PUG CHALLENGE EXCHANGE AMERICAS

OpenEdge SQL Performance Troubleshooting and Understanding Query Plans

26/06/2016

Chandra Sekhar
Mohd Sayeed Akthar
Steven Pittman
Santhosh Kumar Duvvuru

Contents

Executive Summary

In this workshop we will take you through step by step interactive activities where you understand Query plans generated by OpenEdge SQL and demonstrate how important it is to understand query plans while working on performance related problems. In this Workshop you will learn

- Generating and understanding Query plans
- Identifying bad queries which are causing performance degradation
- Importance of statistics
- Troubleshooting common performance problems
- Solving problems using the query plan

Preface

This guide is designed to quickly introduce you to the steps involved in identifying and fixing OE SQL performance problems using Query plans and understand different types of statistics and their role in helping OE SQL optimizer to generate better query plans.

How to use this guide?

This guide is designed to flow from one chapter to the next, because each chapter requires knowledge built in previous chapters. You may decide to skip few chapters if you are aware of the concepts present those chapters. Throughout this workshop, we will be using JDBC client Squirrel for execution of queries.

Prerequisite

This workshop requires basic knowledge on SQL.

Chapter 1 –Why are we here?

Having knowledge of concepts presented in this workshop would really make a big difference when you deal with OE SQL in day to day life. Most of the time in this workshop, you will be working on your machine, understanding query plans and solving performance problems.

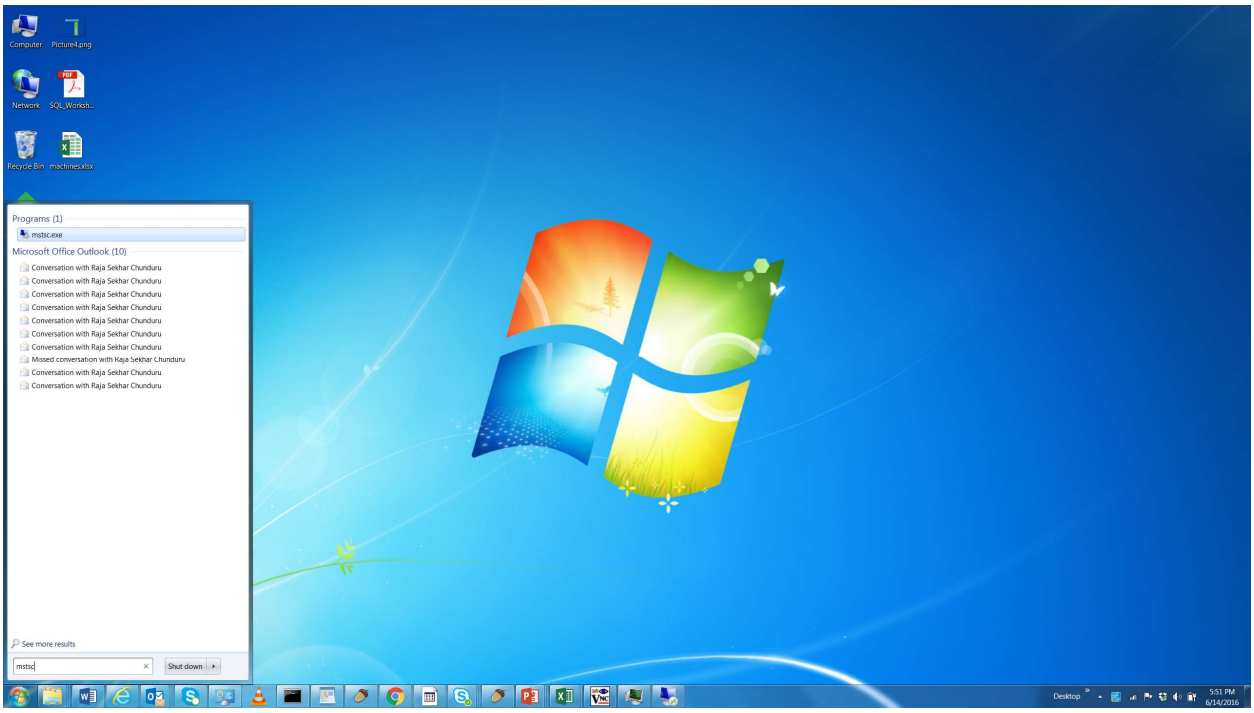
Chapter 2 – Generating query plans

In this chapter, you will learn different ways to generate Query plan. Query plan is the execution strategy determined by SQL optimizer to complete the query. For example, for a query like below, one of query plan can be “Perform the Index scan of CustIdx on table pub.customer with custnum value as 10 and return the custname as the result”.

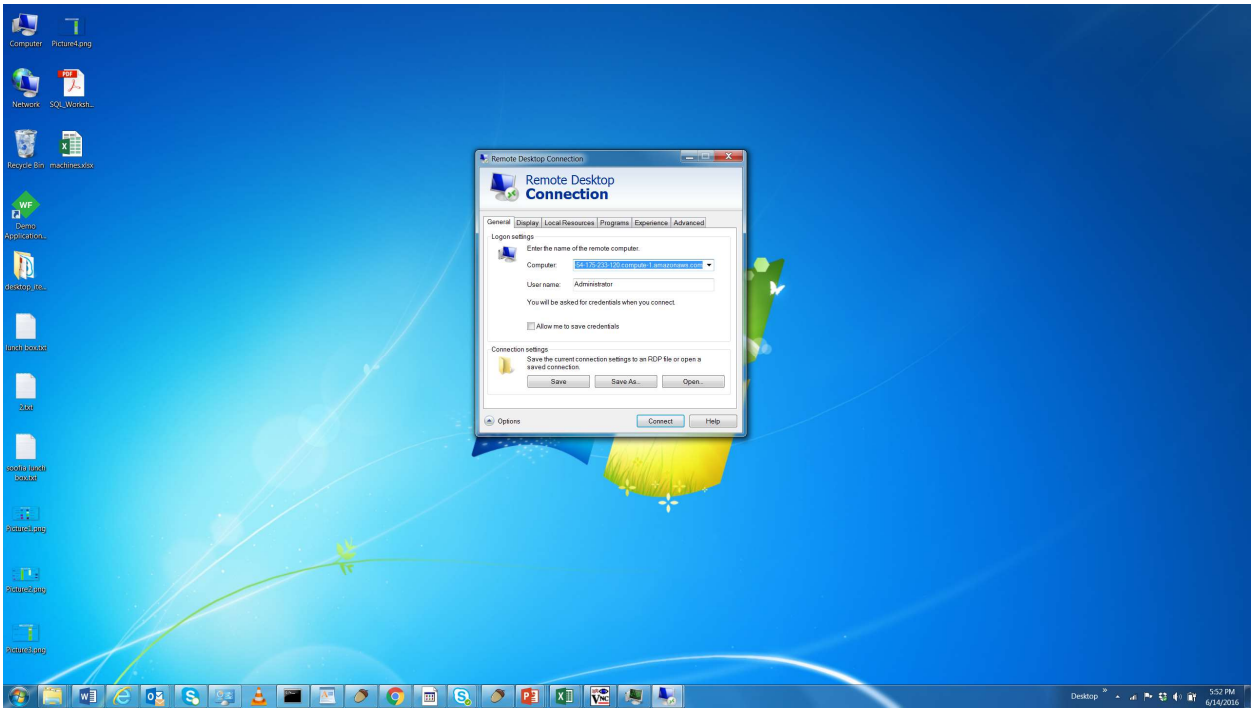
SELECT CustName FROM Pub.Customer WHERE CustNum=10;

Connecting to VM:

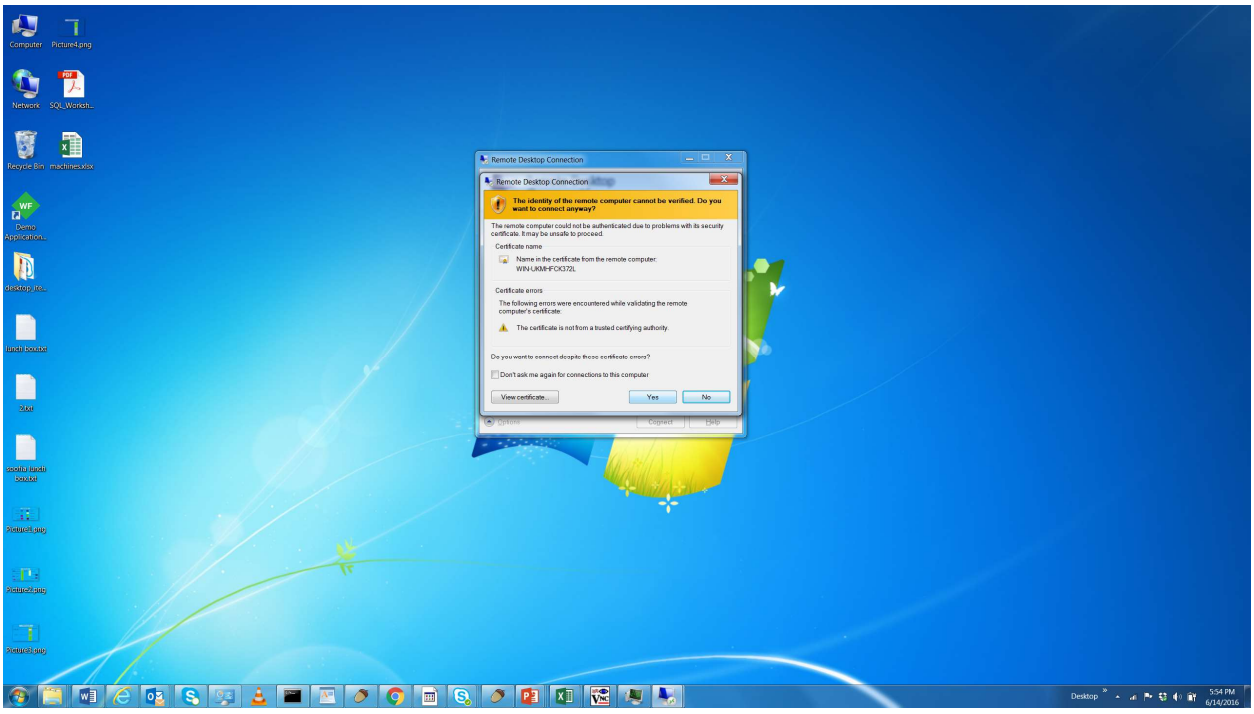
Search for “mstsc” and select.



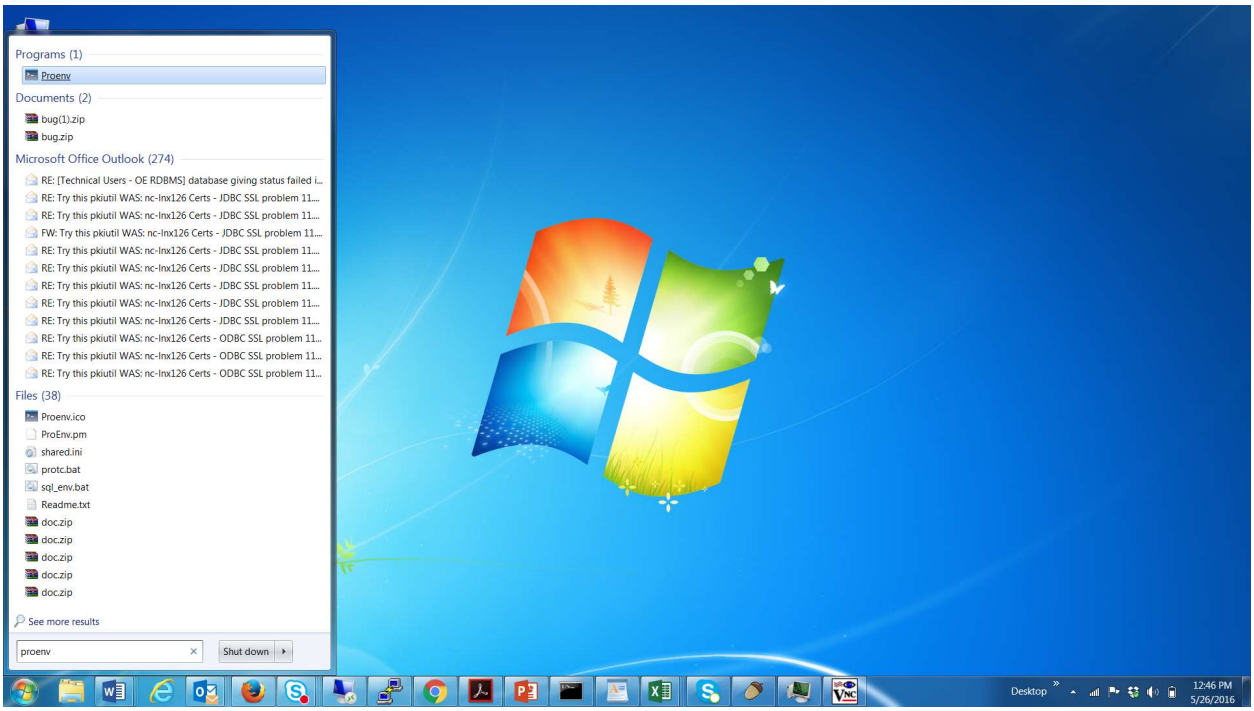
Enter the details of machine, user name and password which is allotted to you and click on connect.



Click on “Yes”.

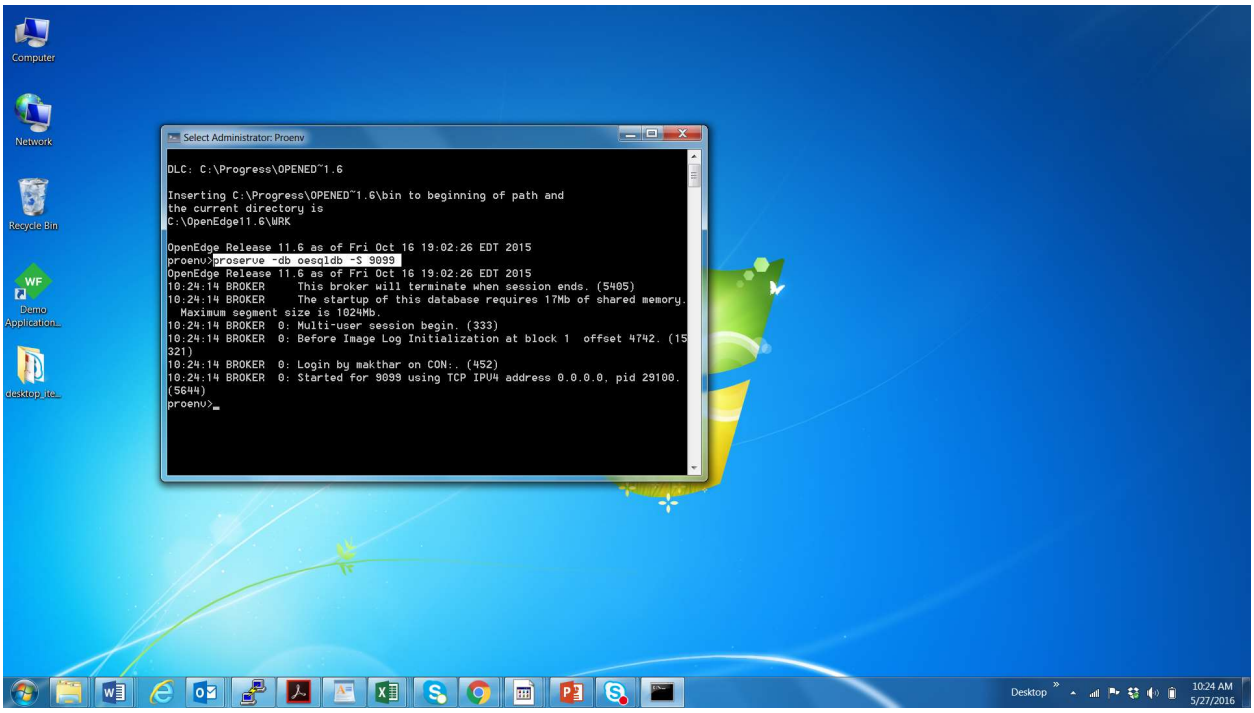


a. Start the database and connect to the database using “Squirrel” as the SQL client.
Step 1: Search for the “proenv” and press “Enter”.

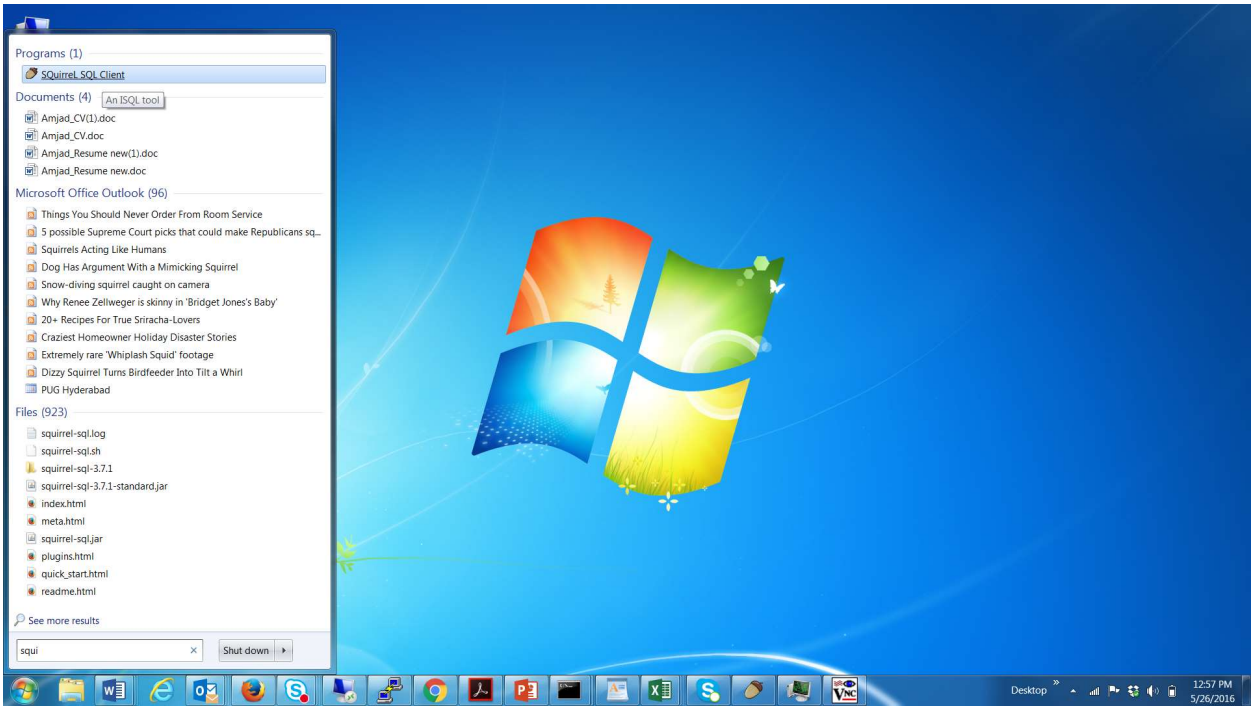


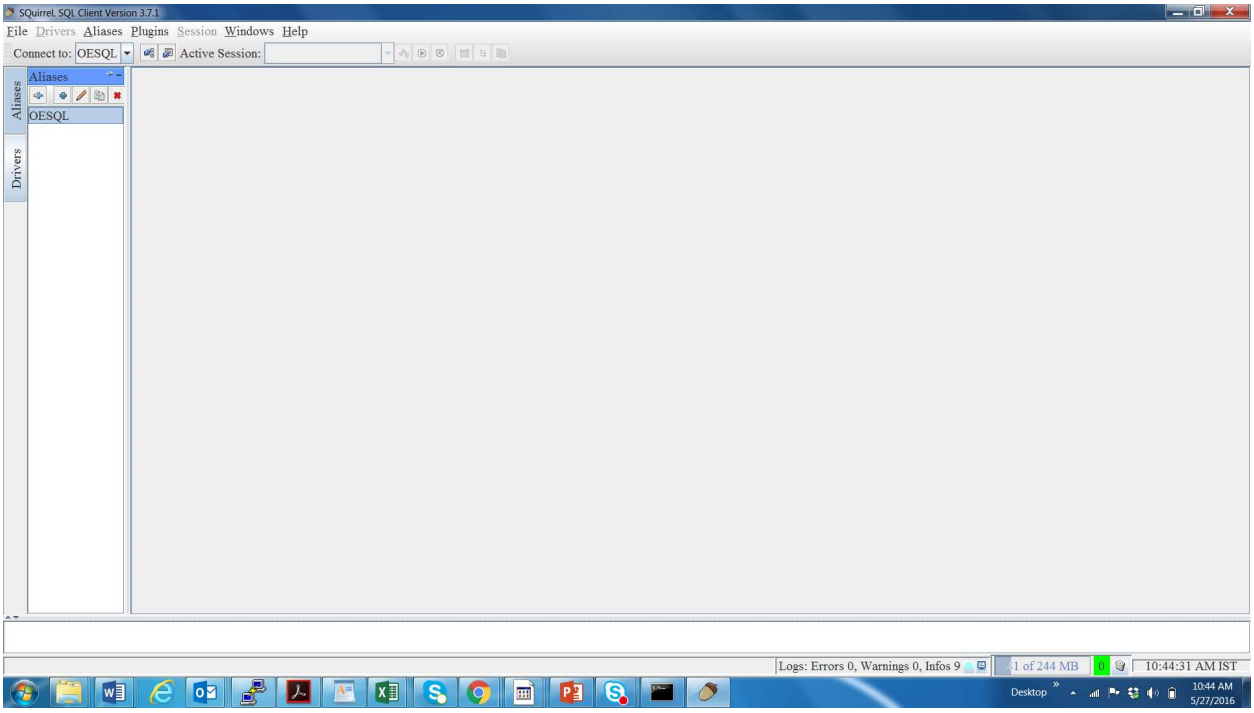
Step 2: Start the database using “proserve” command. This command accepts the database name and the port number where the database listens.

proserve –db oesqldb –S 9099

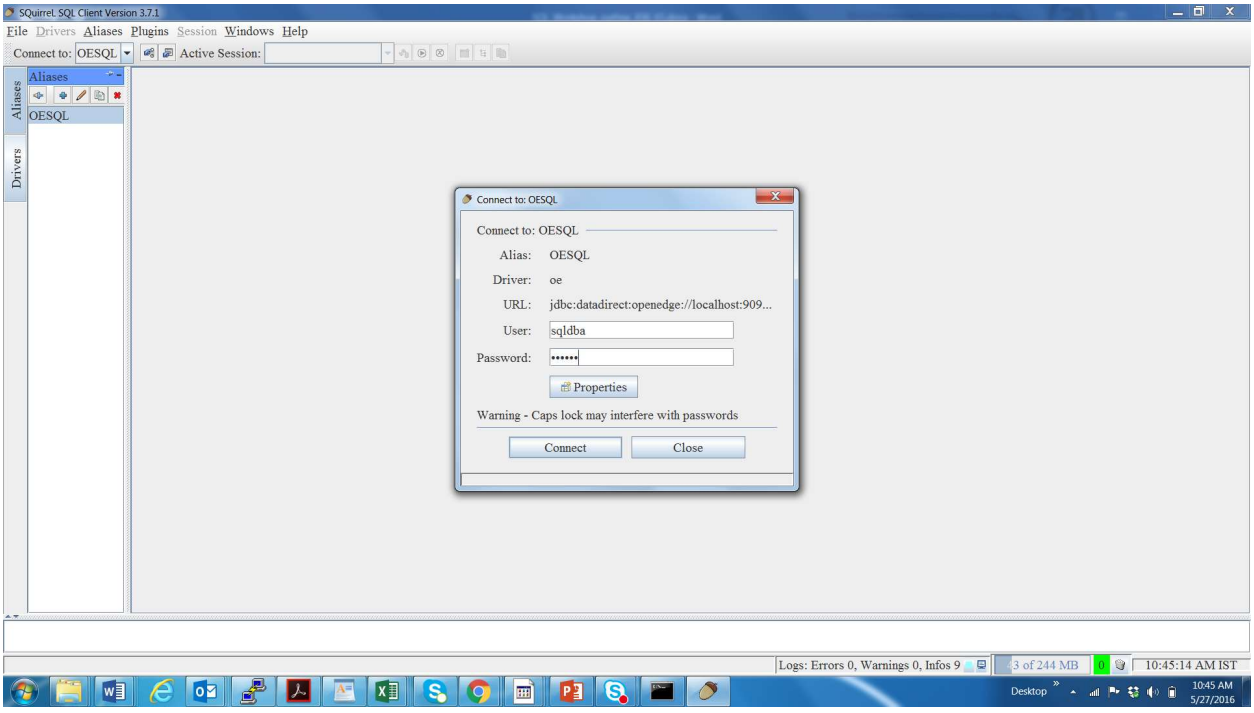


Step 3: Start SQuirrel client by searching “Squirrel”.



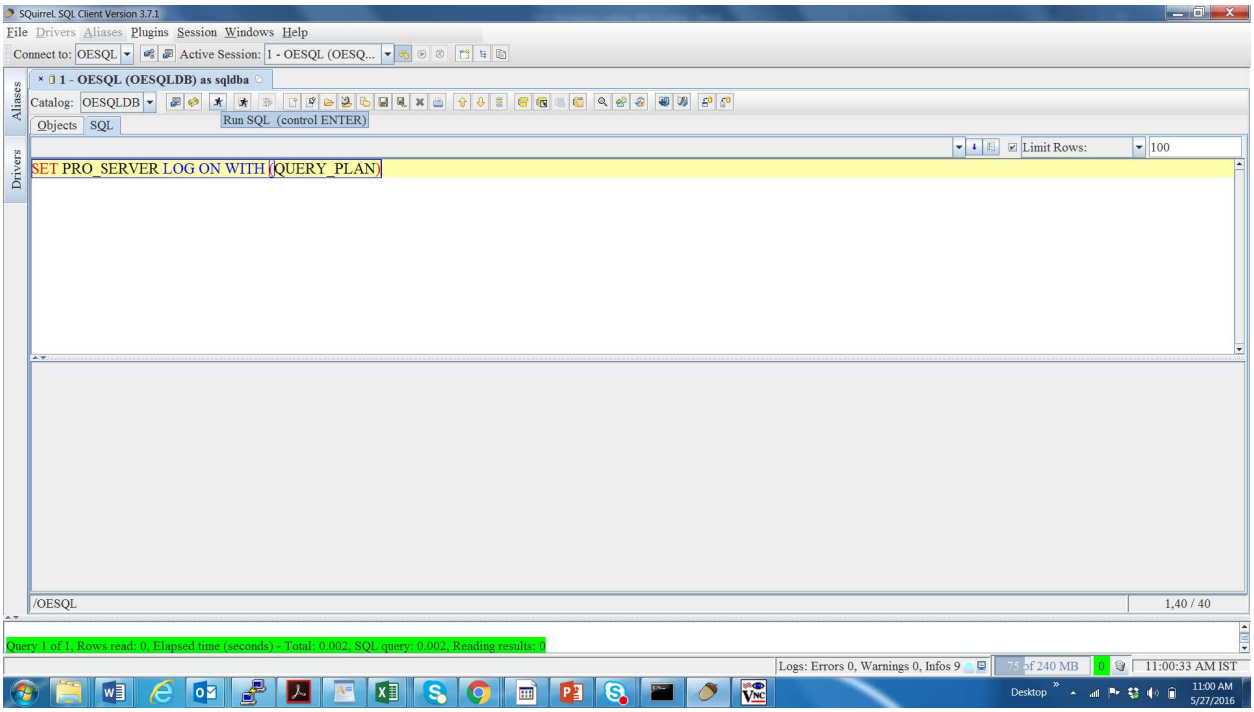


Step 4: Double click on “OESQL” alias name and click on “Connect”. User ‘sqldba’ is an existing user password for this user is “sqldba” (same as user name).



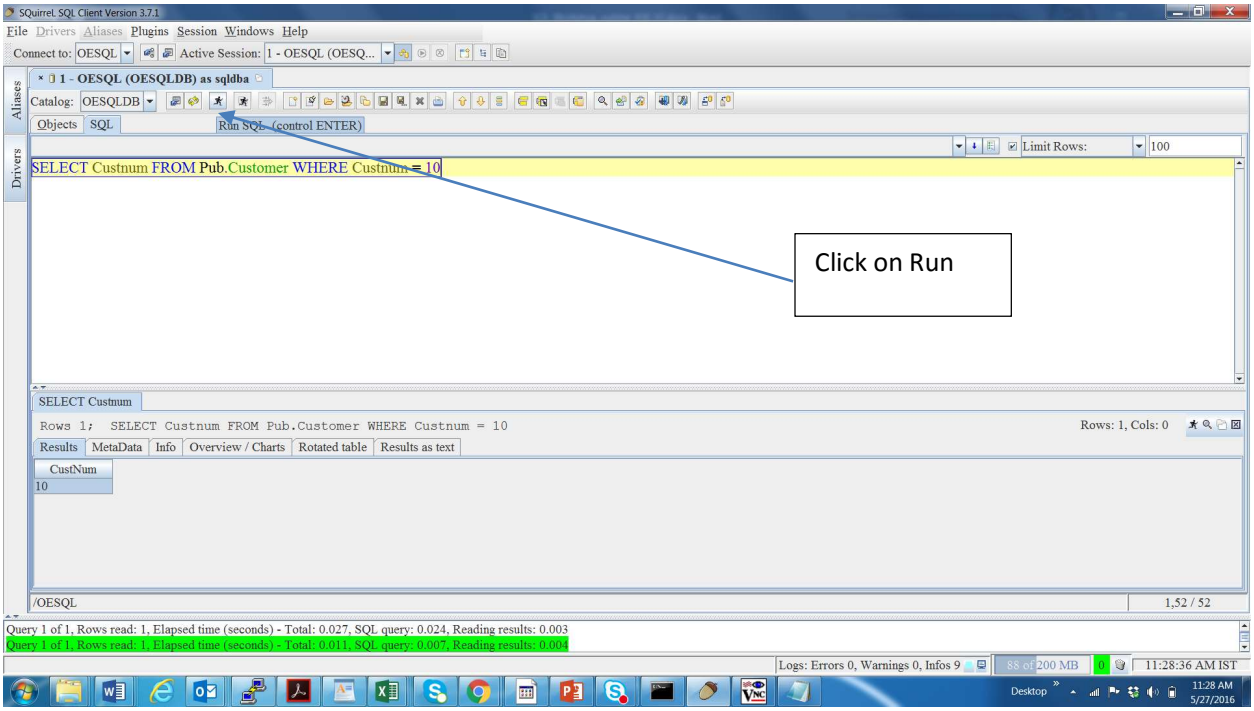
Step 5: Enable server log to display query plans, by executing below mentioned command and then “Run”

SET PRO_SERVER LOG ON WITH (QUERY_PLAN);

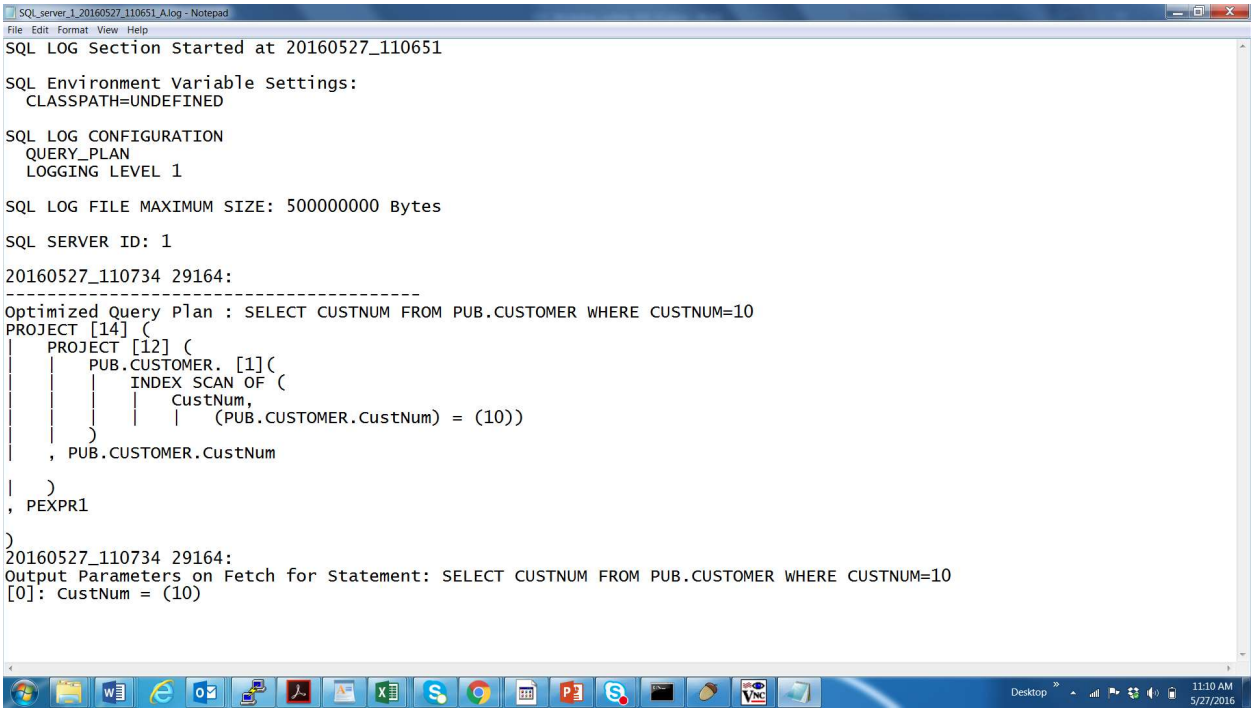


Step 6: Run the below query for which we want to see the query plan then “Run”

SELECT Custnum FROM Pub.Customer WHERE Custnum = 10;



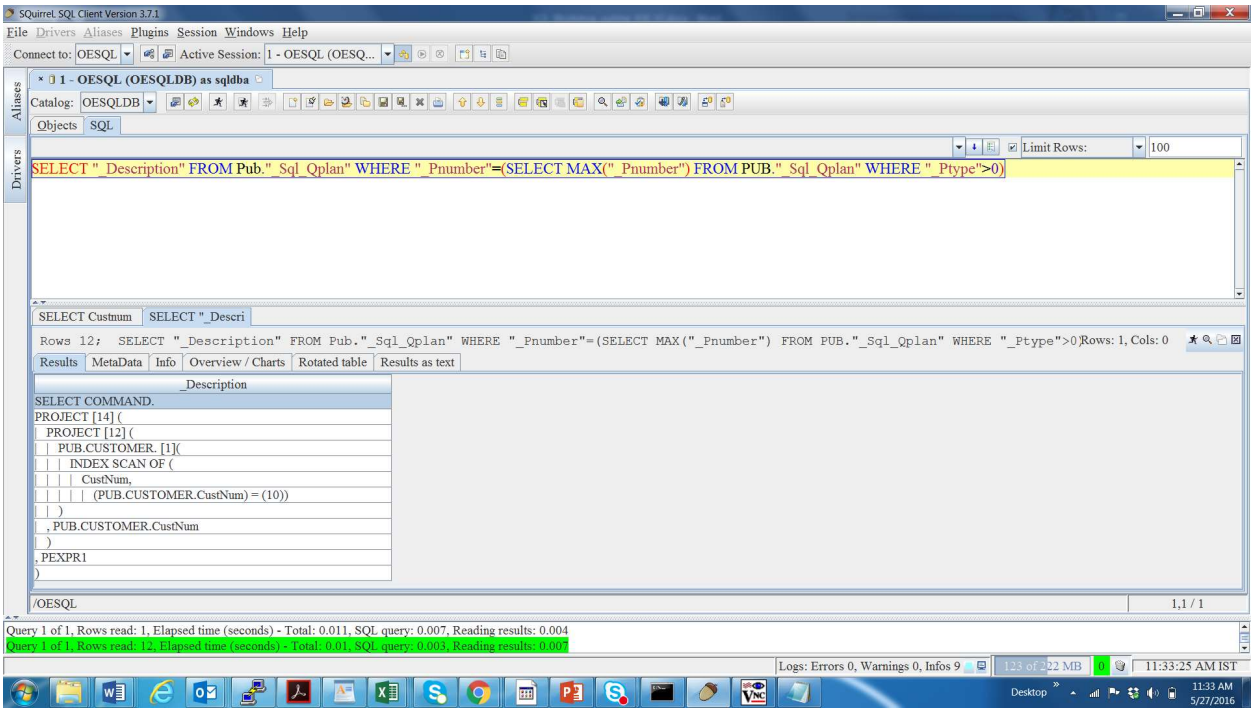
Step 7: Open the log file with name “SQL_server_X_XXXX_XXXX_A.log” at location C:\OpenEdge\WRK



Step 8: Query plan using the System Table, Pub."_Sql_QPlan"

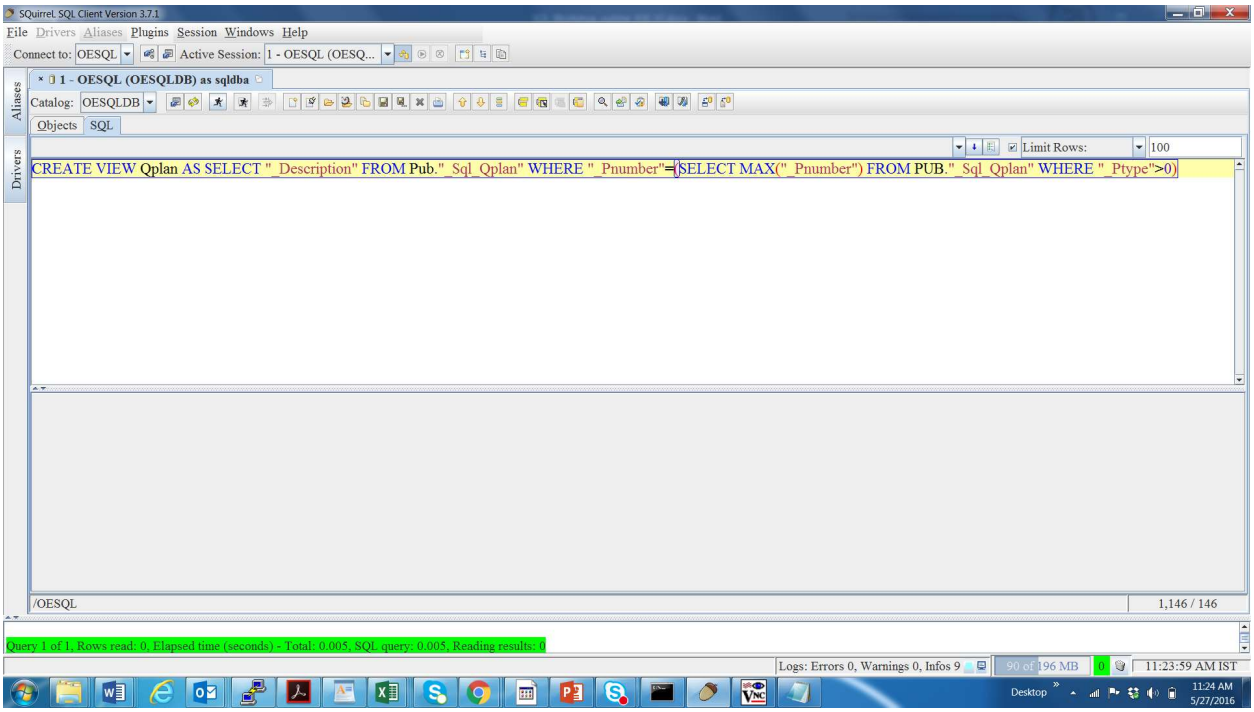
Run the below mentioned query in Squirrel window.

SELECT "_Description" FROM Pub."_Sql_Qplan" WHERE "_Pnumber"=(SELECT MAX("_Pnumber") FROM PUB."_Sql_Qplan" WHERE "_Ptype">0);



Step 2-9: Create the below view which helps us in looking at the query plan with ease. Throughout this session we will be using this view to check the query plan.

