**A**

**PRACTICAL TRAINING PROJECT REPORT**

**ON**

**Ratio of Cost of Job**

**Group Name: SQL-2**

***In partial fulfillment of***

**Bachelor of Technology**

# in

**Computer Engineering**

by

**Gaurav Mishra (PCE18CS057), Rishabh Jain (PCE18CS133)**

**Nupur Khatri (PCE18CS107), Ashutosh Tiwari (PCE18CS034),**

**Ishika Solanki (PCE18CS067)**

under the guidance of

**Mr. HEMANT PAREEK (JIET)**

****

(Session 2020-21)

**Department of Computer Engineering**

**Poornima College of Engineering**

# **ACKNOWLEDGEMENT**

I would like to convey my profound sense of reverence and admiration to my supervisor **Mr. Hemant Pareek, Jodhpur Institute of Engineering & Technology,** for his intense concern, attention, priceless direction, guidance and encouragement throughout this research work.

I am grateful to **Dr. Mahesh Bundele**, Director of Poornima College of Engineering for his helping attitude with a keen interest in completing this dissertation in time.

I extend my heartiest gratitude to all the teachers, who extended their cooperation to steer the topic towards its successful completion. I am also thankful to non-teaching staff of the department to support in preparation of this dissertation work.

My special heartfelt gratitude goes to **Dr. Surendra Kumar Yadav, HOD, Computer Science department, Mr. Hemant Pareek, Project Coordinator, Computer Science Department, Poornima College of Engineering**, for unvarying support, guidance and motivation during the course of this research.

I would like to express my deep sense of gratitude towards management of Poornima College of Engineering including **Dr. S, M. Seth**, Chairman Emeritus, Poornima Group and former Director NIH, Roorkee, **Shri Shashikant Singhi**, Chairman, Poornima Group, **Mr. M. K. M. Shah**, Director Admin & Finance, Poornima Group and **Ar. Rahul Singhi**, Director

Poornima Group for establishment of institute and providing facilities on studies.

I would like to take the opportunity of expressing my thanks to all faculty members of the Department, for their kind support, technical guidance, and inspiration throughout the course.

I am deeply thankful to my parents and all other family members for their blessings and inspiration. At last but not least I would like to give special thanks to God who enabled me to complete my dissertation on time.

**Gaurav Mishra (PCE18CS057), Rishabh Jain (PCE18CS133),**

**Nupur Khatri (PCE18CS107), Ashutosh Tiwari (PCE18CS034),**

**Ishika Solanki (PCE18CS067)**

**Computer Science Department**

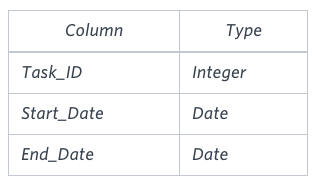
# **TABLE OF CONTENT**

|  |  |  |
| --- | --- | --- |
| **Chapter Number** | **Description** | **Page Number** |
| Title Page | | 1 |
| Acknowledgement | | 2 |
| Table of Contents | | 3 |
| **Project 1** | **Task 1** | 4 |
| **Task 2** | 6 |
| **Task 4** | 9 |
| **Project 2** | **Task 1** | 14 |
| **Task 2** | 18 |
| **Task 3** | 19 |
| **Task 4** | 22 | |
| **Task 5** | 25 | |
| **Project 3** | **Task 1** | 28 | |
| **Task 2** | 29 | |
| **Task 3** | 30 | |
| **Task 4** | 31 |
| **Task 5** | 31 |
| **Task 6** | 32 |
| **Task 7** | 33 |
| **Task 8** | 33 |
| **Task 9** | 34 |
| **Project 4** | **Task 1** | 47 |
| **Task 2** | 55 |
| **Task 4** | 58 |
| **Task 5** | 59 |
| **Conclusion** | | 64 |

**Project 1**

**Task 1:-**

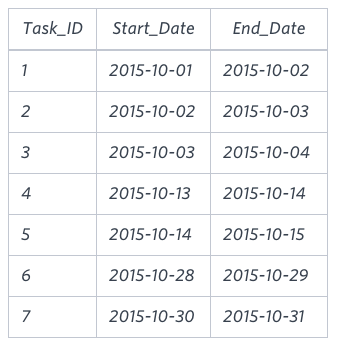
You are given a table, *Projects*, containing three columns: *Task\_ID*, *Start\_Date* and *End\_Date*. It is guaranteed that the difference between the *End\_Date* and the *Start\_Date* is equal to *1* day for each row in the table.



If the *End\_Date* of the tasks are consecutive, then they are part of the same project. Samantha is interested in finding the total number of different projects completed.

Write a query to output the start and end dates of projects listed by the number of days it took to complete the project in ascending order. If there is more than one project that have the same number of completion days, then order by the start date of the project.

