Tutorial Sheet – Week 2
Entity-Relationship (ER) Diagrams

You might find the online module, lecture notes and the file ER Conventions for KIT102-Handout, to be helpful.

Solution Notes

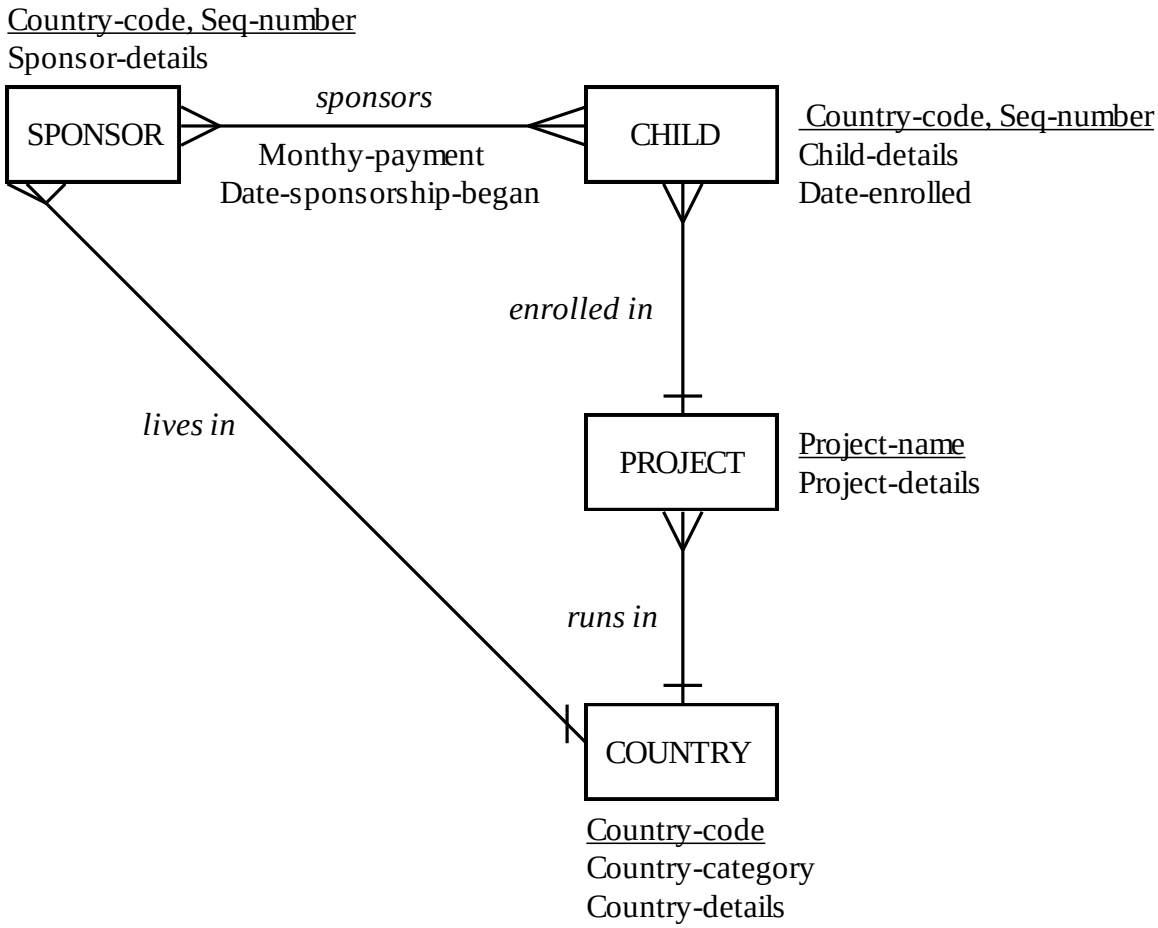
Modelling Scenarios

Note:

Each of the ER diagrams below has attributes and a title.

Your ER diagram in the Final Exam should have attributes and a title.

Help the Children



Notes:

- **We are modelling Help the Children,**
  so Help the Children is the title of our ER diagram.

  HELP THE CHIDREN should *not* be an entity in the ER diagram.

- **Having an attribute Country-name would be a Good Idea.**
  Country names are unique, so Country-name would be an alternate key for COUNTRY.
  There is no convention for showing alternate keys on our ER diagrams – we just have to make a note of the fact that Country-name is an alternate key.

- **We do have a convention for showing composite identifiers, so it is *not* a Good Idea to ignore the information in the scenario:**
  "Each sponsor and each child is given an identifying code which is made up of two parts."
  When we underline two attributes in an ER diagram, we are saying that both attributes are needed to form a unique identifier.

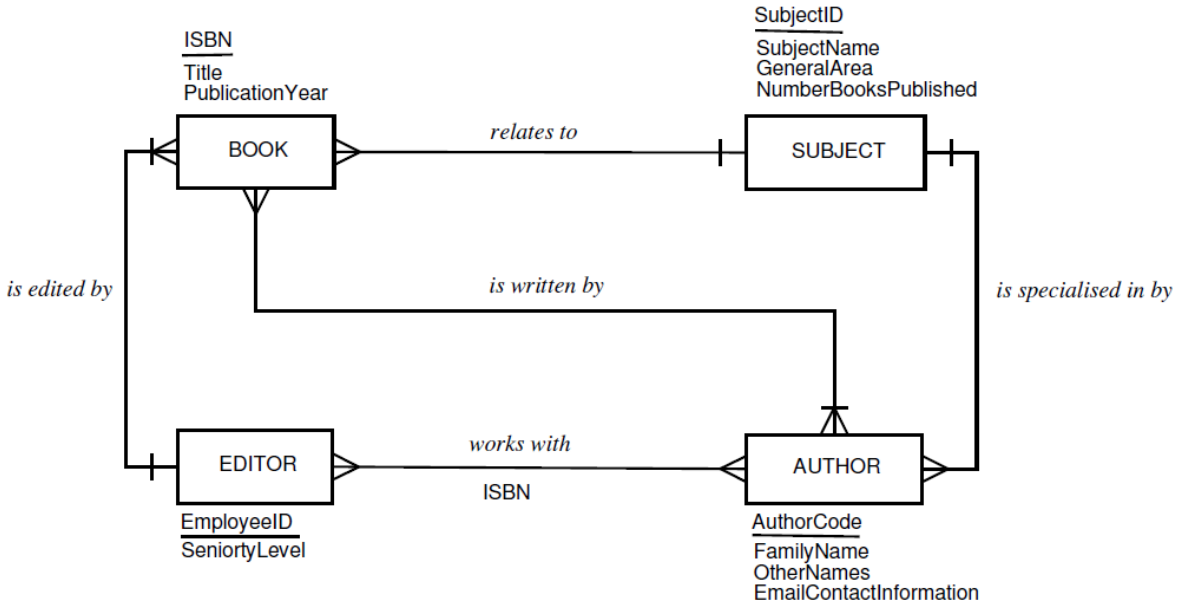
- **We are told that project names are unique, so we use Project-name as the identifier of PROJECT on the conceptual ER Diagram (even though we are almost certain that, at a later step in a database design process, we would allocate a unique code to each project).**

- **We could put in a *lives in* relationship between CHILD and COUNTRY – but this is not necessary because we can derive the relationship: each child is enrolled in one project, and each project runs in one country.**

- **It would be okay to leave out the *lives in* relationship between SPONSOR and COUNTRY because we can derive it from the identifier of SPONSOR.**

- **The most common *mistake* (which was not a mistake) which students identify is that the relationship *sponsors* should be one-to-many, instead of many-to-many, because of the sentence "Each child may have only one sponsor at any one time." in the Scenario. Although a Child cannot have more than one Sponsor at any one time, a child may have several Sponsors, one after the other. *For example,* child HTI0023 may have sponsor AUS0042, then sponsor USA0023, then Sponsor USA0042.**

Newton's Outstanding Publications for Education



ER to Relational Model conversion

Help the Children

Step 1: Each entity becomes a relation

SPONSOR (Country-code, Seq-number, [pk] Sponsor-details)

CHILD (Country-code, Seq-number, [pk] Child-details, Date-enrolled)

PROJECT (Project-name, [pk] Project-details)

COUNTRY (Country-code, [pk] Country-category, Country-details)

Step 2: Each many-to-many relationship becomes a relation

SPONSORS (Sponsor-Country-code [fk1], Sponsor-Seq-number [fk1], Child-Country-code [fk2], Child-Seq-number [fk2], [pk] Monthly-payment, Date-sponsorship-began)

Step 3: Each one-to-many relationship is represented by a foreign key

SPONSOR (Country-code [fk], Seq-number, [pk] Sponsor-details)

CHILD (Country-code[fk], Seq-number, [pk] Child-details, Date-enrolled, Project-name [fk])

PROJECT (Project-name, [pk] Project-details, Country-code [fk])

Step 4: Write out the Final Relational Schema

Help the Children

CHILD (Country-code, Seq-number, [pk] Child-details, Date-enrolled, Project-name [fk])

COUNTRY (Country-code, [pk] Country-category, Country-details)

PROJECT (Project-name, [pk] Project-details, Country-code [fk])

SPONSOR (Country-code [fk], Seq-number, [pk] Sponsor-details)

SPONSORS (Sponsor-Country-code [fk1], Sponsor-Seq-number [fk1], Child-Country-code [fk2], Child-Seq-number [fk2], [pk] Monthly-payment, Date-sponsorship-began)

Solution Notes for Tutorial Week 3
KIT712

Part 1: Relational Algebra

Formulate a Relational Algebra Query on the Location database for the following:

1. Find the name of the person who live in the city "XYZ"

   Π person_name (σ city = xyz (lives))

2. List the person name and the city they work in.

   Π person_name, city (works ⋈company_name **located_in))**

3. Find the name for all persons that **do not** work for the company "Acme"

   Π person_name (σ company_name ≠ ACME (works))

4. Find the name for all persons who live in the same city in which their company is located

   Π person_name (σ lives.city=located_in.city **(lives** ⋈person_name **works**

   ⋈company_name **located_in))**

5. Find the name of companies where **all** its (every single) employees live in the same city where the company is located in (i.e., companies that do not have any commuting employee).