CREATE VIEW Qplan AS SELECT "_Description" FROM Pub."_Sql_Qplan" WHERE "_Pnumber"=(SELECT MAX("_Pnumber") FROM PUB."_Sql_Qplan" WHERE "_Ptype">0)



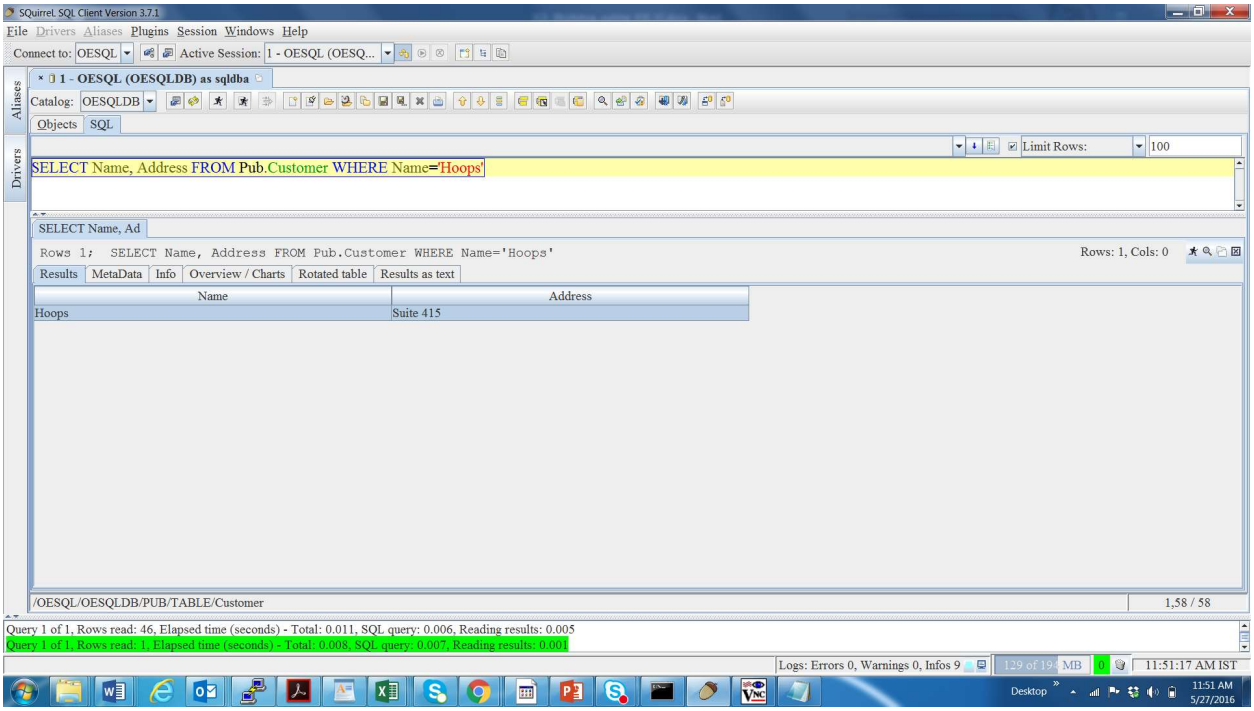
Chapter 3 – Understanding different query operations from Query plan

In this chapter, most of the query operations are explained with examples. You need to execute below queries, where each query will demonstrate one or some set of query operations using query plans.

a. Table scan
b. Index scan
c. Project
d. Restrict
e. Join
f. Aggregation
g. Sort
h. Union
i. TIDUNION
j. Dynamic Index (DIDX)

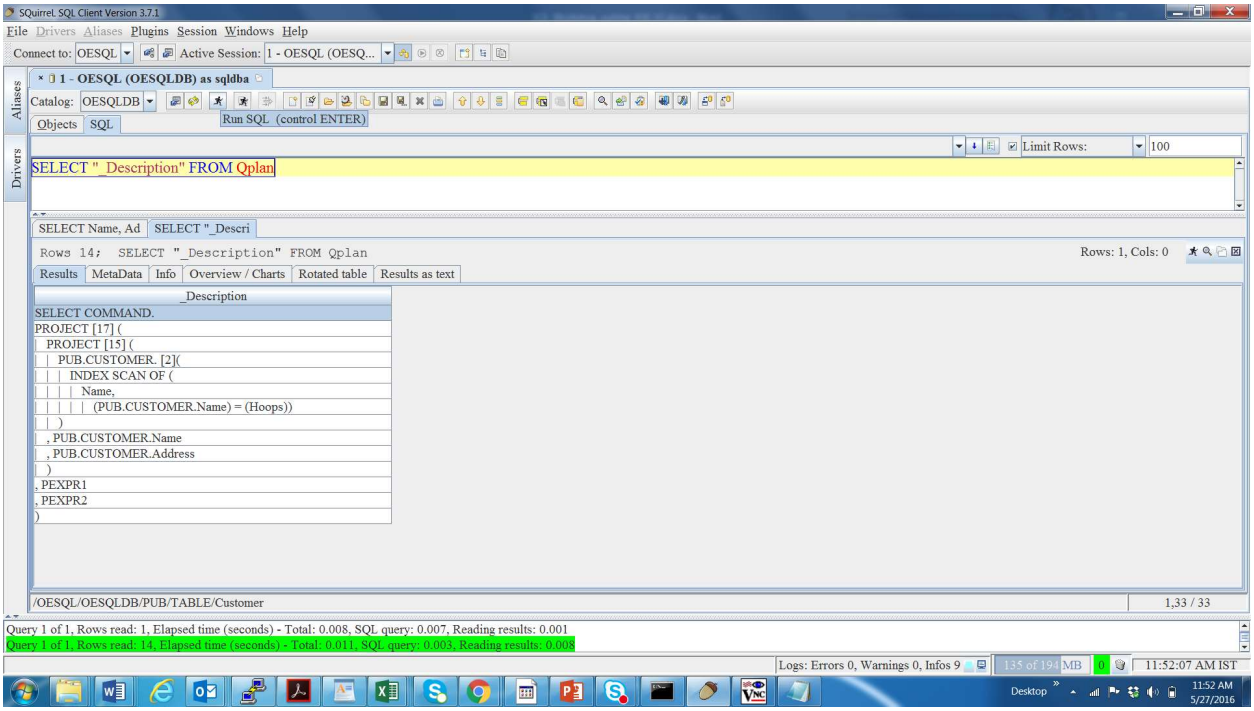
Query 1 – Below query demonstrates “Project” and “Index Scan” operations.

SELECT Name, Address FROM Pub.Customer WHERE Name='Hoops';



Then run select on Qplan view to see the Query plan

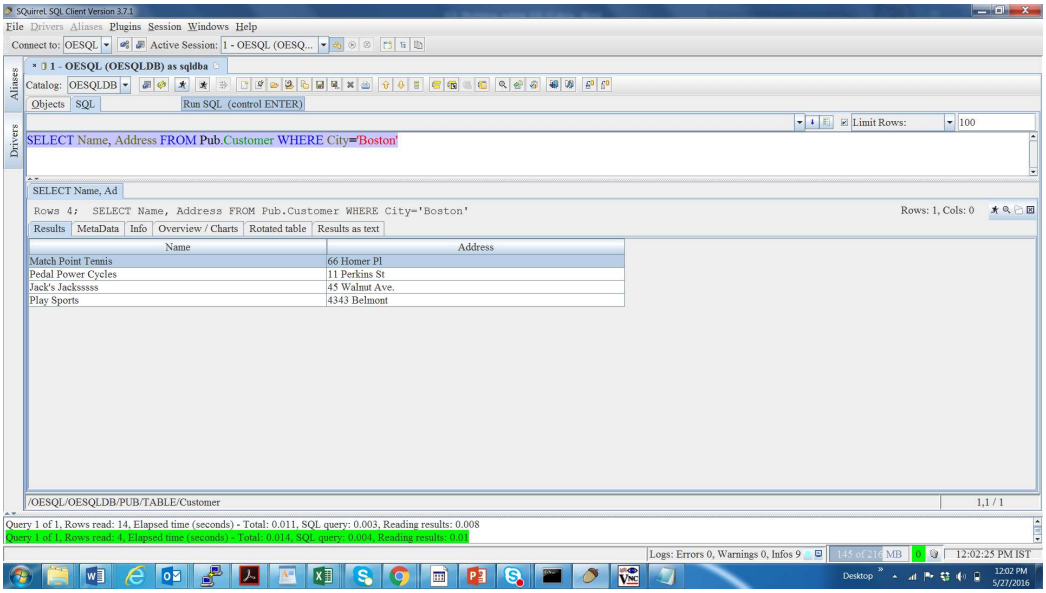
SELECT "_Description" FROM Qplan;



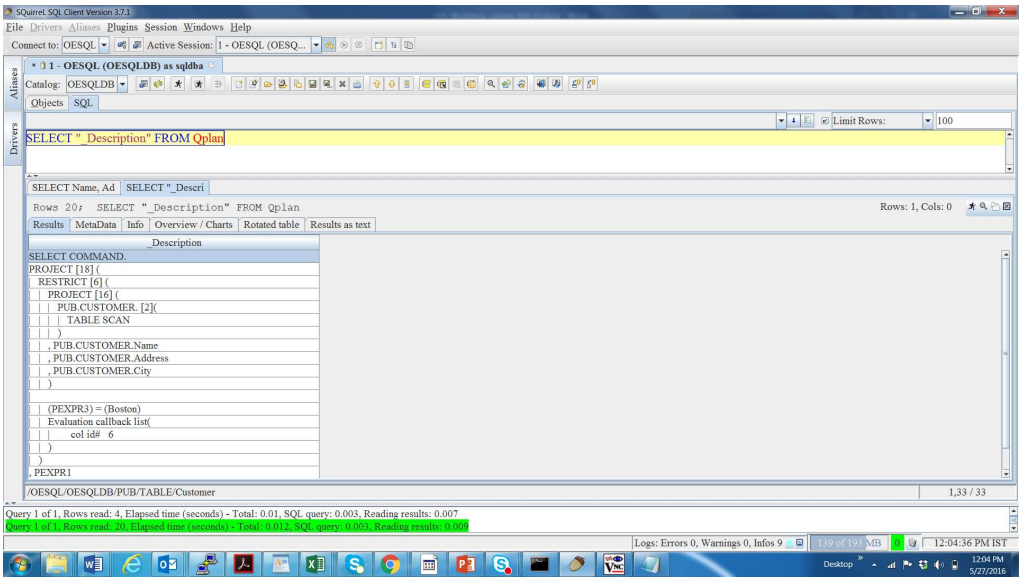

```
SELECT COMMAND.
PROJECT [17] (
|  PROJECT [15] (
|  |  PUB.CUSTOMER. [2](
|  |  |  INDEX SCAN OF (
|  |  |  |  Name,
|  |  |  |  |  (PUB.CUSTOMER.Name) = (Hoops))
|  |  )
|  , PUB.CUSTOMER.Name
|  , PUB.CUSTOMER.Address
|  )
, PEXPR1
, PEXPR2
)
```

Query 2 – Below query demonstrates “Restrict” and “Table Scan” operations.

SELECT Name, Address FROM Pub.Customer WHERE City='Boston';



Now Check Query plan.

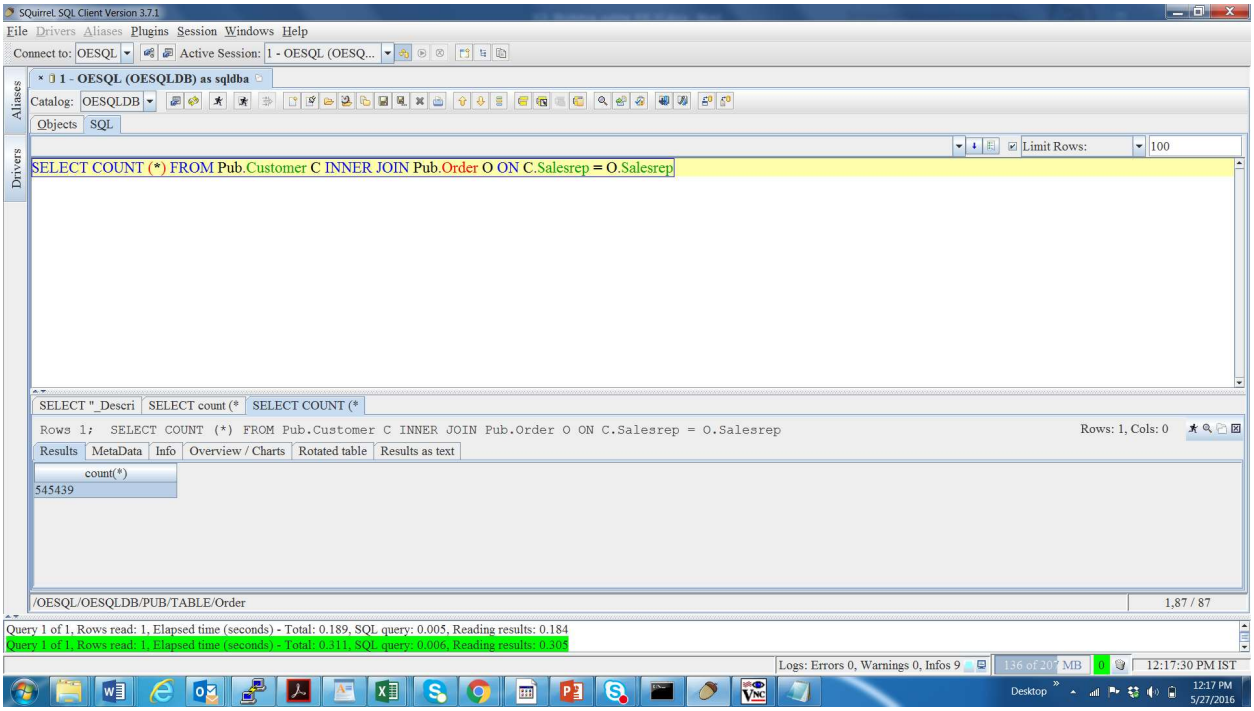


SELECT COMMAND.
PROJECT [18] (
| RESTRICT [6] (
| | PROJECT [16] (
| | | PUB.CUSTOMER. [2](
| | | | TABLE SCAN
| | |)
| | , PUB.CUSTOMER.Name

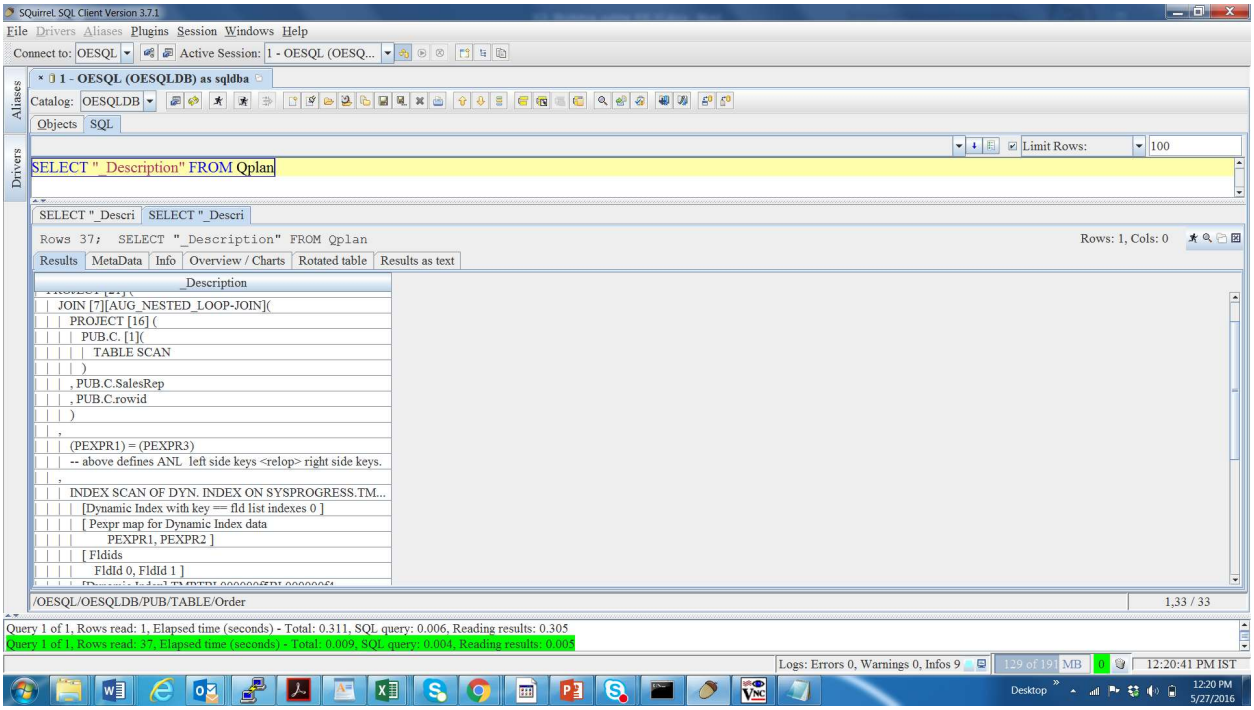
```
|  |  , PUB.CUSTOMER.Address
|  |  , PUB.CUSTOMER.City
|  |  )
|
|  |  (PEXPR3) = (Boston)
|  |  Evaluation callback list(
|  |  |       col id#  6
|  |  )
|  )
, PEXPR1
, PEXPR2
)
```

Query 3 – Below query demonstrates demonstrate “Join” and “DIDX” operations.

SELECT COUNT(*) FROM Pub.Customer C INNER JOIN Pub.Order O ON C.Salesrep = O.Salesrep;



Now Check Query plan

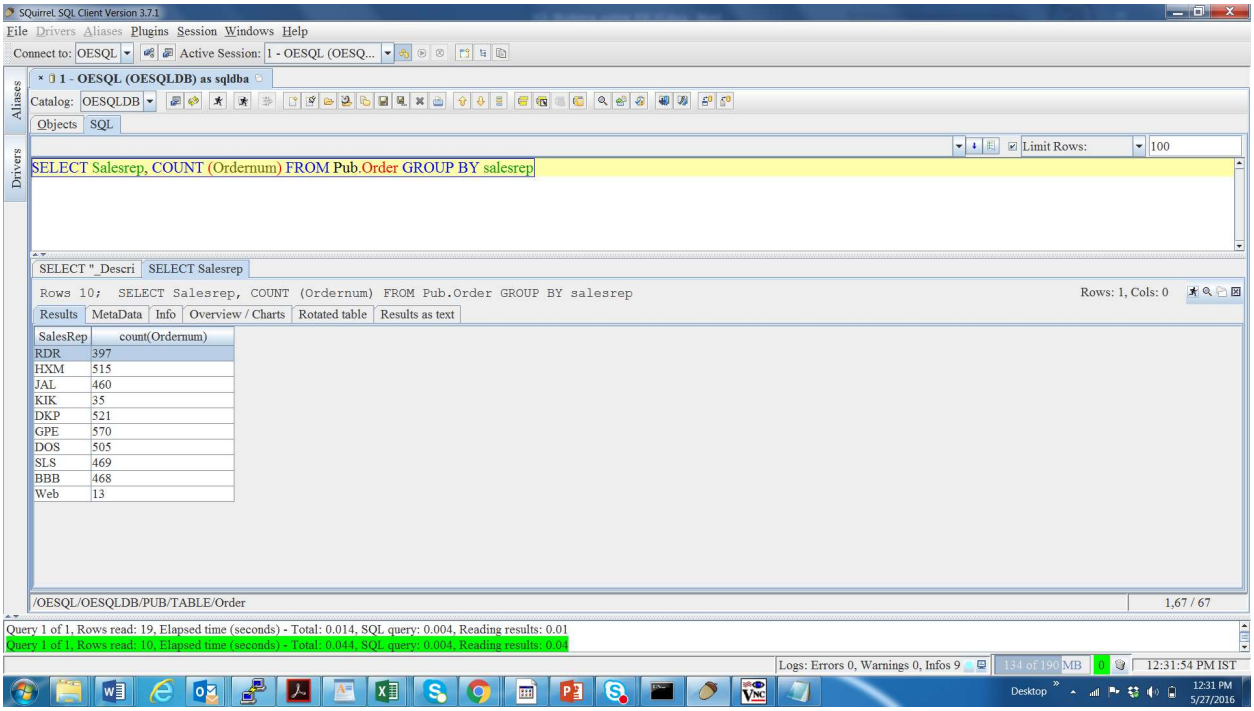


SELECT COMMAND.
PROJECT [22] (
| PROJECT [21] (
| | JOIN [7][AUG_NESTED_LOOP-JOIN](
| | | PROJECT [16] (
| | | | PUB.C. [1](
| | | | | TABLE SCAN
| | | , PUB.C.SalesRep
| | | , PUB.C.rowid
| | ,
| | | (PEXPR1) = (PEXPR3)
| | | -- above defines ANL left side keys <
| | ,
| | | INDEX SCAN OF DYN. INDEX ON SYSPROGRES
| | | | [Dynamic Index with key == fld lis
| | | | [Pexpr map for Dynamic Index data
| | | | PEXPR1, PEXPR2]
| | | | [Fldids
| | | | Fldld 0, Fldld 1]
| | | | [Dynamic Index] TMPTBL000000f5BL00
| | | | | (SYSPROGRESS.TMPTBL000000f4.0)
| | | | PROJECT [19] (
| | | | | PUB.O. [2](
| | | | | | TABLE SCAN

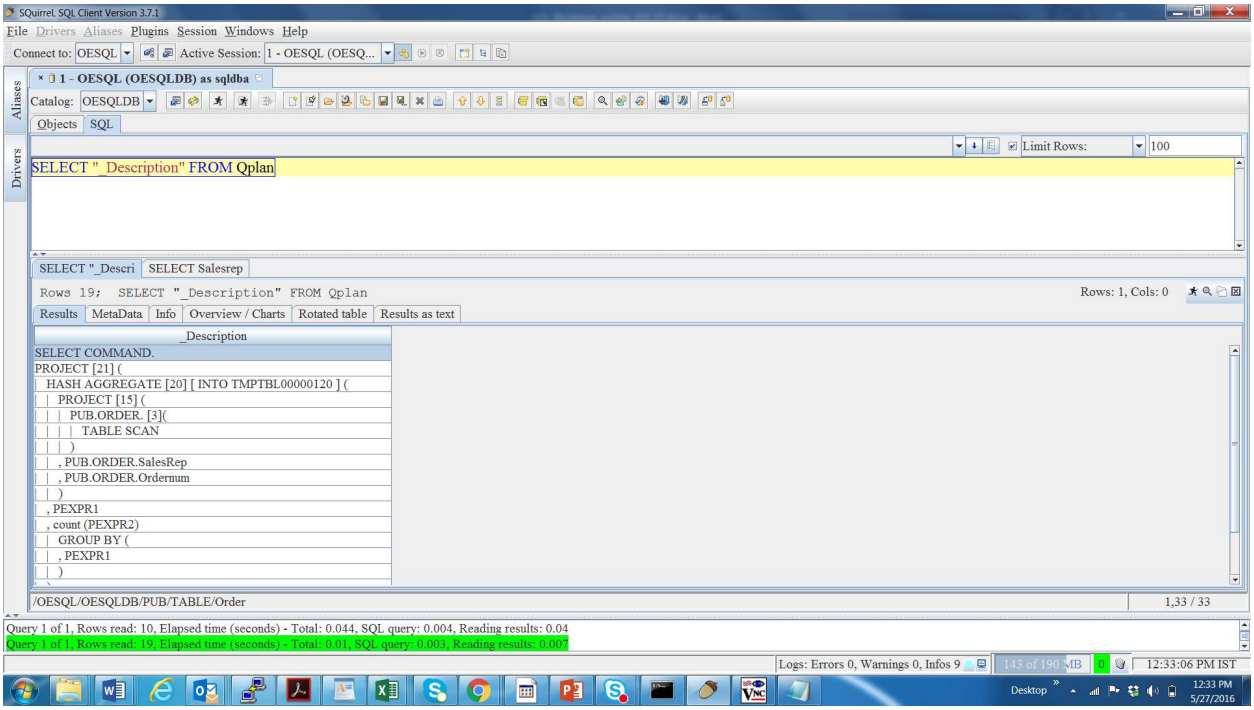
```
|  |  |  |  , PUB.O.SalesRep
|  |  |  |  , PUB.O.rowid
|  |  |  |  )
|  |  |
|  |  |  )
|  |  )
|  , count (*)
|  )
```

Query 4 – Below query demonstrates “Aggregation” operation.

SELECT Salesrep, COUNT (Ordernum) FROM Pub.Order GROUP BY salesrep;



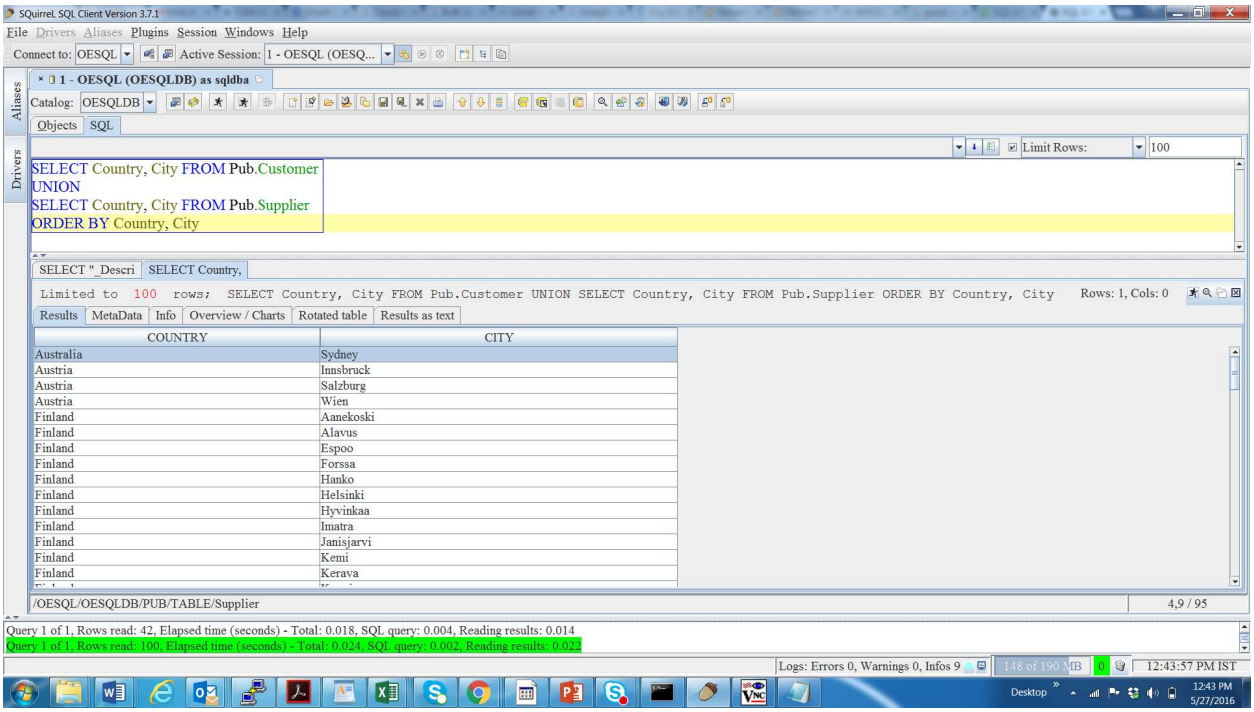
Now check Query plan



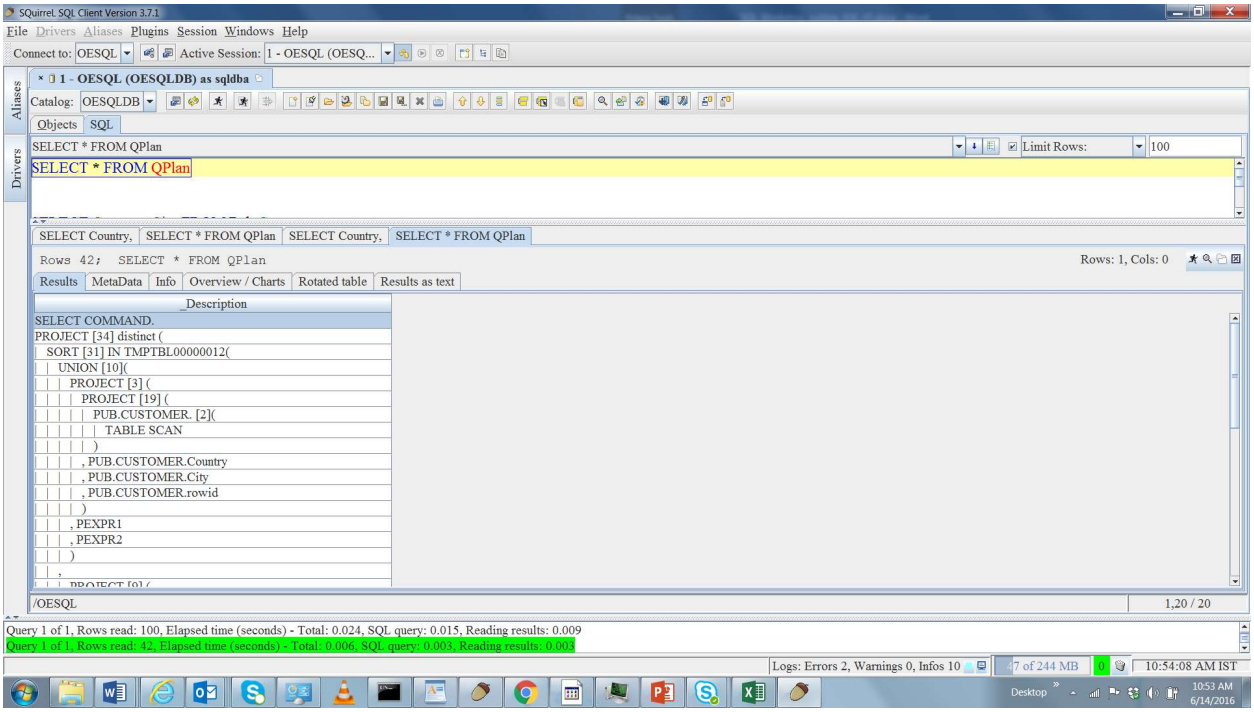
SELECT COMMAND.
PROJECT [21] (
| HASH AGGREGATE [20] [INTO TMPTBL00000120] (
| | PROJECT [15] (
| | | PUB.ORDER. [3](
| | | | TABLE SCAN
| | |)
| | , PUB.ORDER.SalesRep
| | , PUB.ORDER.Ordernum
| |)
| , PEXPR1
| , count (PEXPR2)
| | GROUP BY (
| | , PEXPR1
| |)
|)
, PEXPR1
, PEXPR2
)

Query 5 - Below query demonstrates “Union” and “Sort” operations.

```
SELECT Country, City FROM Pub.Customer
                UNION
SELECT Country, City FROM Pub.Supplier
ORDER BY Country, City;
```



Now check Query plan

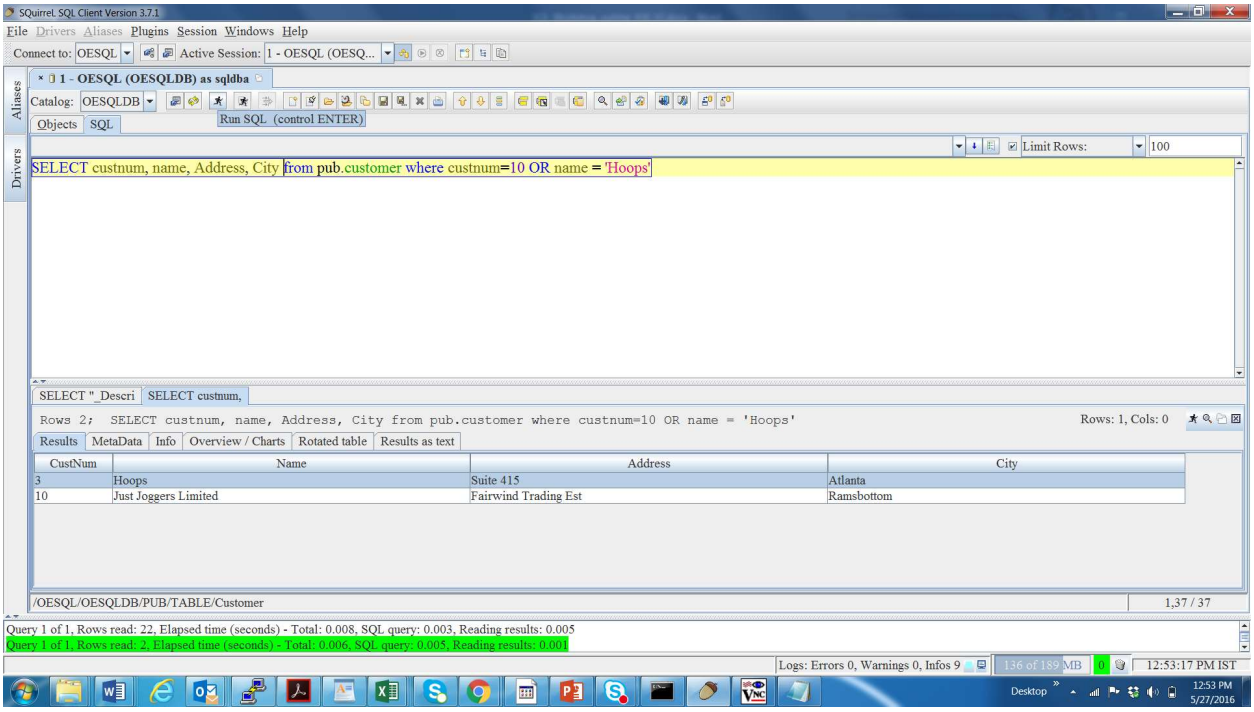


SELECT COMMAND.
PROJECT [34] distinct (
| SORT [31] IN TMPTBL0000017c(
| | UNION [10](
| | | PROJECT [3] (
| | | | PROJECT [19] (
| | | | | PUB.CUSTOMER. [2](
| | | | | | TABLE SCAN
| | | | |)
| | | | , PUB.CUSTOMER.Country
| | | | , PUB.CUSTOMER.City
| | | | , PUB.CUSTOMER.rowid
| | | |)
| | | , PEXPR1
| | | , PEXPR2
| | |)
| | ,
| | | PROJECT [9] (
| | | | PROJECT [27] (
| | | | | PUB.SUPPLIER. [6](
| | | | | | TABLE SCAN
| | | | | , PUB.SUPPLIER.Country
| | | | | , PUB.SUPPLIER.City
| | | | | , PUB.SUPPLIER.rowid
| | | |)
| | | , PEXPR1
| | | , PEXPR2
| | , PEXPR1
| | , PEXPR2
| | SORT BY (
| | , Sort project expression #0 == PEXPR1

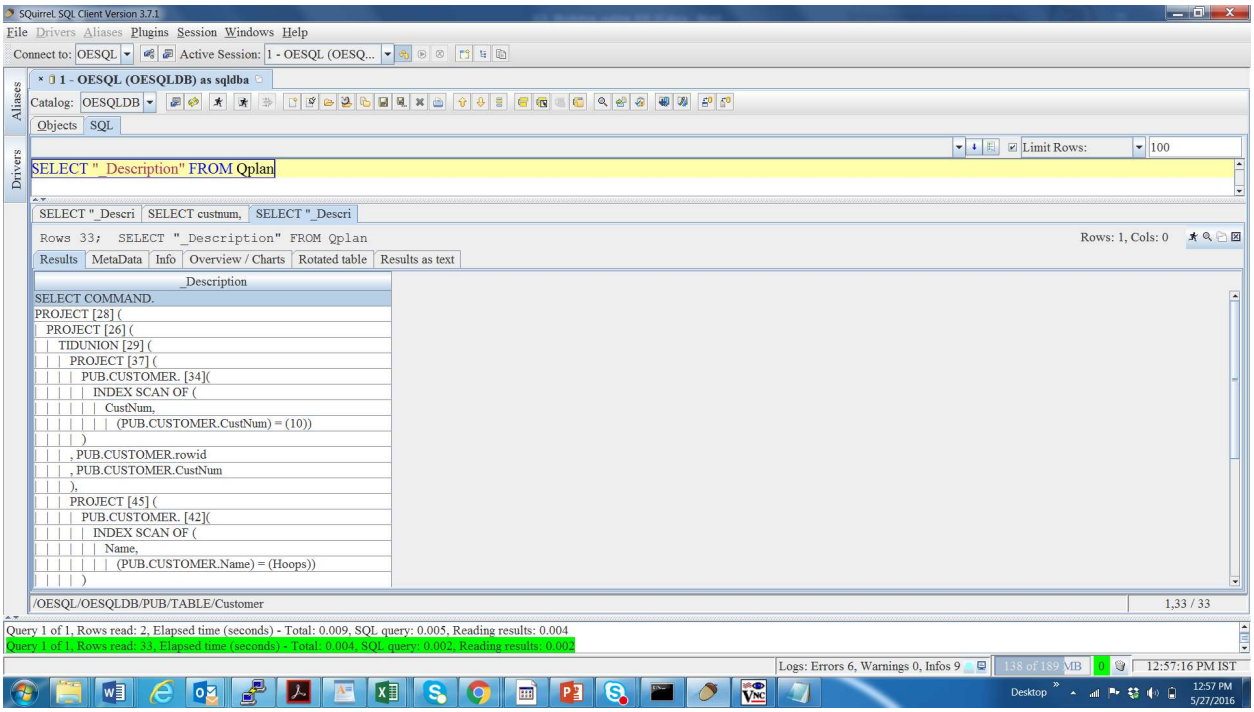
| | , Sort project expression #1 == PEXPR2

Query 6 – Below query demonstrates TIDUNION operation.

SELECT custnum, name, Address, City from pub.customer where custnum=10 OR name = 'Hoops';



Now check the query plan



```
SELECT COMMAND.
PROJECT [28] (
|  PROJECT [26] (
|  |  TIDUNION [29] (
|  |  |  PROJECT [37] (
|  |  |  |  PUB.CUSTOMER. [34](
|  |  |  |  |  INDEX SCAN OF (
|  |  |  |  |  |  CustNum,
|  |  |  |  |  |  |  (PUB.CUSTOMER.CustNum)
|  |  |  |  )
|  |  |  , PUB.CUSTOMER.rowid
|  |  |  , PUB.CUSTOMER.CustNum
|  |  |  ),
|  |  |  PROJECT [45] (
|  |  |  |  PUB.CUSTOMER. [42](
|  |  |  |  |  INDEX SCAN OF (
|  |  |  |  |  |  Name,
|  |  |  |  |  |  |  (PUB.CUSTOMER.Name) =
|  |  |  |  )
|  |  |  , PUB.CUSTOMER.rowid
|  |  |  , PUB.CUSTOMER.Name
|  |  |  )
|  |  )
|  , PUB.CUSTOMER.CustNum
|  , PUB.CUSTOMER.Name
|  , PUB.CUSTOMER.Address
|  , PUB.CUSTOMER.City
|  )
```

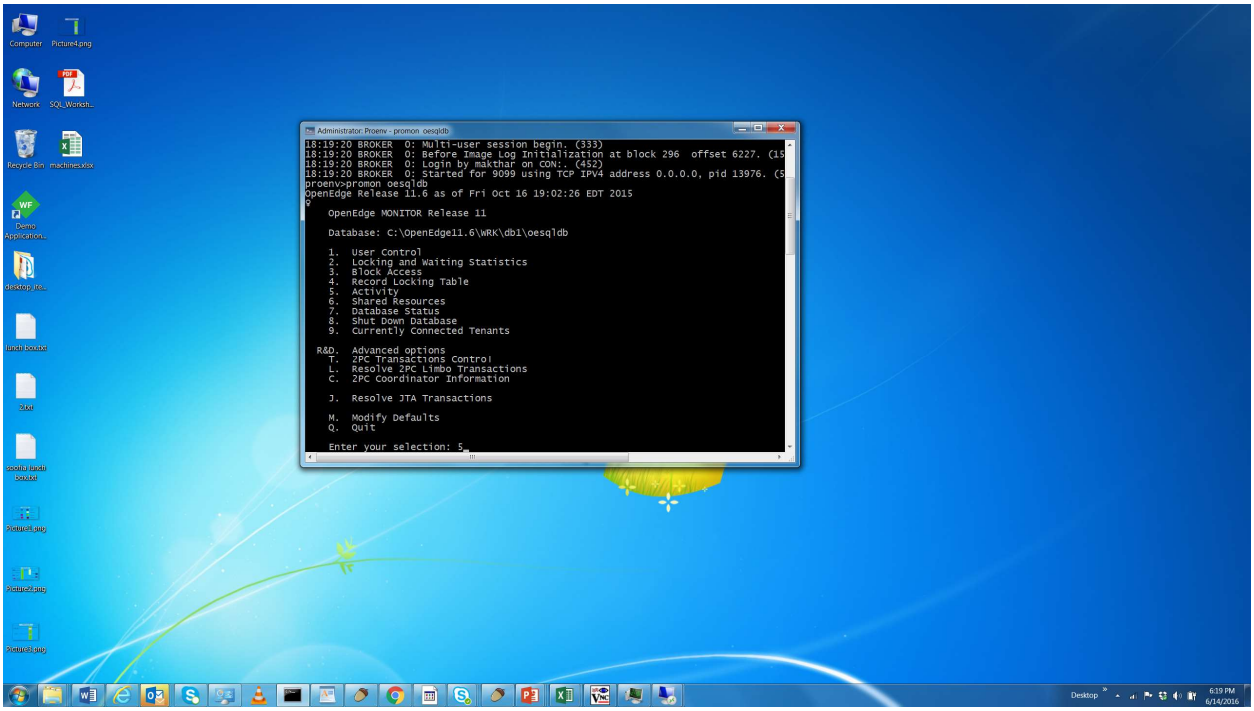
Chapter 4 - Table Scan Vs. Index scan

In this chapter, you will be asked to run a simple query which does table scan and capture the number of IOs using promon utility. Then, you will be asked to create an index and run the same query again and check the number of IOs using promon utility. Then, we compare the number of IOs which took place in table scan versus index scan. In both the cases, we will look at query plan to confirm the query operation.

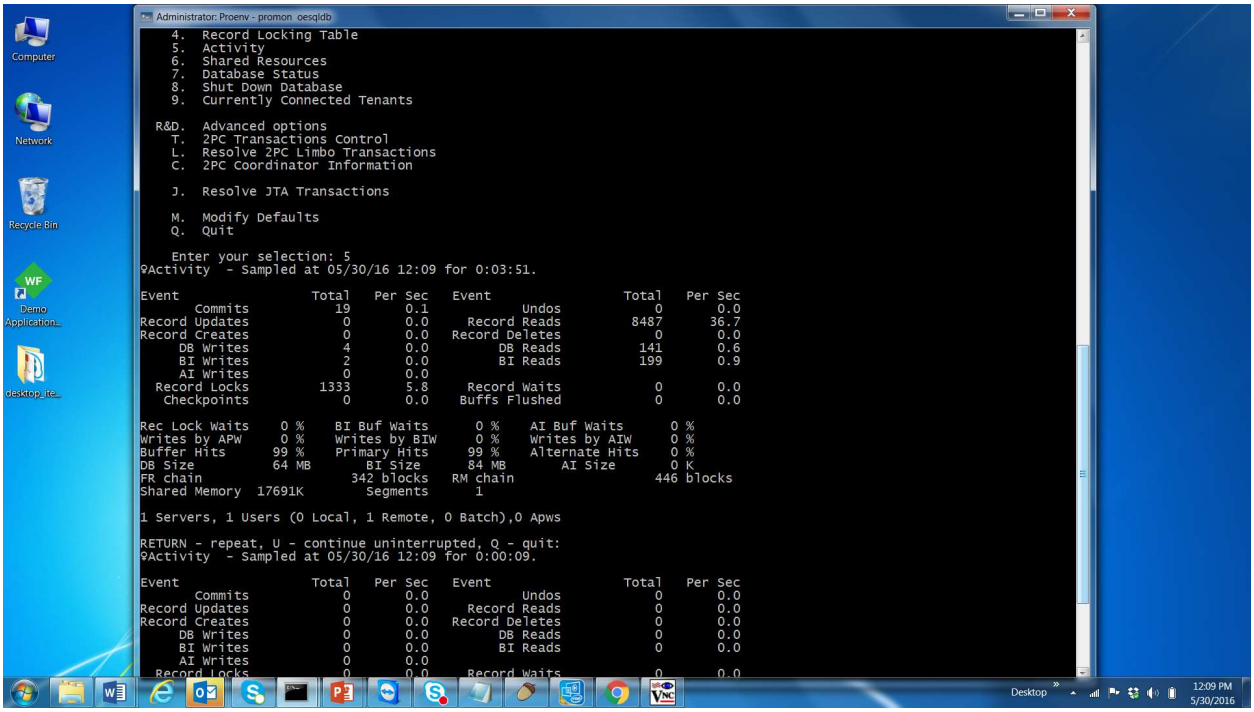
How to start promon?

