Infosys®
POWERED BY INTELLECT
DRIVEN BY VALUES

RDBMS- Day5

- Correlated Sub Queries
- Exists and Not Exists
- Views
- Data Control Language
- Embedded SQL

Education and Research

In today's session we would be discussing about the concept of views, the concept of an index and how it is useful in database implementations and the concept embedded SQL programming using pro*c.

Correlated Sub Queries

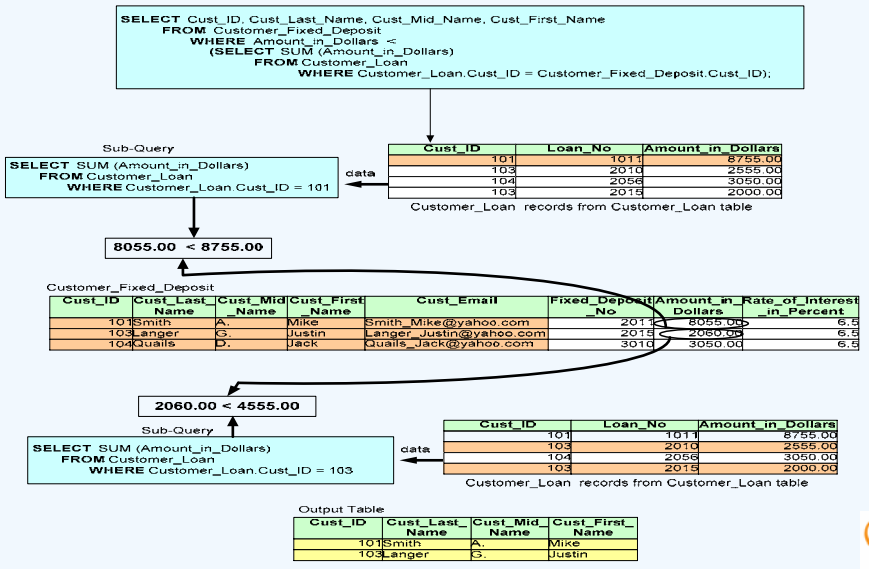
- You can refer to the table in the FROM clause of the outer query in the inner query using Correlated sub-queries.
- The inner query is executed separately for each row of the outer query.
  (i.e. In Co-Related Sub-queries, SQL performs a sub-query over and over again – once for each row of the main query.)

Correlated Sub Queries

To list all Customers who have a fixed deposit of amount less than the sum of all their loans.

```
Select Cust_Id, Cust_Last_Name, cust_Mid_Name, cust_First_Name
From Customer_fixed_Deposit
Where amount_in_dollars
        <
        (Select sum(amount_in_dollars)
        From Customer_Loan
        Where Customer_Loan.Cust_Id = Customer_Fixed_Deposit.Cust_ID);
```

Correlated Sub Queries (Contd…)



Correlated Sub Queries

List customer IDs of all customers who have both a Fixed Deposit and a loan at any of Bank Branches

```
SELECT Cust_ID
FROM Customer_Details
WHERE Cust_ID
      IN
     (SELECT Cust_ID
      FROM Customer_Loan
      WHERE Customer_Loan.Cust_ID  = Customer_Details.Cust_ID)
      AND Cust_ID IN
            (SELECT Cust_ID
             FROM Customer_Fixed_Deposit
           WHERE Customer_Fixed_Deposit.Cust_ID  = Customer_Details.Cust_ID);
```

Correlated Sub Queries ...

Get S# for suppliers supplying some project with P1 in a quantity greater than the average qty of P1 supplied to that project

```
SELECT  DISTINCT  S#
        FROM  Shipments  X
                WHERE  P# = 'P1'
                AND  QTY >  (SELECT  AVG(QTY)
                                FROM  Shipments Y
                                        WHERE  P# = 'P1'
                                        AND  X.J# = Y.J#)
```

Correlated Sub Queries

Get P# for all parts supplied by more than one supplier

```
SELECT P#
       FROM Shipment X
            WHERE P# IN
                  (SELECT P#
                        FROM Shipment Y
                             WHERE Y.S# <> X.S#)
```

Infosys®
POWERED BY INTELLECT
DRIVEN BY VALUES

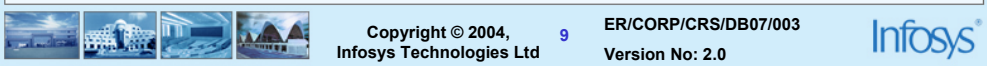
EXISTS Vs NOT EXISTS

Education and Research

Retrieval using EXISTS

List all Customers who have at least one Fixed Deposit more than $3000.00.

```
SELECT Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
FROM Customer_Details S
WHERE
    EXISTS
   (SELECT *
   FROM Customer_Fixed_Deposit O
   WHERE O.Amount_in_Dollars > 3000.00 AND O.Cust_ID = S.Cust_ID);
```



Exists check for the existence of a situation/condition.

Retrieval using EXISTS

List all Customers who have both a Fixed Deposit and a Loan at the Bank

```
SELECT Cust_ID
FROM Customer_Fixed_Deposit
WHERE EXISTS
    (SELECT *
    FROM Customer_Loan
    WHERE Customer_Loan.Cust_ID = Customer_Fixed_Deposit.Cust_ID);
```

Let us take another example:
Find the deptno of those departments having employees who can do some work done by employee in depertment d1.