**Sample Input**



**Sample Output**

2015-10-28 2015-10-29  
2015-10-30 2015-10-31  
2015-10-13 2015-10-15  
2015-10-01 2015-10-04

**Explanation**

The example describes following *four* projects:

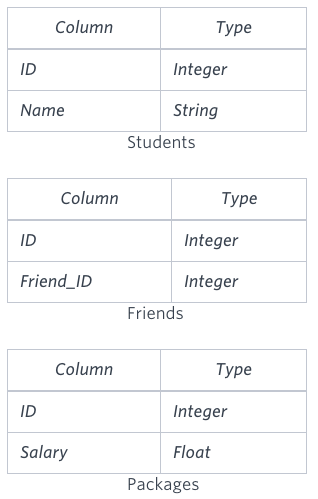
* *Project 1*: Tasks *1*, *2* and *3* are completed on consecutive days, so these are part of the project. Thus start date of project is *2015–10–01* and end date is *2015–10–04*, so it took *3 days* to complete the project.
* *Project 2*: Tasks *4* and *5* are completed on consecutive days, so these are part of the project. Thus, the start date of project is *2015–10–13* and end date is *2015–10–15*, so it took *2 days* to complete the project.
* *Project 3*: Only task *6* is part of the project. Thus, the start date of project is *2015–10–28* and end date is *2015–10–29*, so it took *1 day* to complete the project.
* *Project 4*: Only task *7* is part of the project. Thus, the start date of project is *2015–10–30* and end date is *2015–10–31*, so it took *1 day* to complete the project.

# **Solution :-**

SELECT Start\_Date, min(End\_Date)  
FROM   
(SELECT Start\_Date FROM Projects WHERE Start\_Date NOT IN (SELECT End\_Date FROM Projects)) a ,  
(SELECT End\_Date FROM Projects WHERE End\_Date NOT IN (SELECT Start\_Date FROM Projects)) b  
WHERE Start\_Date < End\_Date  
GROUP BY Start\_Date  
ORDER BY DATEDIFF(min(End\_Date), Start\_Date) ASC, Start\_Date ASC;

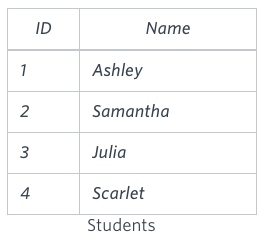
**Task 2:-**

You are given three tables: *Students*,*Friends*and*Packages.* *Students* contains two columns: *ID* and *Name*. *Friends* contains two columns: *ID* and *Friend\_ID* (*ID* of the ONLY best friend). *Packages* contains two columns: *ID* and *Salary* (offered salary in $ thousands per month).



Write a query to output the names of those students whose best friends got offered a higher salary than them. Names must be ordered by the salary amount offered to the best friends. It is guaranteed that no two students got same salary offer.

**Sample Input**



**Sample Output**

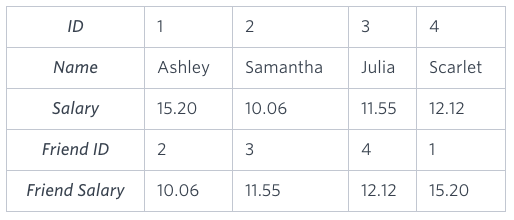
Samantha

Julia

Scarlet

**Explanation**

See the following table:



Now,

* Samantha's best friend got offered a higher salary than her at 11.55
* Julia's best friend got offered a higher salary than her at 12.12
* Scarlet's best friend got offered a higher salary than her at 15.2
* Ashley's best friend did NOT get offered a higher salary than her

The name output, when ordered by the salary offered to their friends, will be:

* Samantha
* Julia
* Scarlet

**Solution:-**  
  
Select S.Name  
From ( Students S join Friends F Using(ID)  
 join Packages P1 on S.ID=P1.ID  
 join Packages P2 on F.Friend\_ID=P2.ID)  
Where P2.Salary > P1.Salary  
Order By P2.Salary;

**Task 4:-**

1. SELECT CON.CONTEST\_ID,
2. CON.HACKER\_ID,
3. CON.NAME,
4. SUM(TOTAL\_SUBMISSIONS),
5. SUM(TOTAL\_ACCEPTED\_SUBMISSIONS),
6. SUM(TOTAL\_VIEWS),
7. SUM(TOTAL\_UNIQUE\_VIEWS)
8. FROM CONTESTS CON
9. JOIN COLLEGES COL ON CON.CONTEST\_ID = COL.CONTEST\_ID
10. JOIN CHALLENGES CHA ON COL.COLLEGE\_ID = CHA.COLLEGE\_ID
11. LEFT JOIN
12. (SELECT CHALLENGE\_ID,
13. SUM(TOTAL\_VIEWS) AS TOTAL\_VIEWS,
14. SUM(TOTAL\_UNIQUE\_VIEWS) AS TOTAL\_UNIQUE\_VIEWS
15. FROM VIEW\_STATS
16. GROUP BY CHALLENGE\_ID) VS ON CHA.CHALLENGE\_ID = VS.CHALLENGE\_ID
17. LEFT JOIN
18. (SELECT CHALLENGE\_ID,
19. SUM(TOTAL\_SUBMISSIONS) AS TOTAL\_SUBMISSIONS,
20. SUM(TOTAL\_ACCEPTED\_SUBMISSIONS) AS TOTAL\_ACCEPTED\_SUBMISSIONS
21. FROM SUBMISSION\_STATS
22. GROUP BY CHALLENGE\_ID) SS ON CHA.CHALLENGE\_ID = SS.CHALLENGE\_ID
23. GROUP BY CON.CONTEST\_ID,
24. CON.HACKER\_ID,
25. CON.NAME
26. HAVING SUM(TOTAL\_SUBMISSIONS) != 0
27. OR SUM(TOTAL\_ACCEPTED\_SUBMISSIONS) != 0
28. OR SUM(TOTAL\_VIEWS) != 0
29. OR SUM(TOTAL\_UNIQUE\_VIEWS) != 0
30. ORDER BY CONTEST\_ID;

The contest 66406 is used in the college 11219. In this college 11219, challenges 18765 and 47127 are asked, so from the view and submission stats:

* Sum of total submissions = 27 + 56 + 28 = 111
* Sum of total accepted submissions = 10 + 18 + 11 = 39
* Sum of total views = 43 + 72 + 26 + 15 = 156
* Sum of total unique views = 10 + 13 + 19 + 14 = 56

Similarly, we can find the sums for contests 66556 and 94828.