Go to proenv window and type below command and select option 5. Option 5 will display the number of records read.

promon oesqldb

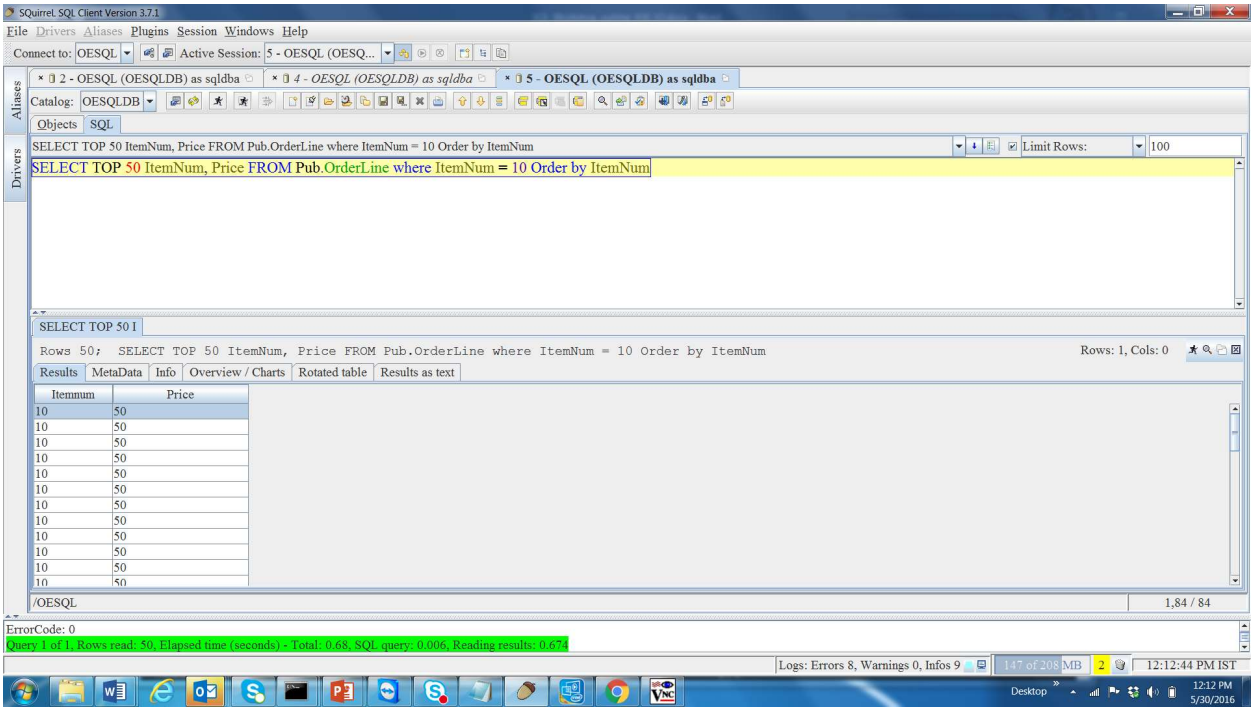


Check the number of DB Requests before running SQL query

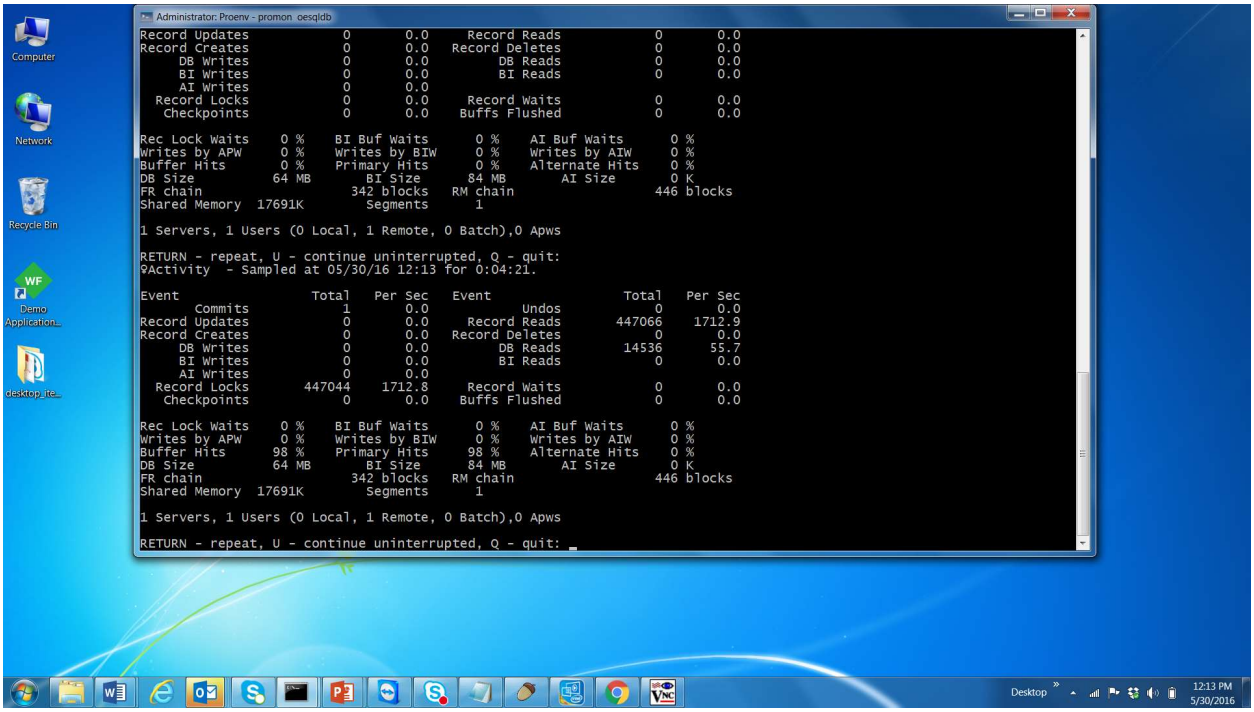


Now, run the below query which results in table scan.

SELECT TOP 50 ItemNum, Price FROM Pub.OrderLine WHERE ItemNum=10 ORDER BY ItemNum;



Now, Check the number of Record read in promon.



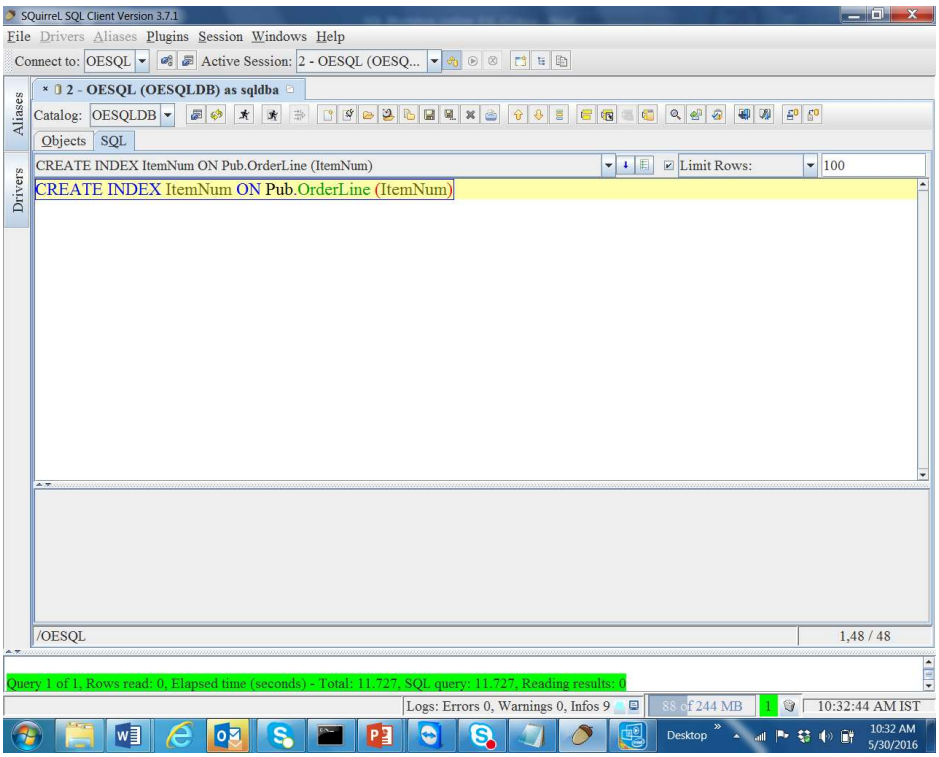
Now check the query plan and confirm that it is doing Table Scan.

```
_Description
----------------------------------------------------
SELECT COMMAND.
PROJECT [25] TOP 50 (
|  SORT [24] IN TMPTBL00000024(
|  |  RESTRICT [6] (
|  |  |  PROJECT [17] (
|  |  |  |  PUB.ORDERLINE. [2](
|  |  |  |  |  TABLE SCAN
|  |  |  |  )
|  |  |  |  , PUB.ORDERLINE.Itemnum
|  |  |  |  , PUB.ORDERLINE.Price
|  |  |  )
|  |
|  |  |  (PEXPR1) = (10)
|  |  |  Evaluation callback list(
|  |  |  |       col id#  4
|  |  |  )
|  |  )
|  | SORT BY (
|  |  , Sort project expression #0 == PEXPR1
|  ),
|  , PEXPR1
```

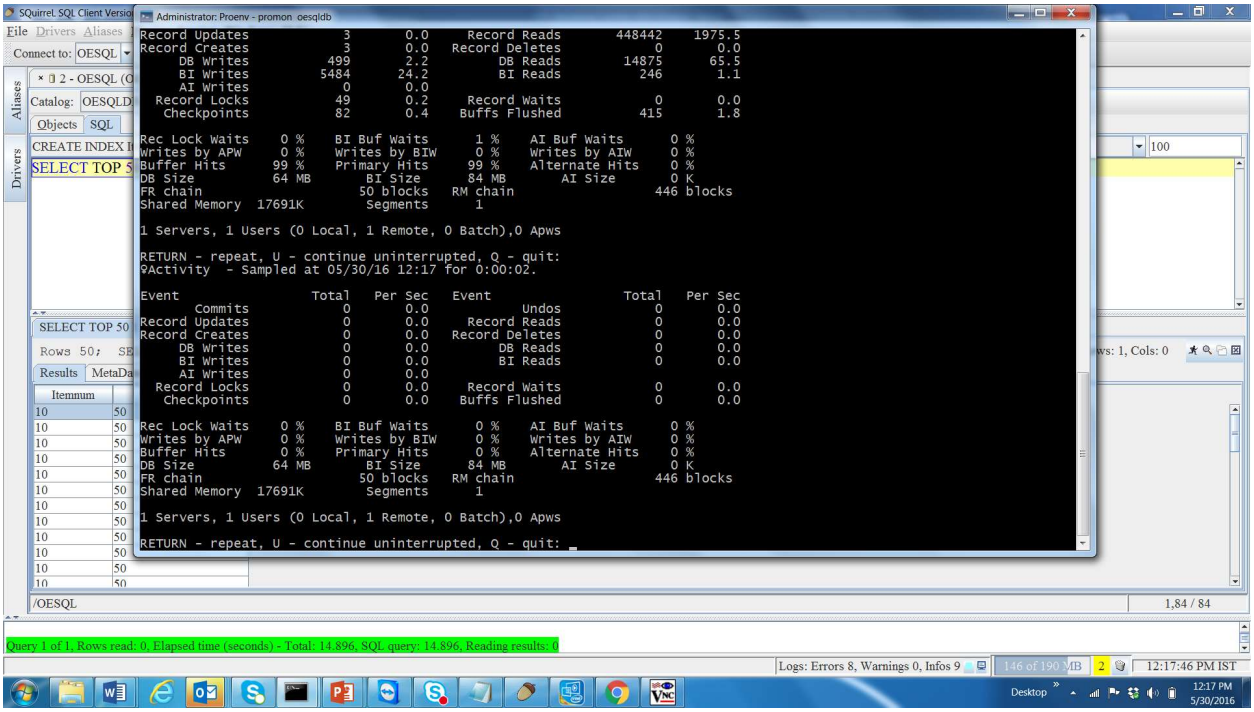
| , PEXPR2
|)
, PEXPR1
, PEXPR2
)

Now, let us create an index on ItemNum column of Pub.OrderLine table.

CREATE INDEX ItemNum ON Pub.OrderLine (ItemNum);

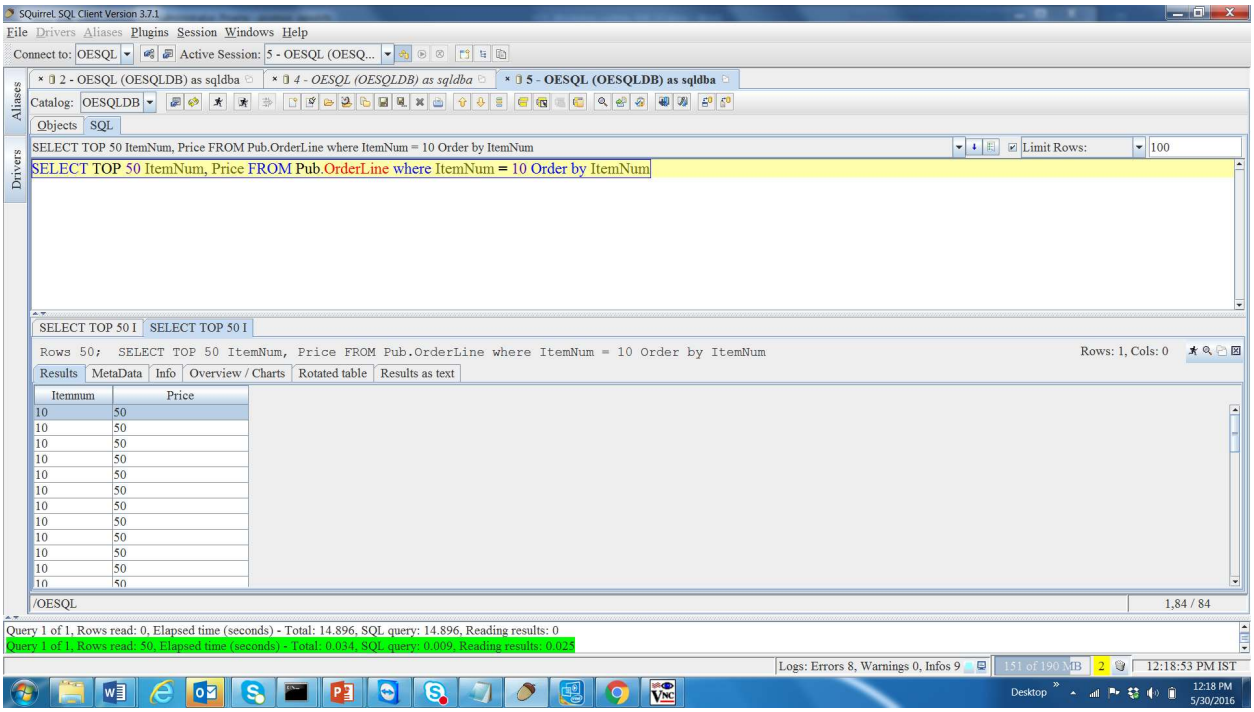


Refresh the promon screen by pressing “Enter”

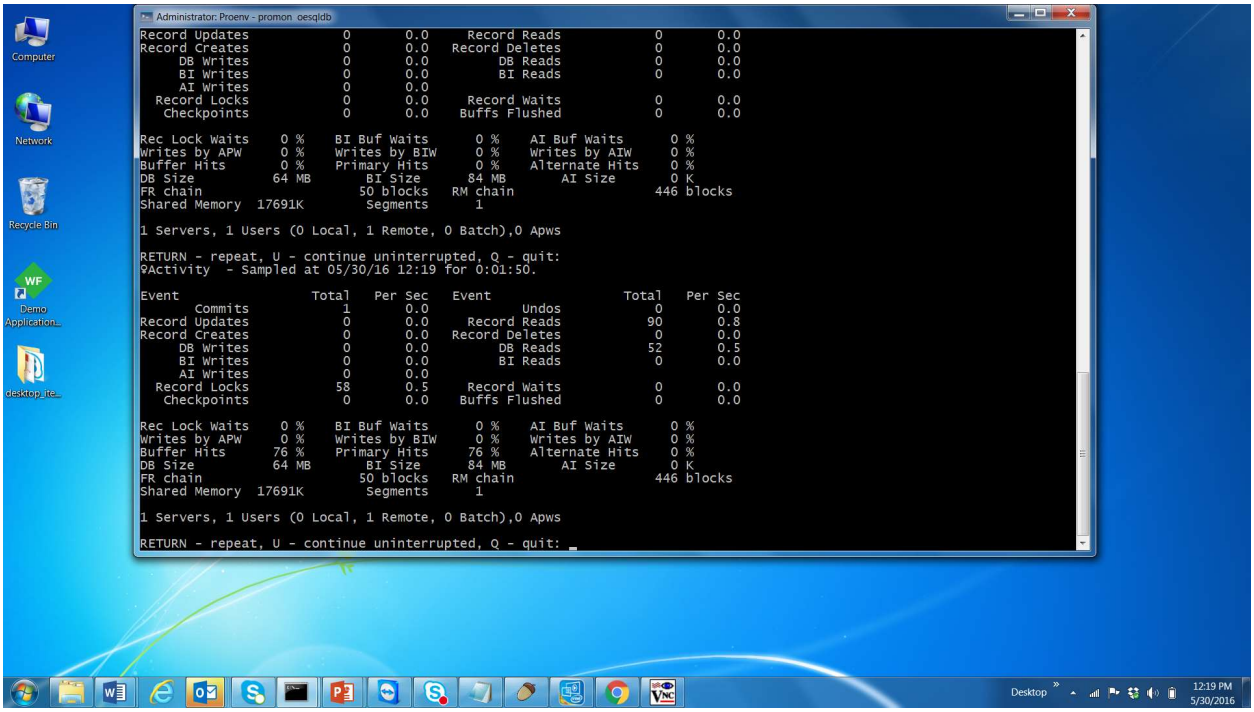


Now, run the same query again and check the number of records read and the time taken.

SELECT TOP 50 ItemNum, Price FROM Pub.OrderLine where ItemNum = 10 Order by ItemNum;



Now, check the number of records read.



Observation – Did you observe the drastic change in the number of records read during Index scan when compared to Table scan? This difference is because, Index scan will have intelligence to read only the required records and skip all other records of a table.

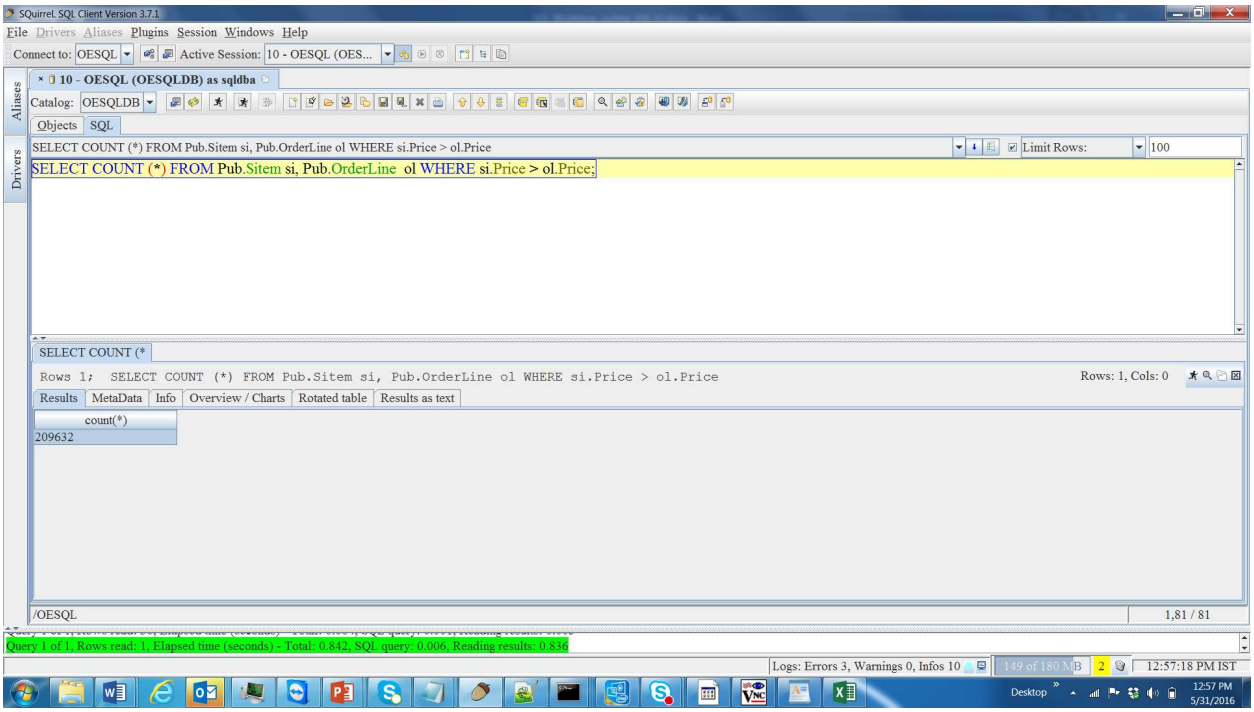
Chapter 5 - Join Methods

In this chapter, we will explain the different join methods used by OE-SQL and you will be learning what are "Good" join methods (Augmented Nested Loops and Merge Joins in OE world) and what is a "Bad" join method (Nested Loop). Basically, "Good" joins are the ones which uses physical index or Dynamic index on the right side of the join to evaluate the join conditions. Here, you will be asked to run a join query which results in "Bad" join and then, create an index and then, asked to run the same query again to see the "Good" join. We will check the join methods through query plans. You will also use promon utility to check the number of IOs in both cases. There can be cases where, SQL optimizer selects "Nested Loop Join" which might be better than other choices.

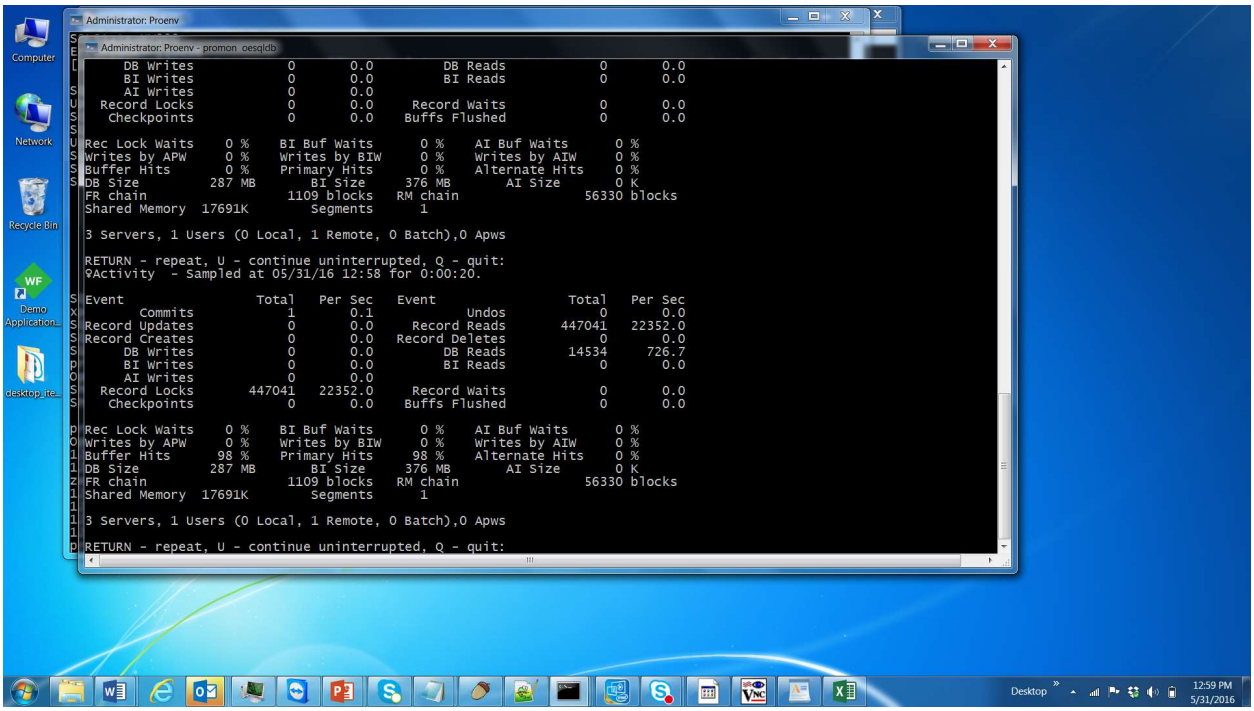
Nested Loop Joins:

Run the Below query which results in Nested Loop Join

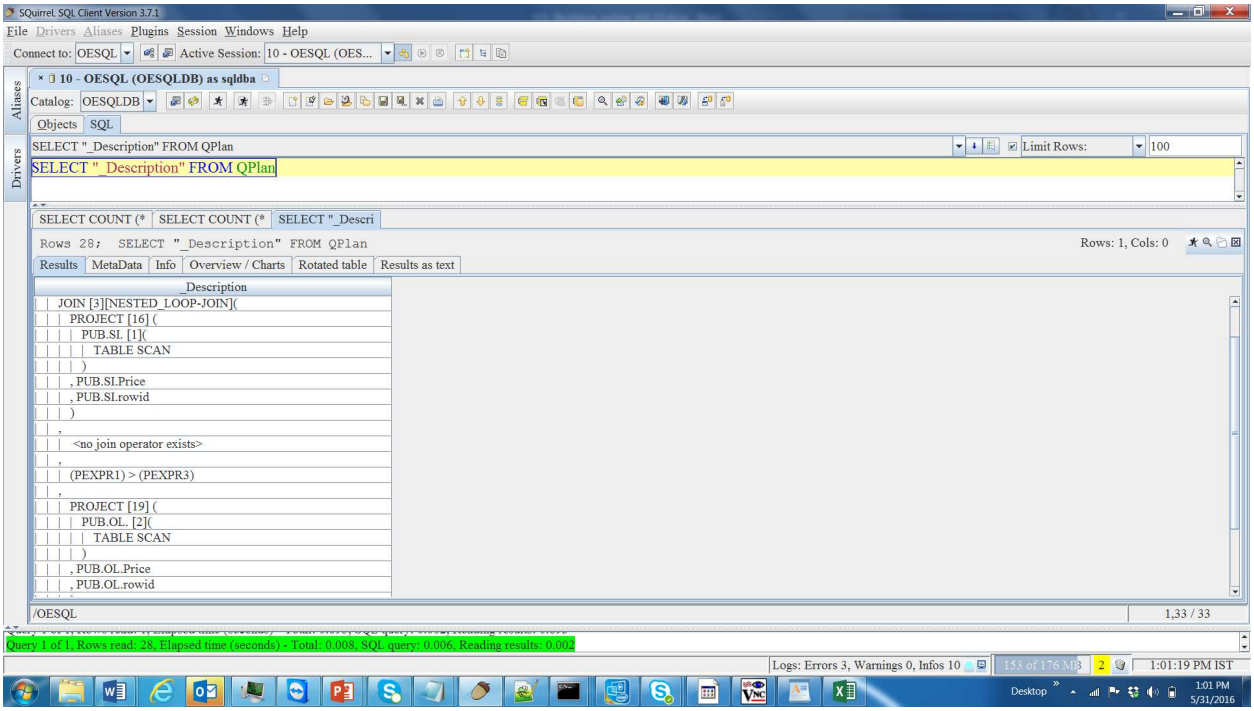
SELECT COUNT (*) FROM Pub.Sitem si, Pub.OrderLine ol WHERE si.Price > ol.Price;



Now check the number of DB Requests.



Now, check the plan



_Description

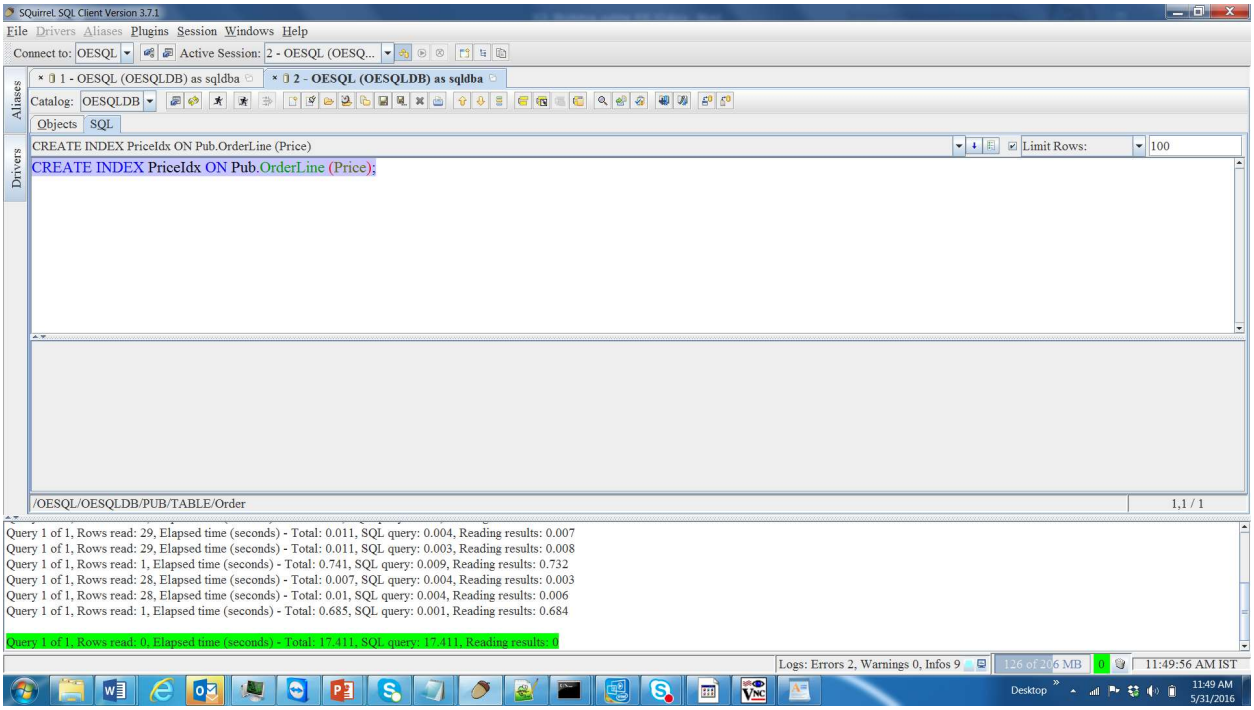
SELECT COMMAND.

```
PROJECT [22] (
|   PROJECT [21] (
|   |   JOIN [3][NESTED_LOOP-JOIN](
|   |   |   PROJECT [16] (
|   |   |   |   PUB.SI. [1](
|   |   |   |   |   TABLE SCAN
|   |   |   |   )
|   |   |   , PUB.SI.Price
|   |   |   , PUB.SI.rowid
|   |   |   )
|   |   ,
|   |   |    <no join operator exists>
|   |   ,
|   |   |   (PEXPR1) > (PEXPR3)
|   |   ,
|   |   |   PROJECT [19] (
|   |   |   |   PUB.OL. [2](
|   |   |   |   |   TABLE SCAN
|   |   |   |   )
|   |   |   , PUB.OL.Price
|   |   |   , PUB.OL.rowid
|   |   |   )
|   |   )
|   , count (*)
|   )
, PEXPR1
```

Augmented Nested Loop Joins:

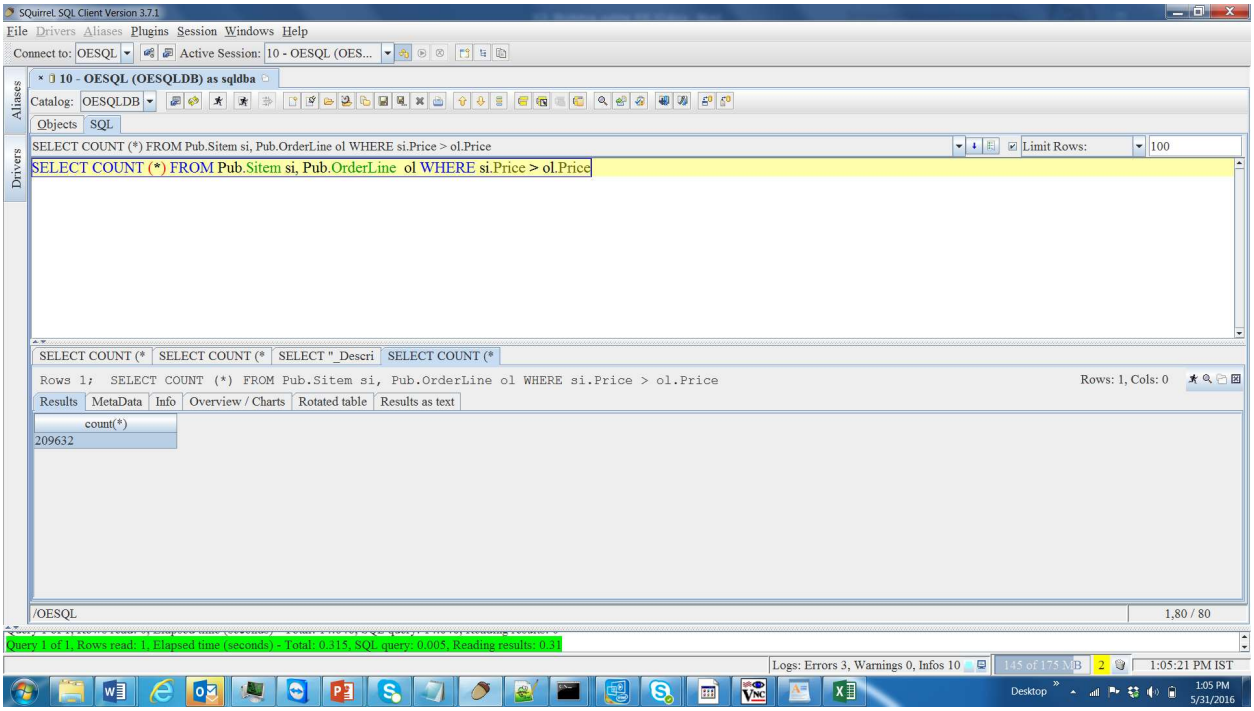
Now, in order to get ANL join, create an index on Pub.OrderLine table

CREATE INDEX PriceIdx ON Pub.OrderLine (Price);

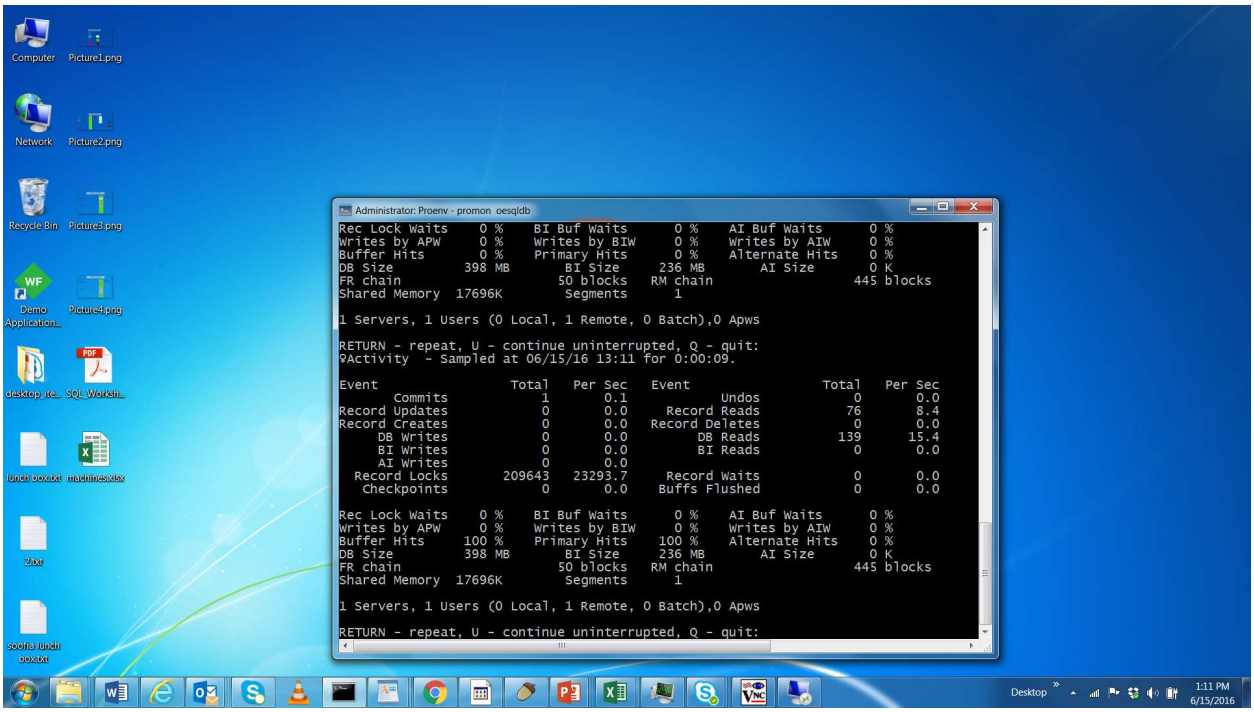


Now, run the same query again. This time, it takes ANL.

