory among processes hence query performance

**There are two tables namely EMP, SALGRADE**

Fields of SALGRADE are
GRADE, LOSAL HISAL
Fields of EMP table are as usual

I want the output like this.

```
    GRADE     LOSAL     HISAL COUNT (*)
--------- --------- --------- ---------
        1       700      1200         4
        2      1201      1400         5
        3      1401      2000         4
        4      2001      3000         8
        5      3001      9999         2
```

Select grade, losal, hisal, count (*) from EMP, salgrade
Where sal between losal and hisal
Group by grade, losal, hisal

**Select even row from emp table**

```
Select * from EMP where rowid in (select decode (mod (rownum, 2), 0, rowid, null)
from EMP);
```

**Select odd row from emp table**

```
Select * from EMP where rowid in (select decode (mod (rownum, 2), 1, rowid, null)
from EMP);
```

**Select Nth row from emp table where N=10**

```
Select * from EMP where rowid = (select rowid from EMP                    where
ROWNUM <= 10
MINUS
Select rowid from EMP where ROWNUM < 10)
```

**Select rows X to Y from emp table where X=5 and Y=7**

```
Select * from EMP where rowid in (select rowid from EMP
Where rowNUM <= 7
MINUS
Select rowid from EMP where rownum < 5);
```

**All even, Odd, Nth rows from a table**

```
Select * from emp where (rowid, 0) in (select rowid, mod (rownum,&n) from emp);
```

**Select N rows from a table**

```
Select * from EMP a
Where &n> (select count (distinct (sal)) from EMP b where a. Sal >= b.sal)
Order by sal desc;
```

Please check this query for the correctness for n=1, 2,3,4,5

**Select Employee having maximum salary in department from emp table**

```
Select deptno, empno, sal from EMP a where (deptno, sal) in (
Select deptno, max (sal) from EMP b
Where a.deptno = b.deptno
Group by b.deptno) order by 1
```

**External Tables**

External tables allow Oracle to query data that is stored outside the database in flat files. The ORACLE_LOADER driver can be used to access any data stored in any format that can be loaded by SQL*Loader. No DML can be performed on external tables but they can be used for query, join and sort operations. Views and synonyms can be created against external tables. They are useful in the ETL process of data warehouses since the data doesn't need to be staged and can be queried in parallel. They should not be used for frequently queried tables.

**Limitations on external tables**

Because external tables are new, Oracle has not yet perfected their use. In Oracle9*i* the feature has several limitations, including:

- No support for DML. External tables are read-only, but the base data can be edited in any text editor.
- Poor response for high-volume queries. External tables have a processing overhead and are not suitable for large tables.

**Example**: The example below describes how to create external files, create external tables, query external tables and create views.

**Step I:** Creating the flat files, which will be queried?

The file "*emp_ext1.dat*" contains the following sample data:

```
101,Andy,FINANCE,15-DEC-1995
102,Jack,HRD,01-MAY-1996
103,Rob,DEVELOPMENT,01-JUN-1996
104,Joe,DEVELOPMENT,01-JUN-1996
```

The file "*emp_ext2.dat*" contains the following sample data:

```
105,Maggie,FINANCE,15-DEC-1997
106,Russell,HRD,01-MAY-1998
107,Katie,DEVELOPMENT,01-JUN-1998
108,Jay,DEVELOPMENT,01-JUN-1998
```

Copy these files under "*C:\EXT_TABLES*"

**Step II:** Create a Directory Object where the flat files will reside

```
SQL> CREATE OR REPLACE DIRECTORY EXT_TABLES AS 'C:\EXT_TABLES';

Directory created.
```

**Step III:** Create metadata for the external table

```
SQL> CREATE TABLE emp_ext
        (
        empcode NUMBER(4),
        empname VARCHAR2(25),
        deptname VARCHAR2(25),
        hiredate date
        )
 ORGANIZATION EXTERNAL
  (
   TYPE ORACLE_LOADER
   DEFAULT DIRECTORY ext_tables
   ACCESS PARAMETERS
   (
    RECORDS DELIMITED BY NEWLINE
    FIELDS TERMINATED BY ','
    MISSING FIELD VALUES ARE NULL
   )
   LOCATION ('emp_ext1.dat','emp_ext2.dat')
  )
 REJECT LIMIT UNLIMITED;
```

Table created.

The REJECT LIMIT clause specifies that there is no limit on the number of errors that can occur during a query of the external data.

**Step V:** Creating Views

```
SQL> CREATE VIEW v_empext_dev AS
                  SELECT * FROM emp_ext
                  WHERE deptname = 'DEVELOPMENT';
View created.
```

**Dropping External Tables**

For an external table, the DROP TABLE statement removes only the table metadata in the database. It has no affect on the actual data, which resides outside of the database.

**Multitable Insert  in Oracle 9i**

Multitable inserts allow a single INSERT INTO.. SELECT statement to conditionally, or non-conditionally, insert into multiple tables. This statement reduces table scans and PL/SQL code necessary for performing multiple conditional inserts compared to previous versions. Its main use is for the ETL process in data warehouses where it can be parallelized and/or convert non-relational data into a relational format:

```
-- Unconditional insert into ALL tables

INSERT ALL
  INTO sal_history VALUES (empid, hiredate, sal)
  INTO mgr_history VALUES (empid, mgr, sysdate)
SELECT employee_id EMPID, hire_date HIREDATE, salary SAL, manager_id MGR
  FROM employees WHERE employee_id > 200;

-- Pivoting insert to split non-relational data
INSERT ALL
  NTO Sales_info VALUES (employee_id, week_id, sales_MON)
  INTO Sales_info VALUES (employee_id, week_id, sales_TUE)
  INTO Sales_info VALUES (employee_id, week_id, sales_WED)
  INTO Sales_info VALUES (employee_id, week_id, sales_THUR)
  NTO Sales_info VALUES (employee_id, week_id, sales_FRI)
SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
    Sales_WED, sales_THUR, sales_FRI
FROM Sales_source_data;

-- Conditionally insert into ALL tables
INSERT ALL
  WHEN SAL>10000 THEN
    INTO sal_history VALUES (EMPID, HIREDATE, SAL)
  WHEN MGR>200 THEN
    INTO mgr_history VALUES (EMPID, MGR, SYSDATE)
SELECT employee_id EMPID, hire_date HIREDATE, salary SAL, manager_id MGR
  FROM employees WHERE employee_id > 200;

-- Insert into the FIRST table with a matching condition
INSERT FIRST
  WHEN SAL > 25000 THEN
    INTO special_sal VALUES (DEPTID, SAL)
  WHEN HIREDATE like ('%00%') THEN
    INTO hiredate_history_00 VALUES (DEPTID, HIREDATE)
  WHEN HIREDATE like ('%99%') THEN
    INTO hiredate_history_99 VALUES (DEPTID, HIREDATE)
  ELSE
    INTO hiredate_history VALUES (DEPTID, HIREDATE)
SELECT department_id DEPTID, SUM (salary) SAL,
    MAX (hire_date) HIREDATE
  FROM employees GROUP BY department_id;
```

The restrictions on multitable Inserts are:

Multitable inserts can only be performed on tables, not on views or materialized views.

You cannot perform a multitable insert via a DB link.
You cannot perform multitable inserts into nested tables.
The sum of all the INTO columns cannot exceed 999.
Sequences cannot be used in the subquery of the multitable insert statement.

**New Join Syntax**

This new join syntax uses the new keywords inner join, left outer join, right outer join, and full outer join, instead of the (+) operator.

