**Lab 10**

1. Find the name of instructors whose department number is null
2. Find the name of the instructors whose department number is not null
3. To find the number of instructors in each department who teach a course in 2010 second semester
4. Find the courses taught in second semester 2009 but not in second semester 2010 (Use subquery)
5. Find the department that has the highest average salary

select dept\_name from instructor group by dept\_name having avg(salary)>=all(select avg(salary) from instructor group by dept\_name);

To test whether a subquery has any tuple in the results.

1. Find all courses taught in both Fall 2009 and Spring 2010

select course\_id

from section as S

where semester=’Fall’ and year=2009

and exists

(select \*

from section as T

where semester=’Spring’ and year=2010 and

S.course\_id =T.course\_id)

The exists construct return true if the result of the subquery is not empty. The not exists construct also available.

Test for absence of duplicate tables

The unique construct return true in the subquery contains no duplicate record

1. Find all the course that were offered at most once in 2009

select T.course\_id

from course as T

where unique (select R.course\_id

from section as R

where T.course\_id=R.course\_id and R.year=2009)

for a course not offered in 2009 the subquery return empty result, and the unique construct will return true for the empty result.

The below query is equivalent to the above query.

select T.course\_id

from course as T

where 1 < = ( select count(R.course\_id)

from section as R

where T.course\_id = R.course\_id and R.year=2009);

1. Find all courses that were offered at least twice in 2009

select T.course\_id

from course as T

where not unique

(select R.course\_id

from section as R

where T.course\_id = R.course\_id

and R.year=2009);

1. Find the average instructors salaries of those department where the average salary is greater than 42000

select dept\_name, avg\_salary

from (select dept\_name, avg(salary) as avg\_salary

from instructor group by dept\_name)

where avg\_salary>42000;

Note: the attribute in the subquery can be used in the outer query Eg. avg\_salary

1. Find the maximum across all departments of the total salary at each department.

select max(tot\_salary)

from (select dept\_name, sum(salary) as tot\_salary from instructor group by dept\_name);

1. find the department with maximum budget

with max\_budget(value) as

(select max(budget) from department)

select budget from department, max\_budget where

deparment.budget=max\_budget.value;

1. To find all department where total salary is greater than the average of the total salary of all the departments

with dept\_total(dept\_name, value) as (select dept\_name, sum(salary) from instructor group by dept\_name),

dept\_total\_avg(value) as

(select avg(value) from dept\_total)

select dept\_name

from dept\_total, dept\_total\_avg

where dept\_total.value > = dept\_total\_avg.value;

1. To list all departments with the number of instructors in each deparments.

select dept\_name, (select count(\*) from insructor where department.dept\_name=instructor.dept\_name) as num\_instructors from department;

**Views**

It is often needed to hide certain part of a database from certain user. We can use view for that purpose.

Examples

1. To create a view named faculty to hide the instructor detail of the instructor

Create view faculty as select id, name,dept\_name from instructor

1. To create a view lists all the courses offered by physics department in the 2009 second semester

SQL> create view physics\_2009 as select course.course\_id,sec\_id,building,room\_number from course,section where course.course\_id=section.course\_id and course.dept\_name=’Physics’ and section.semester=’second’ and section.year=209.

1. ‘The attribute name of the view can be specified explicitly’

Create view dept\_tot\_sal(dept\_name,tot\_salary) as select dept\_name, sum(salary) from instructor group by dept\_name;

insert into faculty values(‘12121’,’Ram’,’Music’);

For salary null value would be inserted.

select \* from faculty;

select course\_id from physics\_2009 where building =’Watson’;

Can use existing views to create another view’

create view physics\_watson as select course\_id from physics\_2009 where building =’Watson’

**Indexing and Sequencing**

Indexing  
An indexing is an ordered list of contents of a column or group of columns in a table.

Creating

* + 1. Simple Index

create index indexfile\_name on table\_name(column\_name)

* + 1. Composite index

create index indexfile\_name on table\_name(column\_name1, column\_name2)

Dropping

drop index indexfile\_name

Sequence

Most applications require the automatic generation of a numeric value. Oracle provides an automatic sequence generator of numeric values.