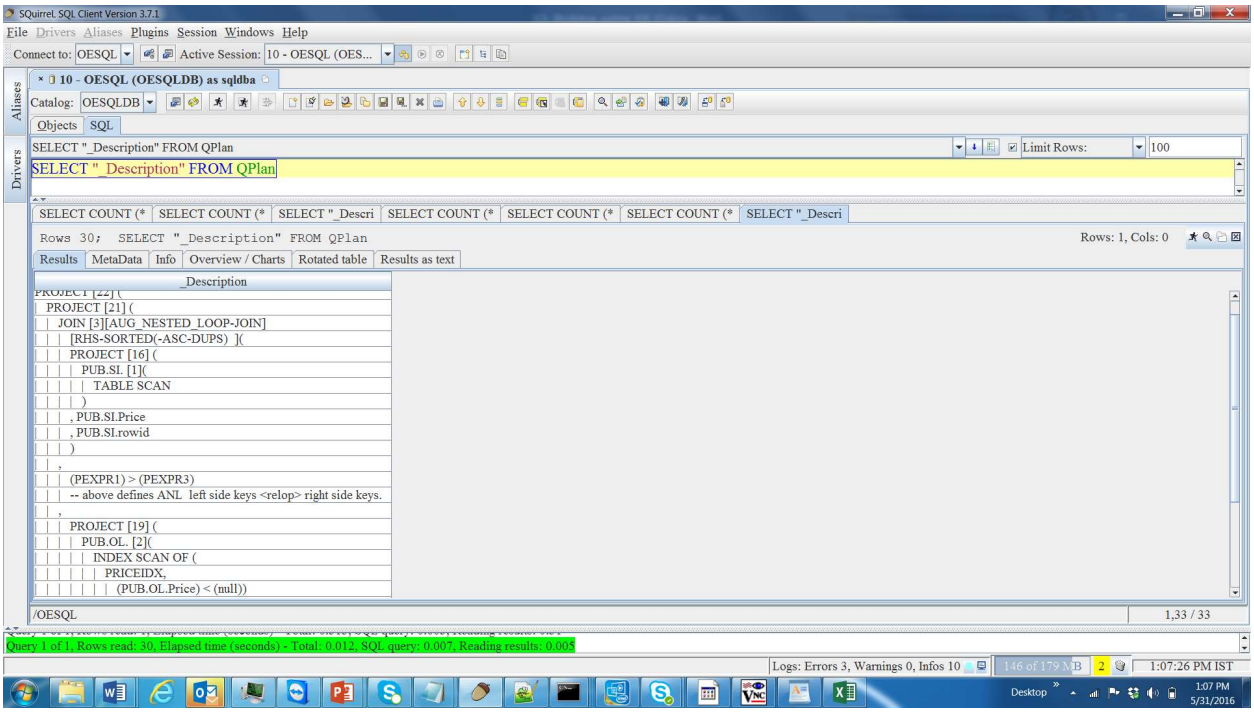
SELECT COUNT (*) FROM Pub.Sitem si, Pub.OrderLine ol WHERE si.Price > ol.Price;



Check the number of DB requests



Check the query plan



```
_Description
-----------------------------------------------------
SELECT COMMAND.
PROJECT [22] (
|   PROJECT [21] (
|   |   JOIN [3][AUG_NESTED_LOOP-JOIN]
|   |   |   [RHS-SORTED(-ASC-DUPS)  ](
|   |   |   PROJECT [16] (
|   |   |   |   PUB.SI. [1](
|   |   |   |   |   TABLE SCAN
|   |   |   |   )
|   |   |   , PUB.SI.Price
|   |   |   , PUB.SI.rowid
|   |   |   )
|   |   ,
|   |   |   (PEXPR1) > (PEXPR3)
|   |   |   -- above defines ANL  left side keys <
|   |   ,
|   |   |   PROJECT [19] (
|   |   |   |   PUB.OL. [2](
|   |   |   |   |   INDEX SCAN OF (
|   |   |   |   |   |   PRICEIDX,
|   |   |   |   |   |   |   (PUB.OL.Price) < (null
|   |   |   |   )
|   |   |   , PUB.OL.Price
|   |   |   , PUB.OL.rowid
|   |   |   )
|   |   )
|   , count (*)
|   )
, PEXPR1
)
```

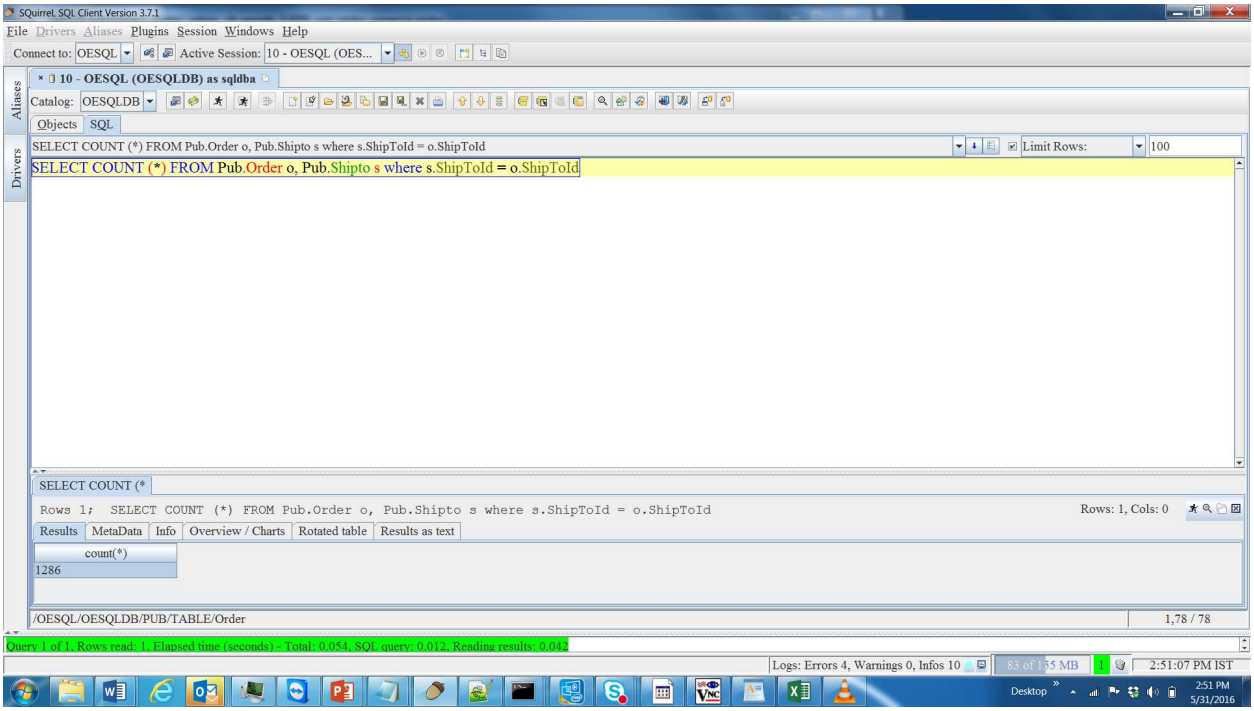
Observation: Did you observe the drastic difference in number of records read? It is because, right side of the join in ANL has the index scan, which reads only the required records from that table.

Augmented Nested Loop joins with Dynamic Index:

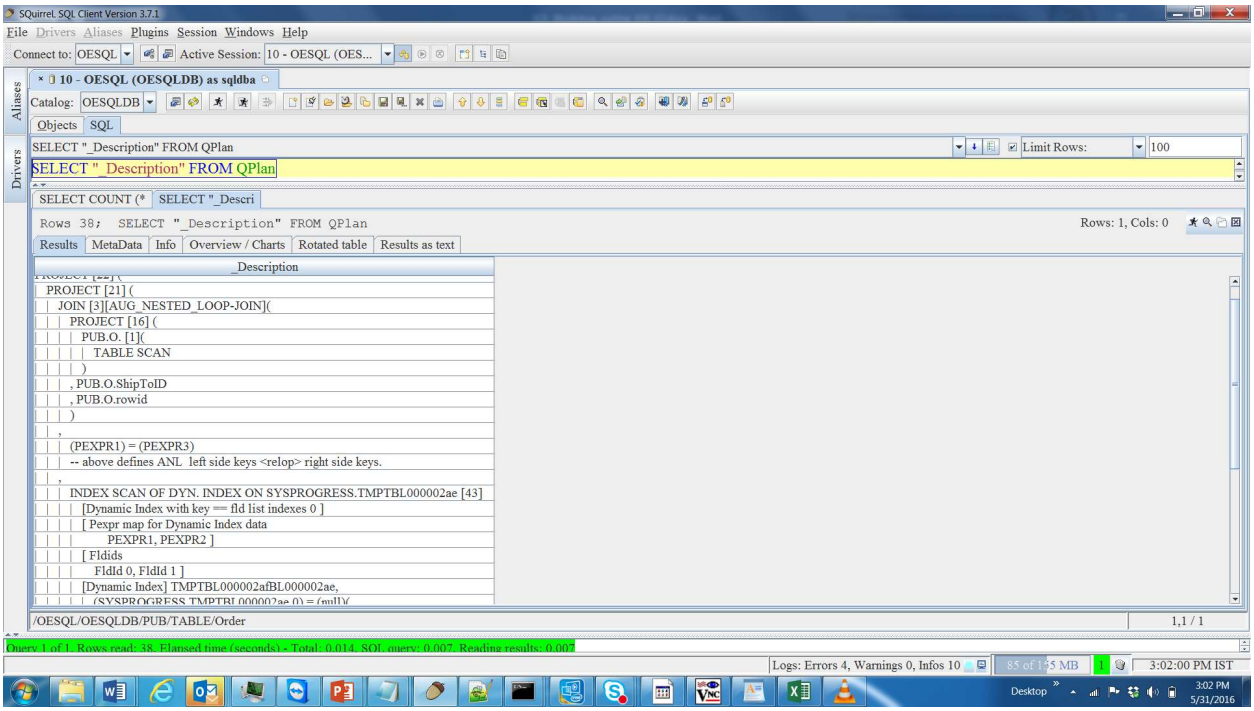
When there is no index available for join, then SQL creates Dynamic Index during the runtime on the Right Side Table of Join. This join is called as ANL with DIDX (DIDX stands for Dynamic Index). This is also called as Hash Join.

Run the below query (Note that, there is no index available on ShipToId column of Pub.Order or Pub.ShipTo).

SELECT COUNT (*) FROM Pub.Order o, Pub.Shipto s where s.ShipToId = o.ShipToId



Check the query plan, it should generate Dynamic Index.




```
_Description
-----------------------------------------------------
SELECT COMMAND.
PROJECT [22] (
|   PROJECT [21] (
|   |   JOIN [3][AUG_NESTED_LOOP-JOIN](
|   |   |   PROJECT [16] (
|   |   |   |   PUB.O. [1](
|   |   |   |   |   TABLE SCAN
|   |   |   |   )
|   |   |   , PUB.O.ShipToID
|   |   |   , PUB.O.rowid
|   |   |   )
|   |   ,
|   |   |   (PEXPR1) = (PEXPR3)
|   |   |   -- above defines ANL  left side keys <
|   |   ,
|   |   |   INDEX SCAN OF DYN. INDEX ON SYSPROGRES
|   |   |   |   [Dynamic Index with key == fld lis
|   |   |   |   [ Pexpr map for Dynamic Index data
```

```
|   |   |   |         PEXPR1, PEXPR2 ]
|   |   |   |   [ Fldids
|   |   |   |      FldId 0, FldId 1 ]
|   |   |   |   [Dynamic Index] TMPTBL00000011BL00
|   |   |   |   |   (SYSPROGRESS.TMPTBL00000010.0)
|   |   |   |   PROJECT [19] (
|   |   |   |   |   PUB.S. [2](
|   |   |   |   |   |   INDEX SCAN OF (
|   |   |   |   |   |   |   custnumshipto,     - F
|   |   |   |   |   )
|   |   |   |   , PUB.S.ShipToID
|   |   |   |   , PUB.S.rowid
|   |   |   |   )
|   |   |
|   |   |   )
|   |   )
|   , count (*)
|   )
, PEXPR1
)
```

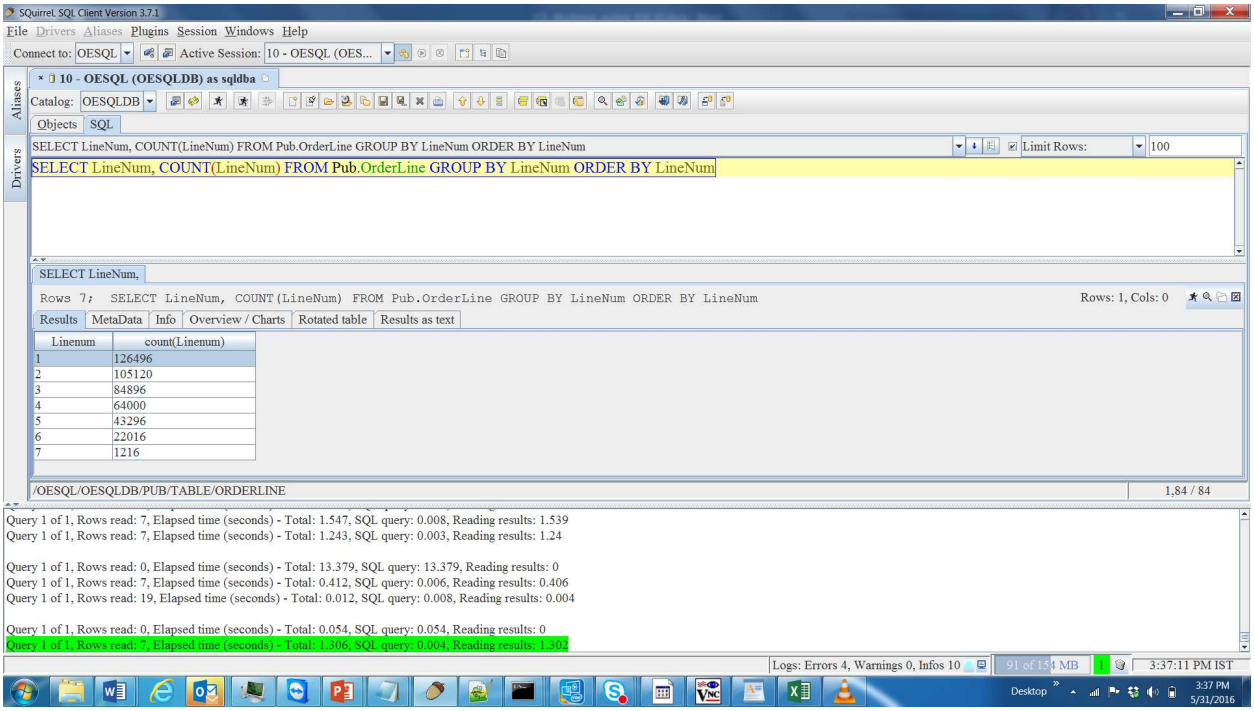
Chapter 6 - Aggregations

In this chapter, we will explain the different aggregation methods used by OE-SQL and then, you will be asked to run a query which results in “Hash Aggregation”. Then, we will create an index and run the same query which now results in “Stream Aggregation”. Basically, “Hash aggregation” uses “Hash Table” to group the same column values, whereas, “Stream Aggregation” uses the “Index scan” operation on the grouping columns and thus does not require any “Hash Table” to group the same column values. We will compare the difference between these two aggregation methods in terms of execution time and the reason behind this difference. All this will be done by analyzing the query plan for both the cases.

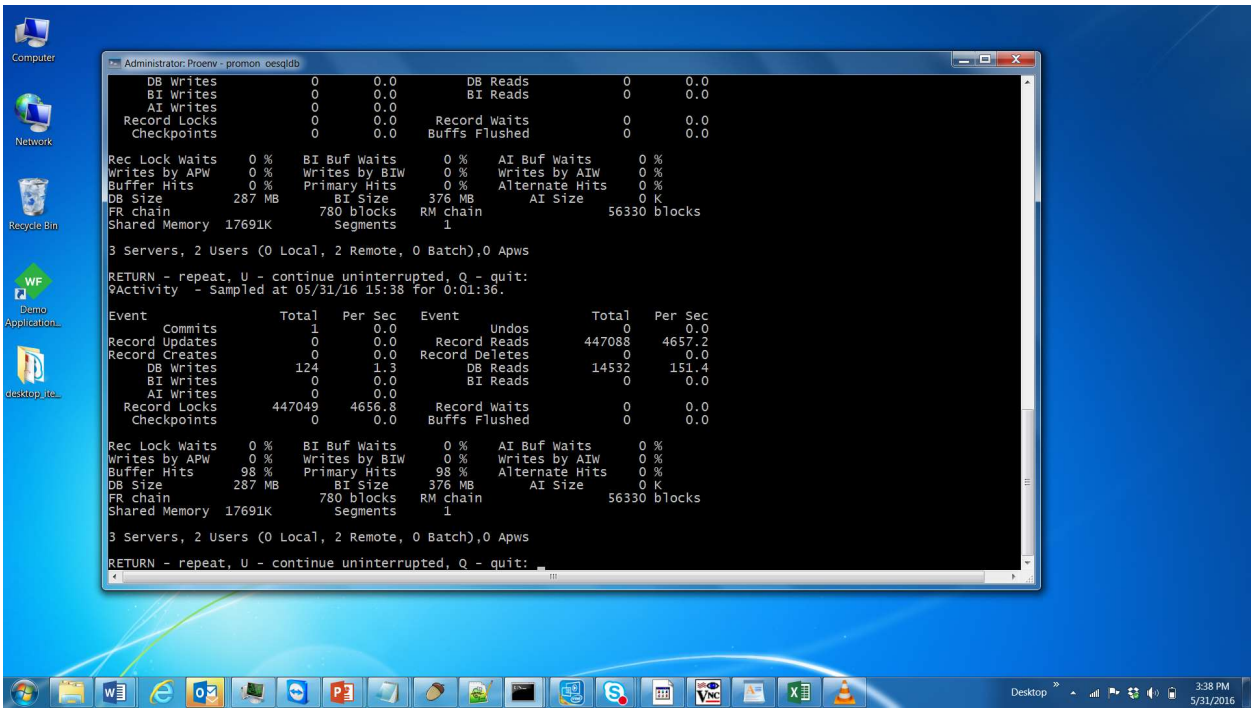
Hash Aggregation:

Run the below query which results in Hash Aggregation.

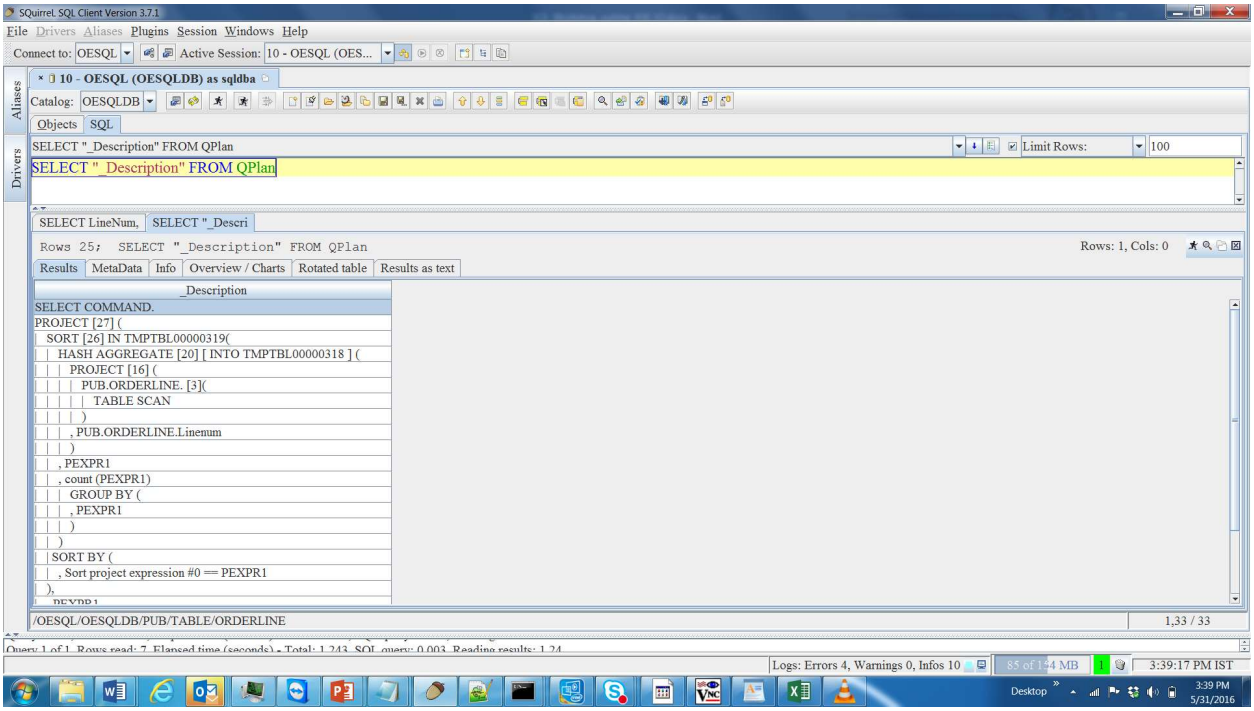
SELECT LineNum, COUNT(LineNum) FROM Pub.OrderLine GROUP BY LineNum ORDER BY LineNum ;



Check number of Records Read



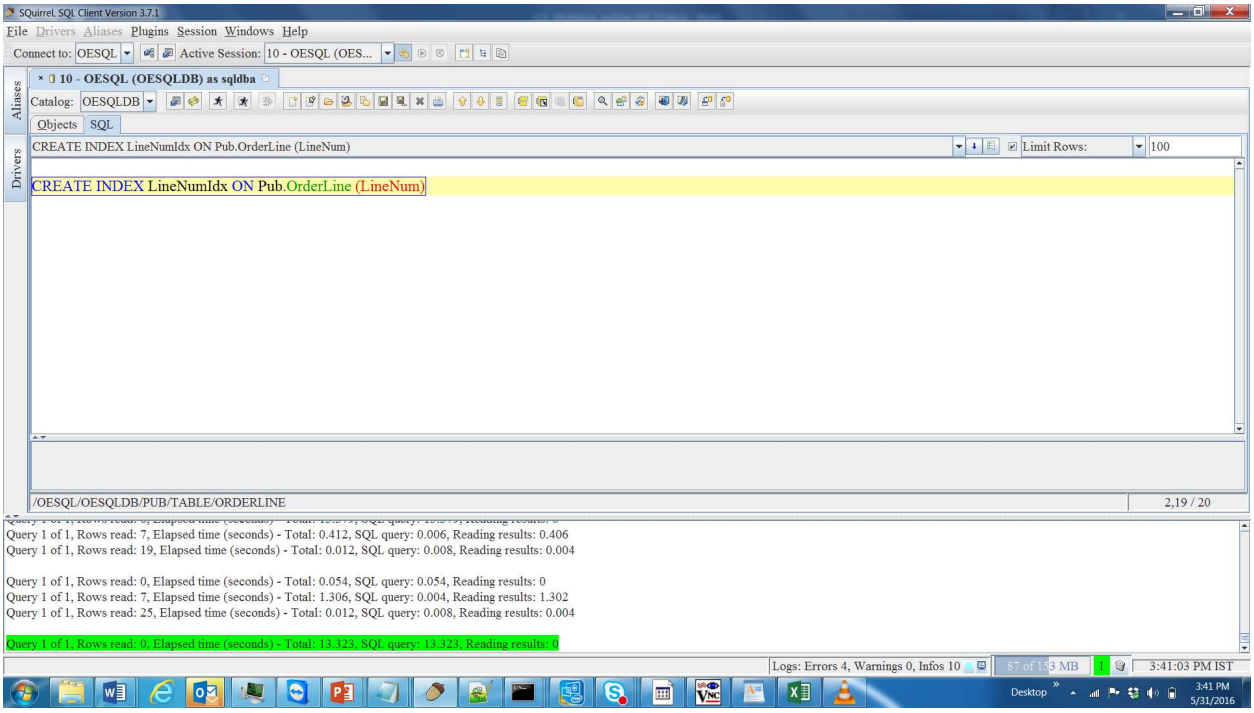
Now, check the query plan.



```
_Description
-----------------------------------------------------
SELECT COMMAND.
PROJECT [27] (
|   SORT [26] IN TMPTBL00000319(
|   |   HASH AGGREGATE [20] [ INTO TMPTBL00000318
|   |   |   PROJECT [16] (
|   |   |   |   PUB.ORDERLINE. [3](
|   |   |   |   |   TABLE SCAN
|   |   |   |   )
|   |   |   , PUB.ORDERLINE.Linenum
|   |   |   )
|   |   , PEXPR1
|   |   , count (PEXPR1)
|   |   |   GROUP BY (
|   |   |   , PEXPR1
|   |   |   )
|   |   )
|   | SORT BY (
|   |   , Sort project expression #0 == PEXPR1
|   ),
|   , PEXPR1
|   , PEXPR2
|   )
, PEXPR1
, PEXPR2
)
```

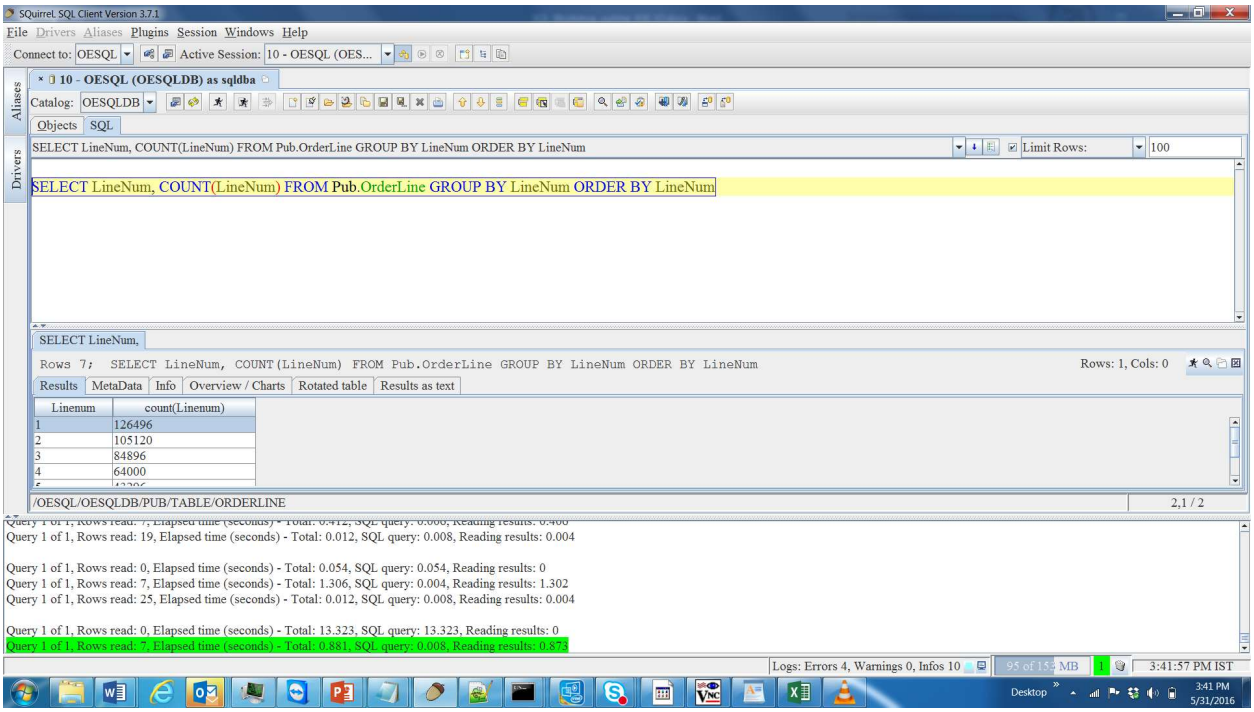
Now, create an index on LineNum column of table Pub.OrderLine.

```
CREATE INDEX  LineNumIdx ON Pub.OrderLine(LineNum);
```

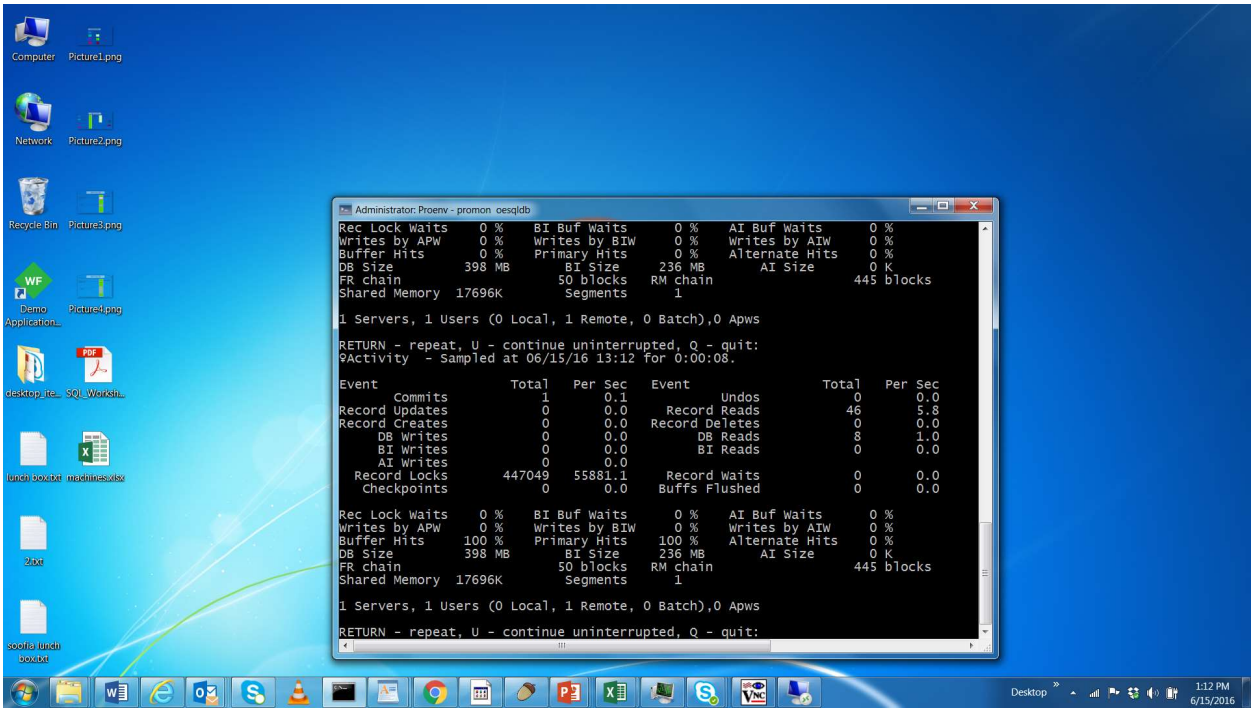


Now, run the same query again.

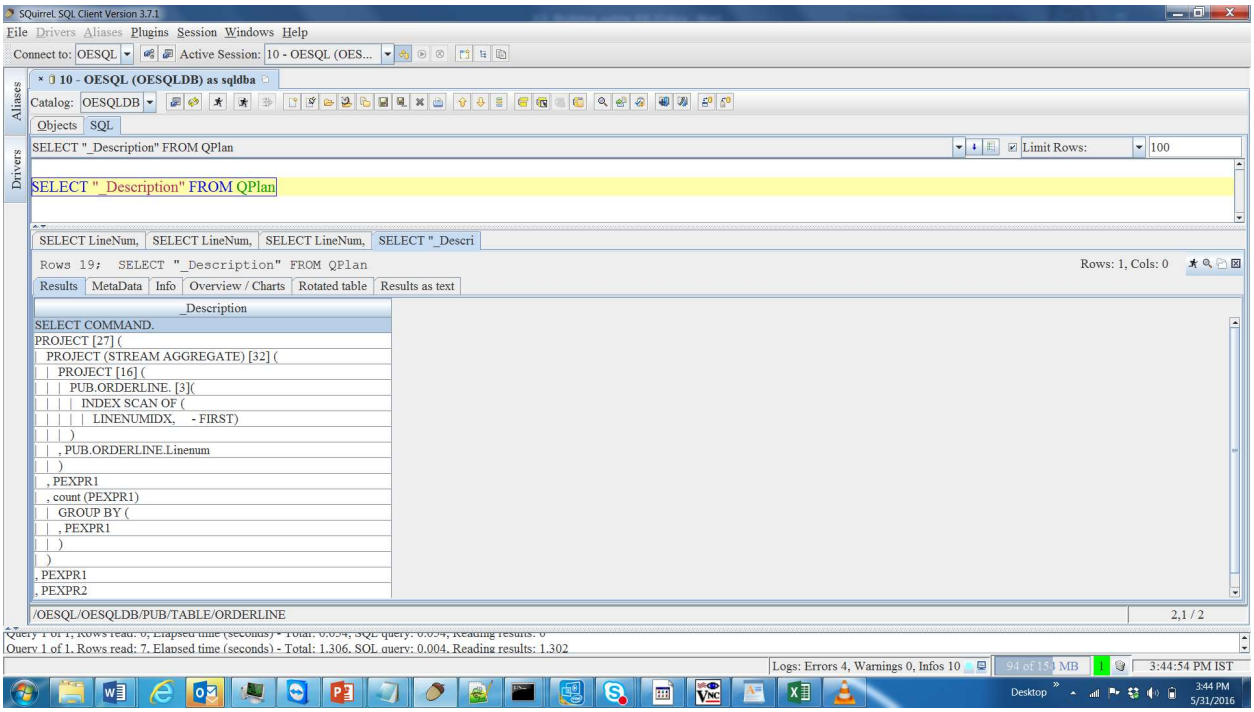
SELECT LineNum, COUNT(LineNum) FROM Pub.OrderLine GROUP BY LineNum ORDER BY LineNum ;



Now, check the number of records read.



Now, check the query plan



```
_Description
-----------------------------------------------------
SELECT COMMAND.
PROJECT [27] (
|   PROJECT (STREAM AGGREGATE) [32] (
|   |   PROJECT [16] (
|   |   |   PUB.ORDERLINE. [3](
|   |   |   |   INDEX SCAN OF (
|   |   |   |   |   LINENUMIDX,    - FIRST)
|   |   |   )
|   |   , PUB.ORDERLINE.Linenum
|   |   )
|   , PEXPR1
|   , count (PEXPR1)
|   |   GROUP BY (
|   |   , PEXPR1
|   |   )
|   )
, PEXPR1
, PEXPR2
)
```

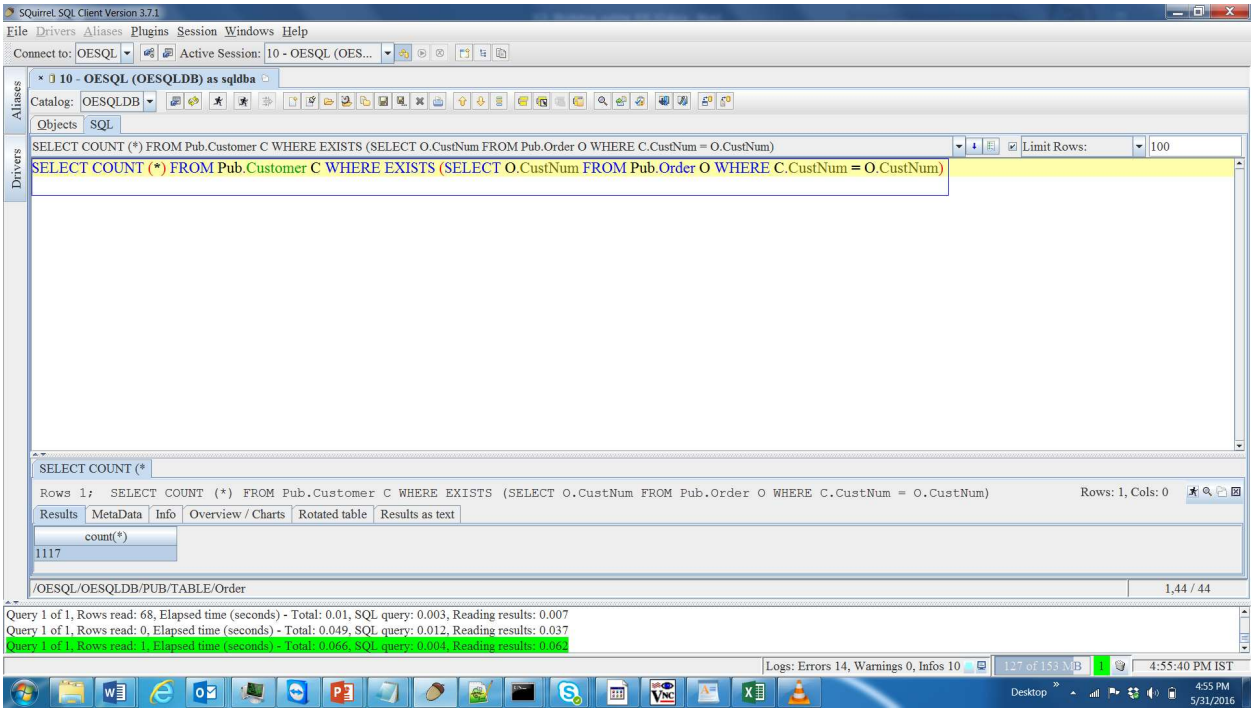
Observation: Did you notice the drastic difference in number of records read? In fact, in case of Stream Aggregation, it did not even read a single record. It is because, all the details which were required by query were already present in Index itself.

Chapter 7 - What happened to my subquery?

In this chapter you will learn how subqueries are transformed into other basic query operations by running below two queries containing subqueries and analyzing query plans generated. Knowing query transformations is important in troubleshooting the performance problems.

Let us start with simple co-related subquery

SELECT COUNT (*) FROM Pub.Customer C WHERE EXISTS (SELECT O.CustNum FROM Pub.Order O WHERE C.CustNum = O.CustNum);

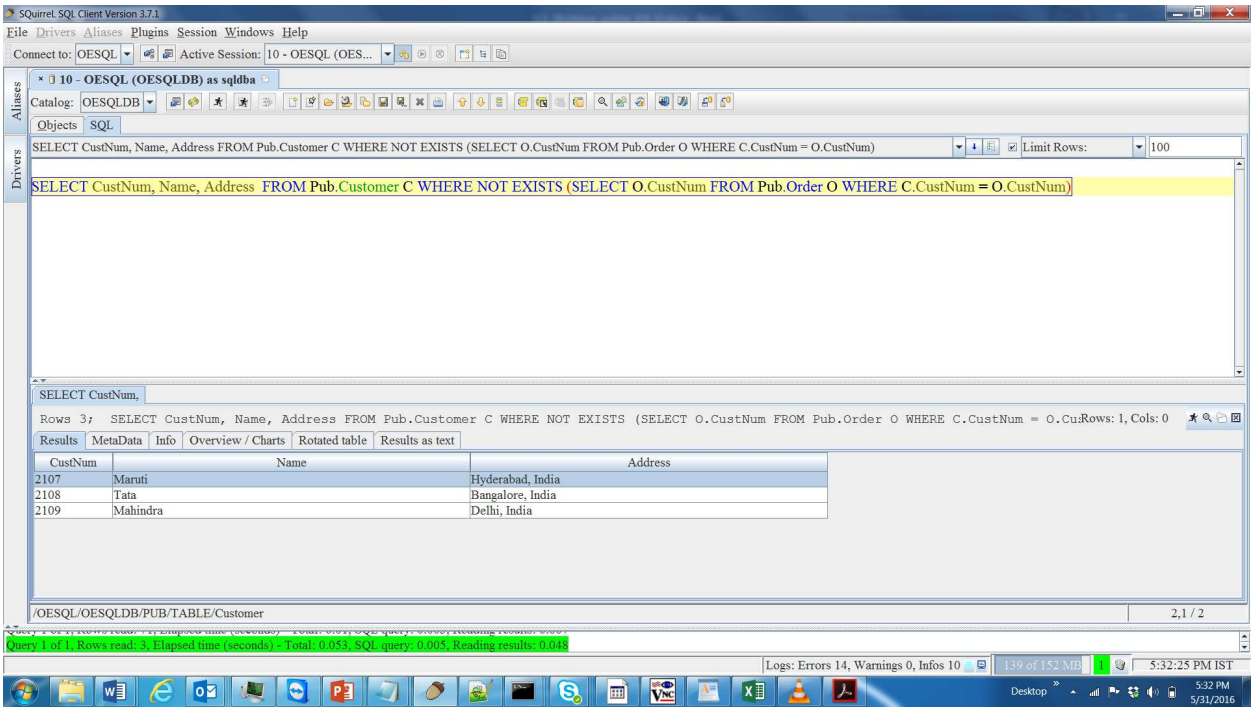


SELECT COMMAND.
PROJECT [55] (
| PROJECT [54] (
| | PROJECT [53] distinct (
| | | JOIN [48][AUG_NESTED_LOOP-JOIN]
| | | | [RHS-SORTED(-ASC-DUPS)](
| | | | RESTRICT [10] (
| | | | | PROJECT [40] (
| | | | | | HASH AGGREGATE [36] [INTO TMPTBL00000387] (
| | | | | | | JOIN [31][AUG_NESTED_LOOP-JOIN]
| | | | | | | | [RHS-SORTED(-ASC-DUPS)](
| | | | | | | | PROJECT [26] (
| | | | | | | | | PUB.O. [3](
| | | | | | | | | | INDEX SCAN OF (
| | | | | | | | | | | CustOrder, - FIRST)
| | | | | | | | |)

```
|  |  |  |  |  |  |  |  , PUB.O.CustNum
|  |  |  |  |  |  |  |  , PUB.O.rowid
|  |  |  |  |  |  |  |  )
|  |  |  |  |  |  |  ,
|  |  |  |  |  |  |  |  (PEXPR1) = (PEXPR3)
|  |  |  |  |  |  |  |  -- above defines ANL  left side keys <relop> right side keys.
|  |  |  |  |  |  |  ,
|  |  |  |  |  |  |  |  PROJECT [30] distinct (
|  |  |  |  |  |  |  |  |  PROJECT [22] (
|  |  |  |  |  |  |  |  |  |  PUB.C. [23](
|  |  |  |  |  |  |  |  |  |  |  INDEX SCAN OF (
|  |  |  |  |  |  |  |  |  |  |  |  CustNum,
|  |  |  |  |  |  |  |  |  |  |  |  (PUB.C.CustNum) = (null))
|  |  |  |  |  |  |  |  |  |  )
|  |  |  |  |  |  |  |  |  , PUB.C.CustNum
|  |  |  |  |  |  |  |  |  )
|  |  |  |  |  |  |  |  , PEXPR1
|  |  |  |  |  |  |  |  )
|  |  |  |  |  |  |  )
|  |  |  |  |  |  , pro_nonempty (*)
|  |  |  |  |  |  , PEXPR3
|  |  |  |  |  |  |  GROUP BY (
|  |  |  |  |  |  |  , PEXPR3
|  |  |  |  |  |  |  )
|  |  |  |  |  |  )
|  |  |  |  |  , PEXPR1
|  |  |  |  |  , PEXPR2
|  |  |  |  |  )
|  |  |  |  |
|  |  |  |  |  (0) < (PEXPR1)
|  |  |  |  )
|  |  |  ,
|  |  |  |  (PEXPR2) = (PEXPR3)
|  |  |  |  -- above defines ANL  left side keys <relop> right side keys.
|  |  |  ,
|  |  |  |  PROJECT [46] (
|  |  |  |  |  PUB.C. [1](
|  |  |  |  |  |  INDEX SCAN OF (
|  |  |  |  |  |  |  CustNum,
|  |  |  |  |  |  |  (PUB.C.CustNum) = (null))
|  |  |  |  |  )
|  |  |  |  , PUB.C.CustNum
|  |  |  |  , PUB.C.rowid
|  |  |  |  )
|  |  |  )
|  |  , PEXPR3
|  |  , PEXPR4
|  |  )
|  , count (*)
|  )
, PEXPR1
)
```

Let us look at second subquery transformation

SELECT CustNum, Name, Address FROM Pub.Customer C WHERE NOT EXISTS (SELECT O.CustNum FROM Pub.Order O WHERE C.CustNum = O.CustNum) ;



SELECT COMMAND.
PROJECT [63] (
| PROJECT [62] distinct (
| | JOIN [55][AUG_NESTED_LOOP-JOIN]
| | | [RHS-SORTED(-ASC-DUPS)](
| | | RESTRICT [12] (
| | | | PROJECT [43] (
| | | | | PROJECT (STREAM AGGREGATE) [251] (
| | | | | | OUTER JOIN [33][AUG_NESTED_LOOP-JOIN][OJ SQL Gen]
| | | | | | | [RHS-SORTED(-ASC-DUPS)](
| | | | | | | PROJECT [32] distinct (
| | | | | | | | PROJECT [24] (
| | | | | | | | | PUB.C. [25](
| | | | | | | | | | INDEX SCAN OF (
| | | | | | | | | | | CustNum, - FIRST)

```
|  |  |  |  |  |  |  |  |  )
|  |  |  |  |  |  |  |  |  , PUB.C.CustNum
|  |  |  |  |  |  |  |  |  )
|  |  |  |  |  |  |  |  , PEXPR1
|  |  |  |  |  |  |  |  )
|  |  |  |  |  |  |  ,
|  |  |  |  |  |  |  |  (PEXPR1) = (PEXPR2)
|  |  |  |  |  |  |  |  -- above defines ANL  left side keys <relop> right side keys.
|  |  |  |  |  |  |  ,
|  |  |  |  |  |  |  |  PROJECT [28] (
|  |  |  |  |  |  |  |  |  PUB.O. [5](
|  |  |  |  |  |  |  |  |  |  INDEX SCAN OF (
|  |  |  |  |  |  |  |  |  |  |  CustOrder,
|  |  |  |  |  |  |  |  |  |  |  |  (PUB.O.CustNum) = (null))
|  |  |  |  |  |  |  |  |  )
|  |  |  |  |  |  |  |  , PUB.O.CustNum
|  |  |  |  |  |  |  |  , PUB.O.rowid
|  |  |  |  |  |  |  |  )
|  |  |  |  |  |  |  )
|  |  |  |  |  |  , count (PEXPR3)
|  |  |  |  |  |  , PEXPR1
|  |  |  |  |  |  |  GROUP BY (
|  |  |  |  |  |  |  , PEXPR1
|  |  |  |  |  |  |  )
|  |  |  |  |  |  )
|  |  |  |  |  , PEXPR1
|  |  |  |  |  , PEXPR2
|  |  |  |  |  )
|  |  |  |
|  |  |  |  (0) = (PEXPR1) [Null-Null EQ]
|  |  |  |  )
|  |  |  ,
|  |  |  |  (PEXPR2) = (PEXPR3)
|  |  |  |  -- above defines ANL  left side keys <relop> right side keys.
|  |  |  ,
|  |  |  |  PROJECT [53] (
|  |  |  |  |  PUB.C. [3](
|  |  |  |  |  |  INDEX SCAN OF (
|  |  |  |  |  |  |  CustNum,
|  |  |  |  |  |  |  |  (PUB.C.CustNum) = (null) [Null-Null EQ])
|  |  |  |  |  )
|  |  |  |  , PUB.C.CustNum
|  |  |  |  , PUB.C.Name
|  |  |  |  , PUB.C.Address
|  |  |  |  , PUB.C.rowid
|  |  |  |  )
|  |  |  )
|  , PEXPR3
|  , PEXPR4
|  , PEXPR5
|  , PEXPR6
|  )
, PEXPR1
, PEXPR2
, PEXPR3
```

)

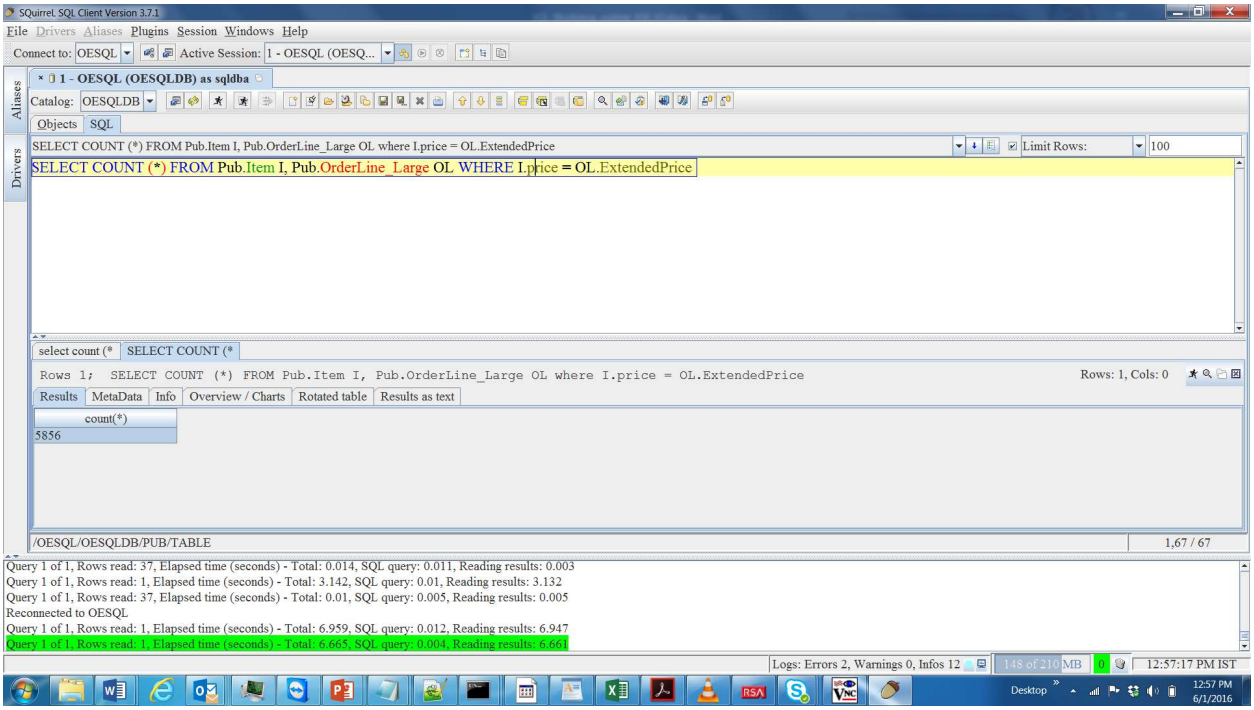
Chapter 8 - Importance of Statistics

In this chapter, you will learn the different types of Statistics available in OE-SQL and their importance in generating good query plans to have better performance. Basically, statistics provides insights on table, index and column data which will be used by OE-SQL optimizer to come up with optimal query plans.