**INNER Join:**

```
SQL> select p.part_id, s.supplier_name
From part p inner join supplier s
On p.supplier_id = s.supplier_id;

PART SUPPLIER_NAME
---- --------------------
P1   Supplier#1
P2   Supplier#2
```

Remember, if we want to retain all the parts in the result set, irrespective of whether any supplier supplies them or not, then we need to perform an outer join. The corresponding outer join query using the new syntax will be:

**OUTER JOIN**

```
SQL> select p.part_id, s.supplier_name from part p left outer join supplier s on p.supplier_id = s.supplier_id;

PART SUPPLIER_NAME
---- --------------------
P1   Supplier#1
P2   Supplier#2
P4
P3
```

This is called a "left outer join" because all the rows from the table on the left (PART) are retained in the result set. If we want to retain all the suppliers in the result set, irrespective of whether they supply any part or not, then we need to perform a "right outer join". That would look like:

```
SQL> select p.part_id, s.supplier_name from part p right outer join supplier s on
p.supplier_id = s.supplier_id;

PART SUPPLIER_NAME
---- --------------------
P1   Supplier#1
P2   Supplier#2
     Supplier#3
```

However, the biggest advantage of the new join syntax is its support for full outer joins. Introduction of the ANSI standard join syntax in Oracle9*i* greatly simplifies the full outer join query. We are no longer limited by unidirectional outer join, and no longer need to use the UNION operation to perform the full outer join.

**FULL OUTER JOIN**

Oracle9*i* introduced the full outer join operation to carry out such operations, as in the following example:

```
SQL> select p.part_id, s.supplier_name
From part p full outer join supplier s
On p.supplier_id = s.supplier_id;

PART SUPPLIER_NAME
---- --------------------
P1   Supplier#1
P2   Supplier#2
P4
P3
     Supplier#3
```

The above SQL statement is not only smaller in size, it is much more elegant and intuitive as well. This ANSI join syntax is also more efficient than the UNION method of achieving a full outer join.

**NATURAL Joins**. A natural join, as its name implies, can be invoked when *two or more tables share exactly the same columns* needed for a successful equijoin. For example, these queries will return all Region and Country information for all countries whose name that contains the string "united":

| **Example: NATURAL Join** | |
|---|---|
| **Traditional Syntax** | **ANSI SYNTAX** |

| | |
|---|---|
| **SELECT r.region_id, r.region_name, c.country_id, c.country_name FROM Countries c, regions r WHERE c.region_id = r.region_id AND LOWER (c.country_name) LIKE '%united%';** | **SELECT region_id, r.region_name, c.country_id, c.country_name FROM countries c NATURAL JOIN regions r WHERE LOWER (c.country_name) LIKE '%united%';** |
| | |

## Multiple block sizes in the same database

•The BLOCKSIZE keyword has been added to the CREATE TABLESPACE command. Valid values are:

2K, 4K, 8K, 16K, 32K (operating system dependent).

•CREATE TABLESPACE MY_TBS ... BLOCKSIZE 16K;
•CREATE TABLESPACE MY_TBS ... BLOCKSIZE 8192;

•The field BLOCK_SIZE has been added to the DBA_TABLESPACES view.
•The DB_BLOCK_SIZE parameter is still needed in the init.ora during database creation. This value is the default blocksize for new tablespaces as well as the required size for the system tablespace, undo tablespace and all temporary tablespaces.
•As with previous releases, DB_BLOCK_SIZE cannot be changed without recreating the database.
•All partitions of a partitioned object must be in tablespaces with the same block size.
•Indexes can be in a tablespace with a different block size than its corresponding table.

## 9I SQL FEATURES

NULL2 Function–

NVL2 lets you determine the value returned by a query based on whether a specified expression is null or not null. If *expr1* is not null, then NVL2 returns *expr2*. If *expr1* is null, then NVL2 returns *expr3*. The argument *expr1* can have any datatype. The arguments *expr2* and *expr3* can have any datatypes except LONG.
If the datatypes of *expr2* and *expr3* are different, then Oracle converts *expr3* to the datatype of *expr2* before comparing them unless *expr3* is a null constant. In that case, a datatype conversion is not necessary.
The datatype of the return value is always the same as the datatype of *expr2*, unless *expr2* is character data, in which case the return value's datatype is VARCHAR2.

NULLIF (expr1, expr2) –Returns NULL if the first argument is equal to the second, otherwise returns the first argument.

COALESCE (expr1, expr2, expr3, ...) –Returns the first non-null argument.

**SELECT... FOR UPDATE Enhancements**
Selecting a record for update that is already locked causes the current session to hang indefinitely until the lock is released. If this situation is unacceptable the NOWAIT keyword can be used to instantly return an error if the record is locked. Oracle9i adds more flexibility by allowing the programmer to specify a maximum time limit to wait for a lock before returning an error. This gets round the problem of indefinite waits, but reduces the chances of lock errors being returned:

```
SELECT * FROM   employees WHERE empno = 20 FOR UPDATE WAIT 30;
```

## Merge Statement

The MERGE statement can be used to conditionally **insert** or **update** data depending on its presence. This method reduces table scans and can perform the operation in parallel. Consider the following example where data from the HR_RECORDS table is merged into the EMPLOYEES table:

```
MERGE INTO employees e
   USING hr_records h
   ON (e.id = h.emp_id)
  WHEN MATCHED THEN
   UPDATE SET e.address = h.address
  WHEN NOT MATCHED THEN
   INSERT (id, address)
   VALUES (h.emp_id, h.address);
```

## Explicitly Named Indexes On Keys

In Oracle9i the index used to support primary and unique keys can be defined independently of the constraint itself by using the CREATE INDEX syntax within the USING UNDEX clause of the CREATE TABLE statement:

```
CREATE TABLE employees
(
  Empno NUMBER (6),
  Name VARCHAR2 (30),
  Deptno NUMBER (2),
  CONSTRAINT emp_pk primary key (empno)
    USING INDEX
    (CREATE INDEX emp_pk_idx ON employee (empno))
);
```

The constraint can subsequently be dropped without dropping the index using either syntax:

```
ALTER TABLE employees DROP PRIMARY KEY KEEP INDEX;
ALTER TABLE employees DROP CONSTRAINT empno_pk;
```

## Share Locks On Unindexed FKs

In previous versions a share lock was issued on the entire child table while the parent table was being updated if the foreign key between them was unindexed. This had the affect of preventing DML operations on the child table until the parent table transaction was complete.

In Oracle9i this situation has been altered such that a table level share lock is issued and instantly released. This action allows Oracle to check that there are no pending changes on the child table, but the instant release means that DML can resume almost instantly once the parent table update has initiated. If multiple keys are updated Oracle issues a share lock and release on the child table for each row.

## PK Lookup During FK Insertion

During insertions foreign key values are checked against the primary keys of referenced tables. This process is optimized in Oracle9i by caching the first 256 PK values of the referenced table on insertion of the second record of a multiple insert. The process is done on the second record to prevent the overhead of managing the cache on a single insert.

## Function Based Index Enhancements

Function Based Indexes are now capable of doing an index-only scan. In previous versions this was only possible if the index creation statement explicitly prevented NULL values. Since each built-in operator knows implicitly whether it can produce null values when all it's input parameters are not null, Oracle can deduce if nulls can be produced and therefore decide if index-only scans are possible based on the columns queried using the function based index.