The solution would look like:

```
Select deptno
        from department
                where exists(
            select *
                                                    from employee x

        where x.dept= 'd1' and exists
                                    (select *
                                                from employee y

                                                    y.job=x.job and
                                        y.dept=department.deptno)

                                    )

          and deptno <> 'd1';
```

Retrieval using NOT EXISTS

List all Customers who don't have a single Fixed Deposit over $3000.00.

```
SELECT Cust_ID, Cust_Last_Name, Cust_Mid_Name, Cust_First_Name
FROM Customer_Details S
WHERE
        NOT EXISTS
  (SELECT *
   FROM Customer_Fixed_Deposit O
   WHERE O.Amount_in_Dollars > 3000.00 AND O.Cust_ID = S.Cust_ID);
```

A not exists checks for the opposite condition than an Exists. It checks for the non-existence of a condition.

Not exists can be used to arrive at a result which can't otherwise be arrived at using exists.

The logic is as follows:
For all x there exists an y such that f(x,y) is true is the same as
There does not exist and x for which there does not exists a y so that f(x,y)is true

Taking a slight variant of the previous example:
Find the deptno of those departments having employees who can do all work done by employees in depertment d1.

```
Select deptno
                from department
                                where not exists(
                                                                                select *
                                                                                from employee x

                where x.dept= 'd1' and not exists(

                                                                select * from employee y

                                                                                y.job=x.job and
                                                                  y.dept=department.deptno)
                                                )
  and deptno <> 'd1'
```

“Views “ is an important and useful feature in RDBMS. It helps achieve sharing with proper security. We will be looking at

What is a view

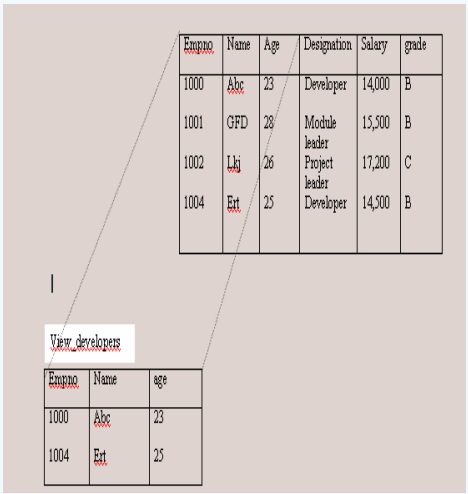
How to create a view

How to update/delete a view

Different types of views

Advantages and disadvantages of views in the next few slides

What is a view?



- A view is a kind of “virtual table”
- Contents are defined by a query like:
  Select Empno, Name, age from Employee
  Where designation=‘developer’;
  As shown in the figure

Views are tables whose contents are taken or derived from other tables.

To the user, the view appears like a table with columns and rows

But in reality, the view doesn’t exists in the database as a stored set of values

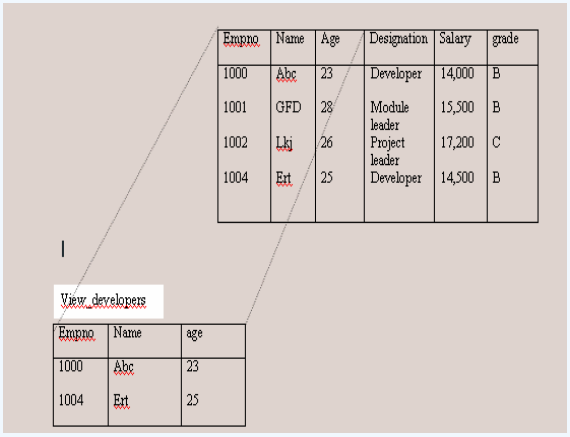
The rows and columns that we find inside a view are actually the results generated by a query that defines the view