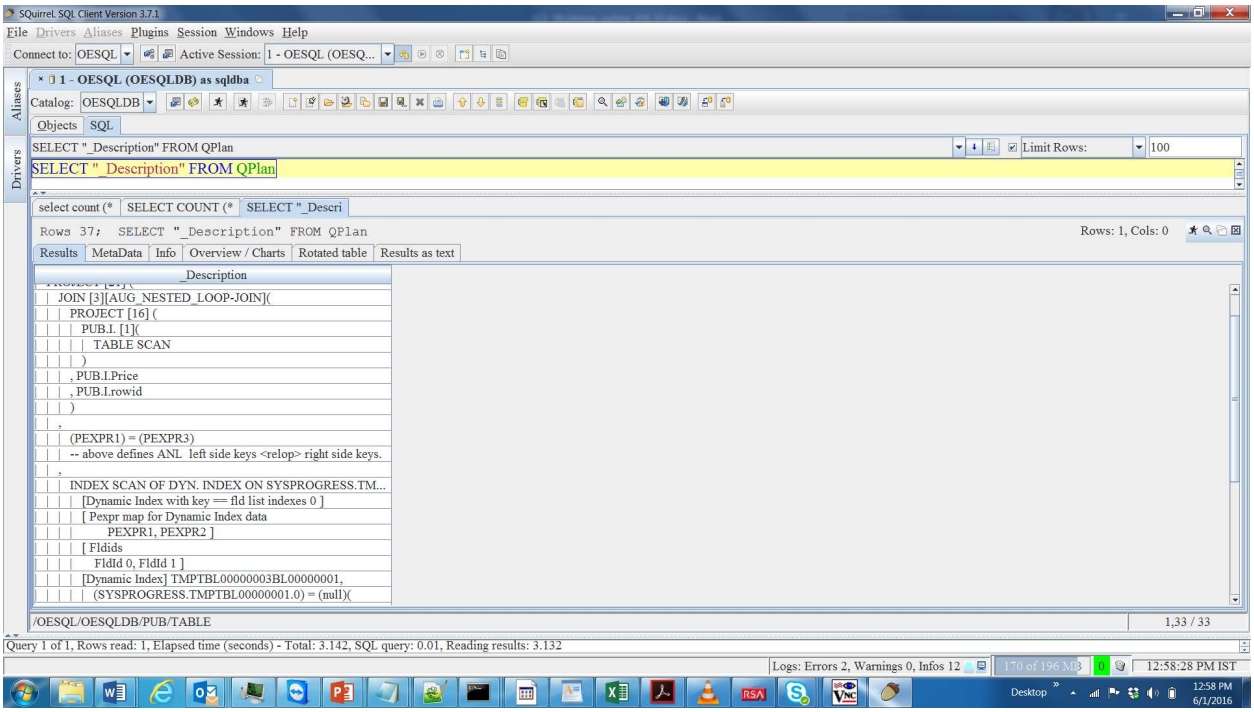
a. **Importance of Table statistics** – To explain the importance of these statistics, you need to run provided join query and check the number of IOs using promon utility. And then, you need to run the table statistics command on tables used by the provided join query and run the same query again to verify the difference between number of IOs took place with former case.

Execute the following query and check the time taken for execution.

SELECT COUNT (*) FROM Pub.Item I, Pub.OrderLine_Large OL where I.price = OL.ExtendedPrice;



Now, check the query plan. You can notice that, DIDX is created on the large table and this caused query to run slower.



_Description

SELECT COMMAND.

PROJECT [22] (

| PROJECT [21] (

| | JOIN [3][AUG_NESTED_LOOP-JOIN](

| | | PROJECT [16] (

| | | | PUB.I. [1](

| | | | | TABLE SCAN

| | | |)

| | | , PUB.I.Price

| | | , PUB.I.rowid

| | |)

| | ,

| | | (PEXPR1) = (PEXPR3)

| | | -- above defines ANL left side keys <

| | ,

| | | INDEX SCAN OF DYN. INDEX ON SYSPROGRES

| | | | [Dynamic Index with key == fld lis

| | | | [Pexpr map for Dynamic Index data

| | | | PEXPR1, PEXPR2]

| | | | [Fldids

| | | | FldId 0, FldId 1]

| | | | [Dynamic Index] TMPTBL00000003BL00

| | | | | (SYSPROGRESS.TMPTBL00000001.0)

| | | | PROJECT [19] (

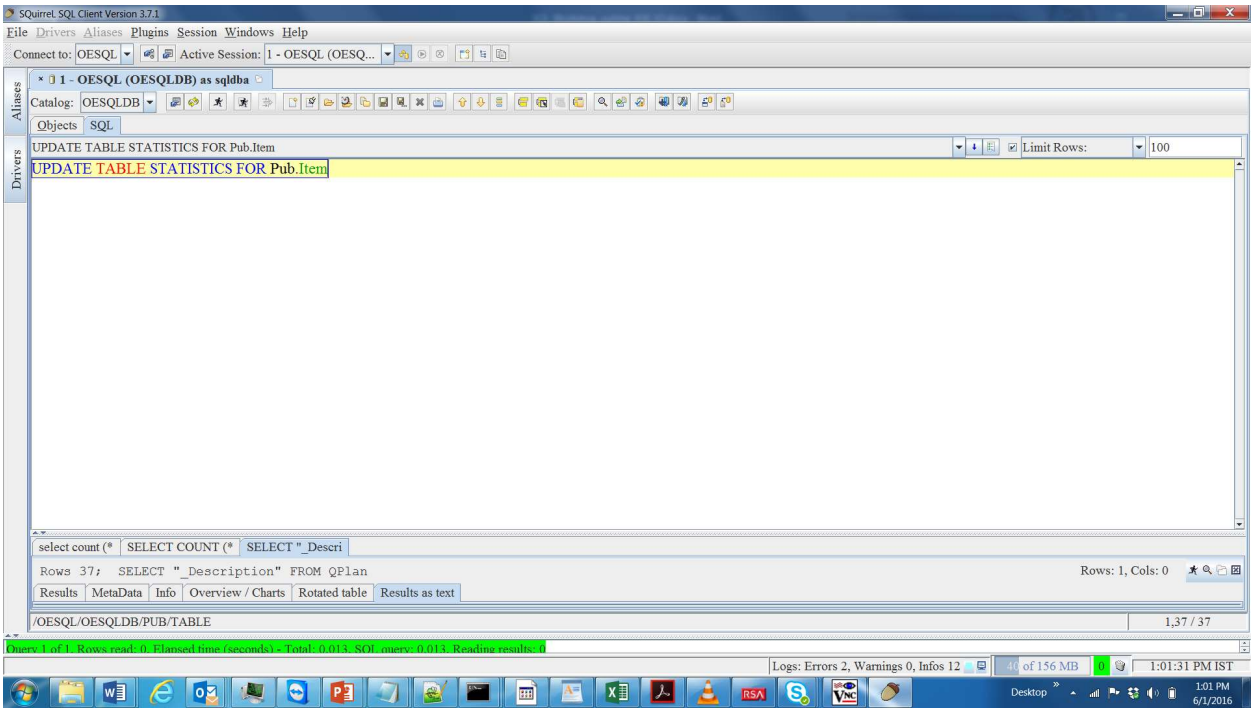
| | | | | PUB.OL. [2](

| | | | | | TABLE SCAN

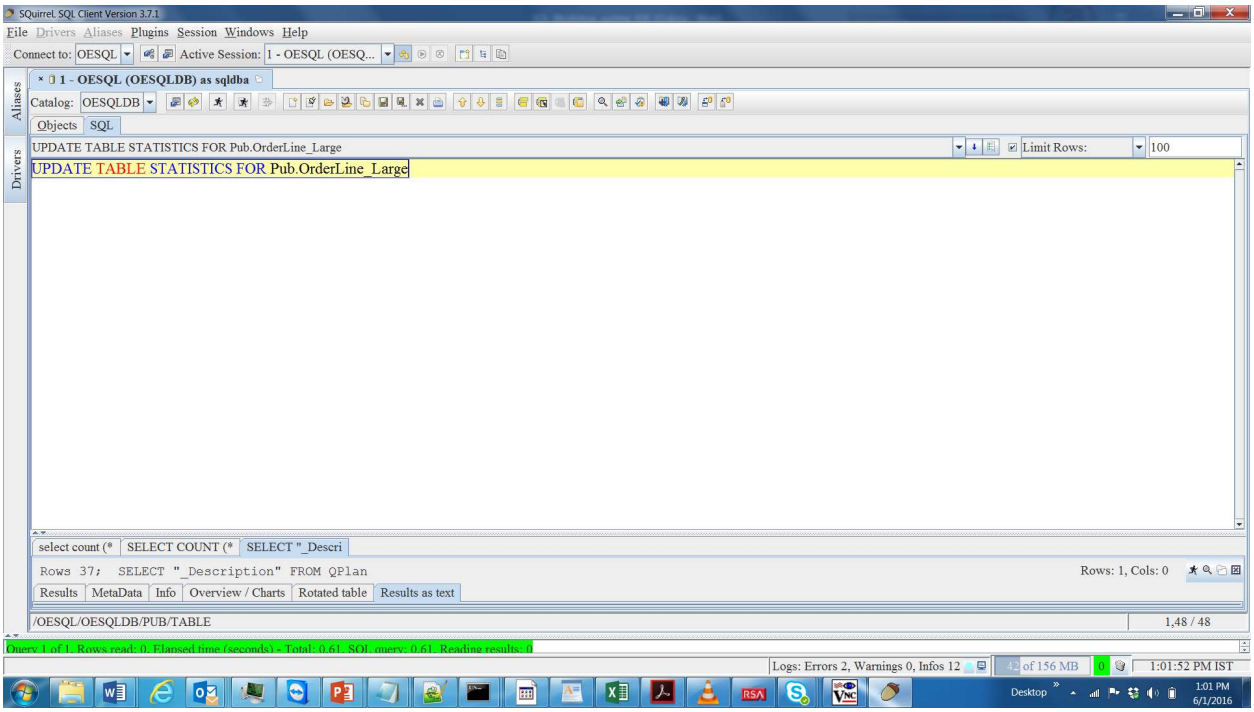
```
|  |  |  |  |  )
|  |  |  |  , PUB.OL.ExtendedPrice
|  |  |  |  , PUB.OL.rowid
|  |  |  |  )
|  |  |
|  |  |  )
|  |  )
|  , count (*)
|  )
, PEXPR1
)
```

Now, let us update the table statistics for the 2 tables involved in the join.

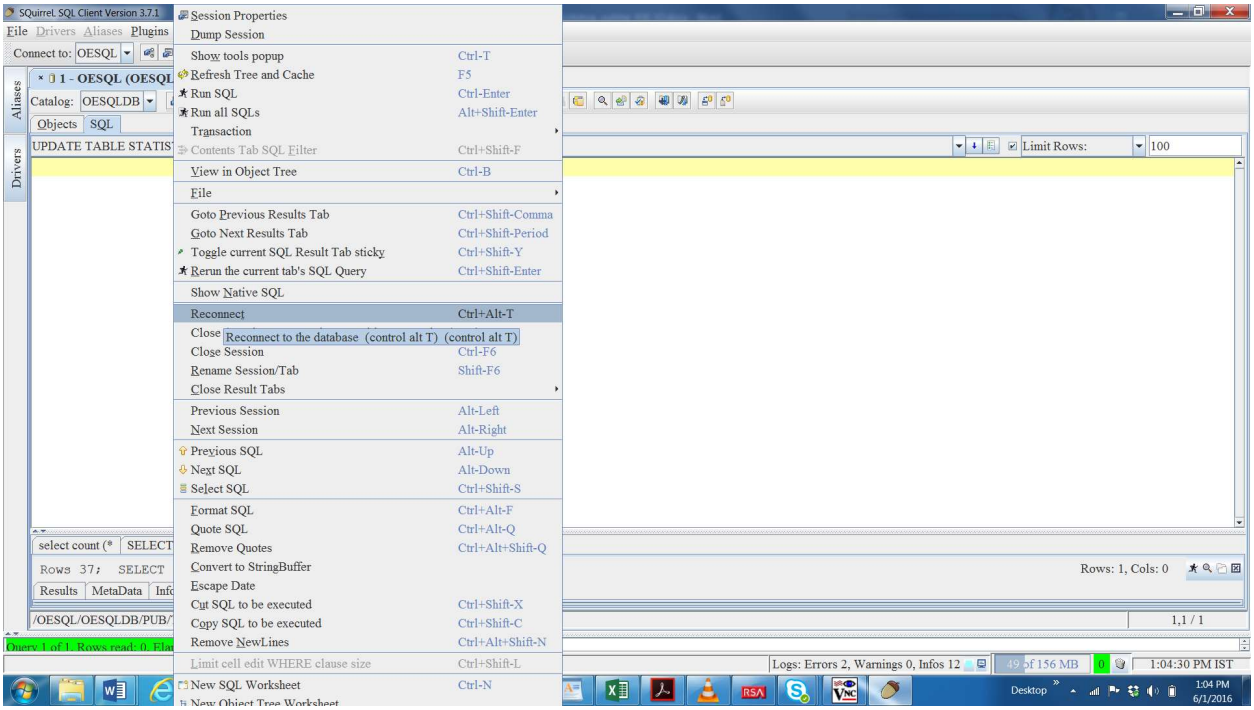
UPDATE TABLE STATISTICS FOR Pub.Item;



UPDATE TABLE STATISTICS FOR Pub.OrderLine_Large;

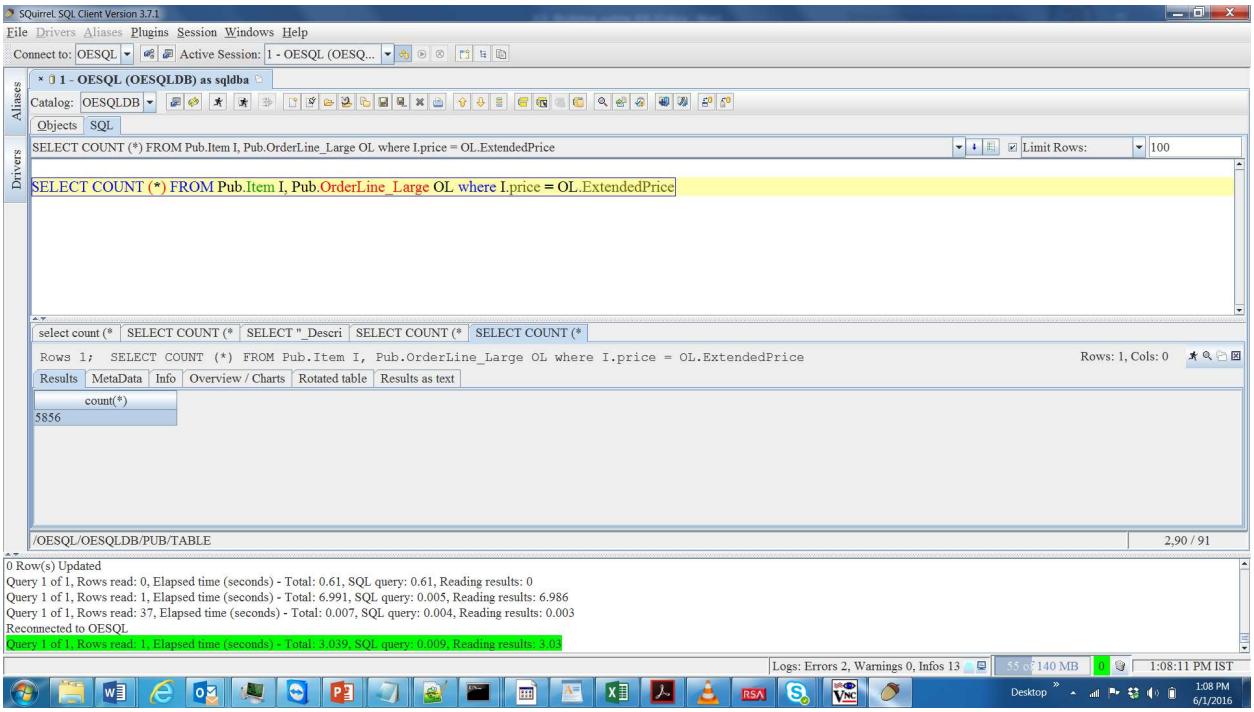


Need to Reconnect to the database in order to see the effect of Table Statistics. For this, go to the "Session" tab of Squirrel client and click on "Reconnect".

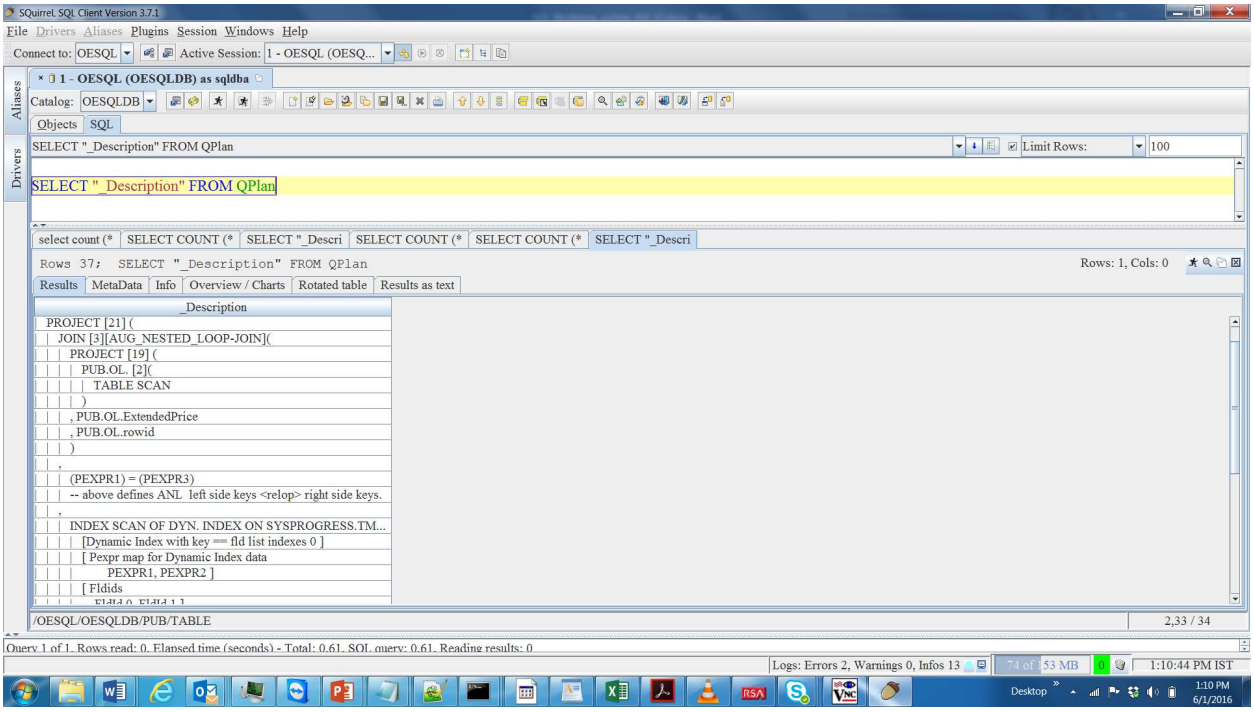


Now, run the same query again and notice the time difference.

SELECT COUNT (*) FROM Pub.Item I, Pub.OrderLine_Large OL where I.price = OL.ExtendedPrice;



Now, check what caused the change in execution time by almost half!! For that, let us check the query plan this time for this same query.

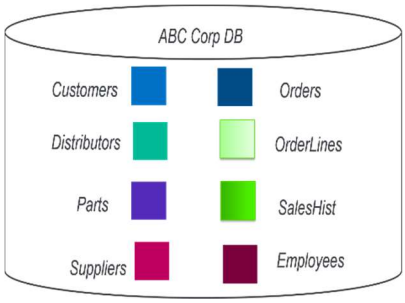


```
_Description
-----------------------------------------------------
SELECT COMMAND.
PROJECT [22] (
|   PROJECT [21] (
|   |   JOIN [3][AUG_NESTED_LOOP-JOIN](
|   |   |   PROJECT [19] (
|   |   |   |   PUB.OL. [2](
|   |   |   |   |   |   TABLE SCAN
|   |   |   |   |   )
|   |   |   |   , PUB.OL.ExtendedPrice
|   |   |   |   , PUB.OL.rowid
|   |   |   |   )
|   |   |   ,
|   |   |   |   (PEXPR1) = (PEXPR3)
|   |   |   |   -- above defines ANL  left side keys <
|   |   |   ,
|   |   |   |   INDEX SCAN OF DYN. INDEX ON SYSPROGRES
|   |   |   |   |   [Dynamic Index with key == fld lis
|   |   |   |   |   [ Pexpr map for Dynamic Index data
|   |   |   |   |           PEXPR1, PEXPR2 ]
|   |   |   |   |   [ Fldids
|   |   |   |   |       FldId 0, FldId 1 ]
|   |   |   |   |   [Dynamic Index] TMPTBL00000002BL00
|   |   |   |   |   |   (SYSPROGRESS.TMPTBL00000001.0)
|   |   |   |   |   PROJECT [16] (
|   |   |   |   |   |   PUB.I. [1](
|   |   |   |   |   |   |   TABLE SCAN
|   |   |   |   |   |   )
|   |   |   |   |   , PUB.I.Price
|   |   |   |   |   , PUB.I.rowid
|   |   |   |   |   )
|   |   |   |
|   |   |   |   )
|   |   |   )
|   , count (*)
|   )
, PEXPR1
```

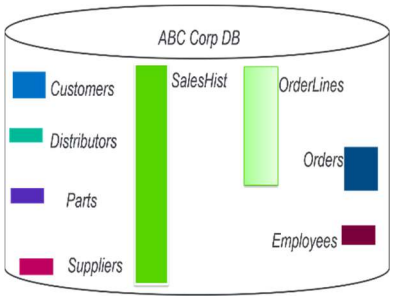
Observation:

We can clearly notice that, having table statistics helped SQL Optimizer to select the join order which is more optimal. That is, SQL optimizer, now identified that creation of Dynamic Index on the Large table amounts to excessive cost, hence, it selected the join order where the Large table is kept on left side and Dynamic Index is created on smaller table.

Before Table Statistics:



After Table Statistics:



b. **Importance of Index statistics** - To explain the importance of these statistics, you need to run provided join query and check the number of IOs using promon utility. And then, you need to run the index statistics command on tables used by the provided join query and run the same query again to verify the difference between number of IOs took place with former case.

Now, let us see how index stats effects the performance of a query.

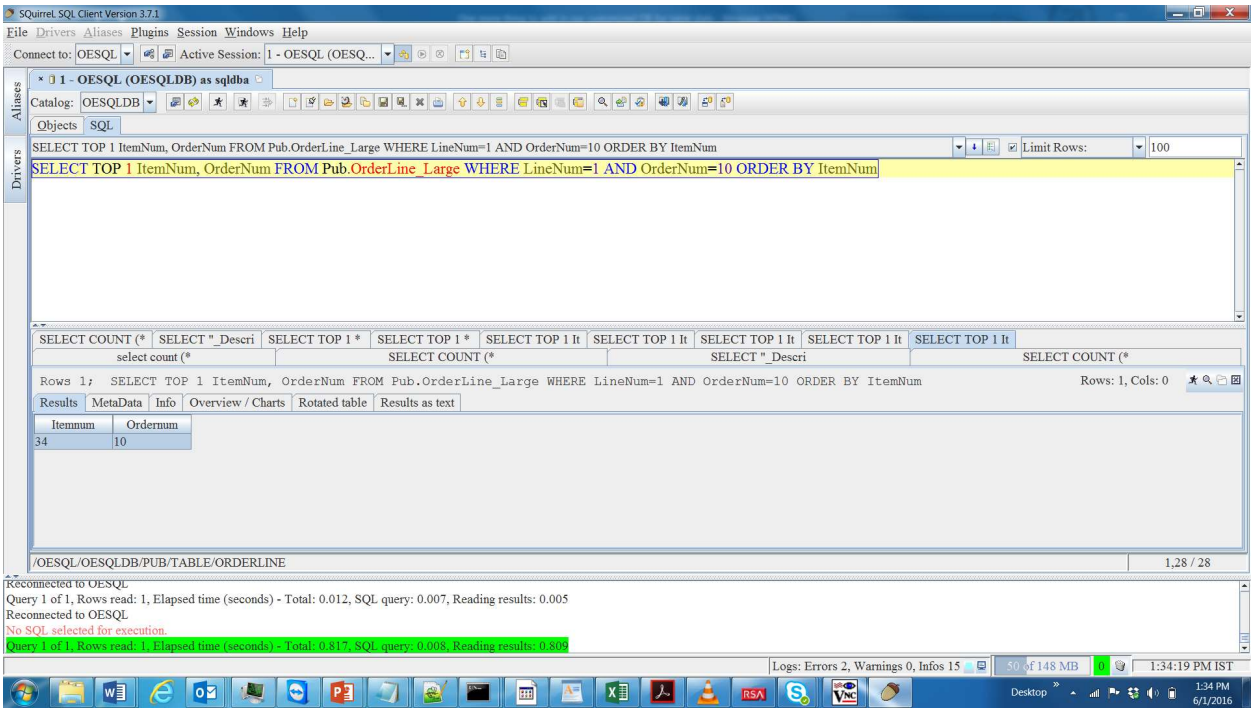
Table Pub.OrderLine_Large already has following 2 indexes.

i) LineNum_Large created on LineNum column

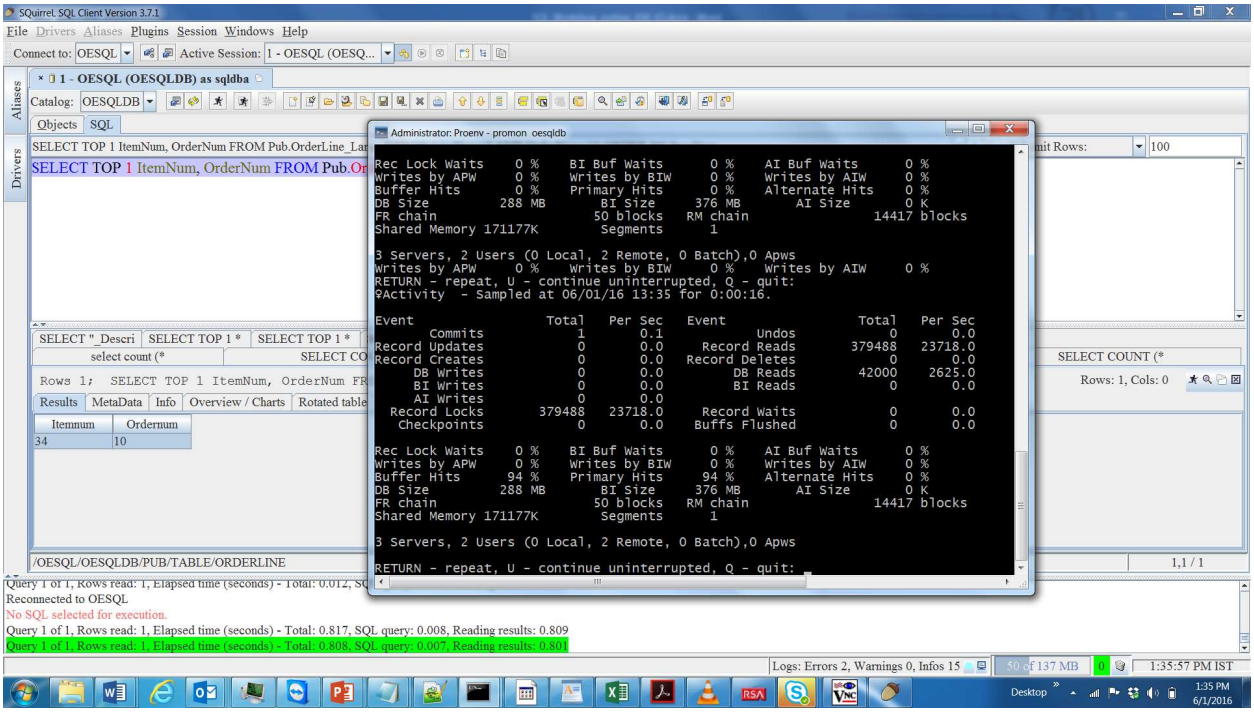
ii) OrderNum_Large created on OrderNum colum.

Now, run the below query.

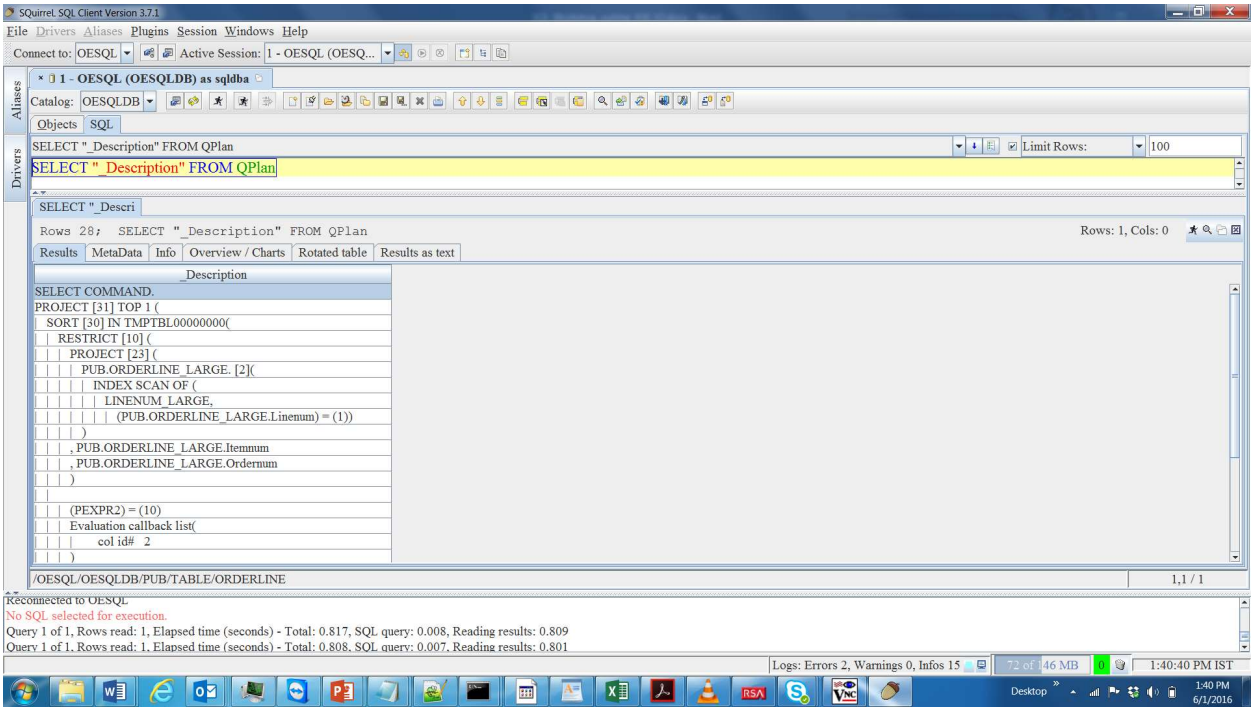
SELECT TOP 1 ItemNum, OrderNum FROM Pub.OrderLine_Large WHERE LineNum=1 AND OrderNum=10 ORDER BY ItemNum;



Now, check the number of Records read



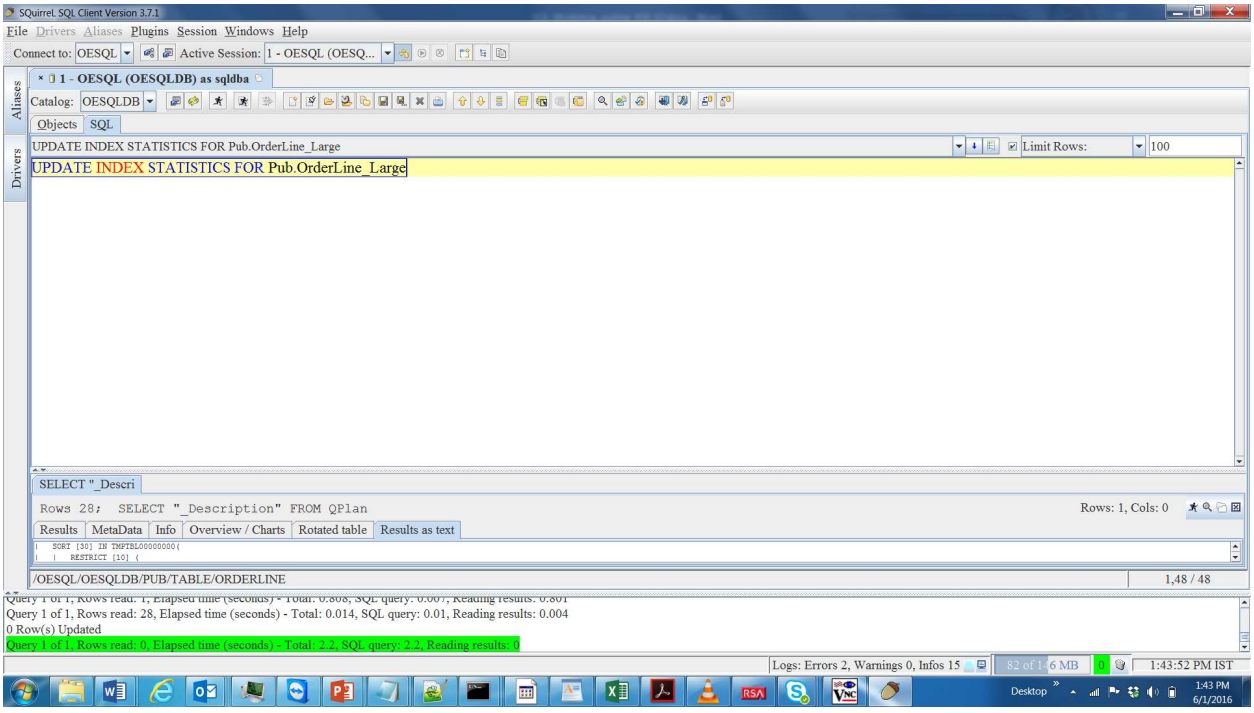
Now, check the query plan. We can notice that it selected index which is created on LineNum column of Pub.OrderLine_Large table.



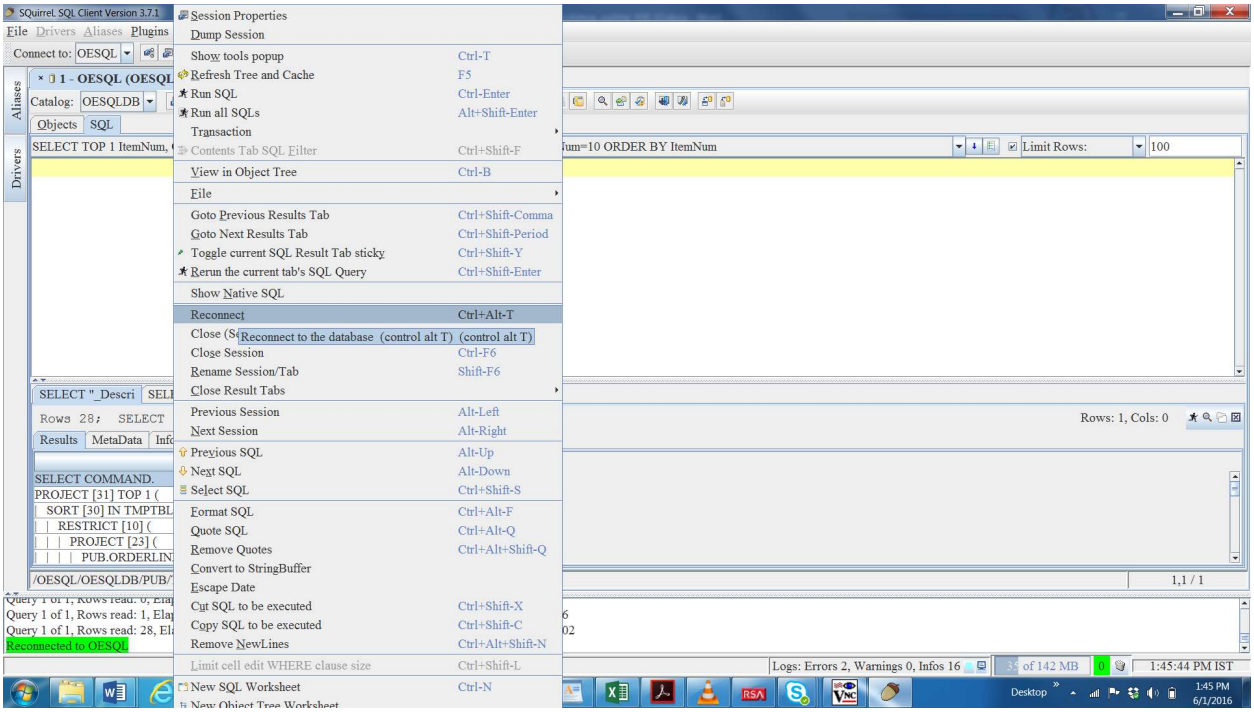
```
_Description
--------------------------------------------------
SELECT COMMAND.
PROJECT [31] TOP 1 (
|  SORT [30] IN TMPTBL00000000(
|  |  RESTRICT [10] (
|  |  |  PROJECT [23] (
|  |  |  |  PUB.ORDERLINE_LARGE. [2](
|  |  |  |  |  INDEX SCAN OF (
|  |  |  |  |  |  LINENUM_LARGE,
|  |  |  |  |  |  |  (PUB.ORDERLINE_LARGE.L
|  |  |  |  )
|  |  |  , PUB.ORDERLINE_LARGE.Itemnum
|  |  |  , PUB.ORDERLINE_LARGE.Ordernum
|  |  |  )
|  |
|  |  |  (PEXPR2) = (10)
|  |  |  Evaluation callback list(
|  |  |  |       col id#   2
|  |  |  )
|  |  )
|  | SORT BY (
|  |  , Sort project expression #0 == PEXPR1
|  ),
|  , PEXPR1
|  , PEXPR2
|  )
, PEXPR1
, PEXPR2
)
```

Now, update the Index Statistics for table Pub.OrderLine_Large.