## View Constraints

Declarative primary key, unique key and foreign key constraints can now be defined against views. The NOT NULL constraint is inherited from the base table so it cannot be declared explicitly. The constraints are not validated so they must be defined with the DISABLE NOVALIDATE clause:

```
CREATE VIEW Emp_view
 (Id PRIMARY KEY DISABLE NOVALIDATE, firstname)
AS SELECT employee_id, first_name
FROM employees
WHERE department_id = 10;

ALTER VIEW Emp_view
ADD CONSTRAINT emp_view_unq
UNIQUE (first_name) DISABLE NOVALIDATE;
```

## How to retrieve DDL from Database

DBMS_METADATA package will work for tables, indexes, views, packages, functions, procedures, triggers, synonyms, and types.

```
DBMS_METADATA.GET_DDL (object_type, name, schema)
DBMS_METADATA.GET_XML (object_type, name, schema)
```

•SELECT DBMS_METADATA.GET_DDL (‘TABLE’, ‘EMP’, ‘SCOTT’) from dual;

**Commit processing** is the way Form Builder attempts to make the data in the database identical to the data in the form. Form Builder's normal cycle of operation is:

1 Read records from the database.

2 Allow the end user to make tentative insertions, updates, and deletions. The tentative changes appear only in the form. The database remains unchanged.

3 Post changes to the database. Form Builder does all of its remaining processing and sends the data to the database. After posting the data, Form Builder can only roll back the changes (via the [Clear Form] function key or CLEAR_FORM built-in) or commit them.

4 Form Builder commits the posted changes. They become permanent changes to the database.

The term commit processing refers to steps 3 and 4 of the above cycle. Normally these steps occur together. In response to the [Commit] key or invocation of the COMMIT_FORM built-in, Form Builder firsts posts, then commits the data to the database.
Posting can occur separately before committing. End users cannot issue posting commands, but triggers can invoke the POST built-in.

Note: Form Builder does not support a commit that does not include the normal Form Builder commit processing. For example, Form Builder does not support a commit issued from an Oracle Precompiled user exit or from a stored procedure.

Processing inserts, updates, and deletes

Posting consists of sending tentative changes from the form to the database. These are records that have been marked for insertion, update, or deletion since the last post. During posting, Form Builder processes inserts, updates, and deletes for all blocks in a form. Form Builder has a standard way to process these changes. Triggers provide a flexible mechanism for altering the standard behavior.
Insert An insert is the pending insertion of a row in the database. Each insert has an associated SQL statement, which Form Builder executes when it posts the insert. The statement has the following form:

```
INSERT INTO table [(column, column, . . .)]
  VALUES (value, value, . . .);
```

table The name of the base table for the current block.
column A column corresponding to a base table item. If an item is a derived column, it does not appear in the column clause.
value The value to insert into the corresponding column.

Update An update is the pending update of an existing row in the database. Each update has an associated SQL statement, which Form Builder executes when it posts the update. The statement has the following form:

```
UPDATE table
  SET (column=value, column=value, . . .)
  WHERE ROWID=rowid_value;
```

table The name of the base table for the current block.
column A column corresponding to a base table item. If an item is a derived column, it does not appear in the column clause.
value The value to update the corresponding column.
rowid_value The ROWID value for the row Form Builder is updating.
Note:

The WHERE clause specifies only a ROWID value. This identifies the unique row that the database should update. Form Builder uses the ROWID construct only when the block's Key Mode property has the value Unique_Key (the default).

If an end user does not have update privileges for a column, and the block's Enforce Column Security property has the value Yes, Form Builder does not include the column in the UPDATE statement. Different end users can have different UPDATE statements, depending on their privileges, but the statement remains unchanged during an end user's Form Builder session.

Delete A delete is the pending deletion of a row in the database. Each delete has an associated SQL statement, which Form Builder executes when it posts the delete. The statement has the following form:

```
DELETE FROM table WHERE ROWID=rowid_value;
```

table The name of the base table for the current block.
rowid_value The ROWID value for the row Form Builder is deleting.
Note: The WHERE clause specifies only a ROWID value. This identifies the unique row that the database should delete. Form Builder does not use the ROWID value for non-Oracle data sources.
Caution: If a commit fails, the ROWID value may not have a null value. Use record status rather than the ROWID value to test for success or failure of the commit.

When commit processing occurs

Form Builder performs commit processing when

A trigger or the [Commit] key invokes the COMMIT_FORM built-in.

Database items in the current block have changed since the last commit, and a trigger invokes the CLEAR_BLOCK built-in (with the DO_COMMIT argument).

Database items in the form have changed since the last commit, and a trigger invokes the CLEAR_FORM built-in (with the DO_COMMIT argument).

An end user answers Yes to the alert that asks

Do you want to commit the changes you have made?

The alert appears as a result of any of the following:

Database items in the current block have changed since the last commit, and any of the following events occurs:

CLEAR_BLOCK (with the ASK_COMMIT argument or with no argument)
COUNT_QUERY
ENTER_QUERY
EXECUTE_QUERY
NEW_FORM

Database items in any block of the form have changed since the last commit, and any of the following events occurs:

CLEAR_FORM (with the ASK_COMMIT argument or with no argument)
Leave the Form

Note: When a PL/SQL block issues a database commit from within Form Builder (via the SQL COMMIT statement), Form Builder commit processing occurs as if the COMMIT_FORM built-in had been invoked.

Changing data during commit processing

Commit processing performs validation and can fire triggers. As a result, a commit event can change database items. For some triggers Form Builder attempts to commit those changes to the database during the current commit event.

Caution: Form Builder can commit unvalidated database changes made by the PRE-DELETE, PRE-INSERT, and PRE-UPDATE triggers.

For example, assume that a PRE-INSERT trigger selects a value into a base table item.

If Form Builder has not already processed the affected record's updates, it commits the changes during the current commit event without validating them.

If Form Builder has already processed the affected record's updates, it does not commit the changes, but it marks the records as having come from the database. Subsequently attempting to update the record before it is committed causes an error.

Caution: Form Builder can commit unvalidated database changes made by the POST-COMMIT, POST-DELETE, POST-INSERT, and POST-UPDATE triggers.

For example, assume that a POST-INSERT trigger selects a value into a base table item.

If the affected record is in a block that the commit has processed or is processing, Form Builder does not commit the changes during the current commit event. If the commit succeeds, Form Builder marks all items and records as Valid. This results in an error during subsequent processing.

If the affected record is in a block that the commit has not yet processed, Form Builder validates the changes and commits them during the current commit event.

Replacing standard commit processing

The ON-INSERT, ON-UPDATE, and ON-DELETE triggers replace standard commit processing.

**Form Level Properties**

## Console Window property

Specifies the name of the window that should display the Form Builder console. The console includes the status line and message line, and is displayed at the bottom of the window.
On Microsoft Windows, the console is always displayed on the MDI application window, rather than on any particular window in the form; however, you must still set this property to the name of a form window to indicate that you want the console to be displayed.

## Menu Source property

Menu Source allows you to specify the location of the .MMX runfile when you attach a custom menu to a form module. Form Builder loads the .MMX file at form startup.

## Menu Module property

Specifies the name of the menu to use with this form.

## Initial Menu property

Specifies the name of the individual menu in the menu module that Form Builder should use as the main, or top-level, menu for this invocation.

## Defer Required Enforcement property

It specifies whether Form Builder should defer enforcement of the Required item attribute until the record is validated.
This property applies only when item-level validation is in effect. By default, when an item has required set to true, Form Builder will not allow navigation out of the item until a valid value is entered. This behavior will be in effect if you set Defer Required Enforcement to No.
If you set Defer Required Enforcement to Yes or 4.5, you allow the end user to move freely among the items in the record, even if they are null, postponing enforcement of the required attribute until validation occurs at the record level.

When Defer Required Enforcement is set to 4.5, null-valued required items are not validated when navigated out of, and the item's Item Is Valid property is unchanged. However, the WHEN-VALIDATE-ITEM trigger (if any) does fire. If it fails (raises Form_Trigger_Failure), the item is considered to have failed validation and Form Builder will issue an error. If the trigger ends normally, processing continues normally. If the item value is still null when record-level validation occurs later, Form Builder will issue an error at that time.