View is like a window through which we see a limited region of the actual table

The table from where the actual data is obtained is called the source table

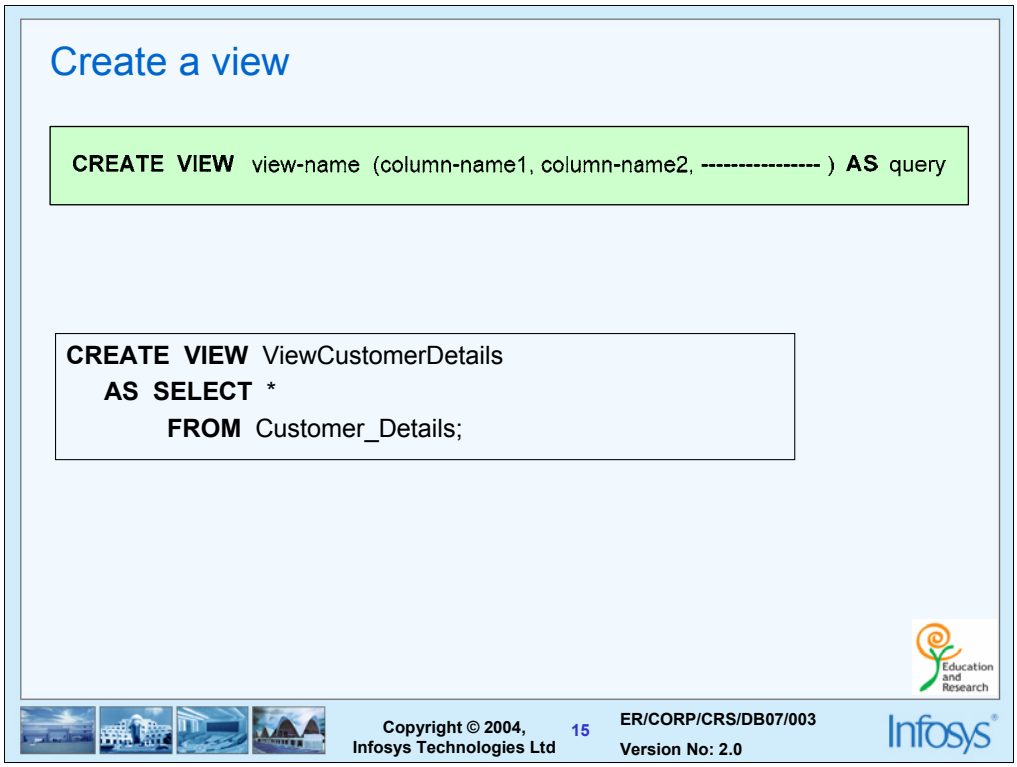
What is a view to the DBMS

- We can use views in select statements like
- Select * from view_employees where age > 23;
- DBMS translates the request to an equivalent request to the source table



For simple queries the results are generated row by row on the fly.

For more complex views the DBMS must generate a temporary table representing the view and later discard it.



You must have permission on the table refereed in the view to successfully create the view

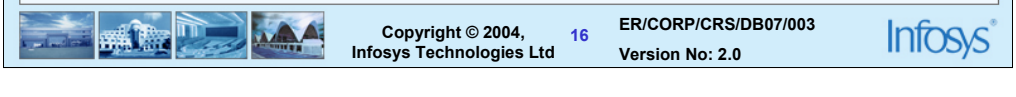
Can assign a new name to each column chosen in the view

Only names can be different. The data type etc remain the same as the source table because, the values are after all going to be derived from that table

If no names are specified for columns, the same name as used in the source table is used

Assigning names to columns

```
Create view vwCustDetails (CustCode,CustLname,CustFName)
    AS  Select Cust_Id,Cust_Last_Name,Cust_First_Name
          From Customer_details
```

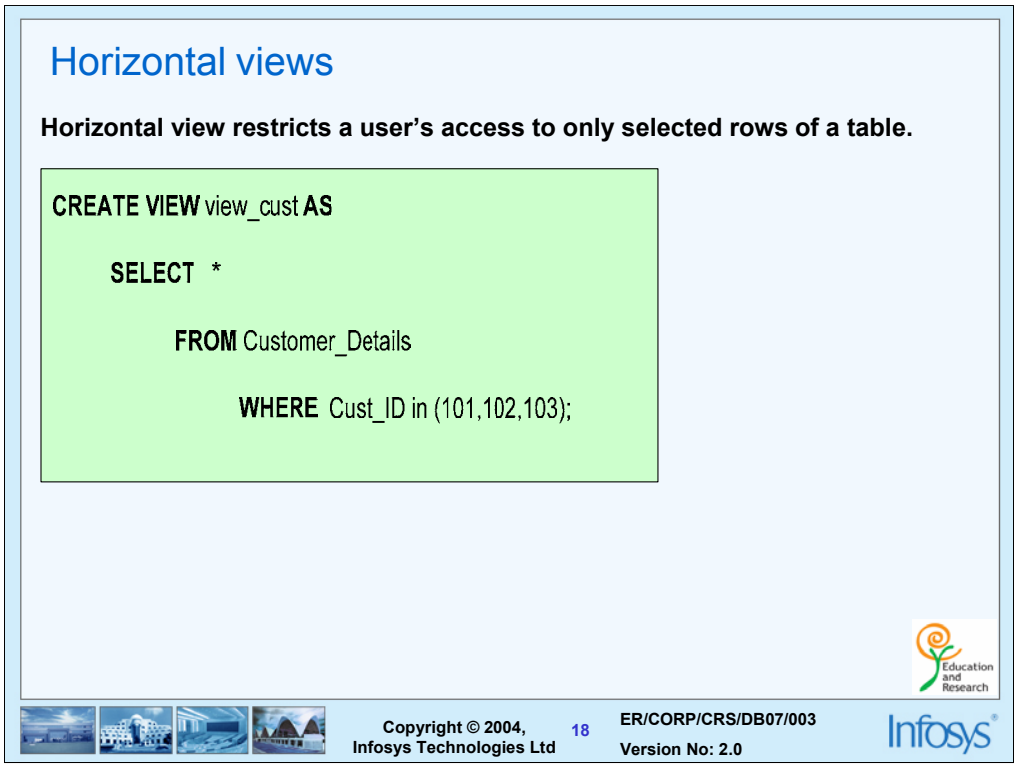


We can assign names for the various columns in the view. This may be entirely different from what has been used in the source table

Types of views

- Horizontal views
- Vertical views
- Row/column subset views
- Grouped views
- Joined views

Based on how the view is created. We will discuss them in detail in the following slides



Here a horizontal subset of the source table is taken to create the view.

Very useful when data belonging to different categories are present in the table.

A private(virtual) table for each category can be created and given access to the person concerned

Vertical views

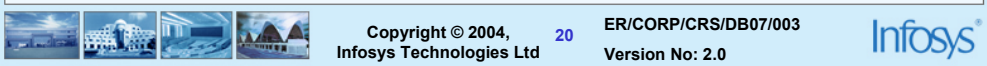
- A view which selects only few columns of a table:
- Vertical view restricts a user's access to only certain columns of a table

```
CREATE VIEW view_cust AS
        SELECT Cust_ID, Account_No, Account_Type
                FROM Customer_Details;
```

Helps restricting access to sensitive information like salary, grade etc.