**Explanation**

The contest 66406 is used in the college 11219. In this college 11219, challenges 18765 and 47127 are asked, so from the view and submission stats:

* Sum of total submissions = 27 + 56 + 28 = 111
* Sum of total accepted submissions = 10 + 18 + 11 = 39
* Sum of total views = 43 + 72 + 26 + 15 = 156
* Sum of total unique views = 10 + 13 + 19 + 14 = 56

Similarly, we can find the sums for contests 66556 and 94828.

### **Analysis**

For Submission\_Stats and View\_Stats table, we can group by challenge\_id to get sum of total\_submissions, total\_accepted\_submissions, total\_views and total\_unique\_views of each challenge. Then we can join results with Contests, Collenges, Challenges tables and group by contest\_id, hacker\_id, name to get the sum we want for each contest. Finally, exclude results whose four sums are 0 and sort by contest\_id.

First, group Submission\_Stats table by challenge\_id to get sum of total\_submissions and total\_accepted\_submissions of each challenge:

**SELECT ss.challenge\_id, SUM(ss.total\_submissions) AS total\_submissions, SUM(ss.total\_accepted\_submissions) AS total\_accepted\_submissions FROM Submission\_Stats AS ss GROUP BY ss.challenge\_id;**

|  |  |
| --- | --- |
|  |  |

Similarly for View\_Stats table:

**SELECT vs.challenge\_id, SUM(vs.total\_views) AS total\_views, SUM(vs.total\_unique\_views) AS total\_unique\_views FROM View\_Stats AS vs GROUP BY vs.challenge\_id;**

Then, join results with Contests, Colleges, Challenges tables and group by contest\_id, hacker\_id, name to get sum of total\_submissions, total\_accepted\_submissions, total\_views and total\_unique\_views for each contest:

**SELECT con.contest\_id, con.hacker\_id, con.name,**

**SUM(sg.total\_submissions), SUM(sg.total\_accepted\_submissions),**

**SUM(vg.total\_views), SUM(vg.total\_unique\_views)**

**FROM Contests AS con**

**JOIN Colleges AS col ON con.contest\_id = col.contest\_id**

**JOIN Challenges AS cha ON cha.college\_id = col.college\_id**

**LEFT JOIN**

**(SELECT ss.challenge\_id, SUM(ss.total\_submissions) AS total\_submissions, SUM(ss.total\_accepted\_submissions) AS total\_accepted\_submissions FROM Submission\_Stats AS ss GROUP BY ss.challenge\_id) AS sg**

**ON cha.challenge\_id = sg.challenge\_id**

**LEFT JOIN**

**(SELECT vs.challenge\_id, SUM(vs.total\_views) AS total\_views, SUM(vs.total\_unique\_views) AS total\_unique\_views**

**FROM View\_Stats AS vs GROUP BY vs.challenge\_id) AS vg**

**ON cha.challenge\_id = vg.challenge\_id**

**GROUP BY con.contest\_id, con.hacker\_id, con.name;**

Exclude results if four sums are 0:

**SELECT con.contest\_id, con.hacker\_id, con.name,**

**SUM(sg.total\_submissions), SUM(sg.total\_accepted\_submissions),**

**SUM(vg.total\_views), SUM(vg.total\_unique\_views)**

**FROM Contests AS con**

**JOIN Colleges AS col ON con.contest\_id = col.contest\_id**

**JOIN Challenges AS cha ON cha.college\_id = col.college\_id**

**LEFT JOIN**

**(SELECT ss.challenge\_id, SUM(ss.total\_submissions) AS total\_submissions, SUM(ss.total\_accepted\_submissions) AS total\_accepted\_submissions FROM Submission\_Stats AS ss GROUP BY ss.challenge\_id) AS sg**

**ON cha.challenge\_id = sg.challenge\_id**

**LEFT JOIN**

**(SELECT vs.challenge\_id, SUM(vs.total\_views) AS total\_views, SUM(vs.total\_unique\_views) AS total\_unique\_views**

**FROM View\_Stats AS vs GROUP BY vs.challenge\_id) AS vg**

**ON cha.challenge\_id = vg.challenge\_id**

**GROUP BY con.contest\_id, con.hacker\_id, con.name**

**HAVING SUM(sg.total\_submissions) +**

**SUM(sg.total\_accepted\_submissions) +**

**SUM(vg.total\_views) +**

**SUM(vg.total\_unique\_views) > 0;**

At last, sort by contest\_id:

**SELECT con.contest\_id, con.hacker\_id, con.name,**

**SUM(sg.total\_submissions), SUM(sg.total\_accepted\_submissions),**

**SUM(vg.total\_views), SUM(vg.total\_unique\_views)**

**FROM Contests AS con**

**JOIN Colleges AS col ON con.contest\_id = col.contest\_id**

**JOIN Challenges AS cha ON cha.college\_id = col.college\_id**

**LEFT JOIN**

**(SELECT ss.challenge\_id, SUM(ss.total\_submissions) AS total\_submissions, SUM(ss.total\_accepted\_submissions) AS total\_accepted\_submissions FROM Submission\_Stats AS ss GROUP BY ss.challenge\_id) AS sg**