Creating

To create a sequence order\_seq which will start generating numbers from 1 to 9999 in ascending order with an interval of 1.

create sequence order\_seq

increment by 1

start with 1

maxvalue 9999

cycle;

Referencing a sequence

This can be done by using select statement

To refer to the next value

select order\_seq.nextval from dual

To refer to the current value

select order\_seq.currval from dual

Using a sequence

Insert values in the sales\_order table, the s\_order\_no must be generated by using the order\_seq sequence

insert into sales\_order(s\_order\_no,s\_order\_date,client\_no)

values(order\_seq.nextval,sysdate,’c0001’);

Altering a sequence

alter sequence order\_seq increment by 2

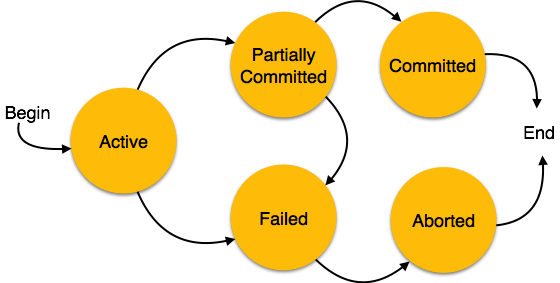
Dropping

drop sequence order\_seq

**Transactions and Concurrency**

**Transactions**

A **transaction** is a set of SQL queries that should be executed as whole. Any transaction ends when it is committed or rolled back, either explicitly with a **COMMIT** or ROLLBACK statement or implicitly when a DDL statement is issued.



1. In this exercise, you will see how to rollback or commit transactions. By default PostgreSQL commits each SQL statement as soon as it is submitted. To prevent the transaction from committing immediately, you have to issue a command begin; to tell PostgreSQL not to commit immediately. You can issue any number of SQL statements after this, and then either commit; to commit the transaction, or rollback; to abort the transaction. To see the effect, execute the following commands one at a time
2. begin ;
3. select \* from student where name = 'Kannan';
4. delete from student where name = 'Kannan';
5. select \* from student where name = 'Kannan';
6. rollback;
7. select \* from student where name = 'Kannan';  
     
   In your submission, explain what you observed and why it happened.

**Concurrency**

The SQL standard defines four levels of transaction isolation. The strictest is Serializable, which says that any concurrent execution of a set of Serializable transactions is guaranteed to produce the same effect as running them one at a time in some order.

Standard SQL Transaction Isolation Levels

| **Isolation Level** | **Dirty Read** | **Nonrepeatable Read** | **Phantom Read** |
| --- | --- | --- | --- |
| Read uncommitted | Possible | Possible | Possible |
| Read committed | Not possible | Possible | Possible |
| Repeatable read | Not possible | Not possible | Possible |
| Serializable | Not possible | Not possible | Not possible |

The other three levels are defined in terms of phenomena, resulting from interaction between concurrent transactions, which must not occur at each level.

The phenomena which are prohibited at various levels are:

Dirty Read

A transaction reads data written by a concurrent uncommitted transaction.

Nonrepeatable Read

A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

Phantom read

A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

PostgreSQL implements concurrency control using read committed isolation level as the default, but also supports other isolation which can be turned on by executing the following command “**set transaction isolation level serializable**” after begin statement.

In the read committed isolation level, each statement sees the effects of all preceding  
transactions that have committed, but does not see the effect of concurrently running transactions(i.e. updates that have not been committed yet). This low level of consistency can cause problems with transactions, and it is safer to use the snapshot isolation level if concurrent updates occur with multiple statement transactions.  
  
The snapshot isolation level offers a higher level of consistency, but does NOT guarantee serializability, even though it is the level that PostgreSQL (and Oracle) use when you set isolation level to serializable.  
In snapshot isolation, where a transaction gets a conceptual snapshot of data at the time it started, and all values it reads are as per this snapshot. Oracle also uses snapshot isolation for concurrency control, although Microsoft SQL Server and IBM DB2 behave a little differently since they use two-phase locking by default.   
  