UPDATE INDEX STATISTICS FOR Pub.OrderLine_Large;

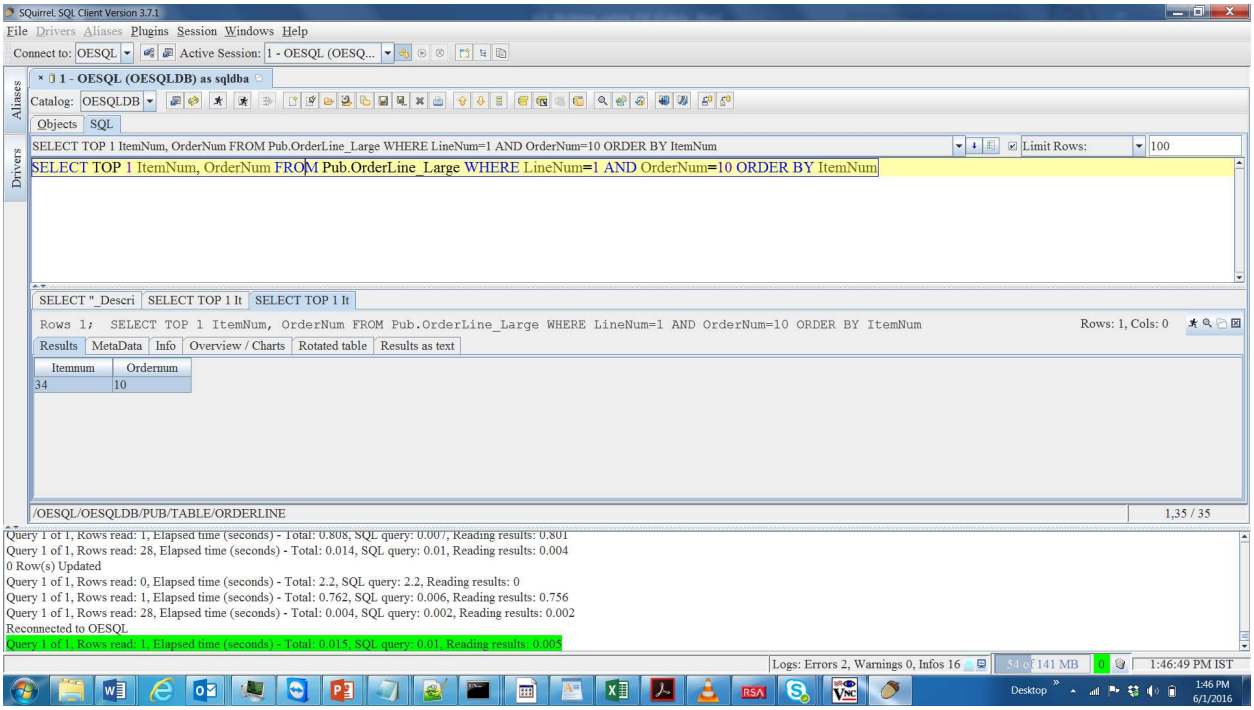


Reconnect to Database in order to get the effects of Index Statistics

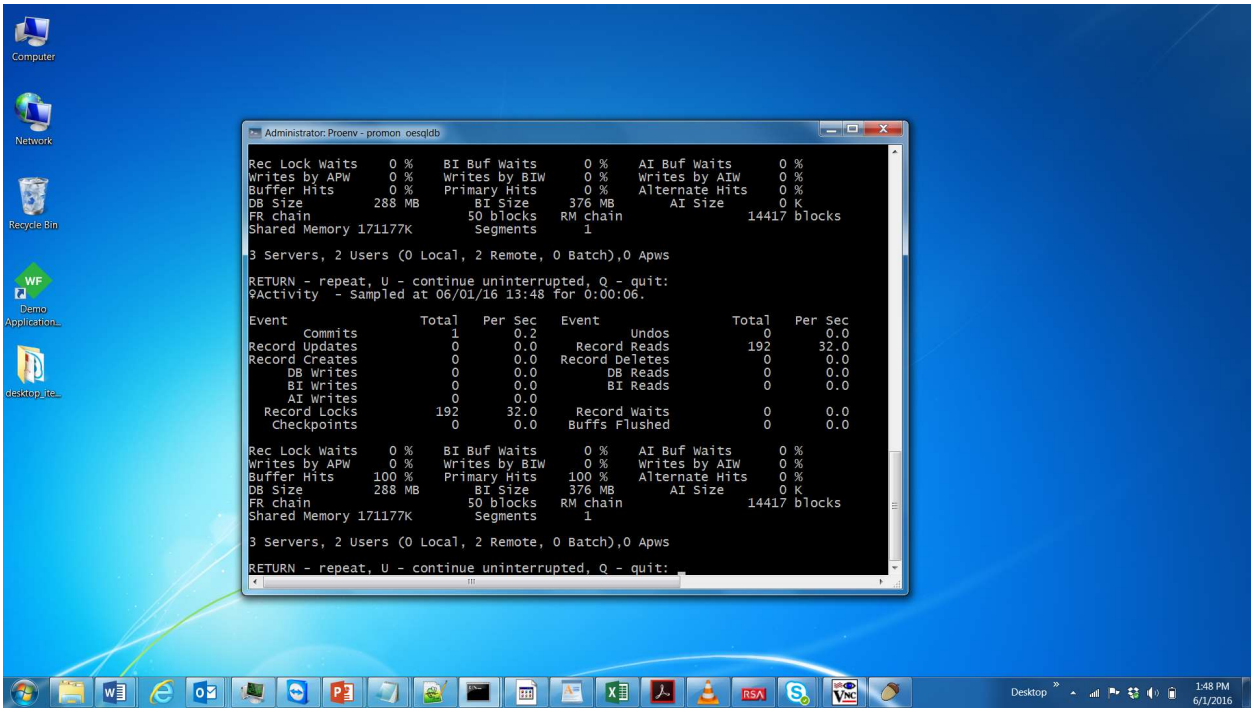


Now, run the same query again.

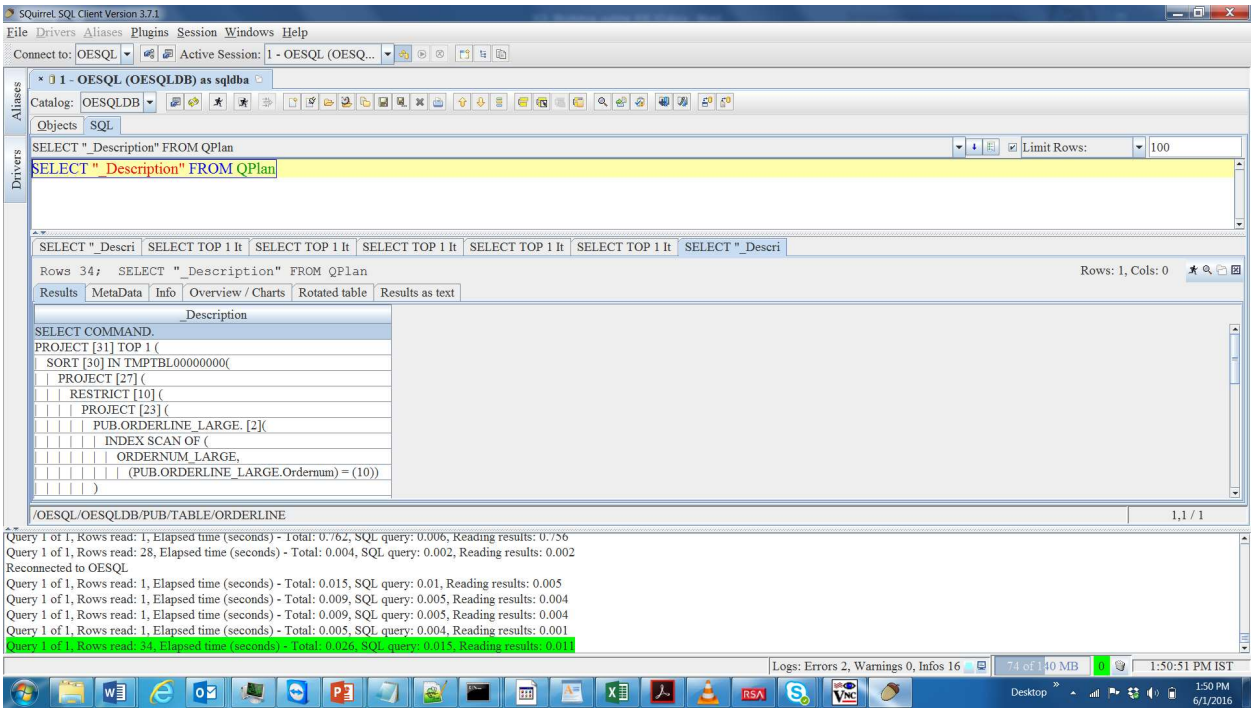
SELECT TOP 1 ItemNum, OrderNum FROM Pub.OrderLine_Large WHERE LineNum=1 AND OrderNum=10 ORDER BY ItemNum;



Now, check the number of Records Read in promon Window.



Now, check the query plan for this same query.

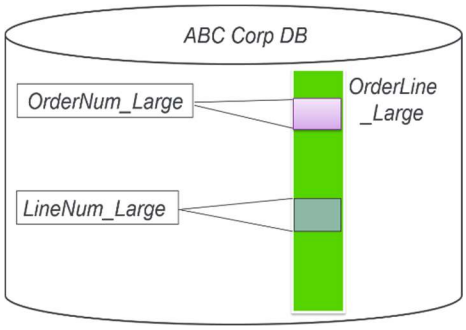


```
_Description
-----------------------------------------------------
SELECT COMMAND.
PROJECT [31] TOP 1 (
|   SORT [30] IN TMPTBL00000000(
|   |   PROJECT [27] (
|   |   |   RESTRICT [10] (
|   |   |   |   PROJECT [23] (
|   |   |   |   |   PUB.ORDERLINE_LARGE. [2](
|   |   |   |   |   |   INDEX SCAN OF (
|   |   |   |   |   |   |   |   ORDERNUM_LARGE,
|   |   |   |   |   |   |   |   |   (PUB.ORDERLINE_LAR
|   |   |   |   |   |   )
|   |   |   |   |   , PUB.ORDERLINE_LARGE.Itemnum
|   |   |   |   |   , PUB.ORDERLINE_LARGE.Ordernum
|   |   |   |   |   , PUB.ORDERLINE_LARGE.Linenum
|   |   |   |   |   )
|   |   |   |
|   |   |   |   |   (PEXPR3) = (1)
|   |   |   |   |   Evaluation callback list(
|   |   |   |   |   |       col id#   3
|   |   |   |   |   |     [can terminate scan]
|   |   |   |   |   )
|   |   |   |   )
|   |   |   )
|   |   , PEXPR1
|   |   , PEXPR2
|   |   )
|   | SORT BY (
|   |   , Sort project expression #0 == PEXPR1
|   ),
|   , PEXPR1
|   , PEXPR2
|   )
, PEXPR1
, PEXPR2
)
```

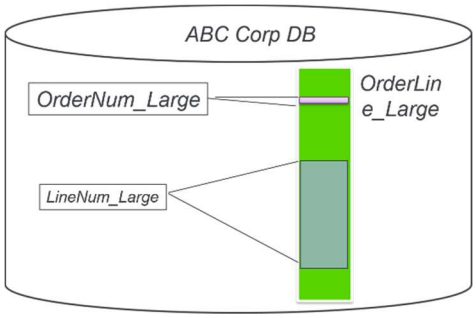
Observation:

What caused the dramatic improvement in query execution time and number of records read!!! With the help of index statistics SQL Optimizer could differentiate and compare the costs of two different indexes and selected the index on OrderNum (OrderNum_Large) as it contains more number of unique values and hence filters more rows.

Before Index Statistics



After Index Statistics

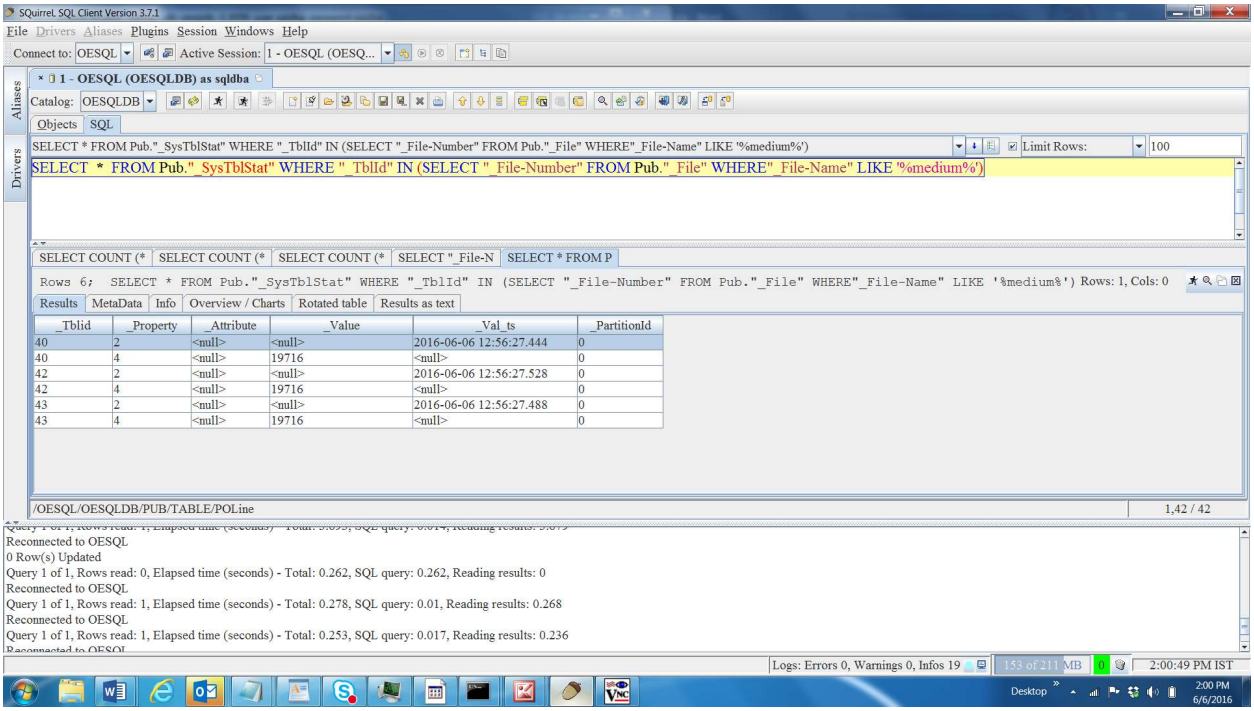


c. **Importance of Column statistics** – There are 2 flavors of column statistics, one which just updates column statistics for columns which has index on them and the other one, which collects column statistics for all the columns of the table. We will use the flavor which collects statistics for all the columns to explain the importance of these statistics. You need to run provided join query and check the number of IOs using promon utility. And then, you need to run the column statistics command on tables used by the provided join query and run the same query again to verify the difference between number of IOs took place with former case.

First, let us check if the Table Statistics exists for the tables which we will be using in our example query to demonstrate the importance of column statistics. If you think, table and index statistics are enough, then you may need to re-think about this perspective after this illustration.

Below query checks if the table statistics are present for tables, Pub.Order_Medium, Pub.OrderLine_Medium and Pub.POLine_Medium.

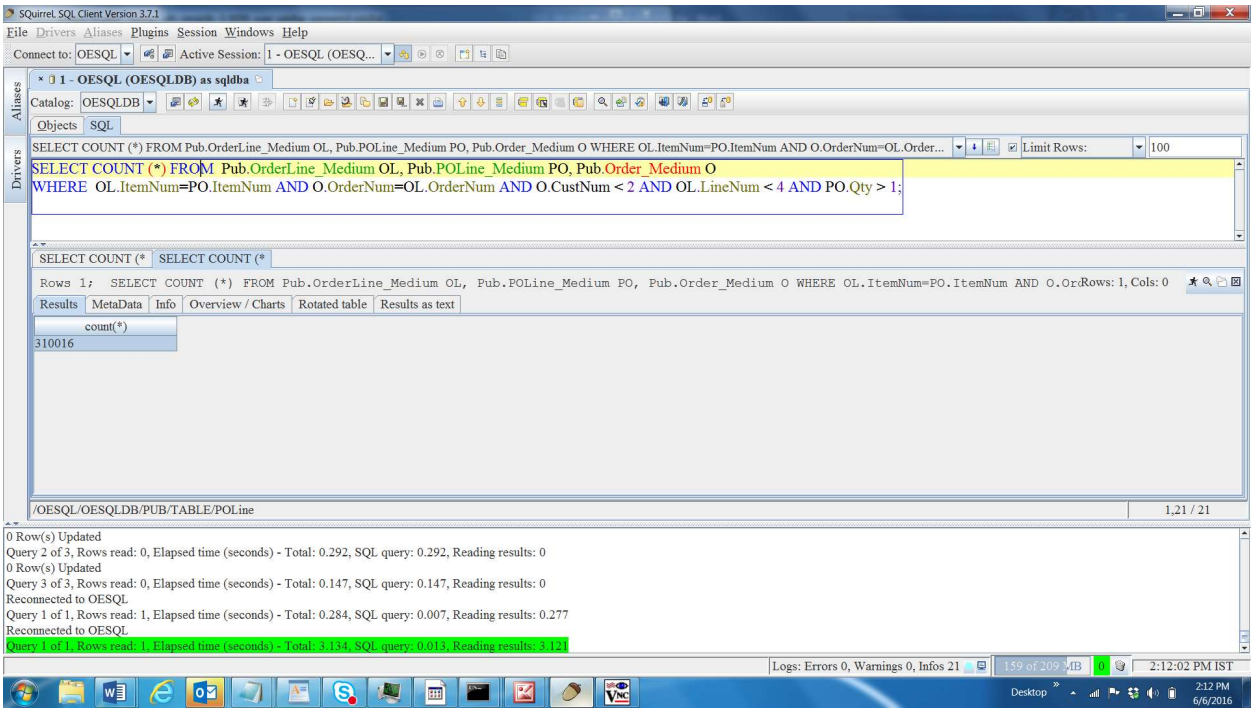
*SELECT * FROM Pub."_SysTblStat" WHERE "_TblId" IN (SELECT "_File-Number" FROM Pub."_File" WHERE"_File-Name" LIKE '%medium%');*



We can notice that each table has same number of rows (19716).

Now, let us run the below query to see the effect of column statistics.

SELECT COUNT (*) FROM Pub.OrderLine_Medium OL, Pub.POLine_Medium PO, Pub.Order_Medium O WHERE OL.ItemNum=PO.ItemNum AND O.OrderNum=OL.OrderNum AND O.CustNum < 2 AND OL.LineNum < 4 AND PO.Qty > 1;



Now, check the query plan.

```
_Description
-----------------------------------------------------
SELECT COMMAND.
PROJECT [53] (
|  PROJECT [52] (
|  |  JOIN [5][AUG_NESTED_LOOP-JOIN](
|  |  |  JOIN [3][AUG_NESTED_LOOP-JOIN](
|  |  |  |  RESTRICT [55] (
|  |  |  |  |  PROJECT [42] (
|  |  |  |  |  |  PUB.OL. [1](
|  |  |  |  |  |  |  TABLE SCAN
|  |  |  |  |  |  )
|  |  |  |  |  , PUB.OL.Itemnum
|  |  |  |  |  , PUB.OL.Ordernum
|  |  |  |  |  , PUB.OL.Linenum
|  |  |  |  |  , PUB.OL.rowid
|  |  |  |  |  )
|  |  |  |  |
|  |  |  |  |  (PEXPR3) < (4)
|  |  |  |  |  Evaluation callback list(
|  |  |  |  |  |       col id#  3
|  |  |  |  |  )
|  |  |  |  )
|  |  |  |  ,
```

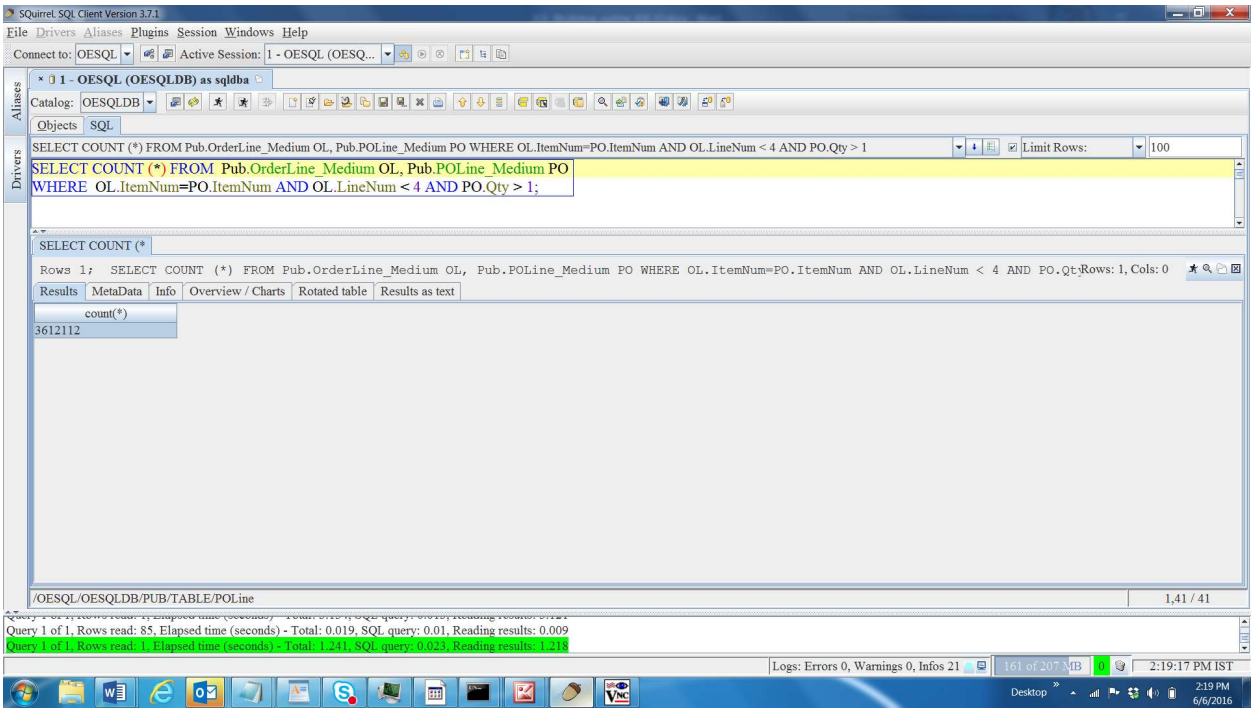
```
|  |  |  |  (PEXPR1) = (PEXPR5)
|  |  |  |  -- above defines ANL  left side ke
|  |  |  ,
|  |  |  |  INDEX SCAN OF DYN. INDEX ON SYSPRO
|  |  |  |  |  [Dynamic Index with key == fld
|  |  |  |  |  [ Pexpr map for Dynamic Index
|  |  |  |  |        PEXPR1, PEXPR2, PEXPR3
|  |  |  |  |  [ Fldids
|  |  |  |  |     FldId 0, FldId 1, FldId 2
|  |  |  |  |  [Dynamic Index] TMPTBL00000003
|  |  |  |  |  |  (SYSPROGRESS.TMPTBL0000000
|  |  |  |  |  RESTRICT [54] (
|  |  |  |  |  |  PROJECT [46] (
|  |  |  |  |  |  |  PUB.PO. [2](
|  |  |  |  |  |  |  |  TABLE SCAN
|  |  |  |  |  |  |  )
|  |  |  |  |  |  , PUB.PO.Itemnum
|  |  |  |  |  |  , PUB.PO.Qty
|  |  |  |  |  |  , PUB.PO.rowid
|  |  |  |  |  |  )
|  |  |  |  |
|  |  |  |  |  |  (PEXPR2) > (1)
|  |  |  |  |  |  Evaluation callback list(
|  |  |  |  |  |  |       col id#   5
|  |  |  |  |  |  )
|  |  |  |  |  )
|  |  |  |
|  |  |  |  )
|  |  |  )
|  |  ,
|  |  |  (PEXPR2) = (PEXPR8)
|  |  |  -- above defines ANL  left side keys <
|  |  ,
|  |  |  INDEX SCAN OF DYN. INDEX ON SYSPROGRES
|  |  |  |  [Dynamic Index with key == fld lis
|  |  |  |  [ Pexpr map for Dynamic Index data
|  |  |  |        PEXPR1, PEXPR2, PEXPR3 ]
|  |  |  |  [ Fldids
|  |  |  |     FldId 0, FldId 1, FldId 2 ]
|  |  |  |  [Dynamic Index] TMPTBL00000004BL00
|  |  |  |  |  (SYSPROGRESS.TMPTBL00000002.0)
|  |  |  |  RESTRICT [56] (
|  |  |  |  |  PROJECT [50] (
|  |  |  |  |  |  PUB.O. [4](
|  |  |  |  |  |  |  TABLE SCAN
|  |  |  |  |  |  )
|  |  |  |  |  , PUB.O.Ordernum
|  |  |  |  |  , PUB.O.CustNum
|  |  |  |  |  , PUB.O.rowid
|  |  |  |  |  )
|  |  |  |
|  |  |  |  |  (PEXPR2) < (2)
|  |  |  |  |  Evaluation callback list(
|  |  |  |  |  |       col id#   3
|  |  |  |  |  )
```

```
|  |  |  |  )
|  |  |
|  |  |  )
|  |  )
|  , count (*)
|  )
, PEXPR1
```

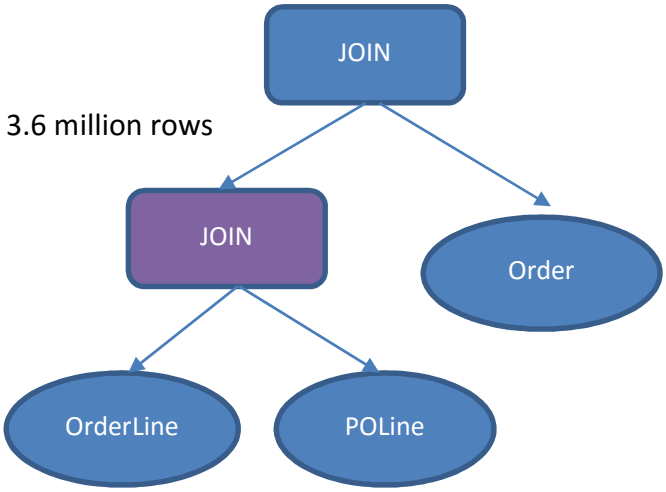
Reason for Slower execution:

If we analyze, the intermediate join (between Pub.OrderLine_Medium and Pub.POLine_Medium) produced around 3.6 million rows!!!

SELECT COUNT(*) FROM Pub.OrderLine_Medium OL, Pub.POLine_Medium PO
WHERE OL.ItemNum=PO.ItemNum AND OL.LineNum<4 AND PO.Qty>1;



3.6 million rows

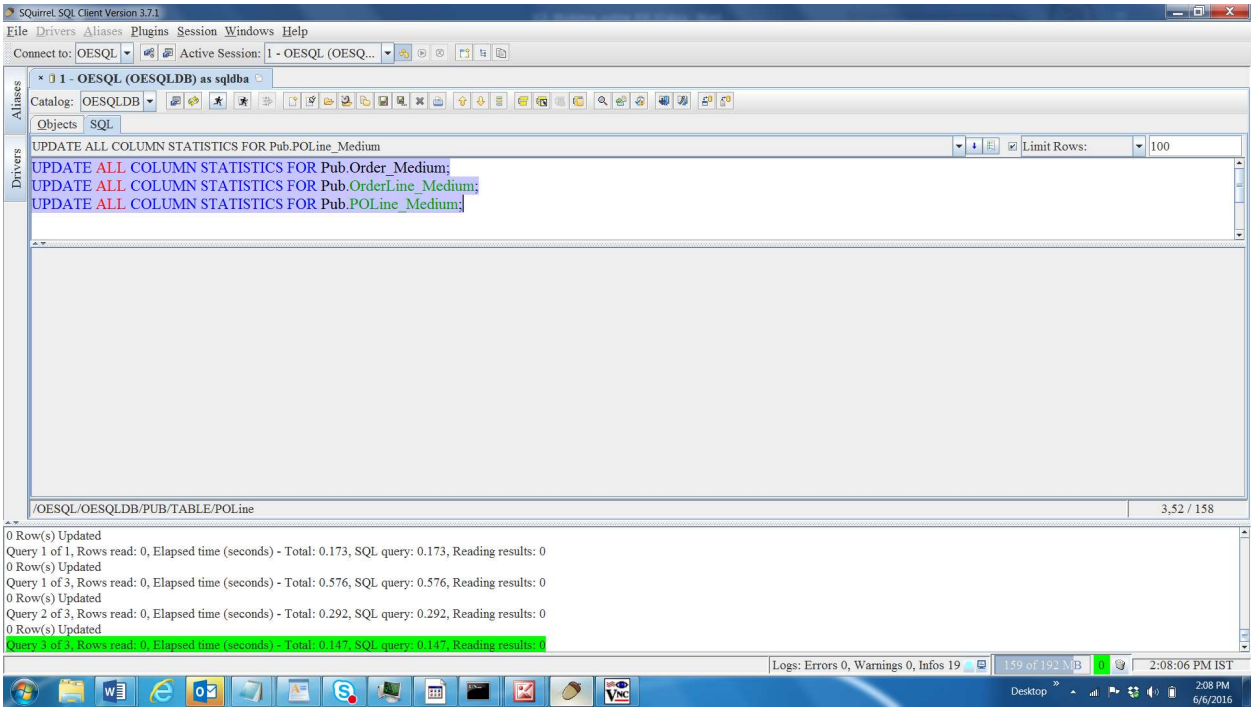


Now, run the column statistics for the 3 tables involved in the query.

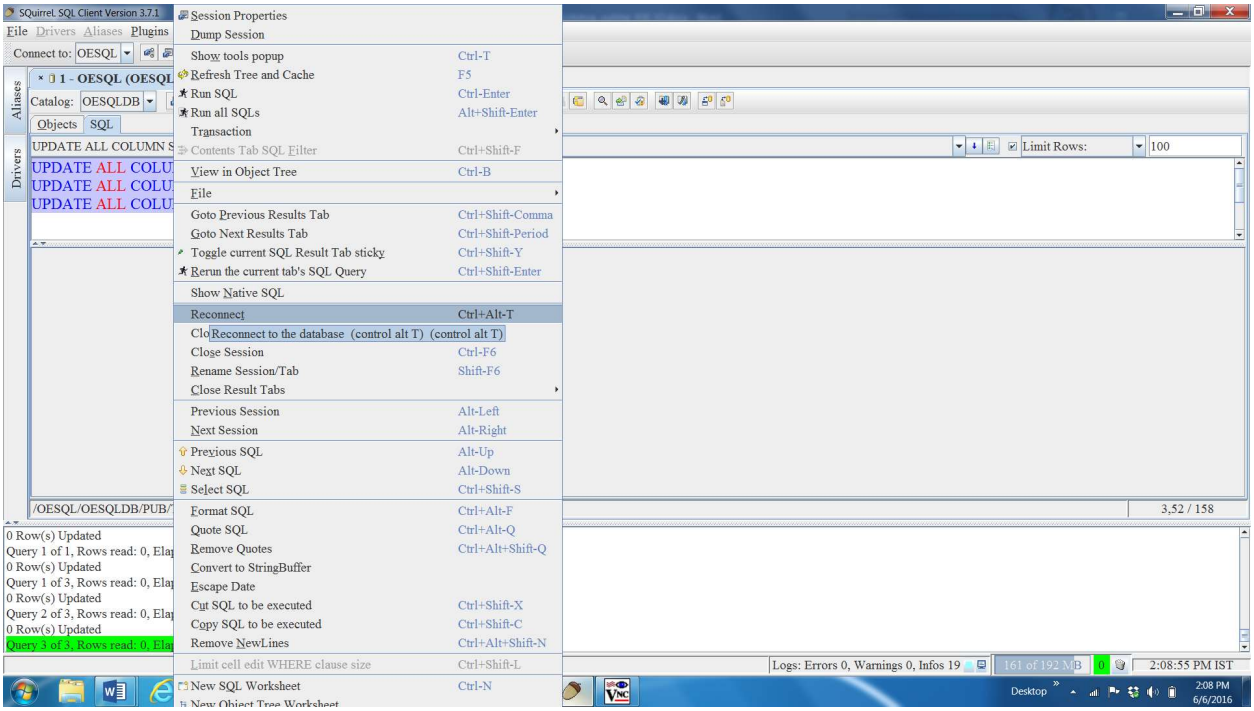
UPDATE ALL COLUMN STATISTICS FOR Pub.Order_Medium;

UPDATE ALL COLUMN STATISTICS FOR Pub.OrderLine_Medium;

UPDATE ALL COLUMN STATISTICS FOR Pub.POLine_Medium;

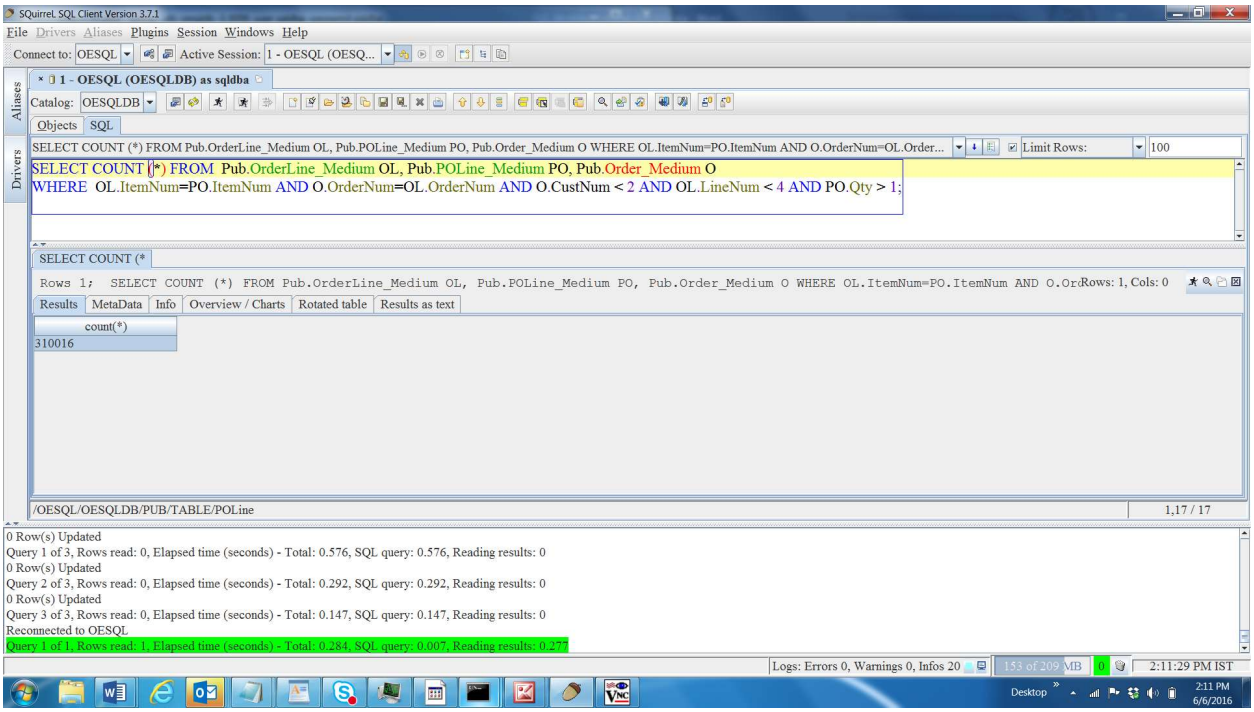


You need to Reconnect to see the effect of Column Statistics.



Now, run the same query again.

SELECT COUNT (*) FROM Pub.OrderLine_Medium OL, Pub.POLine_Medium PO, Pub.Order_Medium O WHERE OL.ItemNum=PO.ItemNum AND O.OrderNum=OL.OrderNum AND O.CustNum < 2 AND OL.LineNum < 4 AND PO.Qty > 1;



Now, check the query plan.

```
_Description
-----------------------------------------------------
SELECT COMMAND.
PROJECT [53] (
|  PROJECT [52] (
|  |  JOIN [66][AUG_NESTED_LOOP-JOIN](
|  |  |  JOIN [65][AUG_NESTED_LOOP-JOIN](
|  |  |  |  RESTRICT [55] (
|  |  |  |  |  PROJECT [42] (
|  |  |  |  |  |  PUB.OL. [1](
|  |  |  |  |  |  |  TABLE SCAN
|  |  |  |  |  |  )
|  |  |  |  |  , PUB.OL.Itemnum
|  |  |  |  |  , PUB.OL.Ordernum
|  |  |  |  |  , PUB.OL.Linenum
|  |  |  |  |  , PUB.OL.rowid
|  |  |  |  |  )
|  |  |  |
|  |  |  |  |  (PEXPR3) < (4)
|  |  |  |  |  Evaluation callback list(
|  |  |  |  |  |       col id#  3
|  |  |  |  |  )
|  |  |  |  )
```

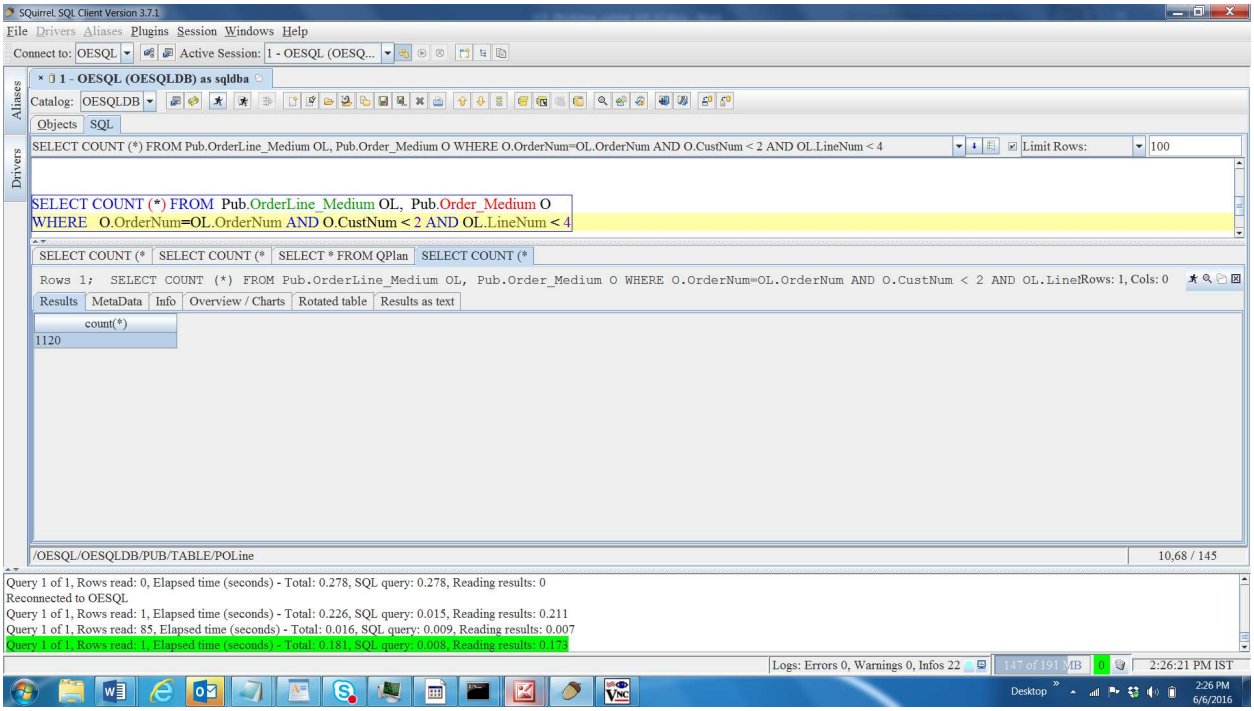
```
|   |   |   ,
|   |   |   |   (PEXPR2) = (PEXPR5)
|   |   |   |   -- above defines ANL  left side ke
|   |   |   ,
|   |   |   |   INDEX SCAN OF DYN. INDEX ON SYSPRO
|   |   |   |   |   [Dynamic Index with key == fld
|   |   |   |   |   [ Pexpr map for Dynamic Index
|   |   |   |   |           PEXPR1, PEXPR2, PEXPR3
|   |   |   |   |   [ Fldids
|   |   |   |   |       FldId 0, FldId 1, FldId 2
|   |   |   |   |   [Dynamic Index] TMPTBL00000003
|   |   |   |   |   |   (SYSPROGRESS.TMPTBL0000000
|   |   |   |   |   RESTRICT [56] (
|   |   |   |   |   |   PROJECT [50] (
|   |   |   |   |   |   |   PUB.O. [4](
|   |   |   |   |   |   |   |   |   TABLE SCAN
|   |   |   |   |   |   |   |   )
|   |   |   |   |   |   , PUB.O.Ordernum
|   |   |   |   |   |   , PUB.O.CustNum
|   |   |   |   |   |   , PUB.O.rowid
|   |   |   |   |   |   )
|   |   |   |   |
|   |   |   |   |   |   (PEXPR2) < (2)
|   |   |   |   |   |   Evaluation callback list(
|   |   |   |   |   |   |       col id#   3
|   |   |   |   |   |   )
|   |   |   |   |   )
|   |   |   |
|   |   |   |   )
|   |   |   )
|   |   ,
|   |   |   (PEXPR1) = (PEXPR8)
|   |   |   -- above defines ANL  left side keys <
|   |   ,
|   |   |   INDEX SCAN OF DYN. INDEX ON SYSPROGRES
|   |   |   |   [Dynamic Index with key == fld lis
|   |   |   |   [ Pexpr map for Dynamic Index data
|   |   |   |           PEXPR1, PEXPR2, PEXPR3 ]
|   |   |   |   [ Fldids
|   |   |   |       FldId 0, FldId 1, FldId 2 ]
|   |   |   |   [Dynamic Index] TMPTBL00000004BL00
|   |   |   |   |   (SYSPROGRESS.TMPTBL00000002.0)
|   |   |   |   RESTRICT [54] (
|   |   |   |   |   PROJECT [46] (
|   |   |   |   |   |   PUB.PO. [2](
|   |   |   |   |   |   |   TABLE SCAN
|   |   |   |   |   |   )
|   |   |   |   |   , PUB.PO.Itemnum
|   |   |   |   |   , PUB.PO.Qty
|   |   |   |   |   , PUB.PO.rowid
|   |   |   |   |   )
|   |   |   |
|   |   |   |   |   (PEXPR2) > (1)
|   |   |   |   |   Evaluation callback list(
|   |   |   |   |   |       col id#   5
```

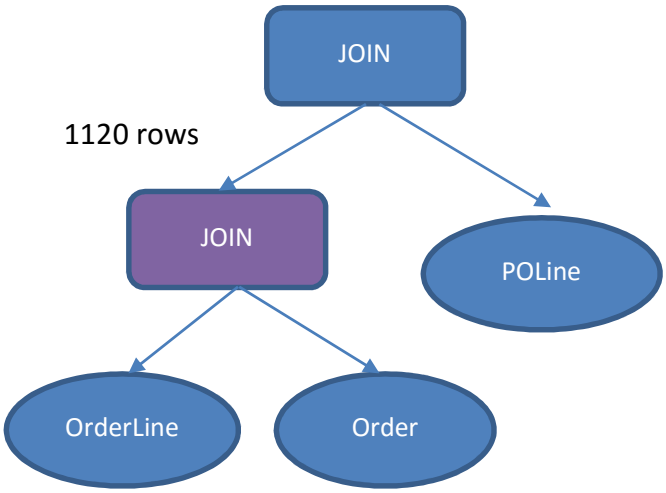
```
|  |  |  |  |  )
|  |  |  |  )
|  |  |
|  |  |  )
|  |  )
|  , count (*)
|  )
, PEXPR1
)
```

Reason for faster execution:

We can notice that, due to column statistics in place intermediate join is now between Pub.Order_Medium and pub.OrderLine_Medium which produces 1120 rows and that is why it is much faster.