**ON cha.challenge\_id = sg.challenge\_id**

**LEFT JOIN**

**(SELECT vs.challenge\_id, SUM(vs.total\_views) AS total\_views, SUM(vs.total\_unique\_views) AS total\_unique\_views**

**FROM View\_Stats AS vs GROUP BY vs.challenge\_id) AS vg**

**ON cha.challenge\_id = vg.challenge\_id**

**GROUP BY con.contest\_id, con.hacker\_id, con.name**

**HAVING SUM(sg.total\_submissions) +**

**SUM(sg.total\_accepted\_submissions) +**

**SUM(vg.total\_views) +**

**SUM(vg.total\_unique\_views) > 0**

### **ORDER BY con.contest\_id;**

### **Solution**

**SELECT con.contest\_id, con.hacker\_id, con.name,**

**SUM(sg.total\_submissions), SUM(sg.total\_accepted\_submissions),**

**SUM(vg.total\_views), SUM(vg.total\_unique\_views)**

**FROM Contests AS con**

**JOIN Colleges AS col ON con.contest\_id = col.contest\_id**

**JOIN Challenges AS cha ON cha.college\_id = col.college\_id**

**LEFT JOIN**

**(SELECT ss.challenge\_id, SUM(ss.total\_submissions) AS total\_submissions, SUM(ss.total\_accepted\_submissions) AS total\_accepted\_submissions FROM Submission\_Stats AS ss GROUP BY ss.challenge\_id) AS sg**

**ON cha.challenge\_id = sg.challenge\_id**

**LEFT JOIN**

**(SELECT vs.challenge\_id, SUM(vs.total\_views) AS total\_views, SUM(vs.total\_unique\_views) AS total\_unique\_views**

**FROM View\_Stats AS vs GROUP BY vs.challenge\_id) AS vg**

**ON cha.challenge\_id = vg.challenge\_id**

**GROUP BY con.contest\_id, con.hacker\_id, con.name**

**HAVING SUM(sg.total\_submissions) +**

**SUM(sg.total\_accepted\_submissions) +**

**SUM(vg.total\_views) +**

**SUM(vg.total\_unique\_views) > 0**

**ORDER BY con.contest\_id;**

**Project 2:-**

**Task 1:-**

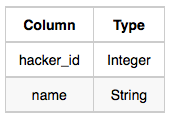
Julia conducted a  days of learning SQL contest. The start date of the contest was *March 01, 2016* and the end date was *March 15, 2016*.

Write a query to print total number of unique hackers who made at least  submission each day (starting on the first day of the contest), and find the *hacker\_id* and *name* of the hacker who made maximum number of submissions each day. If more than one such hacker has a maximum number of submissions, print the lowest *hacker\_id*. The query should print this information for each day of the contest, sorted by the date.

Inserting image...

**Input Format**

The following tables hold contest data:

* *Hackers:* The *hacker\_id* is the id of the hacker, and *name* is the name of the hacker.
* *Submissions:* The *submission\_date* is the date of the submission, *submission\_id* is the id of the submission, *hacker\_id* is the id of the hacker who made the submission, and *score* is the score of the submission.

**Sample Input**

For the following sample input, assume that the end date of the contest was *March 06, 2016*.

*Hackers* Table:

*Submissions* Table:

**Sample Output**

2016-03-01 4 20703 Angela

2016-03-02 2 79722 Michael

2016-03-03 2 20703 Angela

2016-03-04 2 20703 Angela

2016-03-05 1 36396 Frank

2016-03-06 1 20703 Angela

**Explanation**

On March 01, 2016 hackers , , , and made submissions. There are unique hackers who made at least one submission each day. As each hacker made one submission, is considered to be the hacker who made maximum number of submissions on this day. The name of the hacker is Angela.

On March 02, 2016 hackers , , and made submissions. Now and were the only ones to submit every day, so there are unique hackers who made at least one submission each day. made submissions, and name of the hacker is Michael.

On March 03, 2016 hackers , , and made submissions. Now and were the only ones, so there are unique hackers who made at least one submission each day. As each hacker made one submission so is considered to be the hacker who made maximum number of submissions on this day. The name of the hacker is Angela.

On March 04, 2016 hackers , , , and made submissions. Now and only submitted each day, so there are unique hackers who made at least one submission each day. As each hacker made one submission so is considered to be the hacker who made maximum number of submissions on this day. The name of the hacker is Angela.

On March 05, 2016 hackers , , and made submissions. Now only submitted each day, so there is only unique hacker who made at least one submission each day. made submissions and name of the hacker is Frank.

On March 06, 2016 only made submission, so there is only unique hacker who made at least one submission each day. made submission and name of the hacker is Angela.