In snapshot isolation, if two transactions concurrently update the same data item, one of them will be rolled back. However, snapshot isolation does not guarantee serializability; for example, it is possible that transaction T1 reads A and updates B =A, while transaction T2 reads B and updates A=B. In this case, there is no conflict on the update, since different tuples are updated by the two transactions, but the execution may not be serializable: in any serial schedule, A and B will become the same value, but with snapshot isolation, they may exchange values.  
  
For more details on read committed and snapshot isolation in general, read the textbook, and as implemented in PostgreSQL, see https://www.postgresql.org/docs/9.1/static/transaction-iso.html

1. In this exercise you will run transactions concurrently from two different pgAdmin3 windows, to see how updates by one transaction affect another.
2. Open two pgAdmin3 connections to the same database. Execute the following commands in sequence in the first window

begin ;

update student set tot\_cred = 55 where name = 'Kannan';

1. Now in the second window execute

begin;

select \* from student where name = 'Kannan';

1. Look at the value of tot\_cred. Can you figure out why you got the result that you saw? What does this tell you about concurrency control in PostgreSQL?
2. The value of tot\_cred of student 'Kannan' did not change since the different transactions get different snapshot since the command executed in one window was not committed.
3. Now in the first window execute

commit;

1. And in the second window execute

commit;

1. Observe how although the second transaction committed successfully after the first, it got the old value for tot\_cred before commit. Now what is the final state of the tuple at the end of these two sessions?

**After the commit command in both the windows, the tuple was actually updated and in the second window the change was reflected.**

1. Now, let us try to update the same tuple concurrently from two windows. In one window execute

begin;

update student set tot\_cred = 44 where name = 'Kannan'

1. Then in the second window, execute one after another:

begin;

select min(tot\_cred) from student where name = 'Kannan';

update student set tot\_cred = (select min(tot\_cred) from student where name = 'Kannan')+20 where name = 'Kannan'

1. See what happens at this point. The query appears to be hanging: PostgreSQL is waiting for the other query that updates student to complete.

Query is running............ appears at the bottom of the window.

1. Now in the first window, execute   
   commit;   
   and see what happens in the second window.

**The query in second window executes successfully**

1. Then execute commit; in the second window and see what happens. (In snapshot isolation, the above situation is supposed to result in the second transaction aborting. We will see this in the subsequent steps.)

**The query executes successfully**

1. Now in the second window execute   
   select min(tot\_cred) from student where name = 'Kannan';  
   This is the value after the above update was committed. Explain why you got the above value rather than 64.

**This is snapshot isolation level. In the second window the tot\_cred was 55 since in the first window, the updation was not committed before executing the command in the second window. Hence the snapshot received was 55 as the total credit for both windows. So 55+20 gives 75 and not 64.**

1. Next, in both windows execute the command after the begin command  
   set transaction isolation level serializable;  
   and reexecute the above queries, and see what happens. Describe what you observed, and explain why it happened.

**Here the query did not hang in the second window since the isolation level was set to serializable.**

**Thus in both the windows, the query gave the result 64 only as the total credit of the student 'Kannan'**

1. Open two connections (two new query windows) and type the following:

select id, salary from instructor where id in('63395', '78699') and note the results

Begin a transaction

Set the isolation level to serializable

Run this query in window 1: update instructor set salary = (select salary from instructor where id = '63395') where id = '78699';

Run this query in window 2: update instructor set salary = (select salary from instructor where id = '78699') where id = '63395';

commit window 1

commit window 2

Rerun the query: select id, salary from instructor where id in('63395', '78699') and compare the results. Is this equivalent to any serializable schedule?

**Both windows give the same result.**

**This is not equal to any serializable schedule.**

**This result we obtained because the select query was executed before setting the isolation level to serializable. Hence by default the snapshot isolation was set which had 10101's salary as 65000 and 12121's salary as 90000.**

Now, just before the update statement in window 1, add the line

1. select salary from instructor where id = '63395' for update;  
   Now rerun the rest of the steps as above and see what happens.
2. Explain why this happens.

**This result we obtained because the select query was executed before setting the isolation level to serializable. Hence by default the snapshot isolation was set which had 10101's salary as 65000 and 12121's salary as 90000. This record was locked due to select for update statement. When the next commit happens, the lock gets released.**