## Mouse Navigation Limit property

Determines how far outside the current item an end user can navigate with the mouse.
Form (The default): Allow end users to navigate to any item in the current form.
Block: Allows end users to navigate only to items that are within the current block.
Record: Allows end users to navigate only to items that are within the current record.
Item: Prevents end users from navigating out of the current item. This setting prevents end users from navigating with the mouse at all.

## First Navigation Block property

Specifies the name of the block to which Form Builder should navigate at form startup and after a CLEAR_FORM operation.

## Current Record Visual Attribute Group property

Specifies the named visual attribute used when an item is part of the current record. This property can be set at the form, block, or item level, or at any combination of levels. If you specify named visual attributes at each level, the item-level attribute overrides all others, and the block-level overrides the form-level.

## Validation Unit property

The validation unit defines the maximum amount of data that an operator can enter in the form before Form Builder initiates validation.

## Interaction Mode property

Interaction mode dictates how a user can interact with a form during a query. If Interaction Mode is set to Blocking, then users are prevented from resizing or otherwise interacting with the form until the records for a query are fetched from the database. If set to Non-Blocking, then end users can interact with the form while records are being fetched.

Non-blocking interaction mode is useful if you expect the query will be time-consuming and you want the user to be able to interrupt or cancel the query. In this mode, the Forms runtime will display a dialog that allows the user to cancel the query.
You cannot set the interaction mode programmatically, however, you can obtain the interaction mode programmatically using the GET_FORM_PROPERTY built-in.

## Maximum Query Time property

Provides the option to abort a query when the elapsed time of the query exceeds the value of this property.

## Maximum Records Fetched property

Specifies the number of records fetched when running a query before the query is aborted.

## Isolation Mode property

Specifies whether or not transactions in a session will be serializable. If Isolation Mode has the value serializable, the end user sees a consistent view of the database for the entire length of the transaction, regardless of updates committed by other users from other sessions. If the end user queries and changes a row, and a second user updates and commits the same row from another session, the first user sees Oracle error (ORA-08177: Cannot serialize access.).

## Coordinate System property

Specifies whether object size and position values should be interpreted as character cell values, or as real units (centimeters, inches, pixels, or points).
Character: Sets the coordinate system to a character cell-based measurement. The actual size and position of objects will depend on the size of a default character on your particular platform.
Real: Sets the coordinate system to the unit of measure specified by the Real Unit property (centimeters, inches, pixels, or points.)

Changing the coordinate system for the form changes the ruler units displayed on Form Editor rulers, but does not change the grid spacing and snap-points settings.

## Use 3D Controls property

On Microsoft Windows, specifies that Form Builder display items with a 3-dimensional, beveled look.

When Use 3D Controls is set to **Yes,** any canvas that has Visual Attribute Group set to Default will automatically be displayed with background color grey.
In addition, when Use 3D Controls is set to Yes, the bevel for each item automatically appears lowered, even if an item-level property is set, for example, to raised.

## Form Horizontal Toolbar Canvas property

On Microsoft Windows, specifies the canvas that should be displayed as a horizontal toolbar on the MDI application window. The canvas specified must have the Canvas Type property set to Horizontal Toolbar.

## Form Horizontal Toolbar Canvas property

On Microsoft Windows, specifies the toolbar canvas that should be displayed as a vertical toolbar on the MDI application window. The canvas specified must have the Canvas Type property set to Vertical Toolbar.

## Direction property

This property is used to define the test justification for all items.

| Value | Description |
|---|---|
| Default | Direction based on the property shown in the table. |
| Right-To-Left | Direction is right-to-left. |
| Left-To-Right | Direction is left-to-right. |

## Runtime Compatibility Mode property

Specifies the Form Builder version with which the current form's runtime behavior is compatible (either 4.5 or 5.0 +). By default, new forms created with Form Builder 5.0 and later are set to 5.0-compatible. Existing forms that are upgraded from 4.5 are 4.5-compatible. To get these forms to use the new runtime behavior of 5.0 +, set this property to 5.0. The runtime behavior that is affected by this property is primarily validation and initialization.

## Initialization

Form Builder initializes a form by creating a null record in each block of the form. Each item of a null record is null. Form builder replaces the null record with an initialized record the first time it enters the block. At that time it gives initial values to some items. An item's Initial Value property specifies the item's initial value.

Note: If a trigger assigns a value to an item in a null record, the record is no longer null. Form Builder never assigns an initial value to any item in the record. For example, a PRE-FORM trigger fires while the form contains only null records. If it assigns a value to an item, it must ensure that all other items of the record receive appropriate initial values.

Special considerations apply to certain types of items.

**Check boxes**

The initial value determines whether the check box is initially checked or unchecked. You must specify a valid initial value unless one of the following is true:

- The item's Mapping of Other Values property is set to Checked or Unchecked.
- The value associated with Checked or Unchecked is Null.

Note: A check box in a null record appears unchecked.

**List items**

You must specify a valid initial value unless one of the following is true:

- The item's Mapping of Other Values property has a non-null value.
- The value associated with one of the list elements is Null.

Note: A list item in a null record appears blank.

**Radio groups**

You must specify a valid initial value unless one of the following is true:

- The item's Mapping of Other Values property has a non-null value.
- The value associated with one of the radio buttons in the group is Null.

Note: A radio group in a null record appears with no radio button selected.

**Synchronized items**

If items, say B and C, have their Synchronize with Item properties set to the name of another item, say A, then A, B, and C form a set of synchronized items. A is the master item of the set, and B and C are subordinate items. Synchronized items were called mirror items in earlier versions of Form Builder.
The items of a synchronized set share a common data value, but can have different item properties. Form Builder takes data-specific properties and triggers from the master item and ignores those properties and triggers for the subordinate items.

Form Builder initializes synchronized items from the master item's Initial Value property. Form Builder also uses the master item's ON-SEQUENCE-NUMBER trigger.
If the designer specifies an Initial Value property or an ON-SEQUENCE-NUMBER trigger for a subordinate item, Form Builder ignores them and issues a warning.
Form Builder ignores an item's Initial Value property if all of the following are true for the item (or an item synchronized with it):

The item is a check box, poplist (drop-down list), Tlist (Text list), or radio group.
The item has no element corresponding to the specified initial value.
The item's Mapping of Other Values property has a null value.

## Validation

Form Builder validates an item by ensuring that the item conforms to the rules that pertain to it. Triggers can implement additional validation.

**Validation unit**

A form's Validation Unit property controls the granularity of validation in the form. The validation unit can be an item, record, block, or form. Most Form Builder applications validate at the item level.

**When validation occurs**

Form Builder performs validation when

Form Builder navigates out of the validation unit. This occurs when the end user presses certain function keys or clicks the mouse outside the validation unit, or a trigger executes certain built-ins.
A trigger invokes the ENTER built-in, or the end user presses [Enter].
A trigger invokes the COMMIT_FORM built-in, or the end user presses [Commit]. In this case Form Builder validates the form regardless of the validation unit.
A trigger invokes the VALIDATE built-in.

Form Builder does not perform validation in Enter Query mode. This allows the end user to enter query criteria that Form Builder would find invalid.

**How validation proceeds**

Internally, Form Builder maintains a status for each item, record, block, and form. For purposes of validation, each potential validation unit has a validation status with one of two values:

Valid
Unvalidated

The Valid status means that Form Builder does not need to examine the validation unit. The Unvalidated status, a combination of states known internally as New and Changed, means that Form Builder needs to validate the items in the validation unit. The distinction between New and Changed appears only rarely in the validation process (see the note in the Record entry in the table that follows).
A third state occurs if validation fails, but in that case Form Builder does not proceed until the situation is corrected, so the failed status does not enter into the validation process.