SELECT COUNT(*) FROM Pub.OrderLine_Medium OL, Pub.Order_Medium O
WHERE O.OrderNum=OL.OrderNum AND O.CustNum < 2 AND OL.LineNum<4;





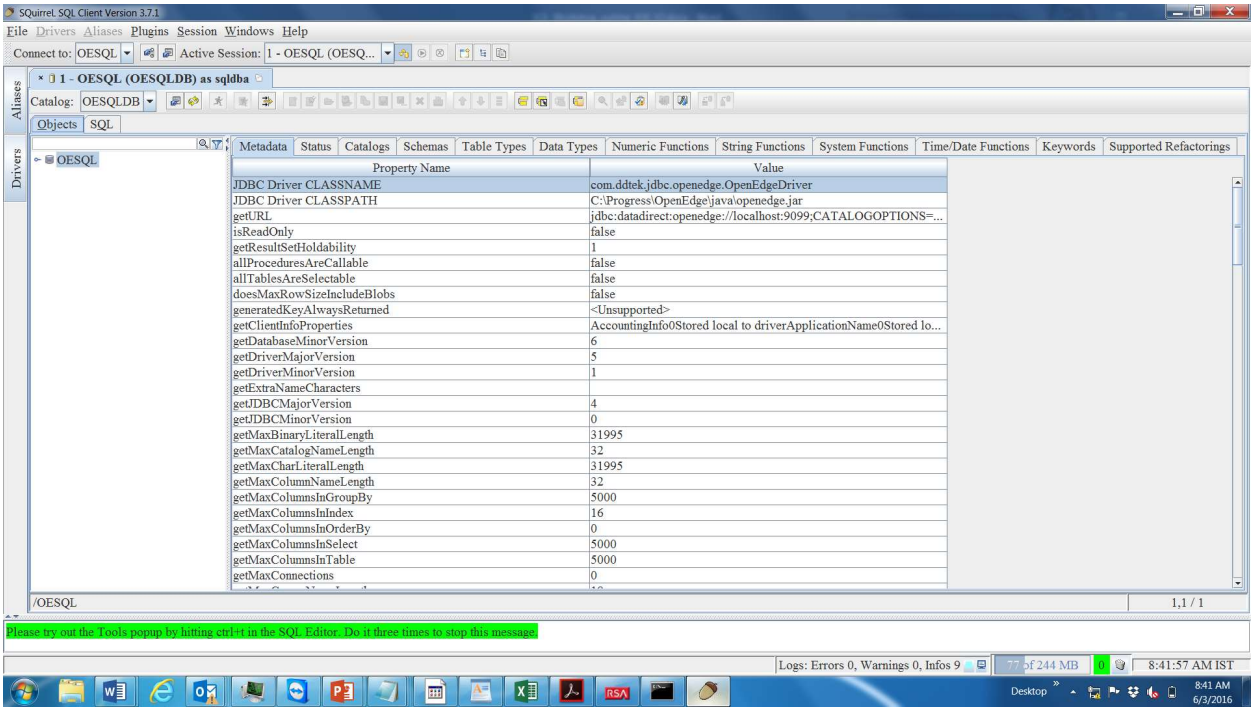
Observation: Having column statistics helped SQL Optimizer to come up with more optimal join order which made the overall query execution much faster. For instance, with the help of column statistics in place, SQL Optimizer could identify that, filter on OrderNum of table Pub.Order can filter more number of rows, because it has more number of unique values and selected this table as part of the intermediate join.

Chapter 9 – What goes into Statistics?

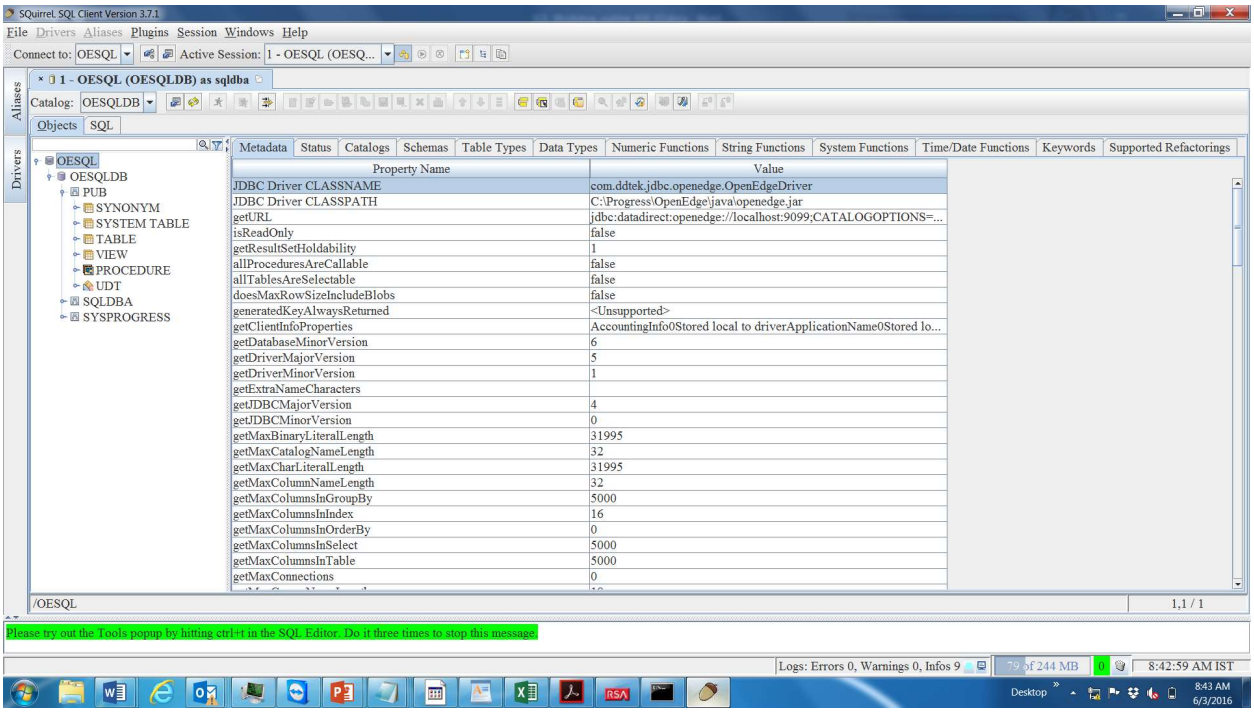
In this chapter you will learn what is inside the statistics and how this information helps SQL Optimizer to come up with optimal query plans.

First let us see about Table Statistics. The System Table involved in this is, Pub."_SysTblStat". Let us see what goes into this table when we update the Table statistics. Go to Squirrel window, and check for this table structure.

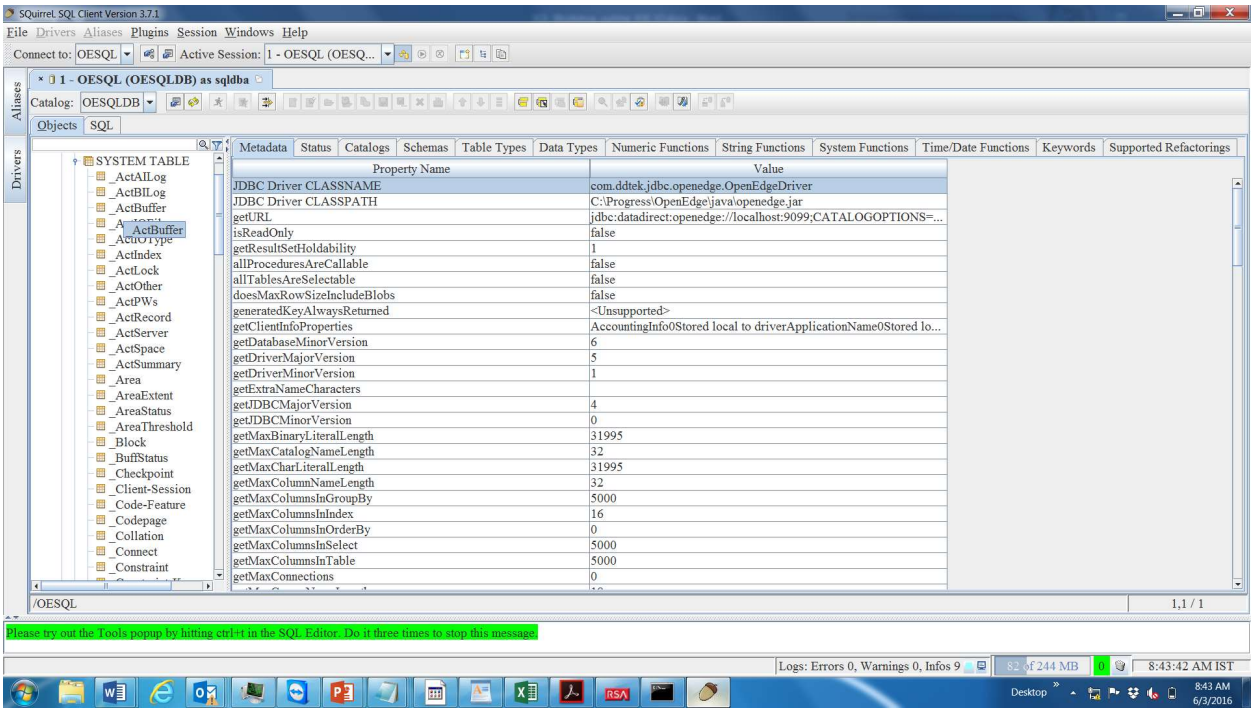
Click on "Objects" Tab (just beside the SQL Tab).



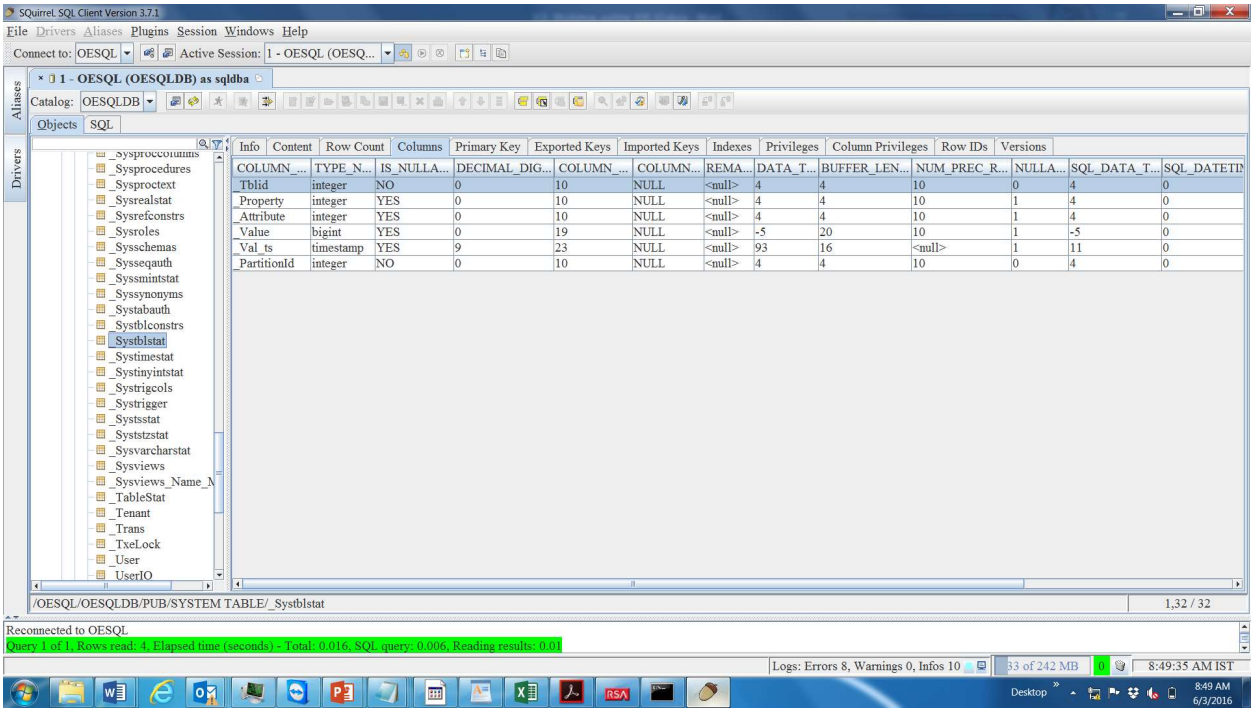
Expand the alias “OESQL”



Expand the “SYSTEM TABLE” Item.



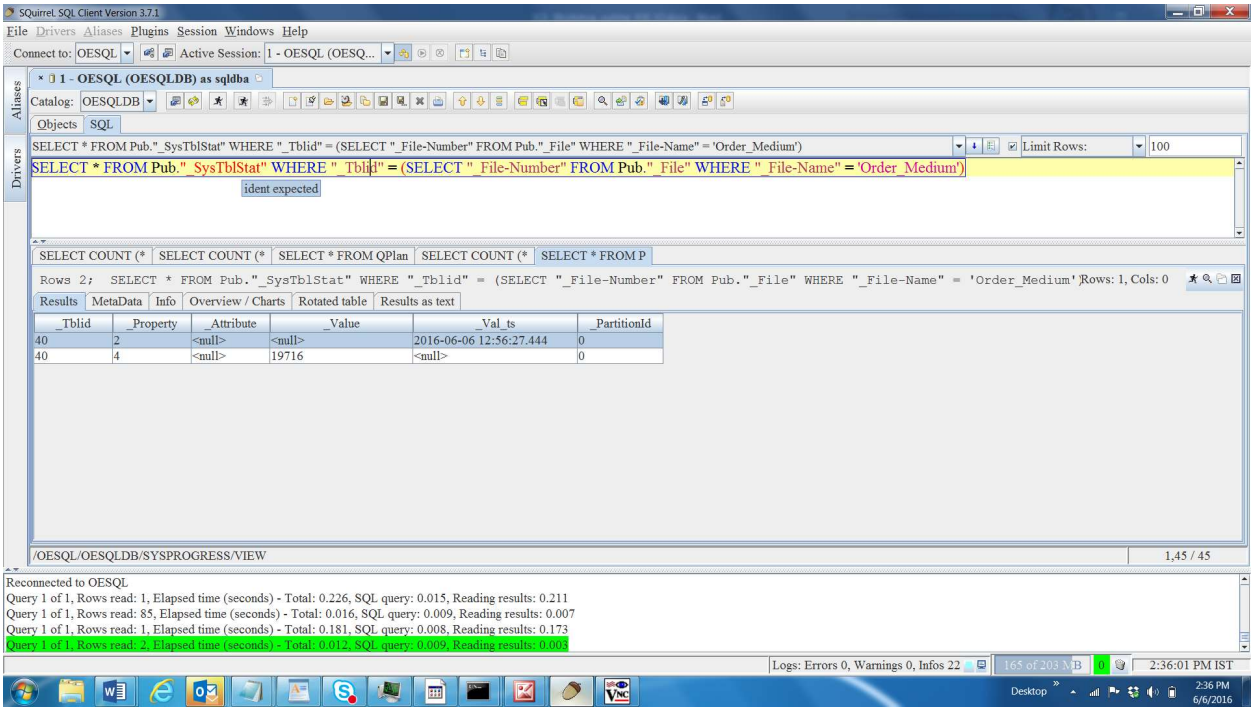
Look for Pub."_SysTblStat" and click on it. Notice the columns of this table.



Columns we are interested in are "_Tblld" & "_Value". Column "_Tblld" denotes the Table Identifier for which Table Statistics are collected and column "_Value" denotes the number of rows the table contains.

Now, go back to “SQL” Tab and run the below query to see table statistics information of table Pub.Order_Medium. The number 19716 denotes the table cardinality and other value denotes the timestamp when table statistics ran last time.

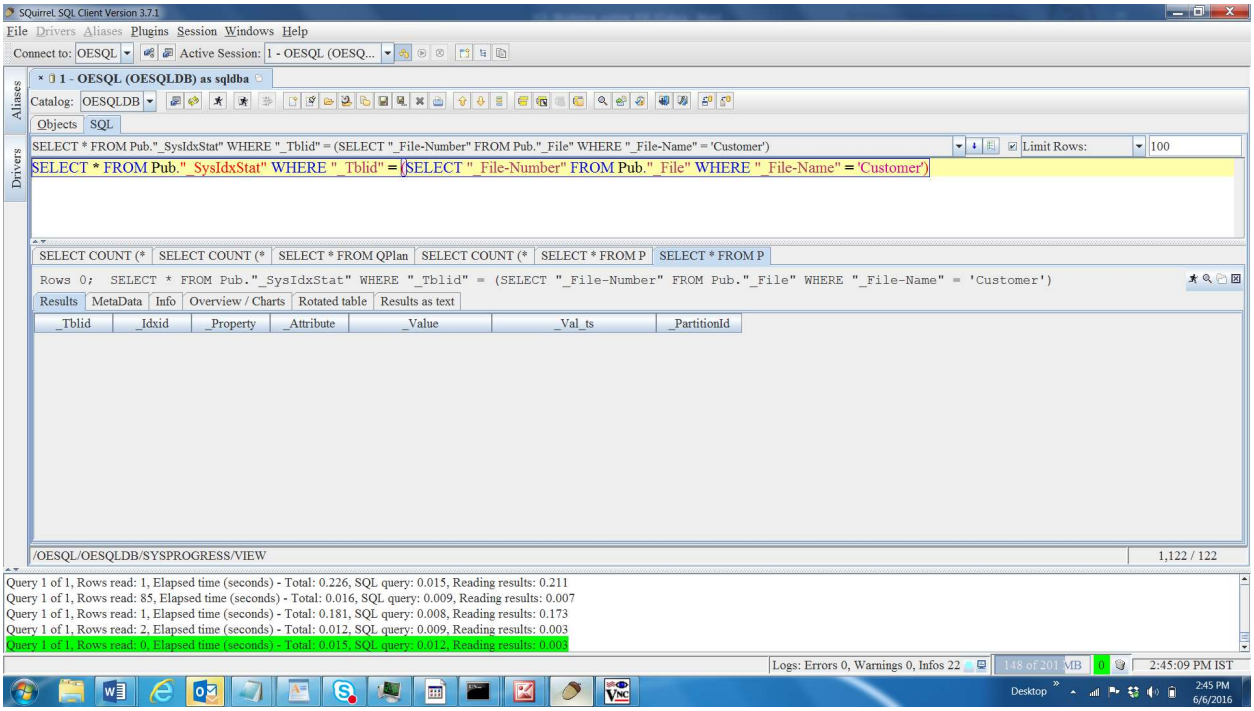
*SELECT * FROM Pub."_SysTblStat" WHERE "_Tblid" = (SELECT "_File-Number" FROM Pub."_File" WHERE "_File-Name" = 'Order_Medium');*



Now, let us see what goes into Index Statistics. The System Table which is used in this is, Pub."_SysIdxStat". Check what goes into this system table when we update Index Statistics.

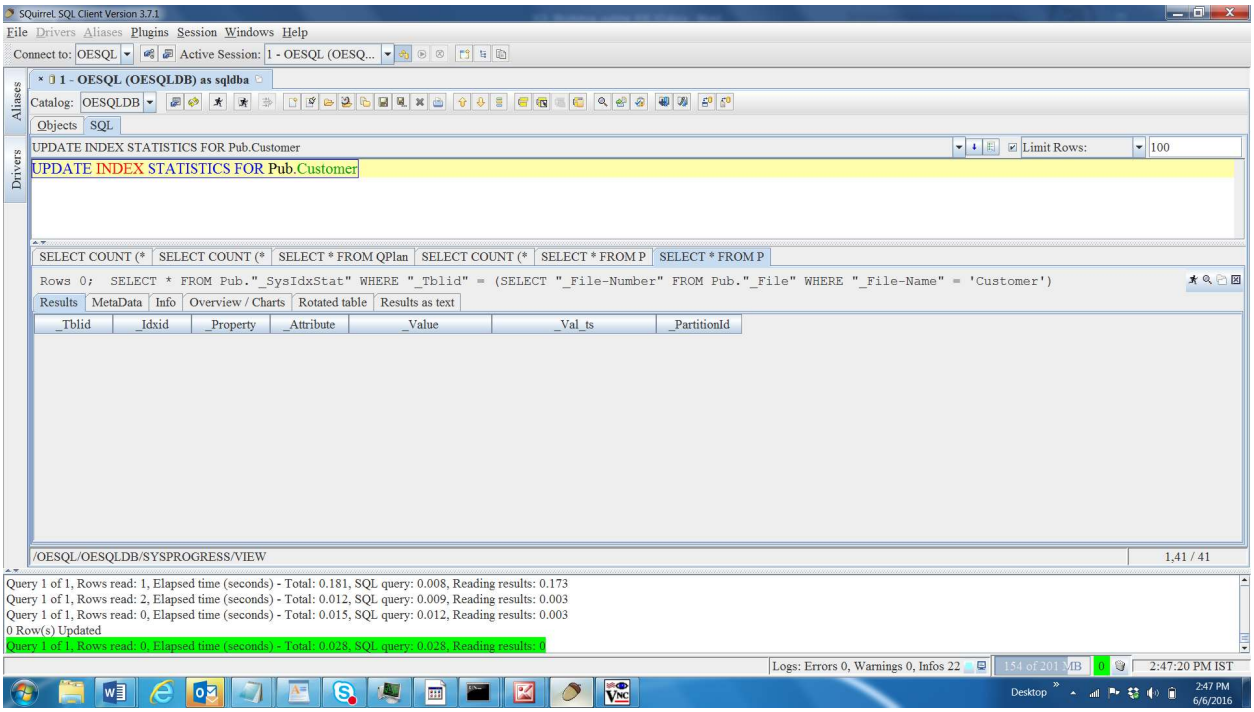
First, see that the index statistics are not present for table Pub.Customer

*SELECT * FROM Pub."_SysIdxStat" WHERE "_Tblid" = (SELECT "_File-Number" FROM Pub."_File" WHERE "_File-Name" = 'Customer');*



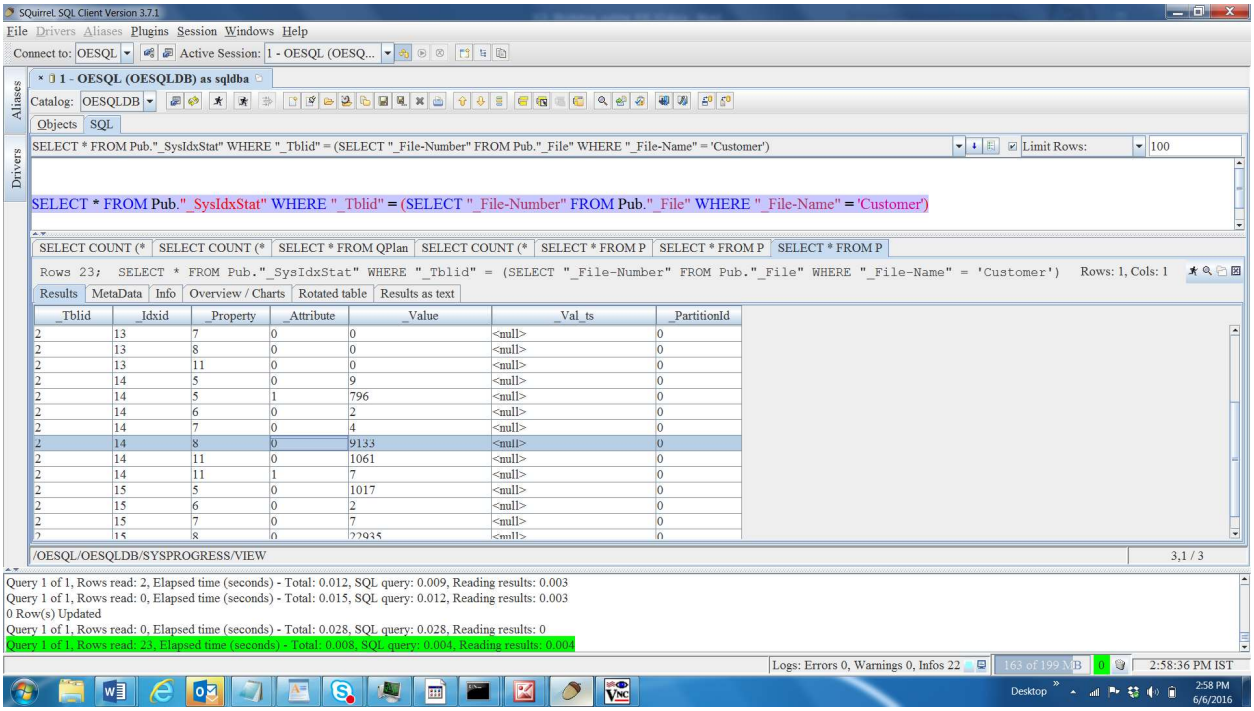
Now, let us run the index statistics for this table, Pub.Order.

UPDATE INDEX STATISTICS FOR Pub.Customer;



Now, run the same query again to see what went into Index Statistics System Table, Pub."_SysIdxStat".

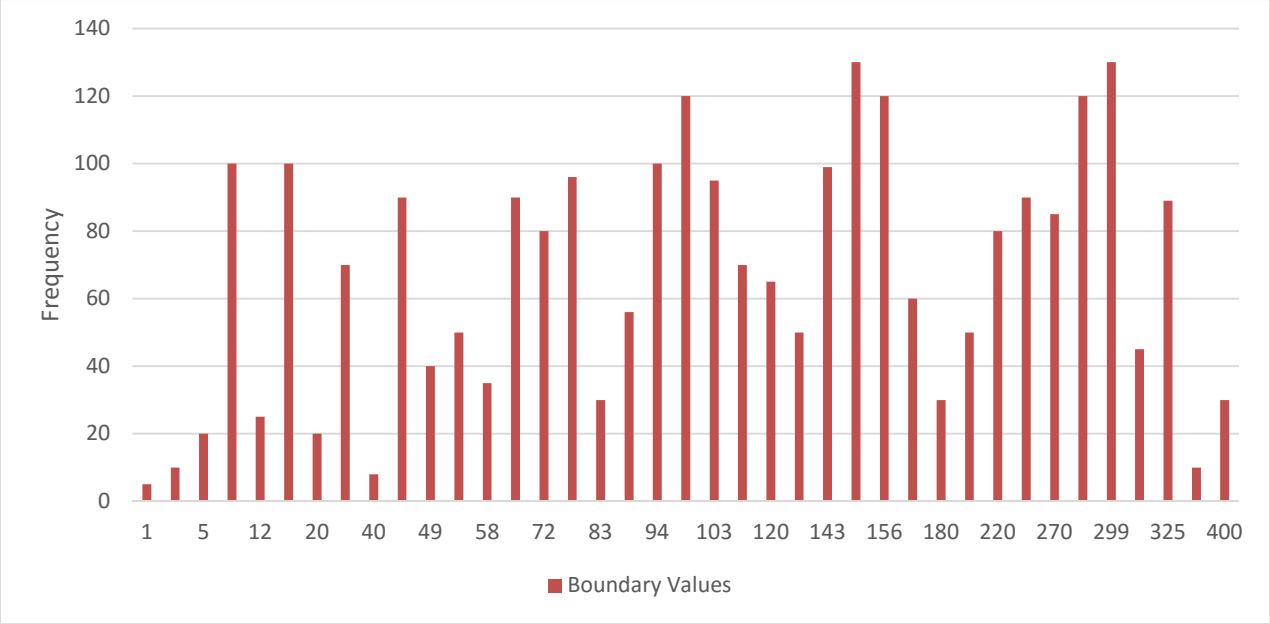
*SELECT * FROM Pub."_SysIdxStat" WHERE "_Tblid" = (SELECT "_File-Number" FROM Pub."_File" WHERE "_File-Name" = 'Customer');*



We can notice that, there are 3 indexes, each containing 5 or more than 5 properties. Property value “5” denotes number of unique values, property “6” denotes the number of levels in index, property “7” denotes the number of blocks this index occupied, property “8” denotes size of the index in bytes and property “11” denotes the max dup (highest number of duplicate values that SQL found).

Now, let us check what goes into Column Statistics. The system table involved in this is, Pub.”_SysColStat” and various system tables specific to data type of column, such as Pub.”_SysIntStat” or Pub.”_SysNumStat”. These System tables basically represents Histogram which stores information about column data. Table, Pub.”_SysColStat” stores the number of unique values in each bucket (there will be total 40 buckets as of OpenEdge 11.6) and systable pub.”_Sys<data_Type>Stat” stores the boundary values of each bucket.

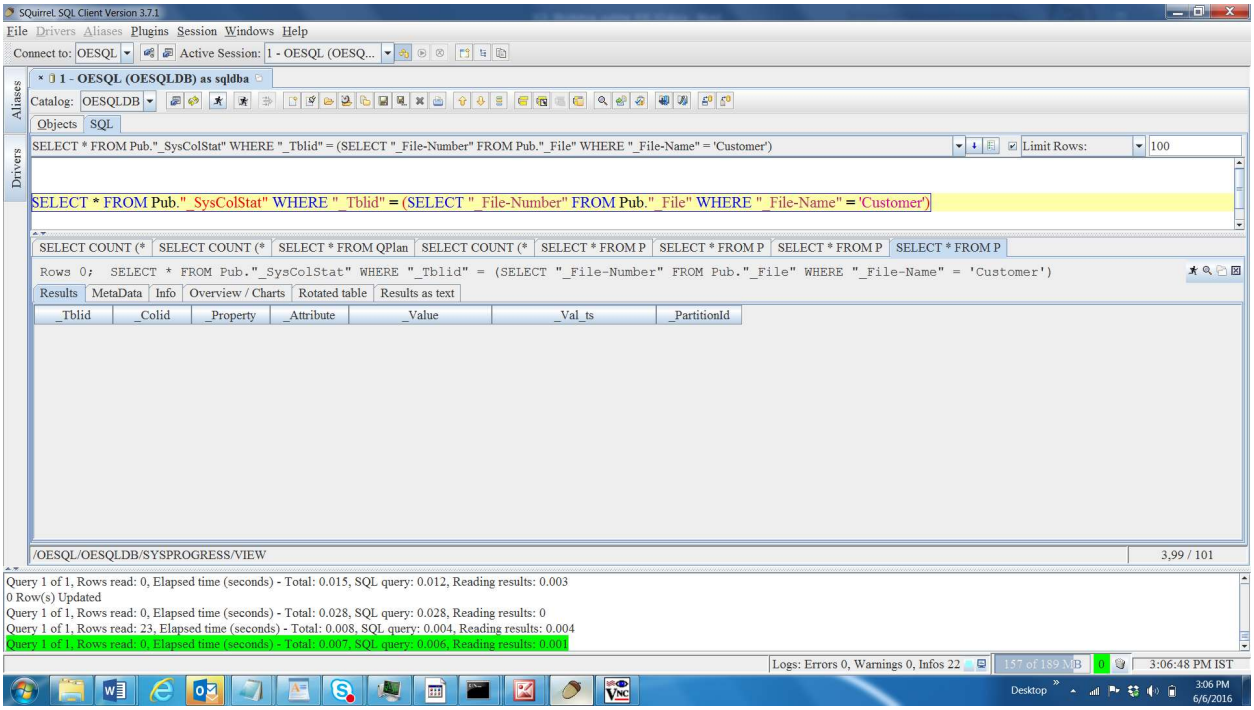
Sample histogram might look like below



Bucket Number	1	2	3	4	5	6	7	…	39	40
Unique count	1	3	5	8	12	15	20	…	380	400
Boundary value	5	10	20	100	25	100	20	…	10	30

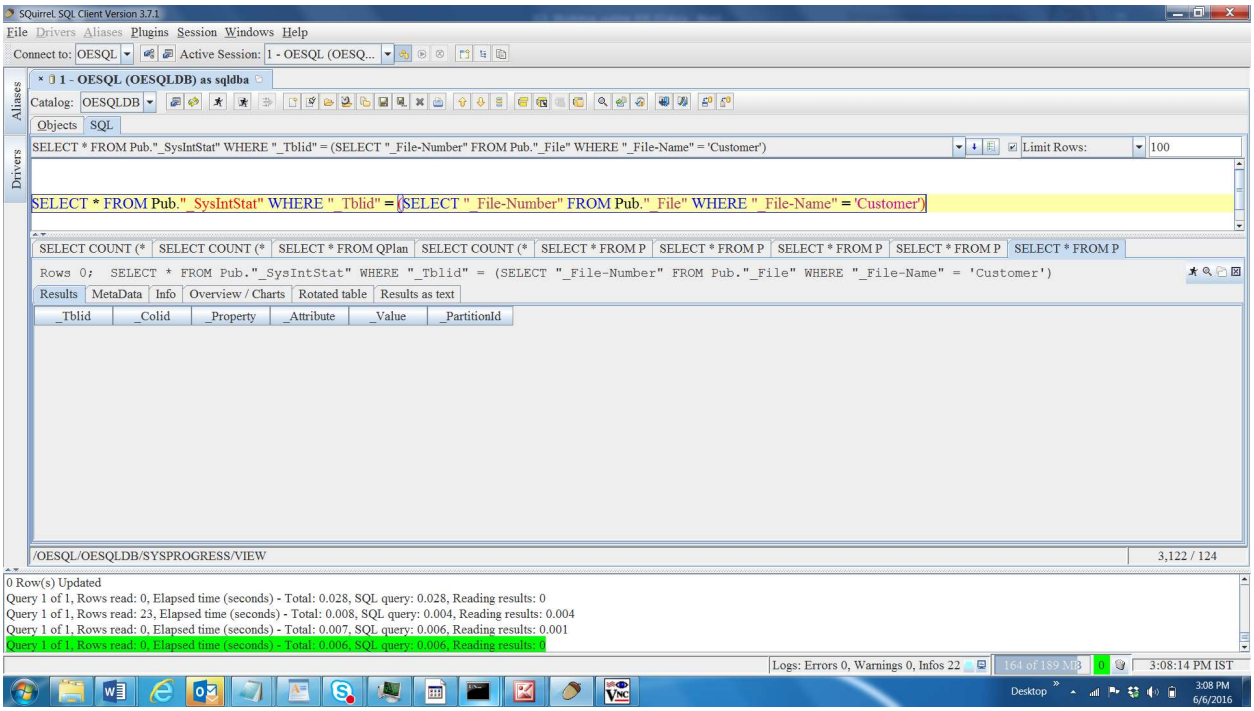
First, check that the column statistics are not present for table Pub.Customer.

*SELECT * FROM Pub."_SysColStat" WHERE "_Tblid" = (SELECT "_File-Number" FROM Pub."_File" WHERE "_File-Name" = 'Customer');*



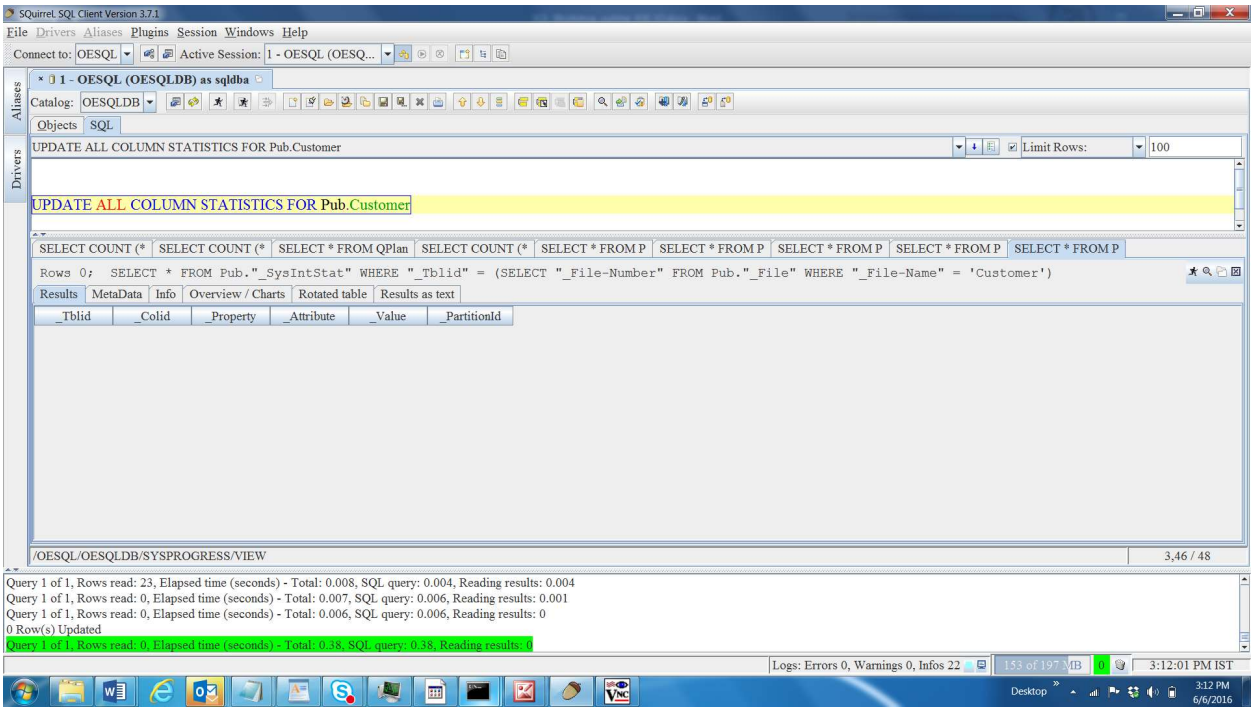
And check the table Pub.”_SysIntStat” for integer columns of Pub.Customer table.