   H1 = Π company_name **(σ** lives.city ≠ located_in.city **(lives** ⋈person_name **works**

   ⋈company_name **located_in))**

   Result = **Π** company_name **(located_in) – H1**

KIT712 Week 4 Tutorial Solution

SQL QUERIES

You need to write SQL queries to get the required information.

Single Table Queries

1. Display the last name, job code, hire date, and employee number for each employee, with the employee number appearing first

```
SELECT employee_id, last_name, job_id, hire_date
StartDate FROM employees;
```

2. Display all unique job codes from the EMPLOYEES table.

```
SELECT DISTINCT job_id FROM employees;
```

3. Display the last name and salary of employees earning more than $12,000.

```
SELECT last_name, salary FROM employees WHERE salary >
12000;
```

4. Display the last name and salary of employees who earn between $5000 and $12,000 and are in department 20 or 50. Label the columns Employee and Monthly Salary, respectively.

```
Select last_name as "Employee", salary as "MonthlySalary"
From employees where (salary BETWEEN 5000 AND 12000) AND
(department_id=20 or department_id=50);
```

5. List the last name and department ID of all employees in departments 20, 30 or 50 in ascending alphabetical order by name.

```
Select last_name, department_id
From employees
Where department_id IN (20,30,50) order by last_name ASC;
```

6. Display the last name and department number of all employees in departments 20 or 50 in ascending alphabetical order by name

```
SELECT last_name, department_id FROM employees WHERE
department_id IN (20, 50) ORDER BY last_name ASC
```

7. List the last name and hire date for all employees who are hired in 2005.

```
Select last_name, hire_date
From employees
Where hire_date like '%05';
```

```
SELECT LAST_NAME, HIRE_DATE FROM HR.EMPLOYEES
WHERE HIRE_DATE<'1-JAN-2006' AND HIRE_DATE >'31-DEC-2004';

SELECT LAST_NAME, HIRE_DATE FROM HR.EMPLOYEES
WHERE EXTRACT(YEAR FROM HIRE_DATE) =2005;
```

8. Create a query that displays the last name and salary of employees who earn more than an amount the user specifies after a prompt.

```
Select last_name, salary
From employees
Where salary>&sal_amt;
```

9. Create a query that prompt the user for a manager ID and generates the employee ID, last name, salary and department for that manager's employees.

```
Select employee_id, last_name, salary, department_id
From employees
Where manager_id=&mgr_num;
```

10. Display minimum and maximum salary given in the company.

```
Select max(salary), min(salary) from employees;
```

11. For each employee, list the last name and calculate the number months between today and the date on which the employee was hired. Label the column months_worked. Round the number of months up to the closes whole number.

```
Select last_name, ROUND (MONTHS_BETWEEN(SYSDATE,
hire_date)) MONTHS_WORKED
From employees
Order by MONTHS_WORKED;
```

Multiple Table Queries (joins)

12. Write a query that list addresses of all the departments. Hint: For this you have to use Locations and countries tables. Show the location ID, street address, city, state and country in the output. Use Natural join to produce the results.

```
SELECT location_id, street_address, city, state_province,
country_name from locations natural join countries;
```

13. Write a query to display the last name, department number and department name for all the employees.

```
Select last_name, department_id, department_name
From employees Join departments using (department_id);
```

14. Display employees's last names and employee number along with their managers's last names and manager number. Label the columns Employee, Emp#, Manager and Mgr#, respectively.

```
Select w.last_name "Employee", w.employee_id
"EMP#",m.last_name "Manager", m.employee_id "Mgr#"
From employees w join employees m
```

```
ON (w.manager_id=m.employee_id);
```

15. Display employee name and id with their manager id including those employees who has no manager.

```
SELECT w.last_name "Employee", w.employee_id "EMP#",
m.last_name "Manager", m.employee_id "Mgr#" FROM
employees w LEFT OUTER JOIN employees m ON (w.manager_id
= m.employee_id)
```

16. Find those department names which have no employees allocated in them.

```
Select department_name from employees m right outer join
departments d on m.department_id= d.department_id where
m.employee_id is null;
```

Group By

17. Display the minimum, maximum, sum, and average salary for each job type.

```
SELECT job_id, ROUND(MAX(salary),0) "Maximum",
ROUND(MIN(salary),0) "Minimum", ROUND(SUM(salary),0)
"Sum", ROUND(AVG(salary),0) "Average" FROM employees
GROUP BY job_id;
```

18. Write a query to display the number of people with the same job id.

```
Select job_id, COUNT(*) from employees GROUP BY job_id;
Generalise about query so that the user in HR department
is prompted for a job title.
Select job_id, COUNT(*) from employees where
job_id='&job_title' GROUP BY job_id;
```

Subqueries

19. List last name and salary of every employee who reports to manager with last name King. Manager King has no manager.

Select last_name, salary from employees
Where manager_id=(Select employee_id from employees where last_name='King' and manager_id is null}

20. Display the department number, last name, and job_ID for every employee in the Executive department.

Select department_id, last_name, job_id from employees
Where department_id IN (select department_id from departments where department_name='Executive');

KIT712 Tutorial 4: Advance SQL Extra Questions

SQL QUERIES

Multiple Table Queries (joins)

1. Write a query in SQL to display those employees who contain a letter z to their first name and also display their last name, department, city, and state province

```
SELECT E.first_name,E.last_name, D.department_name, L.city,
L.state_province FROM employees E JOIN departments D  ON
E.department_id = D.department_id JOIN locations L ON
D.location_id = L.location_id WHERE E.first_name LIKE
'%z%';
```

2. Write a query in SQL to display the first and last name and salary for those employees who earn less than the employee earn whose number is 182.

```
SELECT E.first_name, E.last_name, E.salary FROM employees E,
employees S where E.salary < S.salary AND S.employee_id =
182;
```

3. Write a query in SQL to display the first name, last name, department number and name, for all employees who have or have not any department.

```
SELECT E.first_name, E.last_name, E.department_id,
D.department_name
  FROM employees E
   LEFT OUTER JOIN departments D
     ON E.department_id = D.department_id;
```

Subqueries

4. Write a query in SQL to display the first and last name and salary for those employees who earn less than the employee earn whose number is 182.

```
SELECT E.first_name, E.last_name, E.salary FROM employees E
where E.salary < (select salary from employees S where
S.employee_id = 182);
```

Tutorial Week 6: SQL Tuning

Objective is to understand how SQL tuning done in Oracle. To start first finding how your SQL statement is executed and how much time it is taken for execution. You have to use the "**SH**" schema or user.

Now from the same terminal you started Oracle, typing **SQLPLUS** to start.

In your current session, type following:

set echo on

set timing on

set autotrace on

After that before you do each part of this tutorial, flush the cached data using:

alter system flush shared_pool;

alter system flush buffer_cache;

To create an index: http://www.techonthenet.com/oracle/indexes.php

Tutorial on how to use ***hints*** in oracle:
http://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements006.htm#SQLRF50702

Q1. Type following select statements and compare their execution plans and time. What is difference you find?

a) `select cust_first_name, cust_last_name from customers where cust_id = 1030;`
b) `select cust_first_name, cust_last_name from customers where cust_id <> 1030;`
c) `select cust_first_name, cust_last_name from customers where cust_id < 20000;`
d) `select cust_first_name, cust_last_name from customers where cust_id between 70000 and 80000;`

The results of these queries show that the Oracle optimizer CAN use an index for EQUALITY search (=) and CANNOT use the NOT EQUAL (<>) search.

Moreover, the optimizer sometimes has a problem with the index.

For example, the database may be unable to use an index when handling a range predicate: unbounded range (<=, <, >, and >=). It means that the database will perform a full table scan, ignoring your index.

Q2. Create an index on the CUST_CREDIT_LIMIT column of the CUSTOMERS table. After that, compare and discuss the execution plan and timing of following queries

```
a) select cust_id from customers where cust_credit_limit*1.10=11000;
b) select cust_id from customers where cust_credit_limit=11000/1.10;
```

Execute following SQL statement to create an index on the CUST_CREDIT_LIMIT column of the CUSTOMERS table:

```
Create index CUST_CREDIT_INDX on CUSTOMERS (CUST_CREDIT_LIMIT);
```

The results of these queries show that although the CUST_CREDIT_LIMIT column is indexed, this index cannot be used appropriately in the statement (a) because of an expression (***1.10**) on the CUST_CREDIT_LIMIT column.