Form Builder validates a validation unit according to following processes:

**Item**
If the item's status is Valid, the process stops. If the status is Unvalidated, Form Builder performs standard validation checks, then fires the WHEN-VALIDATE-ITEM trigger. (See, however, the Defer Required Enforcement property.)
**Record**
If the record's status is Valid, the process stops. If the status is Unvalidated, Form Builder validates each item in the record, then fires the WHEN-VALIDATE-RECORD trigger.
Note: if the validation is as a result of a call to the VALIDATE trigger and the record's internal status is New, the process stops.
**Block**
Form Builder validates all records in the block.
**Form**
Form Builder validates all blocks in the form.

**Item validation status**

**Unvalidated**
Form Builder sets the status of an item to Unvalidated when any of the following occurs:

Form Builder creates a record.
The end user types a value into the item, or a trigger causes a value to be stored into the item.

**Note**: Form Builder sets the status of the item to Unvalidated even if the new value is the same as the old value. For example, if an item with Valid status has the value 10 and the end user types 10 into the item, Forms Builder sets the status of the item to Unvalidated.

- When Form Builder duplicates a record (in response to the DUPLICATE_RECORD built-in or the [Duplicate Record] key), any item that has Unvalidated status in the original has Unvalidated status in the duplicate.

**Valid**

Form Builder sets the status of an item to Valid in the following cases:

- When Form Builder successfully validates an item, it sets the item's status to Valid, even if validation changes the item's value (through a WHEN-VALIDATE-ITEM, WHEN-VALIDATE-RECORD, or POST-CHANGE trigger). Form Builder does not re-validate the changed value.

**Caution**: This behavior avoids validation loops, but a trigger error can cause Form Builder to commit an invalid value to the database.

- When Form Builder fetches records from the database, it sets their status to Valid, even if a Post-Change trigger that fires during the fetch changes item values.
- When Form Builder successfully commits data to the database, it sets the status of all items in the form to Valid. Form Builder does not validate changes caused by triggers that fire during the commit transaction.

**Note**: See Changing Data During Commit Processing for exceptions to this rule.

- When Form Builder duplicates a record (in response to the DUPLICATE_RECORD built-in or the [Duplicate Record] key), any item that has Valid status in the original has Valid status in the duplicate.

**Record validation status**

Each record has a validation status of Unvalidated or Valid.

**Note**: a record's validation status is not equivalent to the value of the SYSTEM.RECORD_STATUS system variable.

**Unvalidated**

Form Builder sets the status of a record to Unvalidated in the following cases:

- When Form Builder creates a record, it sets its status to Unvalidated.
- When Form Builder sets the status of an item in the record to Unvalidated, it sets the status of the record to Unvalidated.
- When Form Builder duplicates a record (in response to the DUPLICATE_RECORD built-in or the [Duplicate Record] key), it sets the status of the duplicate record to Unvalidated.

**Valid**

Form Builder sets the status of a record to Valid in the following cases:

When Form Builder validates a record, it validates each item that does not already have a status of Valid. If all items have Valid status at the completion of this process, and if the WHEN-VALIDATE-RECORD trigger, if any, returns without raising the Form_Trigger_Failure exception, Form Builder sets the status of the record to Valid.

Form Builder sets to Valid the status of records fetched from the database.

When Form Builder successfully commits data to the database, it sets the status of each record in the form to Valid. Form Builder does not validate changes caused by triggers that fire during the commit transaction.

Note: See Changing Data During Commit Processing for exceptions to this rule.

- When Form Builder (in response to the DUPLICATE_RECORD built-in or the [Duplicate Record] key) duplicates a record that has Valid status, it sets the status of the duplicate record to Valid.

**Standard validation checks**

Form Builder uses the requirements specified in the Data and List of Values (LOV) sections of the property palette to validate an item.

**Validation sequence for text items**

For text items, Form Builder performs one of the following sequences of validation steps, depending on whether the item's value is null or not-null. If the text item fails a validation step, Form Builder notes the failure and omits the subsequent validation steps.

**When the item's value is null:**

1. If the item's Required property has a setting of Yes, the item fails validation -- unless item-level validation is occurring and one of the following is true:

- The form's Defer Required Enforcement property has the setting Yes or 4.5. (For a discussion of validation behavior in these cases, see Defer Required Enforcement property.)

- The item instance does not allow end-user update.

Under either of the above exceptional conditions, Form Builder defers validating the null item until record-level validation. (Note that if the item instance does not allow end-user update, validation is deferred even if Defer Required Enforcement is set to No.)

2 If the item's Required property has a setting of No, then the WHEN-VALIDATE-ITEM trigger is fired. If the trigger raises the Form_Trigger_Failure exception, the item fails this validation step.

**When the item's value is not null:**

1. Check data type. If the value does not match the item's data type, the item fails this validation step.
Form Builder tries to convert a user-entered value to the item's data type.
If the item has a format mask, Form Builder uses it. If the item has no format mask and its data type is DATE or DATETIME, Form Builder uses the input format mask derived at Form Builder startup.
If the item has no format mask and is of data type NUMBER or ALPHA, Form Builder checks for a valid numeric or alphabetic text string.

2. Check length. If the value does not match the item's Fixed Length property, the item fails this validation step.

3. Check range. If the value does not match the item's Lowest Allowed Value and Highest Allowed Value properties, the item fails this validation step.

4 If the item's Use LOV for Validation property has the value Yes, verify that the value appears in the item's list of values.

5. Fire the POST-CHANGE trigger. If the trigger raises the Form_Trigger_Failure exception, the item fails this validation step.

6. Fire the WHEN-VALIDATE-ITEM trigger. If the trigger raises the Form_Trigger_Failure exception, the item fails this validation step.

**Validation sequence for non-text items**

If the item's Item Type property does not have the value Text Item, Form Builder follows the above sequence, but omits steps 3 through 5. It also omits step 2 unless the Item Type property has the value List Item and the List Style property has the value Combo Box.

**Validation of synchronized items**

If items, say B and C, have their Synchronize with Item properties set to the name of another item, say A, then A, B, and C form a set of synchronized items. A is the master item of the set, and B and C are subordinate items. Synchronized items were called mirror items in earlier versions of Form Builder.
The items of a synchronized set share a common data value, but can have different item properties. Form Builder takes data-specific properties and triggers from the master item and ignores those properties and triggers for the subordinate items.

When validating the value assigned to a synchronized set of items, Form Builder uses the following data-specific properties and triggers from the master item:

- Required property
- Highest Allowed Value property
- Lowest Allowed Value property
- Initial Value
- POST-CHANGE trigger
- WHEN-VALIDATE-ITEM trigger

If the subordinate item has values for these properties or specifications for these triggers, Form Builder ignores them and issues a warning.
Form Builder takes the remaining validation properties from the item through which the data value is set—either programmatically or by end user action. In particular, Form Builder uses the Fixed Length and Validate from List properties of that item.
The SYSTEM.TRIGGER_ITEM variable contains the name of the item through which the data value was set. This allows code in the POST-CHANGE or WHEN-VALIDATE-ITEM trigger to refer to properties of that item, even though these triggers are defined for the master item.

Form Builder considers a value that has not been set programmatically or by the end user to have been set by the master item, because that is the source Form Builder uses for the initial value.
During record-level validation, Form Builder validates the item through which the data value was set. If validation fails, Form Builder gives focus to that item, unless the item instance does not allow end user input. In that case, Form Builder tries to give focus to another item of the synchronized set that does allow end user input. If it fails to find one, it tries to give focus to an item of the set for which the Enabled property has the value Yes.