—

This problem is a little bit tricky as you should get the aggregated value

**Here is the solution**

**SELECT SUBMISSION\_DATE,  
(SELECT COUNT(DISTINCT HACKER\_ID)   
FROM SUBMISSIONS S2   
WHERE S2.SUBMISSION\_DATE = S1.SUBMISSION\_DATE AND   
(SELECT COUNT(DISTINCT S3.SUBMISSION\_DATE)   
FROM SUBMISSIONS S3 WHERE S3.HACKER\_ID = S2.HACKER\_ID AND S3.SUBMISSION\_DATE < S1.SUBMISSION\_DATE) = DATEDIFF(S1.SUBMISSION\_DATE , '2016-03-01')),  
(SELECT HACKER\_ID FROM SUBMISSIONS S2 WHERE S2.SUBMISSION\_DATE = S1.SUBMISSION\_DATE   
GROUP BY HACKER\_ID ORDER BY COUNT(SUBMISSION\_ID) DESC, HACKER\_ID LIMIT 1) AS TMP,  
(SELECT NAME FROM HACKERS WHERE HACKER\_ID = TMP)  
FROM  
(SELECT DISTINCT SUBMISSION\_DATE FROM SUBMISSIONS) S1  
GROUP BY SUBMISSION\_DATE;**

Here we could break down the problem into a few small problems,

* The number of people who has made consecutive submissions in the past few days
* Among the people who had make consecutive submission , who submit the most amount of data

**To solve the first part**

(SELECT COUNT(DISTINCT HACKER\_ID)   
FROM SUBMISSIONS S2   
WHERE S2.SUBMISSION\_DATE = S1.SUBMISSION\_DATE AND   
(SELECT COUNT(DISTINCT S3.SUBMISSION\_DATE)   
FROM SUBMISSIONS S3 WHERE S3.HACKER\_ID = S2.HACKER\_ID AND S3.SUBMISSION\_DATE < S1.SUBMISSION\_DATE) = DATEDIFF(S1.SUBMISSION\_DATE , '2016-03-01'))

Here we select the number of distinct hacker whose on certain date equal than the number of days the contest start

**And the second part**

(SELECT HACKER\_ID FROM SUBMISSIONS S2 WHERE S2.SUBMISSION\_DATE = S1.SUBMISSION\_DATE   
GROUP BY HACKER\_ID ORDER BY COUNT(SUBMISSION\_ID) DESC, HACKER\_ID LIMIT 1) AS TMP,  
(SELECT NAME FROM HACKERS WHERE HACKER\_ID = TMP)FROM  
(SELECT DISTINCT SUBMISSION\_DATE FROM SUBMISSIONS) S1  
GROUP BY SUBMISSION\_DATE;

and incorporate these two selection in the main part of selection

**Task 2:-**

Consider P1(a,b) and P2(c,d) to be two points on a *2D* plane.

* happens to equal the minimum value in *Northern Latitude* (*LAT\_N* in **STATION**).
* happens to equal the minimum value in *Western Longitude* (*LONG\_W* in **STATION**).
* happens to equal the maximum value in *Northern Latitude* (*LAT\_N* in **STATION**).
* happens to equal the maximum value in *Western Longitude* (*LONG\_W* in **STATION**).

Query the [Manhattan Distance](https://xlinux.nist.gov/dads/HTML/manhattanDistance.html) between points P1 and P2 and round it to a scale of  decimal places.

**Input Format**

The **STATION** table is described as follows:

where *LAT\_N* is the northern latitude and *LONG\_W* is the western longitude.

* Write a query to print all *prime numbers* less than or equal to 1000. Print your result on a single line, and use the ampersand (&) character as your separator (instead of a space).

For example, the output for all prime numbers <=10 would be:

The Manhattan Distance is |x1 - x2| + |y1 - y2| = |a - c| + |b - d|.

* |a - c| + |b - d| ==> ABS(MIN(LAT\_N)-MAX(LAT\_N)) + ABS(MIN(LONG\_W)-MAX(LONG\_W))
* round to a scale of 4 decimal places ==> SELECT ROUND(ABS(MIN(LAT\_N)-MAX(LAT\_N)) + ABS(MIN(LONG\_W)-MAX(LONG\_W)), 4)
* from **STATION** table ==> FROM STATION

### **Solution:**

**SELECT ROUND(ABS(MIN(LAT\_N)-MAX(LAT\_N)) + ABS(MIN(LONG\_W)-MAX(LONG\_W)), 4) FROM STATION;**

**Task 3:-**

[Pivot](https://en.wikipedia.org/wiki/Pivot_table) the *Occupation* column in **OCCUPATIONS** so that each *Name* is sorted alphabetically and displayed underneath its corresponding *Occupation*. The output column headers should be *Doctor*, *Professor*, *Singer*, and *Actor*, respectively.

**Note:** Print **NULL** when there are no more names corresponding to an occupation.

**Input Format**

The **OCCUPATIONS** table is described as follows:

*Occupation* will only contain one of the following values: **Doctor**, **Professor**, **Singer** or **Actor**.

**Sample Input**

**Sample Output**

Jenny Ashley Meera Jane

Samantha Christeen Priya Julia

NULL Ketty NULL Maria

**Explanation**  
The first column is an alphabetically ordered list of Doctor names.  
The second column is an alphabetically ordered list of Professor names.  
The third column is an alphabetically ordered list of Singer names.  
The fourth column is an alphabetically ordered list of Actor names.  
The empty cell data for columns with less than the maximum number of names per occupation (in this case, the Professor and Actor columns) are filled with NULL values.