To improve performance, the indexed column should appear clean in the WHERE clause (avoid using function or expression).

By rewriting the query, the index is used in the statement (b) because of the CUST_CREDIT_LIMIT column is clean.

Q3. Create another index on the CUST_LAST_NAME column of the CUSTOMERS table. After that, compare the execution plan and timing of following queries

```
a) select cust_id from customers where substr(cust_last_name,1,1) = 'S';
b) select cust_id from customers where cust_last_name like 'S%';
c) select cust_last_name from customers where cust_last_name like 'S%';
```

Execute following SQL statement to create an index on the CUST_LAST_NAME column of the CUSTOMERS table:

```
Create index CUST_LNAME_INDX on CUSTOMERS (CUST_LAST_NAME);
```

(a) Although the CUST_LAST_NAME column is indexed, this index cannot be used in an appropriate manner in the statement (a) because of the function (**SUBSTR**) on the CUST_LAST_NAME column. Optimiser will use 'Fast Full Index Scan' which is equivalent to a 'Full Table Scan'. By rewriting the query (a) as (b) can allow index to be used.

Queries (a) and (b) are logically equivalent, but in (c) selected column is same as search column. Therefore, you will notice change in execution plan.

In summary, you need to understand that the indexed column should be clean. As soon as you apply any function, index usage is impossible.

Q4. Check the execution plan and reason why index is not used.

```
a) select cust_id from customers where cust_last_name like '%S%'
```

The index is unusable, because the search pattern starts with a wildcard. This makes sense. The index is useless if you do not specify a leading part for the search.

b) `select cust_last_name from customers where cust_id like '7%'`

This is because the CUST_ID column is a numeric column, so the optimizer must apply an implicit data type conversion and rewrites the WHERE clause to read. This will lead to full scan of index. In other words, index will become unusable. To solve the problem in this query, you need to create a new index for TO_CHAR(cust_id) and rewrite the query like following:

```
select cust_last_name from customers where TO_CHAR(cust_id) like '7%'
```

Q5. In 'ORDER BY' clause, in general there is sorting done on all the rows. Try to run following select statement and again check the execution plan:

```
select cust_first_name, cust_last_name, cust_credit_limit from customers
order by cust_credit_limit;
```

Now create an appropriate index on a column and again run above select statement. Do you find any change in performance and execution plan?

You create an index on the CUST_CREDIT_LIMIT column using the following statement.

```
Create index CUST_CREDIT_INDX on CUSTOMERS (CUST_CREDIT_LIMIT);
```

Create an index on `CUST_ID` column of `CUSTOMERS` table **if it does not.** Now execute following SQL statement and compare the execution plan of this query with above.

```
select cust_first_name, cust_last_name, cust_credit_limit from customers
order by cust_id;
```

In general index is used when you are doing 'order by' on an indexed column. You will notice that the index is used in later case but not in previous case. The difference between both queries is that the CUST_ID column is mandatory (NOT NULL) and the CUST_CREDIT_LIMIT column is not. The index on CUST_CREDIT_LIMIT cannot be guaranteed to contain an entry for each customer, because NULL-values are not stored in regular indexes. Thus, only way to find all rows containing NULL values is to perform a full-table scan.

Q6. Compare the execution plan for following statements and explain the difference:

```
select max(cust_credit_limit) from customers;
select max(cust_credit_limit*2) from customers;
select max(cust_credit_limit+1000) from customers;
```

This shows that an index can be useful to retrieve a maximum value (and a minimum value). If no index is available, the optimizer must scan the full table and perform a sort. However, in some databases, for second query index will not be used but index is used in last one. The reason is that optimizer can find equivalent query for last query so that index can be used but optimizer cannot find for second query.

```
max(cust_credit_limit + :x) is always equivalent to
max(cust_credit_limit ) + :x
```

*max(cust_credit_limit * :x) is not always equivalent to*
*max(cust_credit_limit) * :x*

Q7. Compare the execution plan and time of following two statements.

```
select count(*) from products p where prod_list_price<1.15 * (select
avg(unit_cost) from costs c where c.prod_id=p.prod_id);
```

```
select count(*) from products p, (select prod_id, avg(unit_cost) ac from
costs group by prod_id) c where p.prod_id=c.prod_id and p.prod_list_price
<1.15*c.ac;
```

Second query performs better as in the first query as subquery runs for every row found in the query. You may notice, your optimizer may rewrite the query to have join rather than subquery.

Q8. Analyse following SQL statement and identify the best join operation (and join sequence, when relevant), using the hints (USE_MERGE, USE_NL, USE_HASH) to instruct the optimizer.

```
select c.cust_last_name, c.cust_year_of_birth, co.country_name from
customers c, countries co where c.country_id = co.country_id and
co.country_region = 'Americas';
```

You have to use hint like following
select /*+ USE_MERGE(c co) */ c.cust_last_name, c.cust_year_of_birth, co.country_name from customers c, countries co where c.country_id = co.country_id and co.country_region = 'Americas';
For other hints you have to do in similar way. While comparing execution plan, compare their 'cost' to see which one is better than other.

Q9. Analyse the SQL statement

```
select c.cust_last_name, s.time_id, s.prod_id from customers c, sales s
where c.cust_id <> s.cust_id and s.prod_id = 2595 and s.time_id = '01-JAN-
98';
```

Which join operations can be used to execute this join? Experiment with different join orders by using an ORDERED hint, then try a LEADING hint, and find the best choice. There is a significant difference in performance? Do the same with different join operations.

Nested loop is only possibility here due to non-equality. Here you may find, for different types of join hints, oracle may still use most appropriate one which may be different from the one specified in join hint. However, you may find difference when using ORDERED and LEADING. Having CUSTOMERS as outer table will results in more cost, so in this case, there will be very high chance for SALES to be an outer table for join.

Q10. Consider following SQL statement

```
select c1.cust_first_name, c1.cust_last_name, c1.cust_year_of_birth from
customers c1 where c1.cust_year_of_birth = (select
max(c2.cust_year_of_birth) from customers c2 where c1.country_id =
c2.country_id);
```

This statement retrieves the customers with the oldest birth year in every country. This type of statement can be found in many real-life situations. What is happening in the execution plan? Are the results satisfactory?

Create an index on the COUNTRY_ID column, and measure the performance improvement. Is creating the index a better choice? Can you rewrite the query to use join operation? Is there any performance improvement.
***Hint: Move the correlated subquery to the `FROM` clause*.**

You create an index on the COUNTRY_ID column using the following statement.

Create index CUST_COUNTRY_INDX on CUSTOMERS (COUNTRY_ID);

Query can be rewritten like following:

select c1.cust_first_name, c1.cust_last_name, c1.cust_year_of_birth from customers c1, (select max(c2.cust_year_of_birth)as maxyear, c2.country_id as country from customers c2 group by c2.country_id) temp where c1.cust_year_of_birth=temp.maxyear and c1.country_id=temp.country;

In following questions, try to understand different possibilities you need to consider before deciding for which column index needs to be created. When bitmap index to be used? When combined index to be used?

Q11. Create indexes on the following columns: cust_gender, cust_postal_code, cust_credit_limit

Execute following SQL statements to create three indexes:

```
Create Index CUST_GENDER_INDX On Customers(cust_gender);

Create Index CUST_POST_INDX On Customers(cust_postal_code);

Create Index CUST_CREDIT_INDX On Customers(cust_credit_limit);
```

The following statement contains WHERE clause with three predicates. Execute this statement, and take notes about the indexes used, the cost of the execution plan, and the amount of I/O performed.

```
select c.* from customers c where cust_gender = 'M' and cust_postal_code =
40804 and cust_credit_limit = 10000;
```

```
Elapsed: 00:00:00.16

Execution Plan
----------------------------------------------------------
Plan hash value: 2008213504

--------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |     6 |  1086 |   405   (1)| 00:00:05 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |     6 |  1086 |   405   (1)| 00:00:05 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(TO_NUMBER("CUST_POSTAL_CODE")=40804 AND
              "CUST_CREDIT_LIMIT"=10000 AND "CUST_GENDER"='M')
```

Run above statement again with different indexes by giving HINT as follow and take notes about the performance results.

```
select /*+ INDEX (c CUST_CREDIT_INDX) */ c.* from customers c where
cust_gender = 'M' and cust_postal_code = 40804 and cust_credit_limit =
10000;
```

Elapsed: 00:00:00.18
Execution Plan

Plan hash value: 3171069297

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		6	1086	452 (1)	00:00:06
* 1	TABLE ACCESS BY INDEX ROWID	CUSTOMERS	6	1086	452 (1)	00:00:06
2	BITMAP CONVERSION TO ROWIDS					
3	BITMAP AND					
* 4	BITMAP INDEX SINGLE VALUE	CUST_CREDIT_INDX				
5	BITMAP CONVERSION FROM ROWIDS					

```
|*  6 |     INDEX RANGE SCAN                | CUST_GENDER_INDX | |  | 51   (0)            | 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(TO_NUMBER("CUST_POSTAL_CODE")=40804)
   4 - access("CUST_CREDIT_LIMIT"=10000)
   6 - access("CUST_GENDER"='M')
```

Run above statement again with different indexes by giving HINT as follow and take notes about the performance results.

```
select /*+ INDEX (c CUST_GENDER_INDX CUST_POST_INDX CUST_CREDIT_INDX) */
c.* from customers c where cust_gender = 'M' and cust_postal_code = 40804
and cust_credit_limit = 10000;

Elapsed: 00:00:00.09
Execution Plan
----------------------------------------------------------
Plan hash value: 743140819
-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name               | Rows | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                    |   6  |1086   | 217   (0)  | 00:00:03 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| CUSTOMERS            |   6  |1086   | 217   (0)  | 00:00:03 |
|*  2 |   INDEX FULL SCAN                   | CUST_POST_INDX|  89  |       | 133   (0)  | 00:00:02 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("CUST_CREDIT_LIMIT"=10000 AND "CUST_GENDER"='M')
   2 - filter(TO_NUMBER("CUST_POSTAL_CODE")=40804)
```

Drop above indexes and create a new combined index and run above statement without any hints again. Compare the performance results with previous ones. Which is best?