Row/column subset views

```
CREATE VIEW View_Cust_VertHor
 AS   SELECT  Cust_Id,Account_No,Account_Type
            FROM Customer_Details
                    WHERE CUST_ID IN (101,102,103);
```



SQL standard doesn't define any notion of horizontal or vertical views . It is for our own understanding that we have given names for views which select only selected rows/columns from the source table as horizontal/vertical views. There could be a combination of these two concepts where a view selects a subset of rows and columns

Views with Group By clause

- The query contains a group by clause

```
CREATE VIEW  View_GroupBY(Dept,NoofEmp)
AS SELECT Department, count(Employee_ID)
        FROM Employee_Manager
                GROUP BY Department;
```

Grouped views are created based on a grouped query. They group rows of data and produce one row in the result corresponding to each group

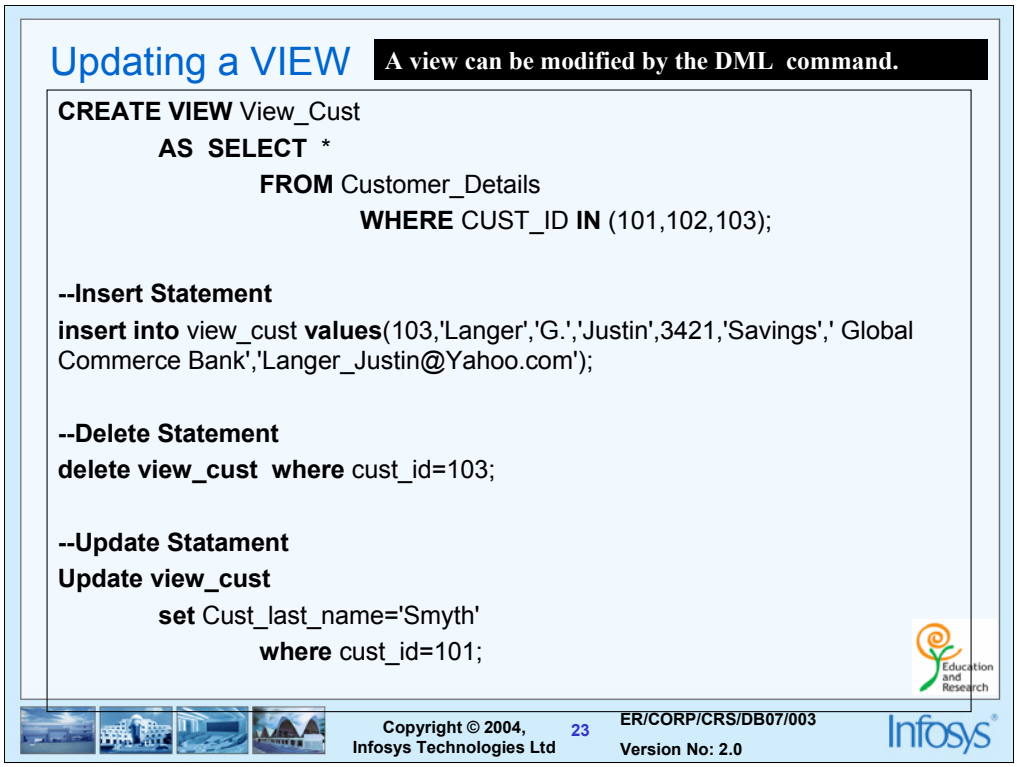
So, the rows in the view don't have a one to one correspondence with the rows of the source table

For this reason , grouped views cannot be updated.

Views with Joins

- Created by specifying a two-table or three-table query in the view creation command

```
Create view View_Cust_Join as
 Select  a.Cust_Id,b.Cust_First_Name,b.Cust_Last_Name,Amount_in_dollars
          from Customer_loan a, customer_details b
                  where a.cust_id = b.cust_id;
```



Depending on the commercial implementation being used, views may or may not be updateable. In some cases, all views are not updateable. In some others a view is updateable if it is from one table, else it is not. In still others, a view is updateable if it is the result of a simple query; if it is defined by some GROUP BY statements, for example, the view is not updateable.

According to ANSI/ISO standards, A view is updatable if

The DBMS is able to trace back the rows and columns of the view to the corresponding rows and columns of the source table.

So a view is updatable if :

DISTINCT is not specified in the query used to create the view

The FROM clause specifies only one source table

The select list doesn't contain expressions/calculated columns

The WHERE clause doesn't include a subquery

The query doesn't include a GROUP BY or HAVING

An example of a view which can't be updated is given in the next slide

Updating View

A view can be updated if the query that defines the view meets all of these restrictions:

- DISTINCT must not be specified; that is, duplicate rows must not be eliminated from the query results
- The FROM clause must specify only one updateable table; the view must have a single underlying source table
- The SELECT list cannot contain expressions, calculated columns, or column functions
- The WHERE clause must not include a sub query; only simple row-by-row search conditions may appear

Dropping Views

Views are dropped similar to the way in which the tables are dropped. However, you must own a view in order to drop it.

DROP VIEW <view name>;

DROP VIEW View_Cust;

if some other views depend on this view, then if you say

DROP VIEW ViewSupplier CASCADE then this view plus all other views that are based on this view are deleted.

If you specify

DROP VIEW ViewSupplier RESTRICT then this view cannot be deleted if other views depend on it.

Checking View Updates :– Check Option

```
CREATE  VIEW  view_customer AS
        SELECT Cust_ID, Cust_Last_Name, Account_No, Account_Type, Bank_Branch
                FROM Customer_Details
                        WHERE Bank_Branch = 'Downtown' ;
```

```
INSERT INTO view_customer
VALUES (115, 'Costner', 107, 'Savings', 'Bridgewater');
```

Will it prevent insertion into Customer_details ?