### **Analysis**

To solve this problem, we can use user-defined variables to help create a new table. Take the sample input as example, the table we want to create looks like below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **RowLine** | **Doctor** | **Professor** | **Singer** | **Actor** |
| 1 | NULL | Ashely | NULL | NULL |
| 2 | NULL | Christeen | NULL | NULL |
| 1 | NULL | NULL | NULL | Jane |
| 1 | Jenny | NULL | NULL | NULL |
| 2 | NULL | NULL | NULL | Julia |
| 3 | NULL | Ketty | NULL | NULL |
| 3 | NULL | NULL | NULL | Maria |
| 1 | NULL | NULL | Meera | NULL |
| 2 | NULL | NULL | Priya | NULL |
| 2 | Samantha | NULL | NULL | NULL |

The *RowLine* represents the line where the name should be put. In addition, because we want to sort names alphabetically for each occupation, the first step of creating the table above is to sort **OCCUPATIONS** table by name. Let’s call the table *t*. Once we have got the table *t*, we can use “SELECT MIN(Doctor), MIN(Professor), MIN(Singer), MIN(Actor) FROM t GROUP BY RowLine” to get the result.

To get table *t*, user-defined variables and CASE operator can help. We create four variables to record the line number *RowLine*, one for each occupation. We use CASE to add variables according to occupation.

### **Solution**

**SET @r1=0, @r2=0, @r3 =0, @r4=0;**

**SELECT MIN(Doctor), MIN(Professor), MIN(Singer), MIN(Actor) FROM**

**(SELECT CASE Occupation WHEN 'Doctor' THEN @r1:=@r1+1**

**WHEN 'Professor' THEN @r2:=@r2+1**

**WHEN 'Singer' THEN @r3:=@r3+1**

**WHEN 'Actor' THEN @r4:=@r4+1 END**

**AS RowLine,**

**CASE WHEN Occupation = 'Doctor' THEN Name END AS Doctor,**

**CASE WHEN Occupation = 'Professor' THEN Name END AS Professor,**

**CASE WHEN Occupation = 'Singer' THEN Name END AS Singer,**

**CASE WHEN Occupation = 'Actor' THEN Name END AS Actor**

**FROM OCCUPATIONS ORDER BY Name) AS t**

**GROUP BY RowLine;**

**Task 4:-**

Amber's conglomerate corporation just acquired some new companies. Each of the companies follows this hierarchy:

Given the table schemas below, write a query to print the *company\_code*, *founder* name, total number of *lead* managers, total number of *senior* managers, total number of *managers*, and total number of *employees*. Order your output by ascending *company\_code*.

**Note:**

* The tables may contain duplicate records.
* The *company\_code* is string, so the sorting should not be **numeric**. For example, if the *company\_codes* are *C\_1*, *C\_2*, and *C\_10*, then the ascending *company\_codes* will be *C\_1*, *C\_10*, and *C\_2*.

**Input Format**

The following tables contain company data:

* *Company:* The *company\_code* is the code of the company and *founder* is the founder of the company.
* *Lead\_Manager:* The *lead\_manager\_code* is the code of the lead manager, and the *company\_code* is the code of the working company.
* *Senior\_Manager:* The *senior\_manager\_code* is the code of the senior manager, the *lead\_manager\_code* is the code of its lead manager, and the *company\_code* is the code of the working company.
* *Manager:* The *manager\_code* is the code of the manager, the *senior\_manager\_code* is the code of its senior manager, the *lead\_manager\_code* is the code of its lead manager, and the *company\_code* is the code of the working company.
* *Employee:* The *employee\_code* is the code of the employee, the *manager\_code* is the code of its manager, the *senior\_manager\_code* is the code of its senior manager, the *lead\_manager\_code* is the code of its lead manager, and the *company\_code* is the code of the working company.

**Sample Input**

*Company* Table:

*Lead\_Manager* Table: *Senior\_Manager* Table:

*Manager* Table:  *Employee* Table:

**Sample Output**

C1 Monika 1 2 1 2

C2 Samantha 1 1 2 2

**Explanation**

**In company** *C1*, the only lead manager is *LM1*. There are two senior managers, *SM1* and *SM2*, under *LM1*. There is one manager, *M1*, under senior manager *SM1*. There are two employees, *E1* and *E2*, under manager *M1*.

In company *C2*, the only lead manager is *LM2*. There is one senior manager, *SM3*, under *LM2*. There are two managers, *M2* and *M3*, under senior manager *SM3*. There is one employee, *E3*, under manager *M2*, and another employee, *E4*, under manager, *M3*.

### **Analysis**

We can join all tables with *company\_code*, *lead\_manager\_code*, *senior\_manager\_code*, *manager\_code* and *employee\_code*. We can use “SELECT from tb1, tb2, … WHERE” to join tables. And we can use “JOIN … ON …” to join tables one by one as well.  
Also,

* number of employees and various managers ==> COUNT(…) GROUP BY …
* table may contain duplicates ==> COUNT(DISTINCT …) GROUP BY …
* order output by ascending *company\_code* ==> ORDER BY c.company\_code