Execute following SQL statement to create a new combined index:

```
Create Index C_INDX On Customers(cust_gender, cust_postal_code,
cust_credit_limit);

Elapsed: 00:00:00.09
Execution Plan
----------------------------------------------------------
Plan hash value: 1353691094
-----------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |    7 |  1267 |   93   (0) | 00:00:02 |
|   1 |  TABLE ACCESS BY INDEX ROWID| CUSTOMERS |    7 |  1267 |   93   (0) | 00:00:02 |
|*  2 |   INDEX RANGE SCAN                  | C_INDX        |    6 |       |   87   (0) | 00:00:02 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
```

2 - access("CUST_GENDER"='M' AND "CUST_CREDIT_LIMIT"=10000)
filter(TO_NUMBER("CUST_POSTAL_CODE")=40804 AND "CUST_CREDIT_LIMIT"=10000)

Q12. Oracle provides several index types.

Hint: to create an index.

http://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_creating.htm

http://www.orafaq.com/wiki/Bitmap_index

In this exercise will try to see how they differ:

Execute following SQL statement to see all existing indexes:

```
SELECT index_name FROM user_indexes;
```

a) Drop previously created indexes in this tutorial. Create two bitmap indexes on the following columns of the `CUSTOMERS` table:
`cust_year_of_birth`
`cust_credit_limit`

Execute following SQL statements to drop indexes:

```
Drop Index CUST_YOB_INDX;
Drop Index CUST_CREDIT_INDX;
```

Execute following SQL statements to create two bitmap indexes:

```
Create Bitmap Index CUST_YOB_INDX On Customers(cust_year_of_birth);

Create Bitmap Index CUST_CREDIT_INDX On Customers(cust_credit_limit);
```

Execute following SQL statements and compare their execution plan and performance:

```
SELECT /*+ INDEX_COMBINE(c) */ c.* FROM customers c
WHERE c.cust_year_of_birth = 1953 OR c.cust_credit_limit = 10000;
```

The indexes are used because the INDEX hint is applied to force the optimizer to use it.

```
SELECT c.* FROM customers c WHERE c.cust_year_of_birth = 1953 OR
c.cust_credit_limit = 10000;
```

The indexes are not used.

b) Drop all the indexes on the `CUSTOMERS` table except its primary key index.
After this, create a concatenated B-tree index on the following columns of the `CUSTOMERS` table, and in the order here:
`cust_last_name`
`cust_first_name`

Check all existing indexes of the CUSTOMERS table and drop all except its primary key index.

```
Drop Index column_names;
```

Compare results of following query before and after creating the index:

```
SELECT c.cust_last_name, c.cust_first_name FROM customers c;
```

A composite index, also called a concatenated index, is an index on multiple columns in a table. Columns in a composite index should appear in the order that makes the most sense for the queries that will retrieve data and need not be adjacent in the table.

Execute following SQL statement to create a concatenated B-tree index:

```
Create Index CUST_NAME_INDX On Customers(cust_last_name,
cust_first_name);
```

The results of the above query show that an index fast full is used when the concatenated B-tree index created.

Delete the concatenated B-tree index and create two new indexes:

```
Create Index CUST_LAST_NAME_INDX On Customers(cust_last_name);

Create Index CUST_FIRST_NAME_INDX On Customers(cust_first_name);
```

The results of the above query show that two above indexes are not used.

Now run following statement and again compare the results:

```
SELECT /*+ INDEX_JOIN(c CUST_LAST_NAME_INDX CUST_FIRST_NAME_INDX) */
c.cust_last_name, c.cust_first_name FROM customers c;
```

The indexes are used because the INDEX_JOIN hint is applied to force the optimizer to combine and use it.

c) (i) Drop all the previously created indexes on the `CUSTOMERS` table except its primary key index. Create one B-tree index on the following column of the `CUSTOMERS` table: `cust_credit_limit`

```
Drop Index column_names;
Create Index CUST_CREDIT_INDX On Customers(cust_credit_limit);
```

Execute the following query, note down the plan and performance statistics.

```
SELECT count(*), cust_credit_limit FROM customers WHERE
cust_credit_limit = 10000 GROUP BY cust_credit_limit;
```

```
Elapsed: 00:00:00.01
Execution Plan
----------------------------------------------------------
Plan hash value: 636859273
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                     |    1 |     4 |    14   (0)| 00:00:01 |
|   1 |  SORT GROUP BY NOSORT        |                     |    1 |     4 |    14   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | CUST_CREDIT_INDX    | 6938 | 27752 |    14   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("CUST_CREDIT_LIMIT"=10000)
```

(ii) Drop all the indexes on the `CUSTOMERS` table except its primary key index. Then, create one bitmap index on the following column of the `CUSTOMERS` table: `cust_credit_limit`

```
Drop Index column_names;
Create Bitmap Index CUST_CREDIT_INDX On Customers(cust_credit_limit);
```

Execute the following query, note down the plan and performance statistics.

```
SELECT count(*), cust_credit_limit FROM customers WHERE
cust_credit_limit = 10000 GROUP BY cust_credit_limit;
```

```
Elapsed: 00:00:00.04
Execution Plan
----------------------------------------------------------
Plan hash value: 2388706395
-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |     1 |     4 |     1 (0)  | 00:00:01 |
|   1 |  SORT GROUP BY NOSORT        |                   |     1 |     4 |     1 (0)  | 00:00:01 |
|   2 |   BITMAP CONVERSION COUNT    |                   |  6938 | 27752 |     1 (0)  | 00:00:01 |
|*  3 |    BITMAP INDEX SINGLE VALUE | CUST_CREDIT_INDX  |       |       |            |          |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("CUST_CREDIT_LIMIT"=10000)
```

(iii) Compare the results of (i) and (ii). Which index is better and why?

A B-Tree index is used in columns with high data sparsity (many different values relative to the total number of rows), while a bitmap index is used in columns with low data sparsity.

Basic PL/SQL

NOTE: We are going to use **HR database**. You need to start SQL Developer for this.

PL/SQL Basics

Today we are going to do some simple exercises involving PL/SQL. It is a programming language for Oracle Database that allows you to do much more complex things than SQL itself.
A PL/SQL blocks looks like following.
[<Block header>]
[**declare**
<Constants>
<Variables>
<Cursors>
<User defined exceptions>]
begin
<PL/SQL statements>
[**exception**
<Exception handling>]
end;
The block header specifies whether the PL/SQL block is a procedure, a function, or a package. If no header is specified, the block is said to be an anonymous PL/SQL block. The scope of declared variables (i.e., the part of the program in which one can refer to the variable) is analogous to the scope of variables in programming languages such as C or Pascal.

To do this tutorial, your online module will be helpful.

We are going to use HR database.

Exercise 1: Syntax Checks
Which of the following Pl/SQL blocks execute successfully?Why?

a) BEGIN
END;
b) DECLARE
V_amountINTEGER(10);
END;
c) DECLARE
BEGIN
END;
d) DECLARE
V_AMOUNT INTEGER(10);
BEGIN
DBMS_OUTPUT.PUT_LINE(V_AMOUNT);
END;

Exercise 2: Print
Create and execute a simple anonymous block that outputs “Hello World”.

```
SET SERVEROUTPUT ON
BEGIN
DBMS_OUTPUT.PUT_LINE(‘Hello World’);
END;
```

What it outputs? Write another PL/SQL block which prints “Hello *YOURNAME*”.