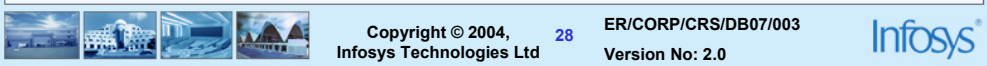
```
SELECT Cust_ID, Cust_Last_Name, Bank_Branch
FROM view_customer;
```

Solution is :

```
CREATE VIEW view_customer AS
    SELECT Cust_ID, Cust_Last_Name, Account_No, Account_Type, Bank_Branch
        FROM Customer_Details
            WHERE Bank_Branch = 'Downtown'
                With CHECK OPTION;
```

Advantages of views

- Security
- Query simplicity
- Structural simplicity



Security: only a limited set of rows/ columns are viewable by certain users

Query simplicity: A view can derive data from many tables. So, subsequently we can use queries on the view as single table queries rather than writing queries against the source tables as multi-table queries

Structural simplicity: views can show only a portion of the table which is relevant to the user there by keeping it simple.

Disadvantages of views

- Performance
- Restrictions

Performance: views based on joins are merely virtual tables. Every time a query is placed against the view, the query representing creation of the view has to be executed . So, complex joins may have to be performed every time a query is placed against the view.

Restrictions: Not all views are updateable

Infosys®
POWERED BY INTELLECT
DRIVEN BY VALUES

Data Control Language (DCL)

Education and Research

GRANT …. Tables or views

```
GRANT {
        [ALTER[, ]]
        [DELETE[, ]]
        [INDEX[, ]]
        [INSERT[, ]]
        [SELECT[, ]]
        [UPDATE [(column-name[,...])][, ]]
        | ALL [PRIVILEGES]]      }
ON [TABLE] {table-name[,...] | view-name[,...]}
TO [AuthID][,...]
[WITH GRANT OPTION]
```


Privileges on a specific table or a view created based on a table.

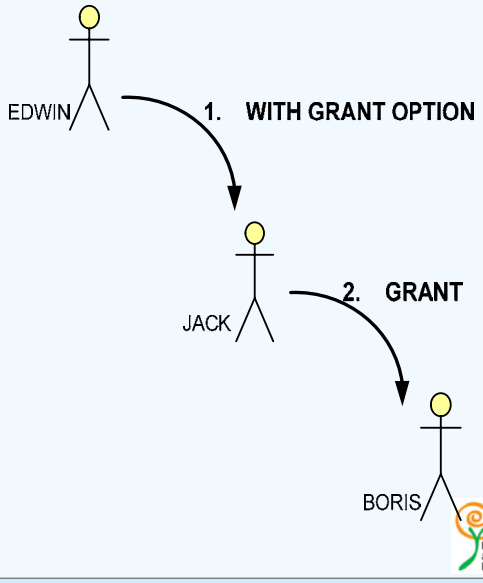
Grant

```
GRANT SELECT, INSERT
        ON Customer_Details
                TO Edwin ;
GRANT ALL PRIVILEGES
        ON Customer_Loan
                TO JACK ;
GRANT ALL
        ON Customer_Loan
                TO PUBLIC ;
```

Grant

- With Grant Option

GRANT SELECT
ON Customer_Loan
TO EDWIN
With GRANT OPTION;



Taking PRIVILIGES away

The syntax of REVOKE command is patterned after GRANT, but with a reverse meaning.

```
REVOKE{
        [ALTER[, ]]
        [DELETE[, ]]
        [INDEX[, ]]
        [INSERT[, ]]
        [SELECT[, ]]
        [UPDATE [(column-name[,...])][, ]]
        |  ALL [PRIVILEGES] }
ON [TABLE] {table-name[,...] | view-name [,...]}
FROM AuthID[,...]
```