**Block Level Properties**

## Navigation Style property

Specifies how a Next Item or Previous Item operation is processed when the input focus is in the last navigable item or first navigable item in the block, respectively.

**Same Record** (Default): A Next Item operation from the block's last navigable item moves the input focus to the first navigable item in the block, in that same record.
**Change Record**: A Next Item operation from the block's last navigable item moves the input focus to the first navigable item in the block, in the next record. If the current record is the last record in the block and there is no open query, Form Builder creates a new record. If there is an open query in the block (the block contains queried records), Form Builder retrieves additional records as needed.
**Change Block**: A Next Item operation from the block's last navigable item moves the input focus to the first navigable item in the first record of the next block. Similarly, a Previous Item operation from the first navigable item in the block moves the input focus to the last item in the current record of the previous block. The Next Navigation Block and Previous Navigation Block properties can be set to redefine a block's "next" or "previous" navigation block.

## Previous Navigation Block property

Specifies the name of the block that is defined as the "previous navigation block" with respect to this block. By default, this is the block with the next lower sequence in the form, as indicated by the order of blocks in the Object Navigator.

## Previous Navigation Block property

Specifies the name of the block that is defined as the "next navigation block" with respect to this block. By default, this is the block with the next higher sequence number in the form, as indicated by the order of blocks listed in the Object Navigator.

## Query Array Size property

Specifies the maximum number of records that Form Builder should fetch from the database at one time.

## Number of Records Buffered property

Specifies the minimum number of records buffered in memory during a query in the block.

## Number of Records Displayed property

Specifies the maximum number of records that the block can display at one time. The default is 1 record. Setting Number of Records Displayed greater than 1 creates a multi-record block.

## Query All Records property

Specifies whether all the records matching the query criteria should be fetched into the data block when a query is executed.
Yes - Fetches all records from query; equivalent to executing the **EXECUTE_QUERY** (ALL_RECORDS) built-in.
No - Fetches the number of records specified by the **Query Array Size block property**.

## Record Orientation property

Determines the orientation of records in the block, either horizontal records or vertical records. When you set this property, Form Builder adjusts the display position of items in the block accordingly.

## Single Record property

Specifies that the control block always should contain one record. Note that this differs from the number of records displayed in a block.

## Database Block property

Specifies that the block is based on any of the following block data source types: table, procedure, transactional trigger, or sub-query. Yes or No.

## Enforce Primary Key (Block) property

Indicates that any record inserted or updated in the block must have a unique key in order to avoid committing duplicate rows to the block's base table.

## Query Allowed (Block) property

Specifies whether Form Builder should allow the end user or the application to execute a query in the block. When Query Allowed is No, Form Builder displays the following message if the end user attempts to query the block:
FRM-40360: Cannot query records here.

## Query Data Source Type property

Specifies the query data source type for the block.  A query data source type can be a Table, Procedure, Transactional Trigger, or FROM clause query.

## Query Data Source Columns property

Specifies the names and datatypes of the columns associated with the block's query data source.  The Query Data Source Columns property is valid only when the Query Data Source Type property is set to Table, Sub-query, or Procedure.

## Query Data Source Arguments property

Specifies the names, datatypes, and values of the arguments that are to be passed to the procedure for querying data.  The Query Procedure Arguments property is valid only when the Query Data Source Type property is set to Procedure.

## Alias property

Establishes an alias for the table that the data block is associated with. The Data Block wizard sets the Alias property to the first letter of the table name.  (For example, a table named DEPT would have a default alias of D.)

## Include REF Item property

Creates a hidden item called REF for this block.  This item is used internally to coordinate master-detail relationships built on a REF link.  This item also can be used programmatically to access the object Id (OID) of a row in an object table.

## WHERE Clause/ORDER BY Clause

The defaults WHERE Clause and default ORDER BY Clause properties specify standard SQL clauses for the default SELECT statement associated with a data block. These clauses are automatically appended to the SELECT statement that Form Builder constructs and issues whenever the operator or the application executes a query in the block.

## Optimizer Hint property

Specifies a hint string that Form Builder passes on to the RDBMS optimizer when constructing queries.  Using the optimizer can improve the performance of database transactions.

## Insert Allowed (Block) property

Specifies whether records can be inserted in the block.

## Update Allowed (Block) property

Determines whether end users can modify the values of items in the block that have the Update Allowed item property set to Yes. (Setting Update Allowed to No for the block overrides the Update Allowed setting of any items in the block.)

## Locking Mode property

Specifies when Form Builder tries to obtain database locks on rows that correspond to queried records in the form. The following table describes the allowed settings for the Locking Mode property:

Automatic (default): Identical to Immediate if the datasource is an Oracle database. For other datasources, Form Builder determines the available locking facilities and behaves as much like Immediate as possible.
Immediate: Form Builder locks the corresponding row as soon as the end user presses a key to enter or edit the value in a text item.
Delayed: Form Builder locks the row only while it posts the transaction to the database, not while the end-user is editing the record. Form Builder prevents the commit action from processing if values of the fields in the block have changed when the user causes a commit action.

## Delete Allowed property

Specifies whether records can be deleted from the block.

## Key Mode property

Specifies how Form Builder uniquely identifies rows in the database. By default, the ORACLE database uses unique ROWID values to identify each row. Non-ORACLE databases do not include the ROWID construct, but instead rely solely on unique primary key values to identify unique rows. If you are creating a form to run against a non-ORACLE data source, you must use primary keys, and set the Key Mode block property accordingly.

Automatic (default): Specifies that Form Builder should use ROWID constructs to identify unique rows in the datasource but only if the datasource supports ROWID.
Non-Updateable: Specifies that Form Builder should not include primary key columns in any UPDATE statements. Use this setting if your database does not allow primary key values to be updated.

Unique: Instructs Form Builder to use ROWID constructs to identify unique rows in an ORACLE database.
Updateable: Specifies that Form Builder should issue UPDATE statements that include primary key values. Use this setting if your database allows primary key columns to be updated and you intend for the application to update primary key values.

## Update Changed Columns Only property

When queried records have been marked as updates, specifies that only columns whose values were actually changed should be included in the SQL UPDATE statement that is sent to the database during a COMMIT. **By default**, Update Changed Columns Only is set to **No**, and all columns are included in the UPDATE statement.

## Enforce Column Security property

Specifies when Form Builder should enforce update privileges on a column-by-column basis for the block's base table. If an end user does not have update privileges on a particular column in the base table, Form Builder makes the corresponding item non-updateable for this end user only, by turning off the Update Allowed item property at form startup.
**Yes**: Form Builder enforces the update privileges that are defined in the database for the current end user.
**No:** Form Builder does not enforce the defined update privileges.

## DML Data Target Type property

Specifies the block's DML data target type. A DML data target type can be a **Table, Procedure, or Transactional Trigger**.

## DML Data Target Name property

Specifies the name of the block's DML data target. The DML Data Target Name property is valid only when the DML Data Target Type property is set to Table.

## Visual Attribute Group property

Specifies how the object's individual attribute settings (Font Name, Background Color, Fill Pattern, etc.) are derived.

**Default:** Specifies that the object should be displayed with default color, pattern, and font settings. When Visual Attribute Group is set to **Default,** the individual attribute settings

reflect the current system defaults. The actual settings are determined by a combination of factors, including the type of object, the resource file in use, and the platform.
**Named visual attribute**: Specifies a named visual attribute that should be applied to the object. Named visual attributes are separate objects that you create in the Object Navigator and then apply to interface objects, much like styles in a word processing program. When Visual Attribute Group is set to a named visual attribute, the individual attribute settings reflect the attribute settings defined for the named visual attribute object. When the current form does not contain any named visual attributes, the poplist for this property will show Default.