*SELECT * FROM Pub."_SysIntStat" WHERE "_Tblid" = (SELECT "_File-Number" FROM Pub."_File" WHERE "_File-Name" = 'Customer');*



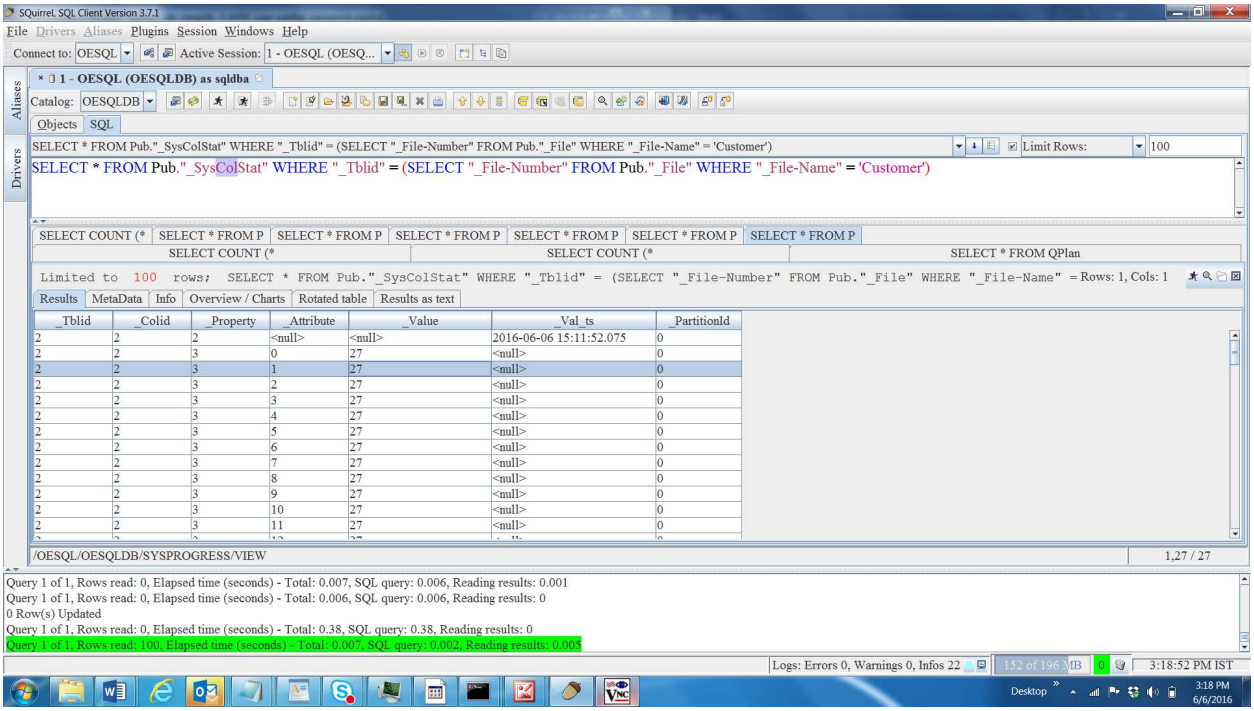
Now, update the column statistics for table Pub.Customer.

UPDATE ALL COLUMN STATISTICS FOR Pub.Customer;

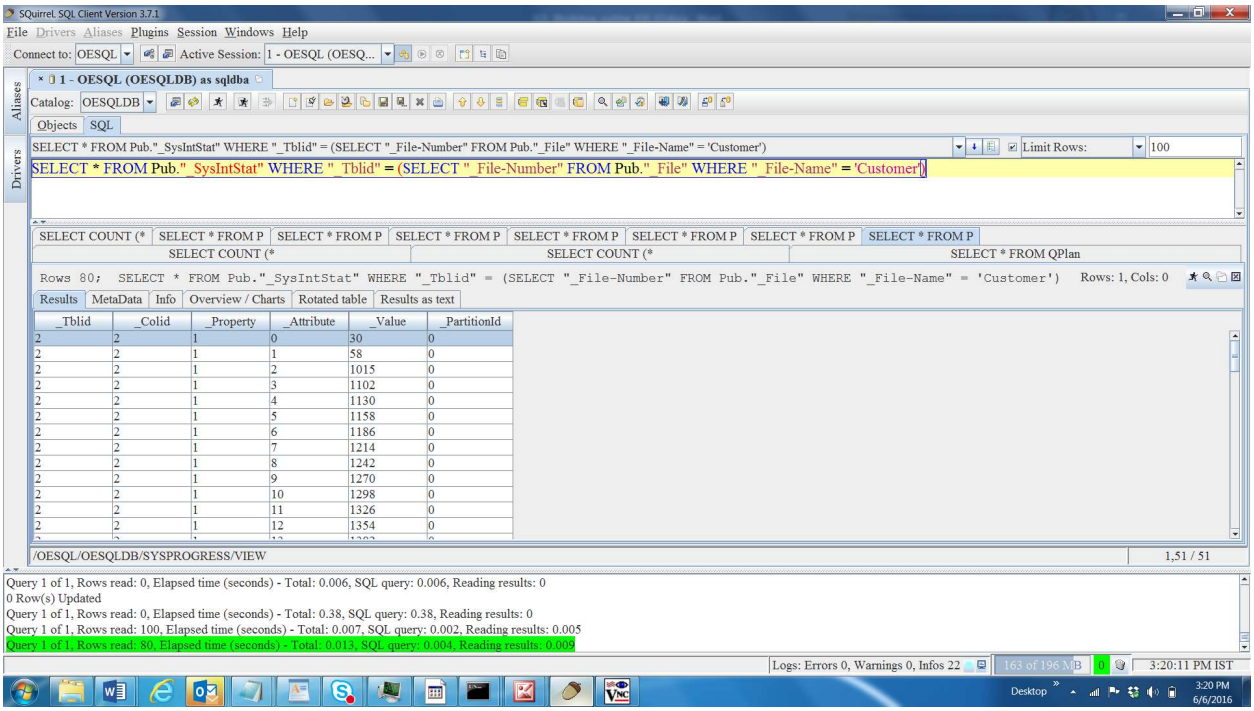


Now, run the same query to check what went into Pub."_SysColStat" and Pub."_SysIntStat" tables.

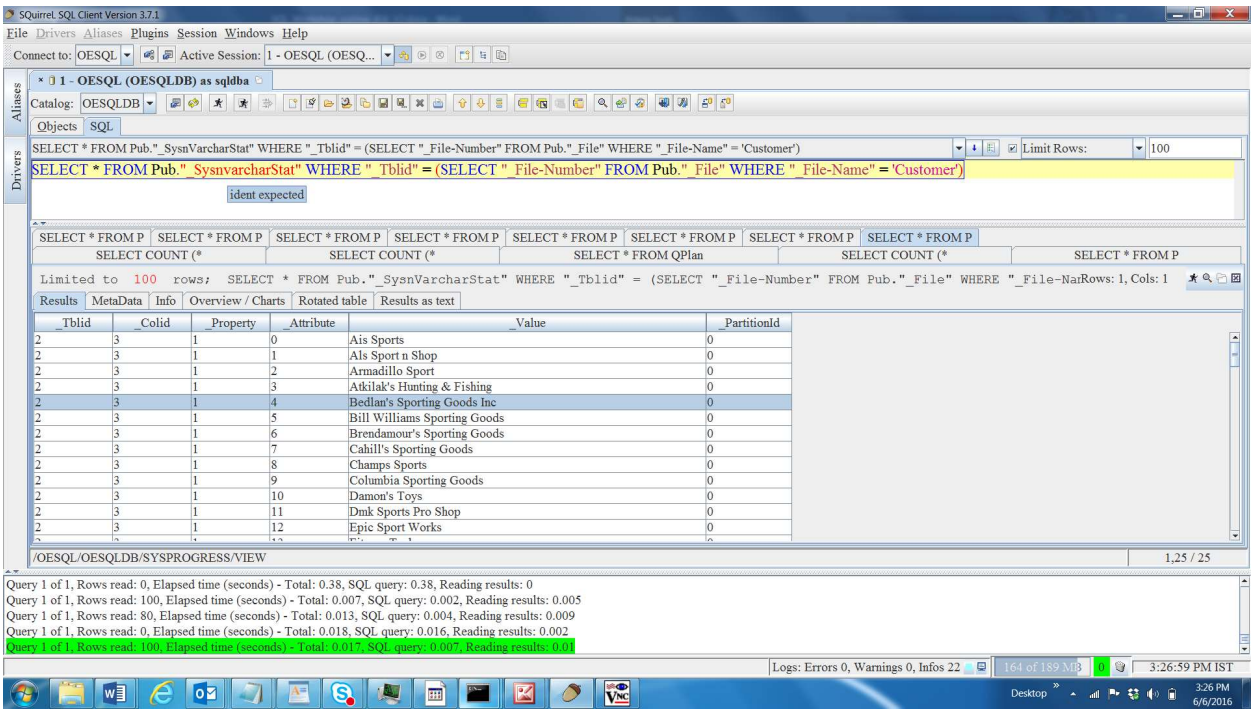
*SELECT * FROM Pub."_SysColStat" WHERE "_Tblid" = (SELECT "_File-Number" FROM Pub."_File" WHERE "_File-Name" = 'Customer');*



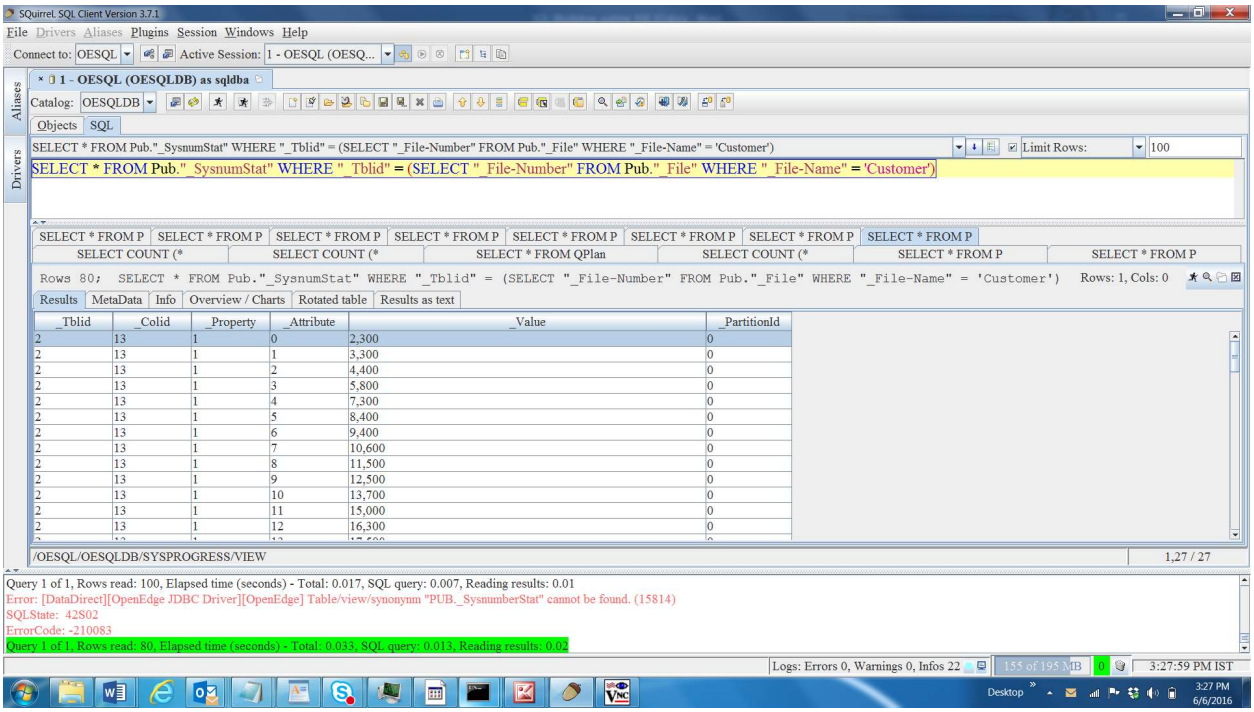
*SELECT * FROM Pub."_SysIntStat" WHERE "_Tblid" = (SELECT "_File-Number" FROM Pub."_File" WHERE "_File-Name" = 'Customer');*



*SELECT * FROM Pub."_SysnVarCharStat" WHERE "_Tblid" = (SELECT "_File-Number" FROM Pub."_File" WHERE "_File-Name" = 'Customer');*



*SELECT * FROM Pub."_SysnumStat" WHERE "_Tblid" = (SELECT "_File-Number" FROM Pub."_File" WHERE "_File-Name" = 'Customer');*



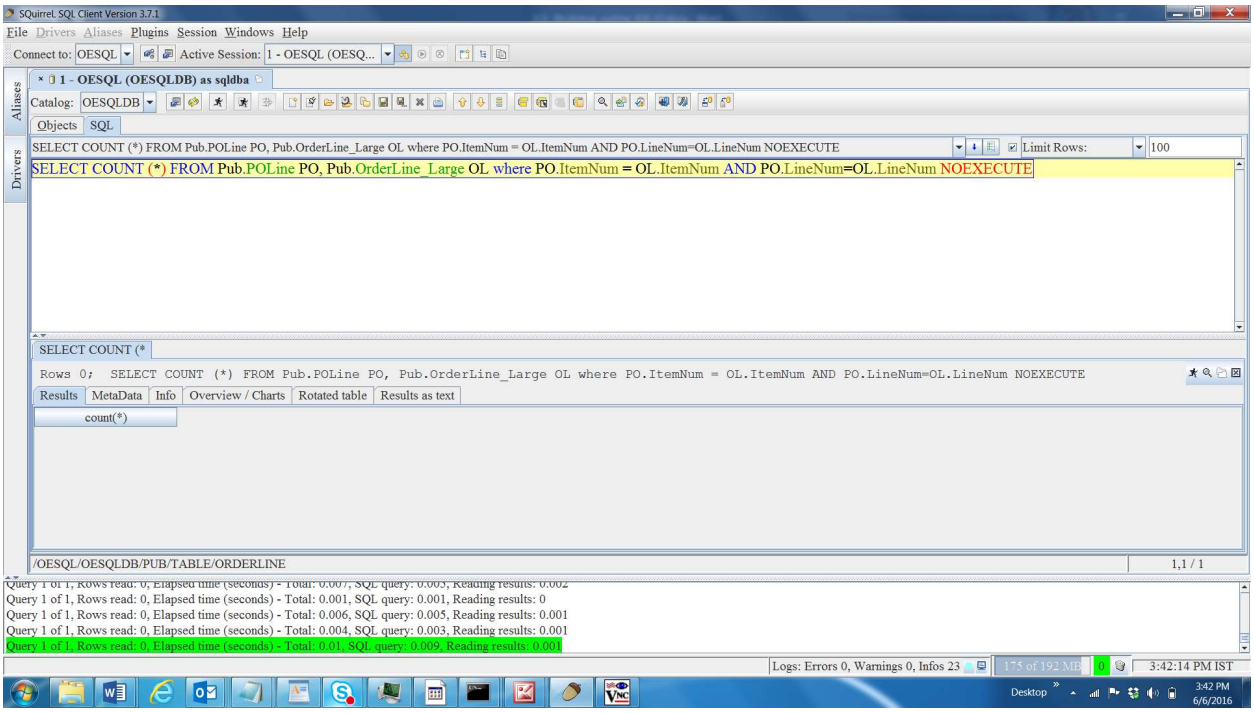
Here, "_Attribute' represents the bucket number of histogram and "_Value" represents the boundary value of that bucket.

Chapter 10 - SQL hints

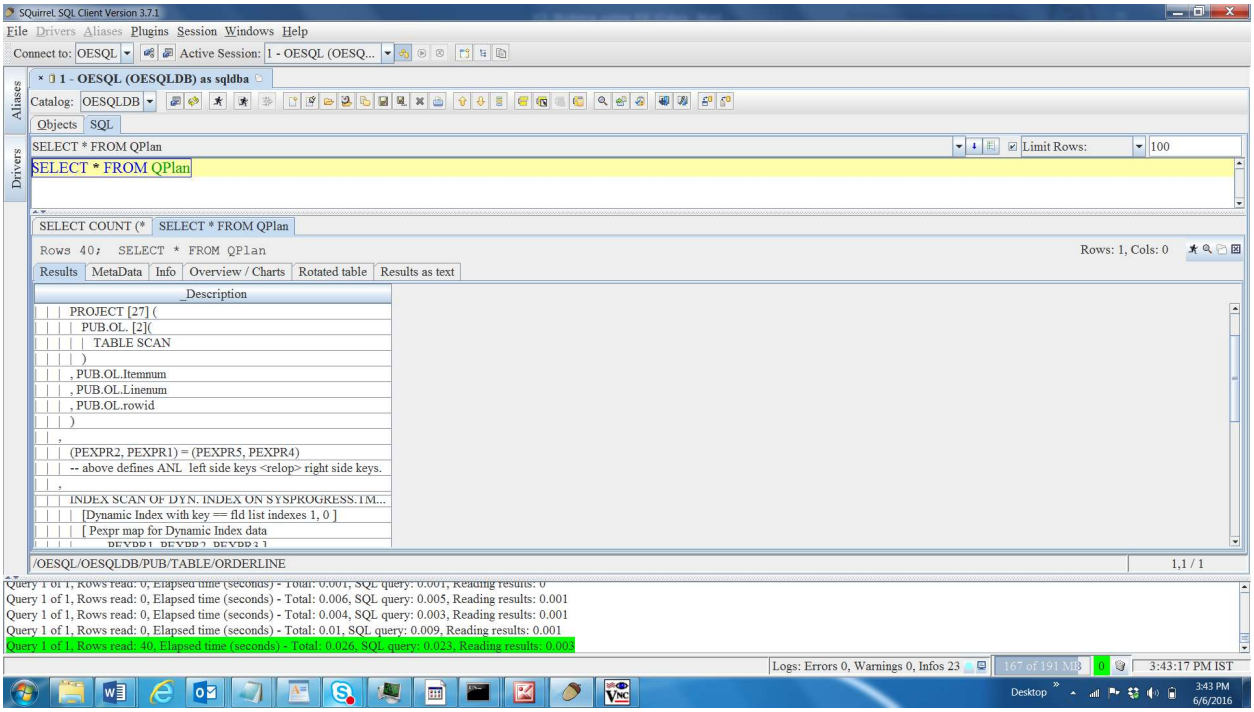
In this chapter you will learn different hints which are supported by OE-SQL and their importance in troubleshooting the performance problems.

a. NoExecute – Useful in cases, where you just want to check the query plan and do not want to execute the actual query. For example, you know some queries are running very slow and you want to know what is the plan used by such queries quickly. Below example will show the usage of “NoExecute”.

SELECT COUNT (*) FROM Pub.POLine PO, Pub.OrderLine_Large OL where PO.ItemNum = OL.ItemNum AND PO.LineNum=OL.LineNum NOEXECUTE;



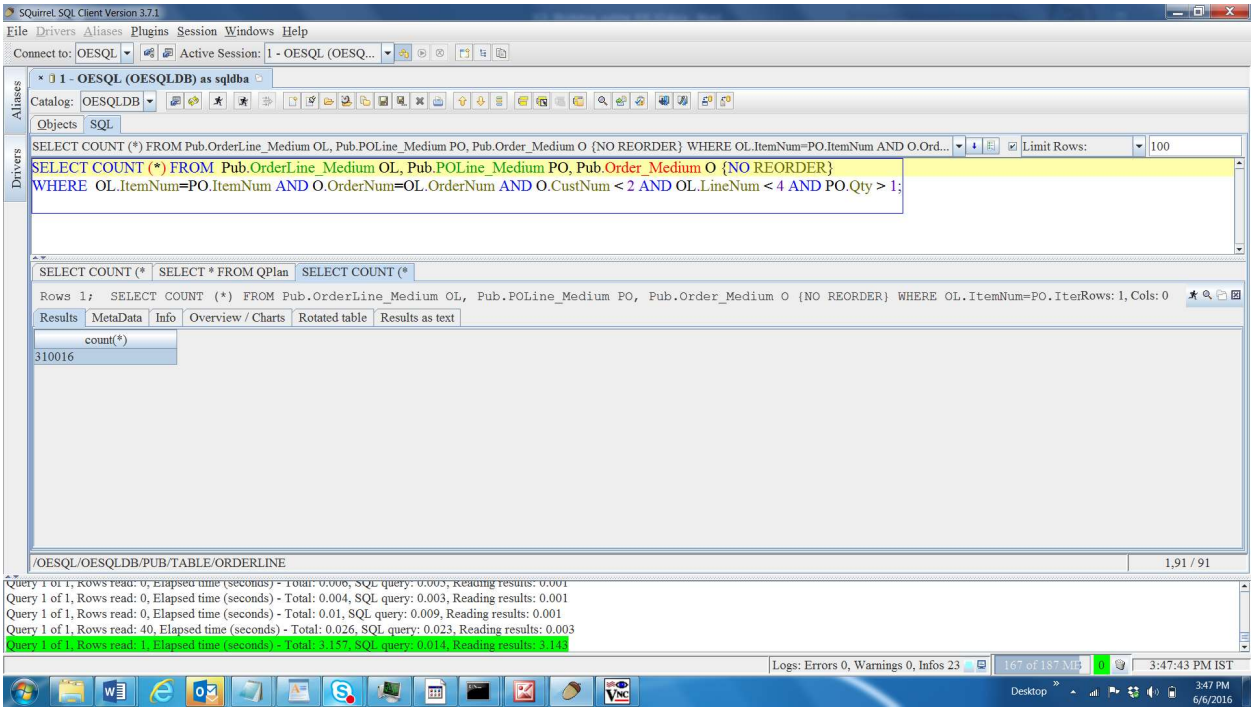
Now, you can check the query plan.



b. No reorder – Useful when you suspect that the join order used by SQL optimizer is not efficient and want to validate the query plan/execution time by forcing SQL optimizer to choose specified join order. Below example will show the usage of "No reorder".

Let us use the same query as the one we used to illustrate the effect of column statistics

SELECT COUNT (*) FROM Pub.OrderLine_Medium OL, Pub.POLine_Medium PO, Pub.Order_Medium O {NO REORDER}
WHERE OL.ItemNum=PO.ItemNum AND O.OrderNum=OL.OrderNum AND O.CustNum < 2 AND OL.LineNum < 4 AND PO.Qty > 1;



Now, if you check the query plan, it uses the given join order instead of re-ordering to create the best possible plan.

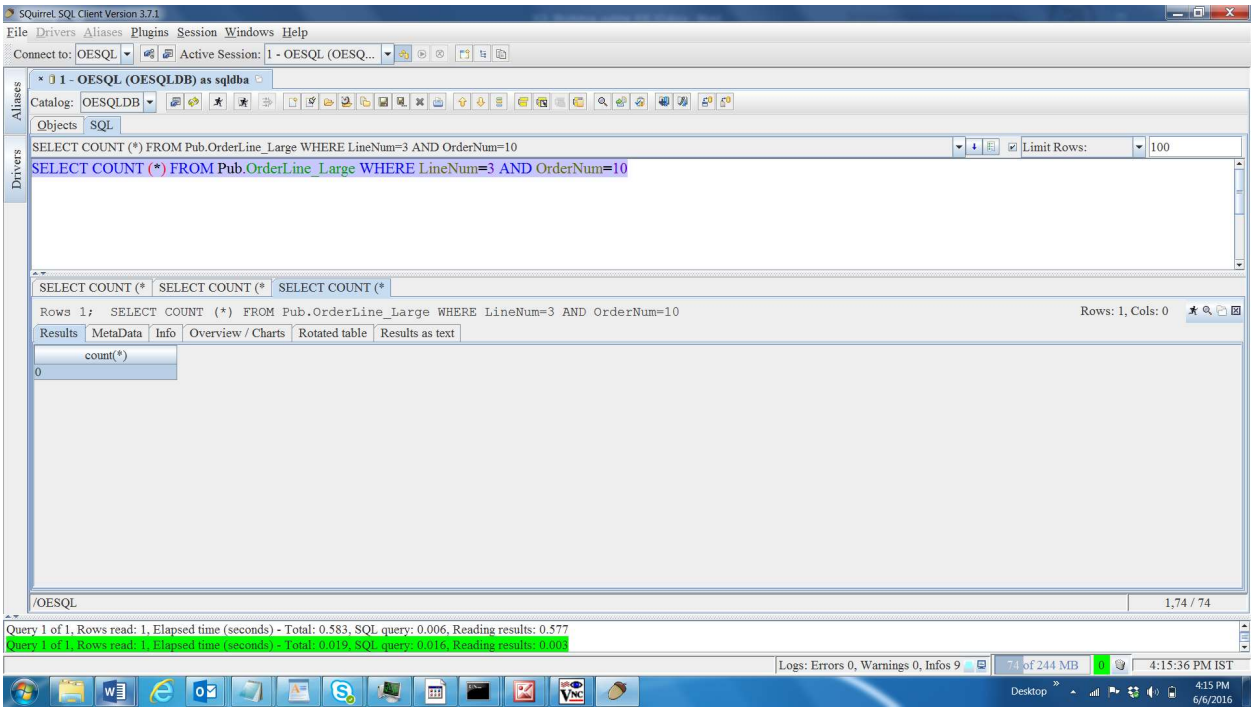
```
_Description
----------------------------------------------------
SELECT COMMAND.
PROJECT [53] (
|  PROJECT [52] (
|  |  JOIN [5][AUG_NESTED_LOOP-JOIN](
|  |  |  JOIN [3][AUG_NESTED_LOOP-JOIN](
|  |  |  |  RESTRICT [55] (
|  |  |  |  |  PROJECT [42] (
|  |  |  |  |  |  PUB.OL. [1](
|  |  |  |  |  |  |  TABLE SCAN
|  |  |  |  |  |  )
|  |  |  |  |  , PUB.OL.Itemnum
|  |  |  |  |  , PUB.OL.Ordernum
|  |  |  |  |  , PUB.OL.Linenum
|  |  |  |  |  , PUB.OL.rowid
```

```
|   |   |   |   |   )
|   |   |   |
|   |   |   |   |   (PEXPR3) < (4)
|   |   |   |   |   Evaluation callback list(
|   |   |   |   |   |       col id#   3
|   |   |   |   |   )
|   |   |   |   )
|   |   |   ,
|   |   |   |   (PEXPR1) = (PEXPR5)
|   |   |   |   -- above defines ANL  left side ke
|   |   |   ,
|   |   |   |   INDEX SCAN OF DYN. INDEX ON SYSPRO
|   |   |   |   |   [Dynamic Index with key == fld
|   |   |   |   |   [ Pexpr map for Dynamic Index
|   |   |   |   |         PEXPR1, PEXPR2, PEXPR3
|   |   |   |   |   [ Fldids
|   |   |   |   |       FldId 0, FldId 1, FldId 2
|   |   |   |   |   [Dynamic Index] TMPTBL00000026
|   |   |   |   |   |   (SYSPROGRESS.TMPTBL0000002
|   |   |   |   |   RESTRICT [54] (
|   |   |   |   |   |   PROJECT [46] (
|   |   |   |   |   |   |   PUB.PO. [2](
|   |   |   |   |   |   |   |   TABLE SCAN
|   |   |   |   |   |   |   )
|   |   |   |   |   |   , PUB.PO.Itemnum
|   |   |   |   |   |   , PUB.PO.Qty
|   |   |   |   |   |   , PUB.PO.rowid
|   |   |   |   |   |   )
|   |   |   |   |
|   |   |   |   |   |   (PEXPR2) > (1)
|   |   |   |   |   |   Evaluation callback list(
|   |   |   |   |   |   |       col id#   5
|   |   |   |   |   |   )
|   |   |   |   |   )
|   |   |   |
|   |   |   |   )
|   |   |   )
|   |   ,
|   |   |   (PEXPR2) = (PEXPR8)
|   |   |   -- above defines ANL  left side keys <
|   |   ,
|   |   |   INDEX SCAN OF DYN. INDEX ON SYSPROGRES
|   |   |   |   [Dynamic Index with key == fld lis
|   |   |   |   [ Pexpr map for Dynamic Index data
|   |   |   |         PEXPR1, PEXPR2, PEXPR3 ]
|   |   |   |   [ Fldids
|   |   |   |       FldId 0, FldId 1, FldId 2 ]
|   |   |   |   [Dynamic Index] TMPTBL00000027BL00
|   |   |   |   |   (SYSPROGRESS.TMPTBL00000025.0)
|   |   |   |   RESTRICT [56] (
|   |   |   |   |   PROJECT [50] (
|   |   |   |   |   |   PUB.O. [4](
|   |   |   |   |   |   |   TABLE SCAN
|   |   |   |   |   |   )
|   |   |   |   |   , PUB.O.Ordernum
```

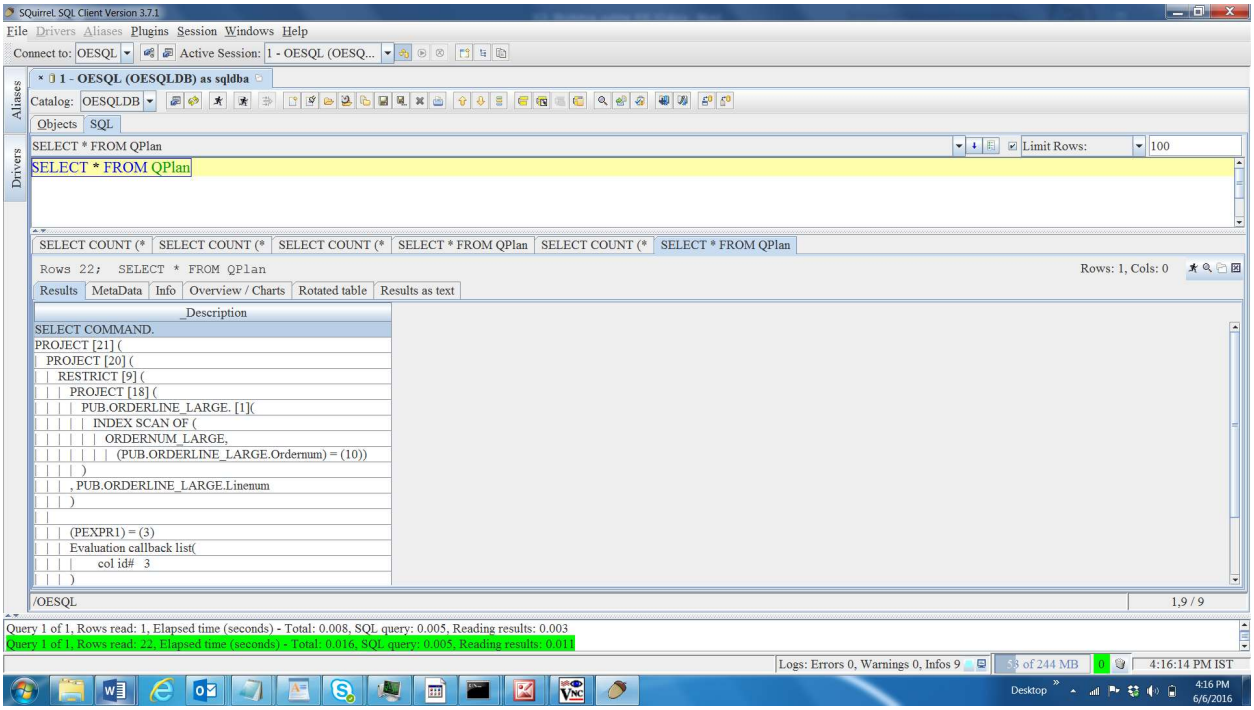
```
|  |  |  |  |  , PUB.O.CustNum
|  |  |  |  |  , PUB.O.rowid
|  |  |  |  |  )
|  |  |  |
|  |  |  |  |  (PEXPR2) < (2)
|  |  |  |  |  Evaluation callback list(
|  |  |  |  |  |      col id#  3
|  |  |  |  |  )
|  |  |  |  )
|  |  |
|  |  |  )
|  |  )
|  , count (*)
|  )
, PEXPR1
)
```

c. Index hints – Useful when you suspect that the index selected by SQL optimizer is less optimal than the index which you expect would perform better. You can specify the index which should be used by SQL optimizer using “Force Index Hints”. Below example will show the usage of “Force index hint”.

SELECT COUNT (*) FROM Pub.OrderLine_Large WHERE LineNum=3 AND OrderNum=10;

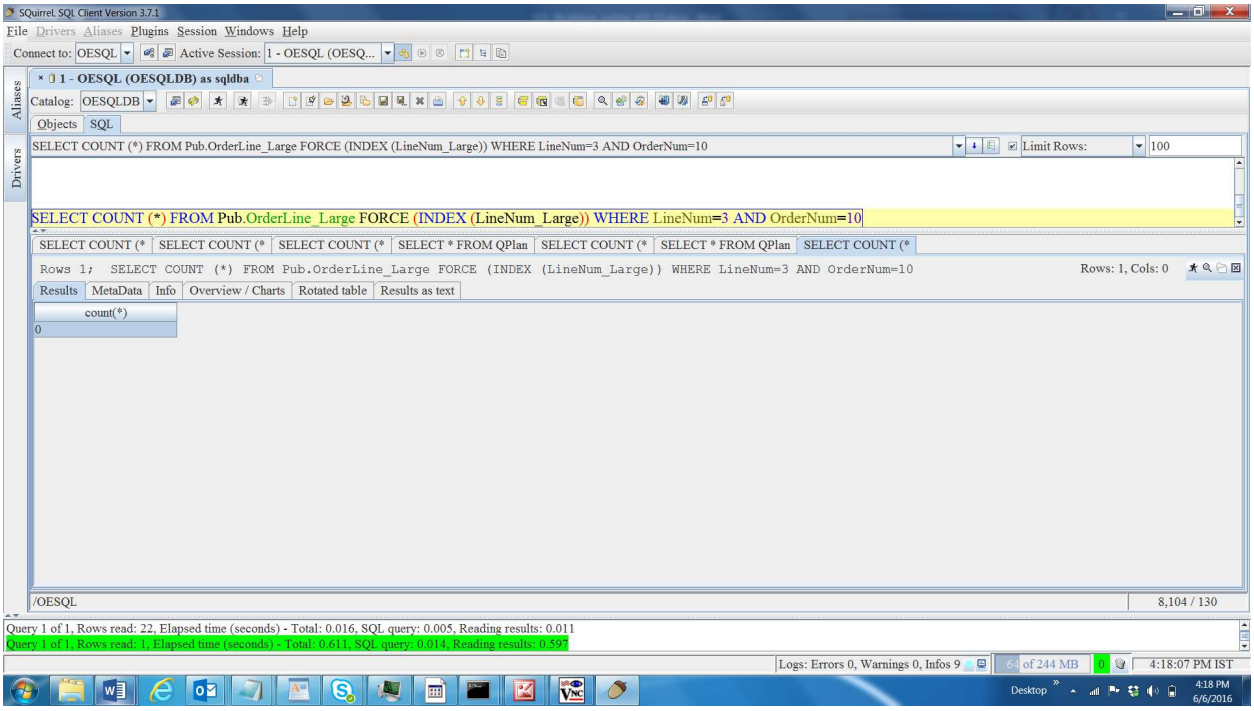


Now, check the query plan. It selected index, OrderNum_Large.

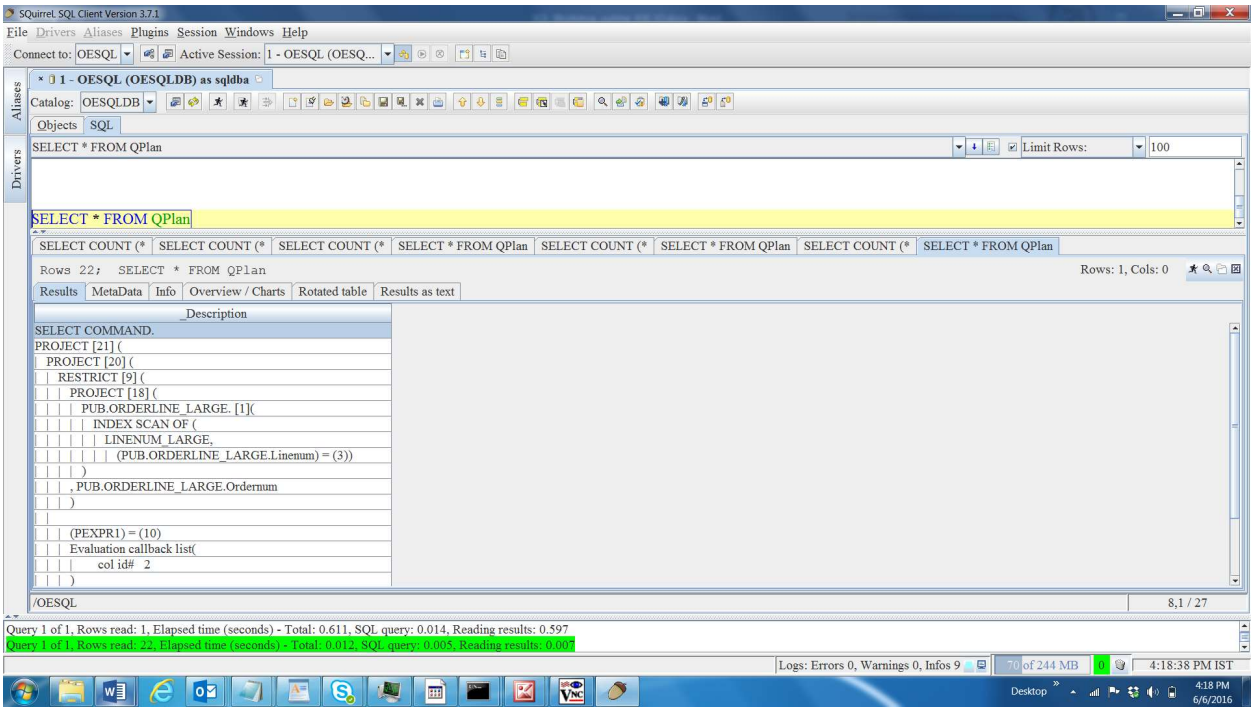


Now, run the same query with force index hints.

SELECT COUNT (*) FROM Pub.OrderLine_Large FORCE (INDEX (LineNum_Large)) WHERE LineNum=3 AND OrderNum=10;



Now, check the query plan.



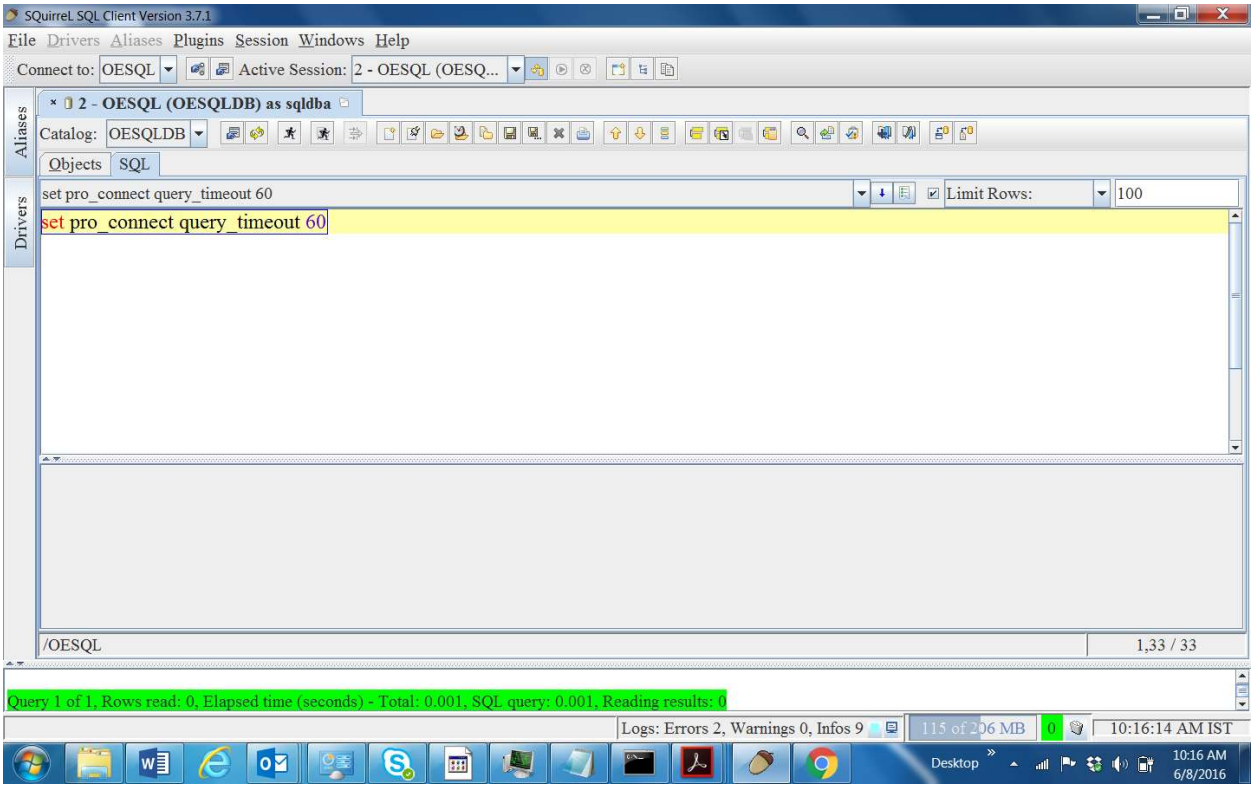
Chapter 11 – Puzzles

In this chapter you will be given two queries having performance problems which performs bad. You are supposed to solve these two performance problems by understanding the reason for bad performance and tuning your database, such that these queries performs better. Note that, obvious way of executing all type of statistics on all the involved tables is not allowed here (This will help us to understand the reason behind the slowness of the query).

Important Note:

Below queries might take more than couple of minutes to complete the execution, so make sure to execute the QUERY_TIMEOUT command, so that query aborts after 60 seconds.

SET PRO_CONNECT QUERY_TIMEOUT 60



Query 1: Below query performs very bad, how do you tune your database to make sure that, it performs better?

select top 30 po.price, po.qty, po.itemnum, po.linenum from pub.poline_large po, pub.orderline_large ol where po.linenum=ol.linenum and po.itemnum = ol.itemnum order by po.price, po.qty;

Query 2: Below query performs very bad, how do you tune your database to make sure that, it performs better?

select count (*) from pub.order_10k O, pub.orderline_10k ol, pub.customer_10k c where ol.country=c.country and o.custnum=c.custnum and o.ordernum < 2;