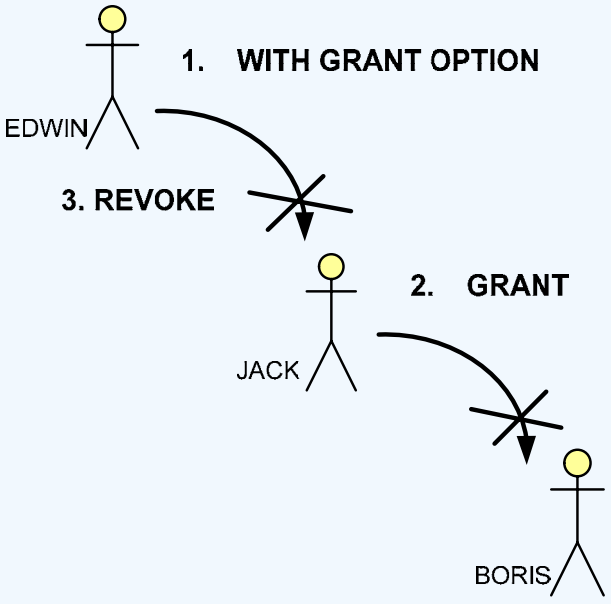
Revoke

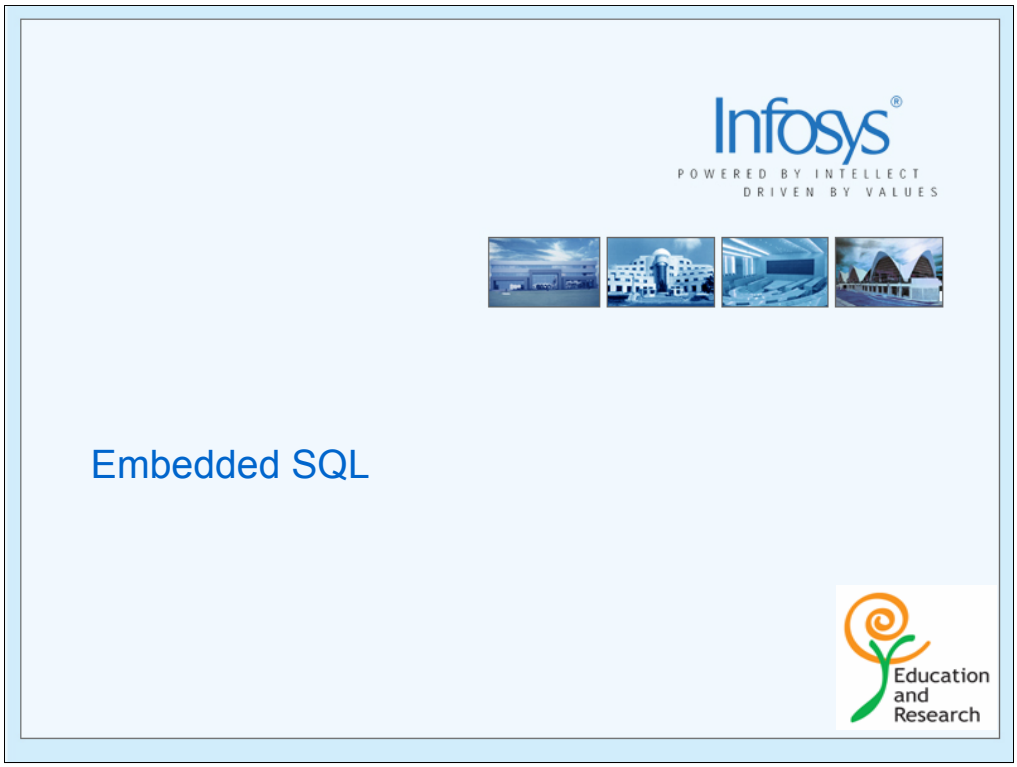
```
REVOKE SELECT, INSERT
    ON Customer_Details
        FROM Edwin ;

REVOKE ALL PRIVILEGES
    ON Customer_Loan
        FROM JACK ;

REVOKE ALL
    ON Customer_Loan
        FROM PUBLIC ;
```

Revoke





SQL can be used in two ways:

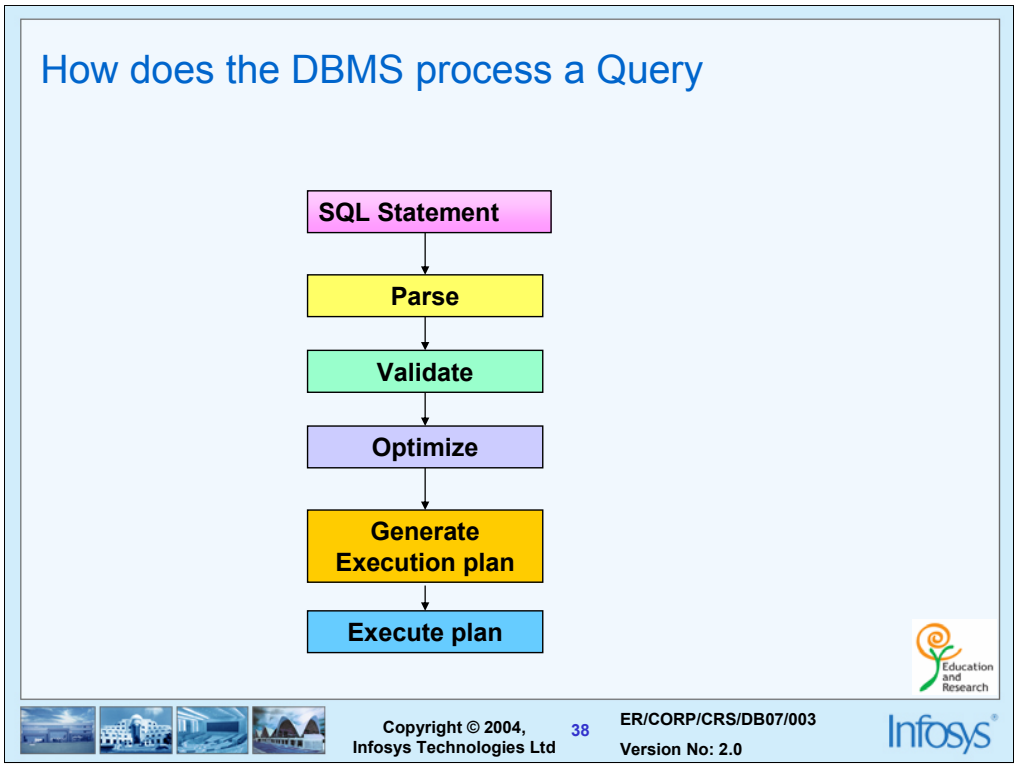
1. As an interactive database language to issue queries against the database and get results online
2. As a programmatic language used along with some high level language for performing database access

Based on this, commercial products support two flavors of using SQL with programming languages

a) Embedded SQL: directly writing SQL commands inside HLL (High level language) program interspersed with the other HLL code. A precompiler would process this kind of a code, separate out the programming language code and sql, a series of tools act on them and the executable is produced

b) API: Application programming interface. The program uses some API function calls and passes the sql as a parameter. This doesn't require any precompiler

The following discussions on embedded SQL are purely from a theoretical point of view. As far as this course is concerned, no practical is involved with respect to embedded SQL.



a) The first step in processing an SQL is **Parsing.** The statement is split into words and checked against syntax. Syntax errors and spelling mistakes can be detected at this stage

b) The second step is **Validation.** Semantic errors are checked. It is verified as to whether the tables referred in the query are present, whether column names are valid etc. The system catalog is refereed for this purpose

c) The next step is optimization. RDBMS explores possibility of using an index, the order of execution of join etc

d) The next step is to generate the 'Application plan'. It is binary representation of the sequence of steps to be executed by the DBMS in executing the query

e) The 'Application plan' is executed.