## White on Black property

Specifies that the object is to appear on a **monochrome bitmap** display device as white text on a black background.

## Listed in Data Block Menu/Data Block Description

Specifies whether the block should be listed in the block menu and, if so, the description that should be used for the block.
Form Builder has a built-in block menu that allows end users to invoke a list of blocks in the current form by pressing [Block Menu]. When the end user selects a block from the list, Form Builder navigates to the first enterable item in the block.

**Item Level Properties**

## Item Type property

Specifies the type of item. An item can be one of the following types:

ActiveX Control (32-bit Windows platforms)
Bean Area
Chart Item
Check Box
Display Item
Hierarchical Tree
Image
List Item
OLE Container
Push Button
Radio Group
Sound
Text Item

User Area
VBX Control (Microsoft Windows 3.1 only)

## Subclass Information property

Specifies the following information about the source object and source module for referenced objects.
**Module:** The name of the source module.
**Storage:** The source module type (Form or Menu) and location (File System or Database)
**Name:** The name of the source object in the source module. (The name of a reference object can be different than the name of its source object.)

## Enabled (Item) property

Determines whether end users can use the mouse to manipulate an item.
On most window managers, Enabled set to No grays out the item.

## Justification property

Specifies the text justification within the item.

**Left**: Left justified, regardless of Reading Order property.
**Center**: Centered, regardless of Reading Order property.
**Right**: Right justified, regardless of Reading Order property.
**Start**: Item text is aligned with the starting edge of the item-bounding box. The starting edge depends on the value of the item's Reading Order property.
Start is evaluated as Right alignment when the reading order is Right To Left, and as Left alignment when the reading order is left to Right.
**End**: Item text is aligned with the ending edge of the item-bounding box. The ending edge depends on the value of the item's Reading Order property.
End is evaluated as Left alignment when the reading order is Right To Left, and as Right alignment when the reading order is left to Right.

## Multi-Line property

Determines whether the text item is a single-line or multi-line editing region.
Setting the Multi-line property Yes allows a text item to store multiple lines of text, but it does not automatically make the item large enough to display multiple lines. It is up to you to set the Width, Height, Font Size, and Maximum Length properties to ensure that the desired number of lines and characters are displayed.
**Single-line**: Pressing the carriage return key while the input focus is in single-line text item initiates a [Next Item] function.

**Multi-line**: Pressing the carriage return key while the input focus is in a multi-line text item starts a new line in the item.

## Wrap Style property

Specifies how text is displayed when a line of text exceeds the width of a text item or editor window.

NONE: No wrapping: text exceeding the right border is not shown.
CHARACTER: Text breaks following the last visible character, and wraps to the next line.
WORD: Text breaks following last visible complete word, and wraps to the next line.

## Case Restriction property

Specifies the case for text entered in the text item or menu substitution parameter.

MIXED: Text appears as typed.
UPPER: Lower case text converted to upper case as it is typed.
LOWER: Upper case text converted to lower case as it is typed.

## Conceal Data property

Hides characters that the operator types into the text item. This setting is typically used for password protection.

| | |
|---|---|
| Yes | Disables the echoing back of data entered by the operator. |
| No | Enables echoing of data entered by the operator. |

## Keep Cursor Position property

Specifies that the cursor position be the same upon re-entering the text item as when last exited.

## Automatic Skip (Item) property

Moves the cursor to the next navigable item when adding or changing data in the last character of the current item. The last character is defined by the Maximum Length property.

## Popup Menu property

Specifies the popup menu to display for the canvas or item.

## Keyboard Navigable property

Determines whether the end user or the application can place the input focus in the item during default navigation. When set to Yes for an item, the item is navigable. When set to No, Form Builder skips over the item and enters the next navigable item in the default navigation sequence.

At runtime, when the Enabled property is set to PROPERTY_FALSE, the Keyboard_Navigable property is also set to PROPERTY_FALSE. However, if the Enabled property is subsequently set back to PROPERTY_TRUE, the keyboard Navigable property is NOT set to PROPERTY_TRUE, and must be changed explicitly.

When Keyboard Navigable is specified at multiple levels (item instance, item, and block), the values are ANDed together. This means that setting Keyboard Navigable to Yes (or NAVIGABLE to PROPERTY_TRUE for runtime) has no effect at the item instance level unless it is set consistently at the item level. For example, your user cannot navigate to an item instance if Keyboard Navigable is true at the instance level, but not at the item level.

You can use the GO_ITEM built-in procedure to navigate to an item that has Keyboard Navigable property set to (PROPERTY_FALSE) for runtime.

## Previous Navigation Item property

Specifies the name of the item that is defined as the"previous navigation item" with respect to the current item. By default, this is the item with the next lower sequence in the form, as indicated by the order of items in the Object Navigator.

## Next Navigation Item property

Specifies the name of the item that is defined as the "next navigation item" with respect to this current item. By default, the next navigation item is the item with the next higher sequence as indicated by the order of items in the Object Navigator.

## Copy Value from Item property

Specifies the source of the value that Form Builder uses to populate the item. When you define a master-detail relation, Form Builder sets this property automatically on the foreign key item(s) in the detail block. In such cases, the Copy Value from Item property names the primary key item in the master block whose value gets copied to the foreign key item in the detail block whenever a detail record is created or queried.

Specify this property in the form <block_name>.<block_item_name>.
Setting the Copy Value from Item property does not affect record status at runtime, because the copying occurs during default record processing.
To prevent operators from de-enforcing the foreign key relationship, set the Enabled property to No for the foreign key items.
To get the Copy Value from Item property programmatically with GET_ITEM_PROPERTY, use the constant ENFORCE_KEY.

## Synchronize with Item property

Specifies the name of the item from which the current item should derive its value. Setting this property synchronizes the values of the two items, so that they effectively mirror each other. When the end user or the application changes the value of either item, the value of the other item changes also.

In earlier releases, this property was called the Mirror Item property.
You can set Synchronize with Item for base table or control blocks. When Synchronize with Item is specified, the current item's Base Table Item property is ignored, and the item derives its value from the mirror item specified, rather than from a column in the database.
If you use the GET_ITEM_PROPERTY built-in to obtain a Base Table Item property, it will obtain the value from the mirror item specified.
You can use mirror item to create more than one item in a block that display the same database column value.

## Calculation Mode property

Specifies the method of computing the value of a calculated item.

**None** (default): Indicates the item is not a calculated item.
**Formula**: Indicates the item's value will be calculated as the result of a user-written formula. You must enter a single PL/SQL expression for an item's formula. The expression can compute a value, and also can call a Form Builder or user-written subprogram.
**Summary**: Indicates the item's value will be calculated as the result of a summary operation on a single form item. You must specify the summary type, and the item to be summarized.

## Primary Key (Item) property

Indicates that the item is a base table item in a data block and that it corresponds to a primary key column in the base table. Form Builder requires values in primary key items to be unique.

## Query Only property

Specifies that an item can be queried but that it should not be included in any INSERT or UPDATE statement that Form Builder issues for the block at runtime.

## Query Length property

Specifies the number of characters an end user is allowed to enter in the text item when the form is Enter Query mode.
You can make the query length greater than the Maximum Length when you want to allow the end user to enter complex query conditions. For example, a query length of 5 allows an end user to enter the query condition !=500 in a text item with a Maximum Length of 3.

## Case Insensitive Query property

Determines whether the operator can perform case-insensitive queries on the text item.
Case-insensitive queries are optimized to take advantage of an index. For example, assume you perform the following steps:

Create an index on the EMP table.
Set the Case Insensitive Query property on ENAME to Yes.
In Enter Query mode, enter the name 'BLAKE' into: ENAME.
Execute the query.