Exercise 3: Variable Declaration

What is the output of following:

```
Set Serveroutput on
Declare
Fname VARCHAR (20);
LnameVARCHAR(15)  DEFAULT  ‘fernandez’;
BEGIN
DBMS_OUTPUT.PUT_LINE(FNAME|| ‘ ‘ || lname);
End;

DECLARE
today DATE:=SYSDATE;
tomorrowtoday%TYPE;
BEGIN
Tomorrow:=today+1;
DBMS_OUTPUT.PUT_LINE(‘Hello World’);
DBMS_OUTPUT.PUT_LINE(‘today is:’ || today);
DBMS_OUTPUT.PUT_LINE(‘tomorrow is’ || tomorrow);
End ;
```

Exercise 4: Variable Names
Identify valid and invalid name of variables(aka. Identifier) in following:

a. Today
b. Last_name
c. Today’s_date
d. Number_of_days
e. #number
f. Number#
g. Number1to7

Exercise 5: Variable Declarations
Identify valid and invalid variable declaration and initialization

```
a. Number_compies PLS_INTEGER;
b. PRINTER_NAME constant VARCHAR2(10);
c. Deliver_to VARCHAR2(10):=Johnson;
d. By_when DATE:CURRENT_DATE+1;
```

Exercise 6: Input at Runtime

Try to run following:

```
DECLARE
my_var VARCHAR2(30);
BEGIN
my_var := '&input';
dbms_output.put_line('Hello'|| my_var );
END;
/
```

What do you observe?What is reason for this output?
Modify above code to take input at the run time your name and your student number, and print out "Hello, *YOURNAME, YOURID"*

```
DECLARE
my_name VARCHAR2(30);
my_id VARCHAR2(30);
BEGIN
my_name := '&input';
my_id := '&input1';
dbms_output.put_line('Hello'|| my_name ||my_id );
END;
/
```

Exercise 7: IF-Then Else Statements
Modify the code given in previous exercise, so that it will only print when input is other than 'No'.

```
DECLARE
my_var VARCHAR2(30);

BEGIN
my_var := '&input';
if(my_var='No') then
dbms_output.put_line('Hello Cannot print the name' );
else
dbms_output.put_line('Hello'|| my_var  );
end if;
END;
/
```

Exercise 8: Reading SQL data
Within PL/SQL block we can also write our regular SQL statement and read the output in a variable. Execute following PL/SQL block

```
DECLARE
F_name varchar(20);
BEGIN
SELECT first_name INTO f_name from employees where
employee_id=100;
End;

/
```

Modify above PL/SQL block to print out the first name of the employee whose id is 100;

```
Set serveroutput on
DECLARE
F_name varchar(20);
BEGIN
SELECT first_name INTO f_name from employees where employee_id=100;
dbms_output.put_line(f_name);
End;
```

Modify above PL/SQL block to print out both first name and last name of the employee whose id is 100;

```
setserveroutput on
DECLARE
F_name varchar(20);
l_name varchar(20);

BEGIN
SELECT first_name, last_name INTO f_name, l_name from employees where
employee_id=100;
dbms_output.put_line(f_name||' '||l_name );
End;
```

HINT: declare two variables for first name and last name. then modify select statement to also retrieve last name of the employee.

Tutorial Week 9-10

PL/SQL Advance

Today we are going to do some simple exercises involving PL/SQL. It is a programming language for Oracle Database that allows you to do much more complex things than SQL itself.

We are going to use HR database.

Exercise 1

Create and execute a simple anonymous block that outputs “Hello World”.

SET SERVEROUTPUT ON

BEGIN

DBMS_OUTPUT.PUT_LINE(‘Hello World’)

END;

Exercise 2

Create a PL/SQL block that selects the maximum department ID in the Department- table and stores it in a variable. Display the maximum department ID. Steps involved in that process:

a) Declare a variable of type NUMBER in the declarative section.
b) Start the executable section with the BEGIN keyword and include a SELECT statement to retrieve the maximum department ID¨
c) Display the variable and end the executable block.

Modify the above PL/SQL block to insert a new department (Name=’EDUCATION’) into the departments table. Use SQL%ROWCOUNT to display the number of rows that are affected. Execute a SELECT statement to check whether new department is inserted or not. Include a DELETE statement to delete the department that you added.

```
SET SERVEROUTPUT ON

DECLARE
v_dept_name departments.department_name%TYPE:= 'Education';
```

```
v_dept_id NUMBER;
v_max_deptno    NUMBER;
BEGIN
   SELECT MAX(department_id)  INTO v_max_deptno  FROM departments;
   DBMS_OUTPUT.PUT_LINE('The maximum department_id is : ' ||
v_max_deptno);
v_dept_id := 10 + v_max_deptno;
   INSERT INTO departments (department_id, department_name,
location_id)
   VALUES (v_dept_id,v_dept_name, NULL);
   DBMS_OUTPUT.PUT_LINE (' SQL%ROWCOUNT gives ' || SQL%ROWCOUNT);
 END;
/
SELECT * FROM  departments  WHERE  department_id=280;
```

Exercise 3 (Loops and IF/ELSE statements)

Create a message table using following command:

```
DROP TABLE messages;
CREATE TABLE messages (results VARCHAR2(80));
```

Write a PL/SQL block to insert numbers in messages table from 1 to 10 excluding 6. Commit before end of the block

```
BEGIN
FOR i in 1..10 LOOP
  IF i = 6 THEN
null;
  ELSE
    INSERT INTO messages(results)
    VALUES (i);
  END IF;
END LOOP;
COMMIT;
END;
/
SELECT * FROM messages;
```

Exercise 4 (Loops and IF/ELSE statements)

Execute following commands to create an emp table.

```
DROP TABLE emp;

CREATE TABLE emp AS SELECT * FROM employees;
```

```
ALTER TABLE    emp  ADD stars    VARCHAR2(50);
```

Create a PL/SQL block that inserts an asterisk in the stars column for every column for every $1000 of an employee's salary. You can use default value for employee number as 176.

```
SET VERIFY OFF
DECLARE
v_empno emp.employee_id%TYPE := 176;
v_asterisk emp.stars%TYPE := NULL;
v_sal emp.salary%TYPE;
BEGIN
  SELECT NVL(ROUND(salary/1000), 0)  INTO v_sal
     FROM emp  WHERE employee_id = v_empno;

     FOR i IN 1..v_sal
      LOOP
           v_asterisk := v_asterisk ||'*';
     END LOOP;

  UPDATE emp SET stars = v_asterisk
  WHERE employee_id = v_empno;
     COMMIT;
END;
/
SELECT employee_id,salary, stars
FROM emp WHERE employee_id =176;
```

Exercise 5 (Cursors)

In this exercise,

- First, you use an explicit cursor to process a number of rows from a table and populate another table with the results using a cursor FOR loop.
- Second, you write a PL/SQL block that processes information with two cursors, including one that uses a parameter.

Create a PL/SQL block to perform following:

a) Declare a cursor which retrieves the last_name, salary, and manager_id of employees working in the department specified (deptno is a varialble)
b) In the executable section, use the cursor for loop to operate on the data retrieved. If the salary of the employee is less than 5,000 and if the manager ID is either 101 or 124, display the message "<<last_name>> Due for a raise". Otherwise, display the message "<<last_name>> Not Due for a raise."
c) Test your PL/SQL block for the following Department IDs: 10,20,50,80

Now modify above PL/SQL block, to write two cursors-one without a parameter and one with a parameter. The first cursor retrieves the department number and the department name from the Department table for all departments whose ID number is less than 100. The second cursor receives the department number as a parameter, and retrieves employee details for those who work in that department and whose employee_id is less than 120. You need to declare variables to hold the values retrieved from each cursor. Use the %Type attribute while declaring variables.

```
SET SERVEROUTPUT ON
SET VERIFY OFF
SET ECHO OFF
DECLARE
v_deptno NUMBER := 10;
CURSOR  c_emp_cursor IS
      SELECT          last_name, salary,manager_id
      FROM       employees
      WHERE      department_id = v_deptno;
BEGIN
 FOR emp_record IN c_emp_cursor
  LOOP
   IF emp_record.salary< 5000 AND (emp_record.manager_id=101 OR
emp_record.manager_id=124) THEN
      DBMS_OUTPUT.PUT_LINE (emp_record.last_name || ' Due for a
raise');
   ELSE
      DBMS_OUTPUT.PUT_LINE (emp_record.last_name || ' Not Due for a
raise');
   END IF;
  END LOOP;
END;

SET SERVEROUTPUT ON
DECLARE
  CURSOR c_dept_cursor IS
        SELECT department_id,department_name
        FROM    departments
        WHERE department_id< 100
        ORDER BY    department_id;

  CURSOR c_emp_cursor(v_deptno NUMBER) IS
        SELECT last_name,job_id,hire_date,salary
        FROM    employees
        WHERE   department_id = v_deptno
        AND employee_id< 120;
v_current_deptno departments.department_id%TYPE;
v_current_dname departments.department_name%TYPE;
v_ename employees.last_name%TYPE;
v_job employees.job_id%TYPE;
v_hiredate employees.hire_date%TYPE;
v_sal employees.salary%TYPE;
```

```
BEGIN
      OPEN c_dept_cursor;
     LOOP
            FETCH c_dept_cursor INTO v_current_deptno,
v_current_dname;
            EXIT WHEN c_dept_cursor%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE ('Department Number : ' ||
v_current_deptno ||  '  Department Name : ' || v_current_dname);

      IF c_emp_cursor%ISOPEN THEN
        CLOSE c_emp_cursor;
       END IF;
       OPEN c_emp_cursor (v_current_deptno);
        LOOP
      FETCH c_emp_cursor INTO  v_ename,v_job,v_hiredate,v_sal;
        EXIT WHEN c_emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (v_ename || '    ' ||  v_job || '   '
||  v_hiredate || '   ' ||  v_sal);
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('-----------------------------------
-------------------------------------------------');
        CLOSE c_emp_cursor;

       END LOOP;

  CLOSE c_dept_cursor;
END;
```

Exercise 6 (Exception)

In this practice, you write a PL/SQL block that applies a predefined exception in order to process only one record at a time. The PL/SQL block selects the name of the employee with a given salary value.