Why Embedded SQL?

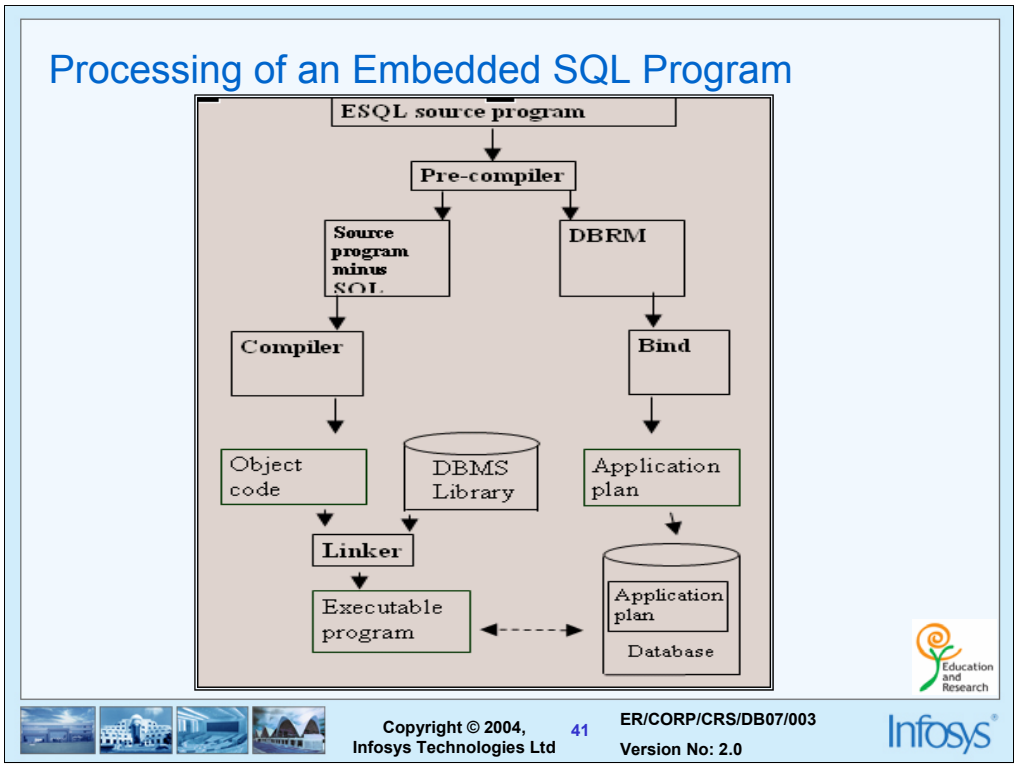
- **SQL has the following limitations:**
  - No provision for declaring variables
  - No unconditional branching/jump statement
  - No **IF** statement for testing conditions
  - No **FOR, DO** or **WHILE** statements to construct loops
  - No block structure

What is Embedded SQL

- Interleaving SQL statements along with the other code in a host language program
- Variables declared in the host program can be used in sql to send /receive values from/to the DBMS

Special variables are used to pass and receive “NULL”

New SQL statements not found in interactive SQL are added to enable processing of query results.



1. The Embedded SQL program is submitted to a precompiler. This is a language specific tool. Ir we have separate precompiler for c, for

   Cobol etc.

2. The precompiler generates two files as output. One is the source file in which all SQL commands are removed and replaced with calls to some special DBMS routines. These routines provide the necessary run-time link with the DBMS. This is aut0-generated by the precompiler

The other file is the DBRM (Data Base Request Module). This contains all the SQL commands removed from the program