Form Builder constructs the following statement:

```
SELECT * FROM EMP WHERE UPPER (ENAME) = 'BLAKE' AND
  (ENAME LIKE 'Bl%' OR ENAME LIKE 'bL%' OR
  ENAME LIKE 'BL%' OR ENAME LIKE 'bl%');
```

The last part of the WHERE clause is performed first, making use of the index. Once the database finds an entry that begins with bl, it checks the UPPER (ENAME) = 'BLAKE' part of the statement, and makes the exact match.

## Update Only if NULL property

Indicates that operators can modify the value of the item only when the current value of the item is NULL.

## Lock Record property

Specifies that Form Builder should attempt to lock the row in the database that corresponds to the current record in the block whenever the text item's value is modified, either by the end user or programmatically.

Set this property to Yes when the text item is a control item but you still want Form Builder to lock the row in the database that corresponds to the current record in the block.
Useful for lookup text items where locking underlying record is required.
To set the Lock Record property with SET_ITEM_PROPERTY, use the constant LOCK_RECORD_ON_CHANGE.

## List of Values property

Specifies the list of values (LOV) to attach to the text item. When an LOV is attached to a text item, end users can navigate to the item and press [List of Values] to invoke the LOV.

## Validate from List property

Specifies whether Form Builder should validate the value of the text item against the values in the attached LOV.
When Validate from List is Yes, Form Builder compares the current value of the text item to the values in the first column displayed in the LOV whenever the validation event occurs:
If the value in the text item matches one of the values in the first column of the LOV, validation succeeds, the LOV is not displayed, and processing continues normally.
If the value in the text item does not match one of the values in the first column of the LOV, Form Builder displays the LOV and uses the text item value as the search criteria to automatically reduce the list.
Note, however, that a When-Validate-Item trigger on the item still fires, and any validation checks you perform in the trigger still occur.

## Editor property

Specifies that one of the following editors should be used as the default editor for this text item:
A user-named editor that you defined in the form or
A system editor outside of Form Builder that you specified by setting the SYSTEM_EDITOR environment variable

## Visible property

Indicates whether the object is currently displayed or visible. Set Visible to Yes or No to show or hide a canvas or window.

You cannot hide the canvas that contains the current item.

You can hide a window that contains the current item.

When you use GET_WINDOW_PROPERTY to determine window visibility, Form Builder uses the following rules:

A window is considered visible if it is displayed, even if it is entirely hidden behind another window.

A window that has been iconified (minimized) is reported as visible to the operator because even though it has a minimal representation, it is still mapped to the screen.

When you use GET_VIEW_PROPERTY to determine canvas visibility, Form Builder uses the following rules:
A view is reported as visible when it is a) in front of all other views in the window or b) only partially obscured by another view.
A view is reported as not visible when it is a) a stacked view that is behind the content view in the window or b) completely obscured by a single stacked view.
The display state of the window does not affect the setting of the canvas VISIBLE property. That is, a canvas may be reported visible even if the window in which it is displayed is not currently mapped to the screen.

## Rendered property

Use the Rendered property to conserve system resources. A rendered item does not require system resources until it receives focus. When a rendered item no longer has focus, the resources required to display it are released.

## Hint (Item) property

Specifies item-specific help text that can be displayed on the message line of the root window at runtime. Hint text is available when the input focus is in the item.

## Display Hint Automatically property

Determines when the help text specified by the item property, Hint, is displayed:

- Set Display Hint Automatically to Yes to have Form Builder display the hint text whenever the input focus enters the item.
- Set Display Hint Automatically to No to have Form Builder display the hint text only when the input focus is in the item and the end user presses [Help] or selects the Help command on the default menu.

## Tooltip property

Specifies the help text that should appear in a small box beneath the item when the mouse enters the item.

**Canvas Level Properties**

## Primary Canvas property

Specifies the canvas that is to be the window's primary content view. At runtime, Form Builder always attempts to display the primary view in the window. For example, when you display a window for the first time during a session by executing the SHOW_WINDOW built-in procedure, Form Builder displays the window with its primary content view.
If, however, Form Builder needs to display a different content view because of navigation to an item on that view, the primary content view is superseded by the target view.

## Window Style property

Specifies whether the window is a Document window or a Dialog window. Document and dialog windows are displayed differently on window managers that support a Multiple Document Interface (MDI) system of window management.
**MDI** applications display a default parent window, called the application window. All other windows in the application are either document windows or dialog windows.
**Document windows** always remain within the application window frame. If the operator resizes the application window so that it is smaller than a document window, the document window is clipped. An operator can maximize a document window so that is occupies the entire workspace of the application window.
**Dialog windows** are free-floating, and the operator can move them outside the application window if they were defined as Movable. If the operator resizes the application window so that it is smaller than a dialog window, the dialog window is not clipped.

## Modal property

Specifies whether a window is to be modal. Modal windows require an end user to dismiss the window before any other user interaction can continue.

## Hide on Exit property

For a modeless window, determines whether Form Builder hides the window automatically when the end user navigates to an item in another window.

## Close Allowed property

Specifies whether the window manager-specific Close command is enabled or disabled for a window.

- Setting Close Allowed to Yes enables the Close command so that the Close Window event can be sent to Form Builder when the operator issues the Close command. However, to actually close the window in response to this event, you must write a When-Window-Closed trigger that explicitly closes the window. You can close a window programmatically by calling HIDE_WINDOW, SET_WINDOW_PROPERTY, or EXIT_FORM.
- On Microsoft Windows, if the operator closes the MDI parent window, Form Builder executes DO_KEY('Exit_Form') by default.

## Move Allowed property

Specifies whether or not the window can be moved. Windows can be moved from one location to another on the screen by the end user or programmatically by way of the appropriate built-in subprogram.

## Resize Allowed property

Specifies that the window is to be a fixed size and cannot be resized at runtime. The Resize Allowed property prevents an end user from resizing the window, but it does not prevent you from resizing the window programmatically with RESIZE_WINDOW or SET_WINDOW_PROPERTY.

## Maximize Allowed property

Specifies that end users can resize the window by using the zooming capabilities provided by the runtime window manager.

## Minimize Allowed property

Specifies that a window can be iconified on window managers that support this feature.

## Minimized Title property

Specifies the text string that should appear below an iconified window.

## Icon Filename property

Specifies the name of the icon resource that you want to represent the iconic button, menu item, or window.

## Inherit Menu property

Specifies whether the window should display the current form menu on window managers that support this feature.

**Canvas Level Properties**

## Canvas Type property

Specifies the type of canvas. The type determines how the canvas is displayed in the window to which it is assigned, and determines which properties make sense for the canvas.
**Content**: The default. Specifies that the canvas should occupy the entire content area of the window to which it is assigned. Most canvases are content canvases.
**Stacked**: Specifies that the canvas should be displayed in its window at the same time as the window's content canvas. Stacked views are usually displayed programmatically and overlay some portion of the content view displayed in the same window.
**Vertical Toolbar Canvas**: Specifies that the canvas should be displayed as a vertical toolbar under the menu bar of the window. You can define iconic buttons, pop-lists, and other items on the toolbar as desired.
**Horizontal Toolbar Canvas**: Specifies that the canvas should be displayed as a horizontal toolbar at the left side of the window to which it is assigned.

## Raise on Entry property

For a canvas that is displayed in the same window with one or more other canvases, Raise on Entry specifies how Form Builder should display the canvas when the end user or the application navigates to an item on the canvas.

- When Raise on Entry is No, Form Builder raises the view in front of all other views in the window only if the target item is behind another view.

- When Raise on Entry is Yes, Form Builder always raises the view to the front of the window when the end user or the application navigates to any item on the view.