1) Execute following command to re-create the message table

   ```
   DROP TABLE messages;
   CREATE TABLE messages (results VARCHAR2(80));
   ```

2) In the declarative section, declare two variables: v_ename of type employees.last_name and emp_sal of type employees.salary. Initialize the latter to 6000.
3) In the executable section, retrieve the last names of employees whose salaries are equal to the value emp_sal. If the salary entered returns only one row, insert into the message table the employee's name and the salary amount.

   Note: Do not use explicit cursors.

4) If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the messages table the message "No employee with a salary of <salary>".

5) f the salary entered returns multiple rows, handle the exception with an appropriate exception handler and insert into the messages table the message More than one employee with a salary of <salary>.
6) Handle any other exception with an appropriate exception handler and insert into the messages table the message Some other error occurred.
7) Display the rows from the messages table to check whether the PL/SQL block has executed successfully.
8) Change the initialized value of emp_salto 2000 and re-execute.

```
SET VERIFY OFF
DECLARE
v_ename    employees.last_name%TYPE;
v_emp_sal  employees.salary%TYPE := 6000;
BEGIN
  SELECT   last_name
  INTO           v_ename
  FROM           employees
  WHERE          salary = v_emp_sal;

  INSERT INTO messages (results)
  VALUES (v_ename || ' - ' || v_emp_sal);

EXCEPTION
  WHEN no_data_found THEN
    INSERT INTO messages (results)
    VALUES ('No employee with a salary of '|| TO_CHAR(v_emp_sal));
  WHEN too_many_rows THEN
    INSERT INTO messages (results)
    VALUES ('More than one employee with a salary of '||
             TO_CHAR(v_emp_sal));
  WHEN others THEN
    INSERT INTO messages (results)
    VALUES ('Some other error occurred.');
END;
/
SELECT * FROM messages;
```

Exercise 7 (Trigger)

We would like to create a trigger on insert and update operation on employee table such that the employees' salary should remain within a range according to their job type. Please follow these steps for that:

a) Create a trigger on employees table that fires before insert or update operation on each row. Trigger must should check the salary whether it is between the minimum and maximum range for salary for a specified job. If job salary does not fall into the range, then raise an error. To raise an application error, you can use following statement:

RAISE_APPLICATION_ERROR(-20100, "error in the salary range);

```
CREATE OR REPLACE TRIGGER check_salary_trg
BEFORE INSERT OR UPDATE OF job_id, salary
ON employees
FOR EACH ROW
Declare
v_minsal jobs.min_salary%type;
  v_maxsal jobs.max_salary%type;

BEGIN
  SELECT min_salary, max_salary INTO v_minsal, v_maxsal
  FROM jobs
  WHERE job_id = UPPER(:new.job_id);
  IF :new.salary NOT BETWEEN v_minsal AND v_maxsal THEN
    RAISE_APPLICATION_ERROR(-20100,
      'Invalid salary $' ||:new.salary ||'. '||
      'Salaries for job '|| :new.job_id ||
      ' must be between $'|| v_minsal ||' and $' || v_maxsal);
  END IF;

END;
```

Testing the trigger:

a. Now try by updating the salary of employee 115 to 2000. What is the result
b. Now update the salary of employee 115 to 2800. What is the result?

Now we want to write a trigger that prevent rows from being deleted during business hours i.e. Monday to Friday (9AM to 6PM). To get the hours from current date you use following function: TO_NUMBER(TO_CHAR(SYSDATE, 'HH24'));

```
CREATE OR REPLACE TRIGGER delete_emp_trg
BEFORE DELETE ON employees
DECLARE
  the_day VARCHAR2(3) := TO_CHAR(SYSDATE, 'DY');
  the_hour PLS_INTEGER := TO_NUMBER(TO_CHAR(SYSDATE, 'HH24'));
BEGIN
   IF (the_hour BETWEEN 9 AND 18) AND (the_day NOT IN ('SAT','SUN'))
THEN
     RAISE_APPLICATION_ERROR(-20150,
      'Employee records cannot be deleted during the business hours
of 9AM and 6PM');
   END IF;
END;
/
SHOW ERRORS


DELETE FROM employees
  WHERE job_id = 'SA_REP'
  AND   department_id IS NULL;
```

Week 10: PL/SQL Advanced Solution

This time we are again going to use HR database for answering following questions

Procedure

a) Modify the following to a procedure. Execute and invoke the procedure. Then Drop the procedure.

```
DECLARE
   v_dept_name departments.department_name%TYPE:= 'Education';
   v_dept_id NUMBER;
   v_max_deptno  NUMBER;
BEGIN
   SELECT MAX(department_id)  INTO v_max_deptno  FROM departments;
   DBMS_OUTPUT.PUT_LINE('The maximum department_id is : ' ||
   v_max_deptno);
   v_dept_id := 10 + v_max_deptno;
   INSERT INTO departments (department_id, department_name,
location_id)
   VALUES (v_dept_id,v_dept_name, NULL);
   DBMS_OUTPUT.PUT_LINE (' SQL%ROWCOUNT gives ' || SQL%ROWCOUNT);
 END;
/
```

b) Modify the procedure created above so that it takes a parameter (v_name). Print this parameter with 'Hello'. Create an anonymous block to invoke greet procedure.

```
CREATE PROCEDURE greet IS
   v_today DATE:=SYSDATE;
   v_tomorrow v_today%TYPE;
BEGIN
   v_tomorrow:=v_today +1;
   DBMS_OUTPUT.PUT_LINE(' Hello World ');
   DBMS_OUTPUT.PUT_LINE('TODAY IS : '|| v_today);
   DBMS_OUTPUT.PUT_LINE('TOMORROW IS : ' || v_tomorrow);
END;
```

c) Create, compile and invoke a procedure called ADD_JOB to insert a new job into the JOBS table. Provide ID and job title using two parameters. Check the procedure using IT_DBA as Job ID and DATABASE ADMIN as job title.

```
CREATE OR REPLACE PROCEDURE add_job (
  p_jobid jobs.job_id%TYPE,
  p_jobtitle jobs.job_title%TYPE) IS
BEGIN
  INSERT INTO jobs (job_id, job_title)
  VALUES (p_jobid, p_jobtitle);
  COMMIT;
     END add_job;
```

```
EXECUTE add_job ('IT_DBA', 'Database Administrator')
```

```
SELECT * FROM jobs WHERE job_id = 'IT_DBA';
```

d) Create a procedure called UPD_JOB to insert a new job into the JOBS table. Provide ID and job title using two parameters. Handle exception when job id not found.

```
CREATE OR REPLACE PROCEDURE upd_job(
  p_jobid IN jobs.job_id%TYPE,
  p_jobtitle IN jobs.job_title%TYPE) IS
BEGIN
  UPDATE jobs
  SET    job_title = p_jobtitle
  WHERE  job_id = p_jobid;
  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20202, 'No job updated.');
  END IF;
END upd_job;
/
```

e) Create a procedure that returns a value from the SALARY and JOB_ID columns for a specified employee ID. Execute the procedure using host variables with two OUT parameters-one for salary and other for job ID.

```
CREATE OR REPLACE PROCEDURE get_employee
    (p_empid IN  employees.employee_id%TYPE,
     p_sal   OUT employees.salary%TYPE,
     p_job   OUT employees.job_id%TYPE) IS
BEGIN
  SELECT  salary, job_id
  INTO    p_sal, p_job
  FROM    employees
  WHERE   employee_id = p_empid;
END get_employee;
```