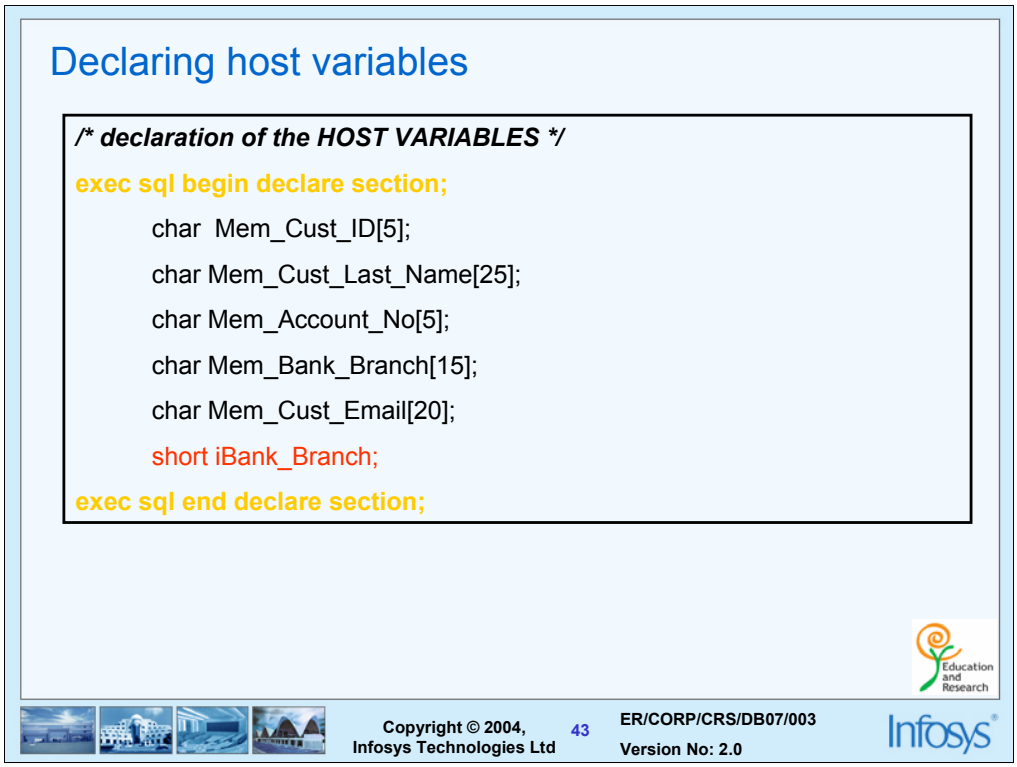
3. Now, this source file is compiled by the compiler to generate the object file.
4. The linker links the object code with other library routines (which includes the DBMS library) and generates the executable
5. parallely, the DBRM is submitted to a “Bind” program. This program parses each SQL statement, validates, optimizes and generates the binary DBMS-executable form called the application plan. After all statements are thus converted , the combined application plan is stored under the DBMS with the same name as the application name from where the SQL statements were stripped off

6. The run-time communication between the executable form of source program and the Application plan is totally hidden .

Using embedded SQL

- **EXEC SQL**
- Keyword indicates to the pre compiler that the next line of code is for the SQL handling.
- The block of sql code is ended with a **semi-colon (;)**

This depends on the host language . EXEC SQL is a common way to begin. But the way to end is language specific.



An example is as shown below:

```
exec sql begin declare section;
                int             emp_no;
                varchar         emp_name[20];
                varchar  emp_addr[20];
                char            emp_des[3];
                char            emp_dept[3];
                int             emp_sal;
                short   emp_addri;
                short   emp_desi;
                short   emp_depti;
                short   emp_sali;
                varchar username[20];
                varchar password[20];
exec sql end declare section;
```

The data type of the variable which is going to be used in a query should be compatible with the data type of the column of the table for which it is going to be substituted/ compared with

For example in the query shown in the slide, we are passing values for supplier number , name and city through the variables declared in the c program named: Sup_No, Sup_Name etc . If the data type of the column SNO in the Supplier table is INTEGER then the data type of the variable Sup_No should be long …

The variables declared in the host language are preceded with a : when they are used in a query.

Example of embedded SQL

```
/*   execute the SQL query */
/* HOST VARIABLES are preceded by a colon (:) e.g.   :Mem_Cust_ID */
/* HOST Variable followed by a companion host indicator variable e.g.
:Mem_Bank_Branch  :iBank_Branch */
EXEC SQL
SELECT Cust_ID, Cust_Last_Name, Account_No, Bank_Branch, Cust_Email
    FROM Customer_Details
        WHERE Cust_ID = :Mem_Cust_ID
                INTO  :Mem_Cust_ID,  :Mem_Cust_Last_Name,
                :Mem_Account_No, :Mem_Bank_Branch :iBank_Branch ,
                :Mem_Cust_Email ;
```

The below code snippet shows how a connection to a DBMS is established by supplying the userid and password.

```
main()
{
        exec sql whenever sqlerror goto sqlerrorlabel;
        strcpy(username.arr,"pramodv");
        username.len = strlen(username.arr);
        strcpy(password.arr,"pramodv");
        password.len=strlen(password.arr);
        exec sql connect :username identified by :password;
```

After this step, the remaining sql statements are written.

The answer to the question in the slide lies in the next few slides..

Indicator variables

- Variable to indicate whether the returned value is NULL.

```
/* checking the value of the INDICATOR VARIABLE */
IF (iBank_Branch  < 0)
        printf("Bank Branch is NULL\n");
ELSE
        printf("Bank Branch  %s\n", Mem_Bank_Branch);
```

Indicator Value

0	**-**	**Host Variable Contains a Valid Value**
- ve Value	**-**	**Host Variable assumed to have a NULL Value**
+ ve Value	**-**	**host variable contains a valid value, which may have been rounded off or truncated**

The indicator variables are used to indicate the status of the value received into the host variable.

Zero Host variable has been assigned the retrieved value. Host variable can be used in the application program.

Negative Retrieved value is NULL. Host variable value not reliable.

Positive Retrieved value is in host variable, but this indicates a warning, rounding off or truncation for example.

The following highlights the declaration of indicator variables

```
exec sql begin declare section;
        ……
        short   emp_addri;
        short   emp_desi;
        short   emp_depti;
        short   emp_sali;
        varchar username[20];
        varchar password[20];
exec sql end declare section;
```

Indicator variables are required only for NULLABLE columns. They are also used with a prefixed colon

Summary

- Views create a window into the table thereby allowing restricted access
- Index helps speed up the search process
- Embedded SQL programs are written in some HLL in which SQL statements are intermixed
- A special processing separates the SQL , converts and generates executable code

Thank You!