#### **Solution 1**

**SELECT c.company\_code, c.founder,**

**COUNT(DISTINCT l.lead\_manager\_code), COUNT(DISTINCT s.senior\_manager\_code),**

**COUNT(DISTINCT m.manager\_code), COUNT(DISTINCT e.employee\_code)**

**FROM Company c, Lead\_Manager l, Senior\_Manager s, Manager m, Employee e**

**WHERE c.company\_code = l.company\_code AND**

**l.lead\_manager\_code = s.lead\_manager\_code AND**

**s.senior\_manager\_code = m.senior\_manager\_code AND**

**m.manager\_code = e.manager\_code**

**GROUP BY c.company\_code, c.founder ORDER BY c.company\_code;**

|  |
| --- |
|  |

#### **Solution 2**

**SELECT c.company\_code, c.founder,**

**COUNT(DISTINCT l.lead\_manager\_code), COUNT(DISTINCT s.senior\_manager\_code),**

**COUNT(DISTINCT m.manager\_code), COUNT(DISTINCT e.employee\_code)**

**FROM Company c JOIN Lead\_Manager l ON c.company\_code = l.company\_code JOIN**

**Senior\_Manager s ON l.lead\_manager\_code = s.lead\_manager\_code JOIN**

**Manager m ON s.senior\_manager\_code = m.senior\_manager\_code JOIN**

**Employee e ON m.manager\_code = e.manager\_code**

**GROUP BY c.company\_code, c.founder ORDER BY c.company\_code;**

**Task 5:-**

You are given three tables: *Students*,*Friends*and*Packages.* *Students* contains two columns: *ID* and *Name*. *Friends* contains two columns: *ID* and *Friend\_ID* (*ID* of the ONLY best friend). *Packages* contains two columns: *ID* and *Salary* (offered salary in $ thousands per month).

Write a query to output the names of those students whose best friends got offered a higher salary than them. Names must be ordered by the salary amount offered to the best friends. It is guaranteed that no two students got same salary offer.

**Sample Input**

**Sample Output**

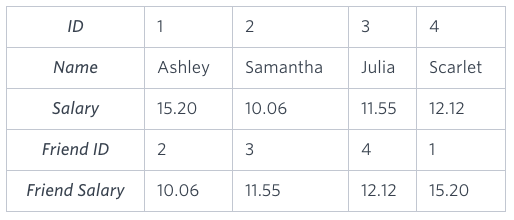
Samantha

Julia

Scarlet

**Explanation**

See the following table:



Now,

* Samantha’s best friend got offered a higher salary than her at 11.55
* Julia’s best friend got offered a higher salary than her at 12.12
* Scarlet’s best friend got offered a higher salary than her at 15.2
* Ashley’s best friend did NOT get offered a higher salary than her

The name output, when ordered by the salary offered to their friends, will be:

* Samantha
* Julia
* Scarlet

### **Analysis**

To solve this problem, we can join Students and Friends, and then join Packages twice to get students’ and their best friends’ salaries. And find students whose salary is fewer than their friend’s. At last, sort the result by friends’ salaries.

First, we need to output students’ names:

**SELECT Name FROM Students;**

Then, join Packages to get students’ salaries:

**SELECT s.Name FROM Students AS s**

**JOIN Packages AS sp ON s.ID = sp.ID;**

Also, join Friends to get corresponding Friend\_ID:

**SELECT s.Name FROM Students AS s**

**JOIN Packages AS sp ON s.ID = sp.ID**

**JOIN Friends AS f ON s.ID = f.ID;**

Next, join Packages again to get friends’ salaries:

**SELECT s.Name FROM Students AS s**

**JOIN Packages AS sp ON s.ID = sp.ID**

**JOIN Friends AS f ON s.ID = f.ID**

**JOIN Packages AS fp ON f.Friend\_ID = fp.ID;**

Additionally, find students whose friends have higher salary:

**SELECT s.Name FROM Students AS s**

**JOIN Packages AS sp ON s.ID = sp.ID**

**JOIN Friends AS f ON s.ID = f.ID**

**JOIN Packages AS fp ON f.Friend\_ID = fp.ID**

**WHERE sp.Salary < fp.Salary;**

Finally, sort result by friends’ salaries:

**SELECT s.Name FROM Students AS s**

**JOIN Packages AS sp ON s.ID = sp.ID**

**JOIN Friends AS f ON s.ID = f.ID**

**JOIN Packages AS fp ON f.Friend\_ID = fp.ID**

**WHERE sp.Salary < fp.Salary**

### **ORDER BY fp.Salary;**

### **Solution**

**SELECT s.Name FROM Students AS s**

**JOIN Packages AS sp ON s.ID = sp.ID**

**JOIN Friends AS f ON s.ID = f.ID**

**JOIN Packages AS fp ON f.Friend\_ID = fp.ID**

**WHERE sp.Salary < fp.Salary**

**ORDER BY fp.Salary;**

**Project 3:-**

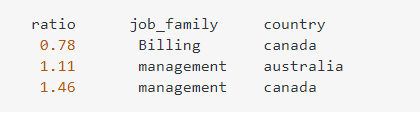
**Task 1:-**

**Display ratio of cost of job family in percentage by India and international (refer simulation data).**

We have data table as:



Now i want ratio of ctc of each service type internationally to india. example: desired output



Beacause Billing total ctc of canada is 30000 and ctc for billing in india is 38000 so ratio is 0.78 Management ctc total in australia is 29000 and in india is 26000 so ratio is 1.11 Management total is 38000 in canada and in india is 26000 so ratio is 1.46

**Select job\_family, country,**

**sum(ctc) \* 1.0 / sum(case when country = 'India' then sum(ctc) end) over (partition by job\_family) as ratio**

**from t**

**group by job\_family, country;**

**Task 2:-**

**Find ratio of cost and revenue of a BU month on month.**

To benchmark your business, you’ll want to compute week-over-week, month-over-month and year-over-year growth rates. In the case of Silota, we are not only interested in the number of charts created monthly, but also their growth rates on a month-to-month basis.