Function

a) Create a function GET_JOB that take jobid as parameter and return a job title.

```
CREATE OR REPLACE FUNCTION get_job (p_jobid IN
jobs.job_id%type)
 RETURN jobs.job_title%type IS
  v_title jobs.job_title%type;
BEGIN
  SELECT job_title
  INTO v_title
  FROM jobs
  WHERE job_id = p_jobid;
  RETURN v_title;
END get_job;
```

b) Create a function that return annual salary of an employee and take salary and commission as input using following formula:
[Salary*12+commission=annual salary]

```
CREATE OR REPLACE FUNCTION get_annual_comp(
  p_sal  IN employees.salary%TYPE,
```

```
  p_comm IN employees.commission_pct%TYPE)
 RETURN NUMBER IS
BEGIN
  RETURN (NVL(p_sal,0) * 12 + (NVL(p_comm,0) * nvl(p_sal,0) * 12));
END get_annual_comp;
     /
```

c) Use the above function in a select statement against Employees table for employees in department 30.

```
SELECT employee_id, last_name,
       get_annual_comp(salary,commission_pct) "Annual
Compensation"
FROM   employees
WHERE department_id=30
/
```

d) Create a function that validate a particular department id and return true if department exist otherwise false. Create a procedure that add an employee to Employees table. Row should be only added if function for validation return true, otherwise alert the user with an appropriate message. Call the procedure with the name "Jane Harris" in department 18 and email jaharris.

```
CREATE OR REPLACE FUNCTION valid_deptid(
  p_deptid IN departments.department_id%TYPE)
  RETURN BOOLEAN IS
  v_dummy  PLS_INTEGER;
BEGIN
  SELECT  1
  INTO    v_dummy
  FROM    departments
  WHERE   department_id = p_deptid;
  RETURN  TRUE;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
     END valid_deptid;


CREATE OR REPLACE PROCEDURE add_employee(
   p_first_name employees.first_name%TYPE,
   p_last_name  employees.last_name%TYPE,
   p_email      employees.email%TYPE,
   p_job        employees.job_id%TYPE         DEFAULT 'SA_REP',
   p_mgr        employees.manager_id%TYPE     DEFAULT 145,
   p_sal        employees.salary%TYPE         DEFAULT 1000,
   p_comm       employees.commission_pct%TYPE DEFAULT 0,
   p_deptid     employees.department_id%TYPE  DEFAULT 30) IS
BEGIN
 IF valid_deptid(p_deptid) THEN
   INSERT INTO employees(employee_id, first_name, last_name, email,
     job_id, manager_id, hire_date, salary, commission_pct,
department_id)
   VALUES (employees_seq.NEXTVAL, p_first_name, p_last_name,
p_email,
```

```
     p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm, p_deptid);
 ELSE
   RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. Try
again.');
 END IF;
END add_employee;
/

EXECUTE add_employee('Jane', 'Harris', 'JAHARRIS', p_deptid=> 15)
```

Tutorial Week 12

Data Administration

Following link is a good tutorial on common administration commands:
https://www.oracletutorial.com/oracle-administration/

In this tutorial we will do only a subset of them.

PART 1: User Management

In this part of the tutorial, you will understand how to create users, how create profiles to limit resource usage, how to create roles and how to assign privileges. You will be using following commands for this exercise.

```
CREATE PROFILE profile_name
LIMIT { resource_parameters | password_parameters};

CREATE USER username
    IDENTIFIED BY password
    [DEFAULT TABLESPACE tablespace]
    [QUOTA {size | UNLIMITED} ON tablespace]
    [PROFILE profile]
    [PASSWORD EXPIRE]
    [ACCOUNT {LOCK | UNLOCK}];

CREATE ROLE role_name
[IDENTIFIED BY password]
[NOT IDENTIFIED]

GRANT {system_privileges | object_privileges} TO role_name;

GRANT role_name TO another_role_name;
```

This part only contains some examples for granting privileges. More details of different types of privileges can be found here: https://docs.oracle.com/database/121/TTSQL/privileges.htm#TTSQL345

Some examples of granting privileges: https://www.oracletutorial.com/oracle-administration/oracle-grant/

In this part, you need to create a user account for Jenny Goodman, the new human resources department manager. There are also two new clerks in the human resources department, David Hamby and Rachel Pandya. All three must be able to log in to the oracle database and to read data

from, and update records in, the HR.EMPLOYEES table. The manager also needs to be able to create new and remove employee records.

Step 1. Profile Creation

1. Create a profile named **CLERKPROFILE** that allows only 5 minutes idle time.

```
CREATE PROFILE CLERKPROFILE LIMIT
  SESSIONS_PER_USER          2
  CPU_PER_SESSION            UNLIMITED
  IDLE_TIME                  5;
```

2. Create a profile named MANPROFILE that allows unlimited sessions and unlimited idle time.

Step 2. Role Management

Now, create the **HRCLERK** and **HRMANAGER** roles that will be used in the next step. Here you have to think what permissions to be given to each role for each table. In this tutorial, we need to give permissions to clerks to read (i.e. Select operation) and update (Update operation) from EMPLOYEES tables. Similarly, you need to think in the context of manager. Following are the steps to implement these permissions.

1. Create the role named **HRCLERK**

```
Create role HRCLERK;
```

2. Grant object privileges to the role

```
Grant SELECT, UPDATE on HR.EMPLOYEES to HRCLERK;
```

   Similarly, you can grant any other privileges including system privileges.

3. Create the role named **HRMANAGER** with **INSERT** and **DELETE** permissions on the **HR.EMPLOYEES** table. Grant the **HRCLERK** role to the **HRMANAGER** role.. [Note that you have used one role to be assigned to another role why?]

```
Grant HRCLERK to HRMANAGER;
```

Step 3. User Management and Creation

Creating and Configuring Users

In this practice, you create the following users and assign appropriate profiles and roles to these users:

Name	Username	Description
David Hamby	DHAMBY	A new HR Clerk
Rachel Pandya	RPANDYA	A new HR Clerk
Jenny Goodman	JGOODMAN	A new HR Manager

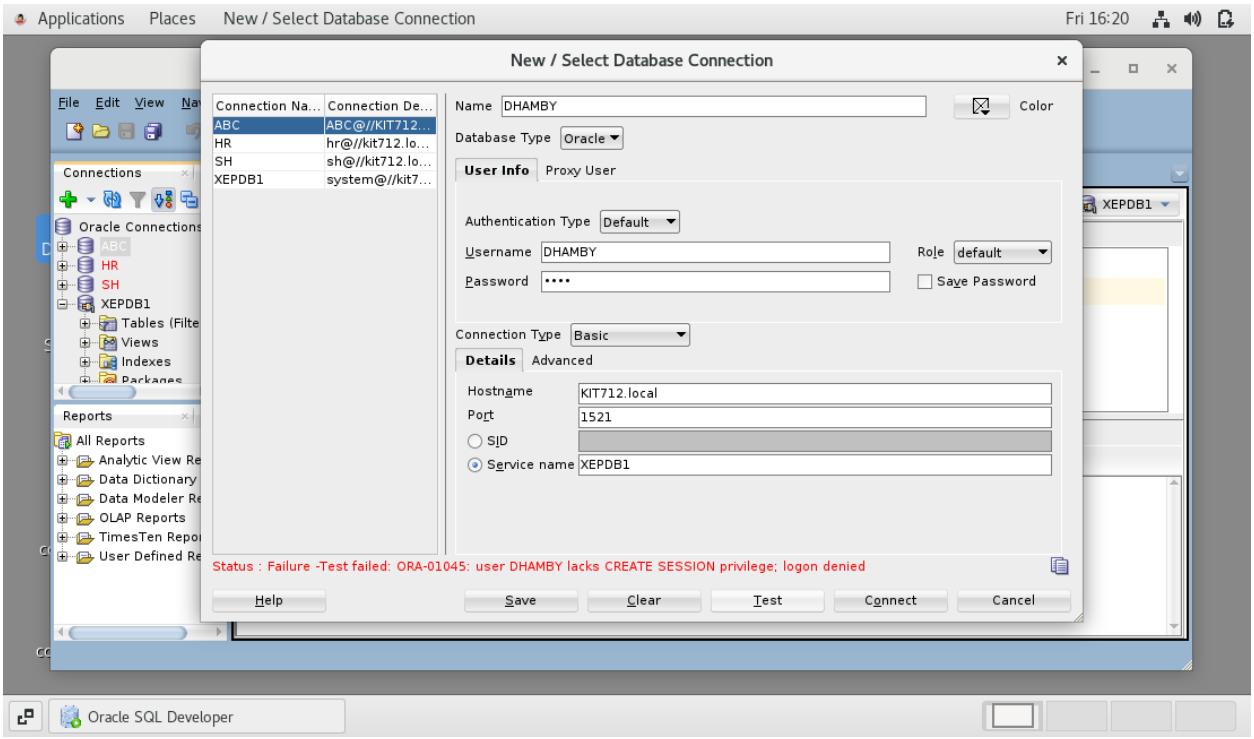
1) Create an account for David Hamby, a new HR clerk (i.e. assign HRCLERK and use CLERKPROFILE). You can use password as *'abcd'*.

```
CREATE USER DHAMBY IDENTIFIED BY abcd PROFILE CLERKPROFILE;

GRANT HRCLERK to DHAMBY;

GRANT CREATE SESSION TO DHAMBY; --system privileges to allow
connection with database
```

Without create session, you may get following error during connection:

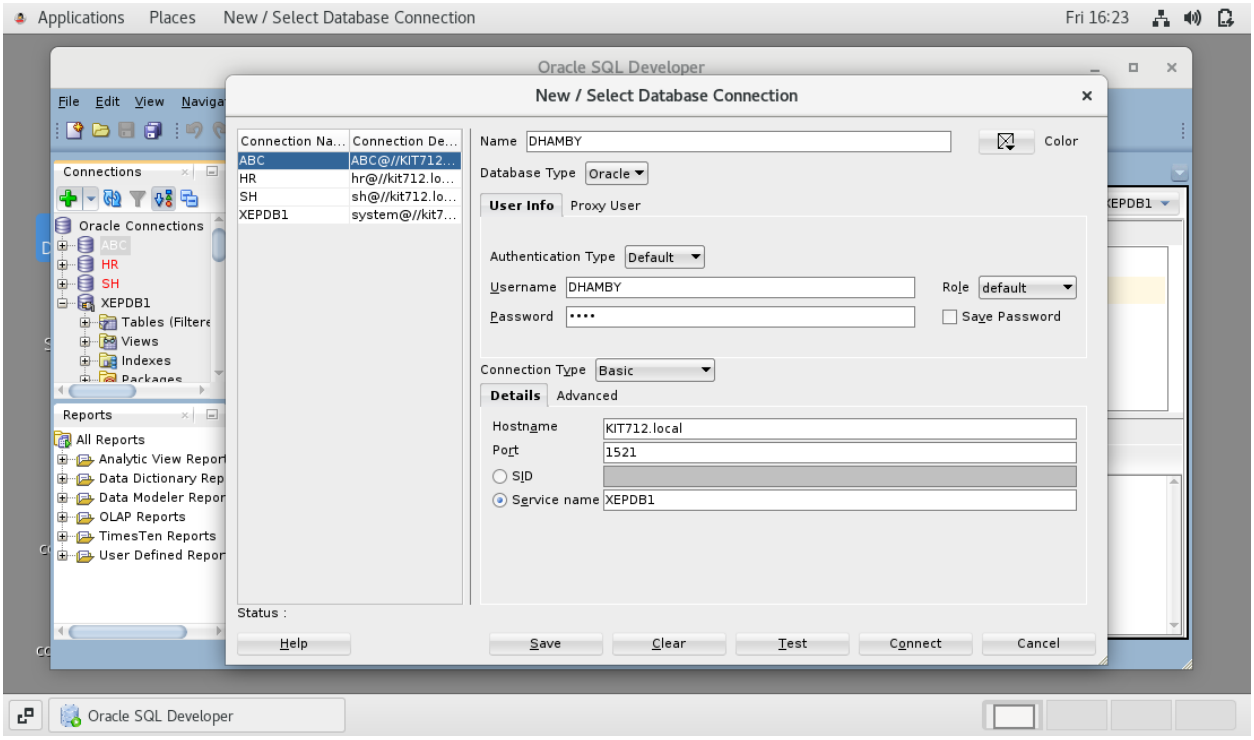


2) Create an account for Rachel Pandya, another new HR clerk. Repeat the steps shown above but with **RPANDYA** as the username and password *'pqr'*.

3) Create an account for Jenny Goodman, the new HR manager. Repeat the steps above but use **JGOODMAN** as the username and select the **HRMANAGER** role instead of the **HRCLERK** role. Use password as *'pwd'*; Also use MANPROFILE for profile of the user.

4) Test the new users in SQL Developer. Connect to the XEPDB1 database as the DHAMBY user. Select the row with **EMPLOYEE_ID=197** from the **HR.EMPLOYEES** table. Then attempt to delete it. (You should get the"insufficient privileges" error.)

To create a connection for DHAMBY user, you need to click first on + sign in your sql developer. And give information shown in following image. Password is same as the one you gave during creation of the user.



5) Repeat the test as the **JGOODMAN** user.

6) Leave DHAMBY connected for more than 5mins. HRPROFILE specifies that users whose sessions are inactive for more than 5 minutes will automatically be logged out. Verify that the user was automatically logged out by trying to select from the **HR.EMPLOYEES** table again. Try the same test with JGOODMAN user. What do you find?

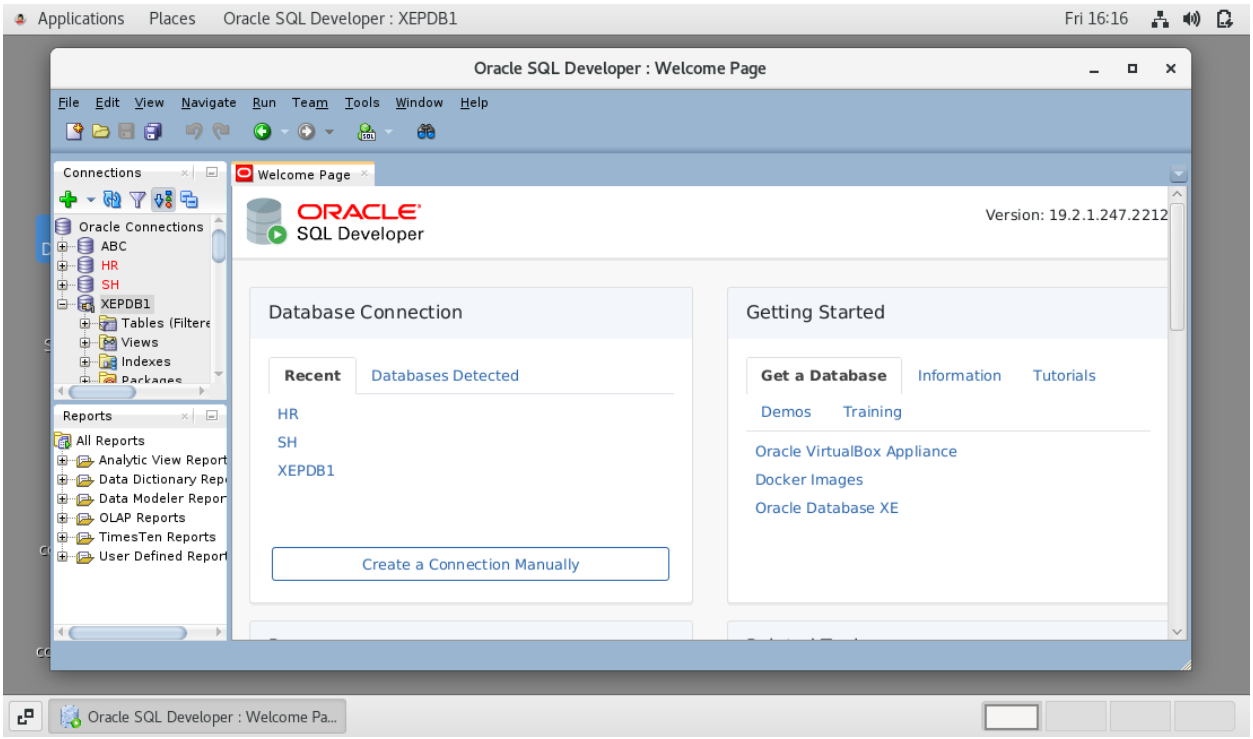
6) You can check whether users and roles are created successfully by using *dba_users*, *dba_tab_privs*, *dba_sys_privs* and *dba_roles* tables. Create SQL queries that check whether the above created users and roles are successfully created and privileges are successfully granted. For example,

```
select USERNAME, CREATED, PROFILE from dba_users where
USERNAME='DHAMBY';
select * from dba_roles where ROLE like 'HRCLERK';
select GRANTOR, grantee,privilege, table_name from dba_tab_privs
where TABLE_NAME='EMPLOYEES';
```

Part 2. Data Dictionary

For this part of the tutorial, you need to login as system user in sqlplus or sqldeveloper. Password for 'system' user is 'student'. This tutorial's objective is to teach you basics of data dictionary. There are several data dictionary tables and views. You can get full list from here: *http://www.oracle.com/pls/tahiti/tahiti.catalog_views*.

To login as System user in SQL Developer, you need to first open SQL Developer using icon in your oracle labshare virtual machine. Then double click on XEPDB1.



Q1. What information can you gather from the following query?Try this query after login as HR and then SH user/schema.

```
SELECT constraint_name, constraint_type, status, table_name FROM
user_constraints
```

Q2. Find which user have access to the JOBS table. *Hint: 'dba_tab_privs' table contain this information*
Find out what are constraints on JOBS table; *Hint: dba_constraints table contains this*

information

Q3. How can you find out whether the procedure 'ADD_JOB_HISTORY' did compile successfully? *Hint: the status of the procedure is then VALID. You can use 'DBA_OBJECTS' table for this*
Can you retrieve its source code from the dictionary? Hint: You can use 'dba_source' for this.

Part 3. Managing Database Storage Structures

More information can be found in your lecture slides

Creating a Tablespace

Information about creating Tablespaces can be found at
https://www.oracletutorial.com/oracle-administration/oracle-create-tablespace/

Create a new, locally managed tablespace (LMT) called **INVENTORY** of size **5 MB with following information:**

- Use **INVENTORY** as the tablespace name,
- Extent Management is **Locally Managed**,
- Type is **Permanent**,
- Datafile size 5 MB
  .

- Using following commands, execute following commands in your SQL Developer or SQLPLUS. Login as SYSTEM user:

```
set echo on

create table x (a char(1000)) tablespace inventory

/

insert into x values('a')

/

insert into x select * from x
/
insert into x select * from x
/
insert into x select * from x
/
insert into x select * from x
/
insert into x select * from x
```

```
/
insert into x select * from x
/
insert into x select * from x
/
insert into x select * from x
/
insert into x select * from x
/
insert into x select * from x
/
insert into x select * from x
/
insert into x select * from x
/
commit
/
quit
```

Do you see any error?. IF yes correct it by changing amount of space available for INVENTORY tablespace to 40MB. Rerun insert commands again.