**Select date\_trunc('month', timestamp) as date, count(\*) as count from events where event\_name = 'created chart' group by 1 order by 1**

|  |  |
| --- | --- |
| **date** | **count** |
| 2016-01-01 | 10 |
| 2016-01-02 | 12 |
| 2016-01-03 | 15 |
| ... | ... |

The above query should give us a neat table with the number of charts created every month. To compute the growth rates, we use window functions and the lag function. First to understand how the lag function works:

**Select date\_trunc('month', timestamp) as date, count(\*) as count, lag(count(\*), 1) over timestamp from events where event\_name = 'created chart' group by 1 order by 1**

|  |  |  |
| --- | --- | --- |
| **date** | **count** | **lag** |
| 2016-01-01 | 10 |  |
| 2016-01-02 | 12 | 10 |
| 2016-01-03 | 15 | 12 |

The lag function returns a value evaluated at the row that is definable offset before the current row within the partition. In this particular we have simply picked the value from the previous row (offset of 1). To compute growth rates, it’s just a matter of subtracting the current value from the previous value:

**Select date\_trunc('month', timestamp) as date, count(\*) as count, 100 \* (count(\*) - lag(count(\*), 1) over (order by timestamp)) / lag(count(\*), 1) over (order by timestamp)) || '%' as growth from events where event\_name = 'created chart' group by 1 order by 1**

|  |  |  |
| --- | --- | --- |
| **date** | **count** | **growth** |
| 2016-01-01 | 10 |  |
| 2016-01-02 | 12 | 20% |
| 2016-01-03 | 15 | 25% |
| ... | ... | ... |

**Task 3:-**

**Show headcounts of sub band and percentage of headcount (without join, subquery and inner query).**

If a result that only shows the highest salary abo ve the average (as opposed to *all* salaries above average) is acceptable, then this can be done without a sub-select:

**Select salary, salary - avg(salary) over () as diff\_to\_average, avg(salary) over () as average\_salary from employees order by 2 desc fetch first 1 row only**

(The above is standard ANSI SQL)

The drawback is that you can't remove the diff\_to\_average column as you can't use the alias in a where clause on the same level (you can remove the average\_salaray though). The whole question doesn't really make sense though.

One solution that does not use a sub-select, but only a derived table is:

**Select \* from (select salary, avg(salary) over () as average\_salary from employees) where salary > average\_salary order by salary**

The derived table is only necessary because SQL does not allow to (re)use a column alias in the WHERE clause on the same level.

However depending on the DBMS, the query in your question might be more efficient as the window function in the derived table typically requires some sort of buffering which would not happen when using the sub-select from your question.

I created a table with three columns: id, name ans salary and a million rows and then compared the two queries. I did not create an index on the salary column.

**Task 4:-**

**Find top 5 employees according to salary (without order by).**

**SELECT \* FROM table WHERE(salary IN ( SELECT TOP (5) salary FROM table as table1 GROUP BY sal ORDER BY sal DESC ))**

**Task 5:-**

**Swap value of two columns in a table without using third variable or a table.**

Instead of having to move a lot of data around, it may be easier to create a view with the names you want:

**CREATE VIEW myview AS SELECT lastname AS name, name AS lastname FROM mytable**

**Task 6:-**

**Create a user, create a login for that user provide permissions of DB\_owner to the user.**

**-- create the user on the master database**

**USE [master]**

**GO**

**CREATE LOGIN [MyUserName] WITH PASSWORD=N'MyPassword'**

**CREATE USER [MyUserName] FOR LOGIN [MyUserName]**

**GO**

**-- create the user on the target database for the login**

**USE [MyDatabaseName]**

**GO**

**CREATE USER [MyUserName] FOR LOGIN [MyUserName]**

**GO**

**-- add the user to the desired role**

**USE [MyDatabaseName]**

**GO**

**ALTER ROLE [db\_owner] ADD MEMBER [MyUserName]**

**GO**

**Task 7:-**

**Find Weighted average cost of employees month on month in a BU.**

I need to calculate the cost for each project for each month, the project consists of tasks. I have an employee table, tasks table and project table. But I don't know how to calculate the cost for a project each month, so far I have this:

**SELECT P.PROJECT\_NAME, SUM(T.HOURS\_WORKED \* E.HOURLY\_RATE) COSTFROM PROJECT P, TASKS T, EMPLOYEE E WHERE E.EMPLOYEE\_ID = T.EMPLOYEE\_ID AND P.PROJECT\_ID = T.PROJECT\_IDGROUP BY P.PROJECT\_NAME;**

but that doesn't work out how much is charged for a project each month,it just works out the overall cost by calculating the hours worked by the employees by the employee hourly rate. In the task table I do have a date\_worked column which displays the day,month and year but I don't know if that needs to be used or not.

**Task 8 -**

**Samantha was tasked with calculating the average monthly salaries for all employees in the EMPLOYEES table, but did not realize her keyboard's 0  key was broken until after completing the calculation. She wants your help finding the difference between her miscalculation (using salaries with any zeroes removed), and the actual average salary.**

**Write a query calculating the amount of error (i.e.:  actual – miscalculated average monthly salaries), and round it up to the next integer.**

**SELECT ROUND(AVG(SALARY)) - ROUND(AVG(REPLACE(SALARY, '0',''))) FROM EMPLOYEES;**

**Task 9:-**

**Copy new data of one table to another( you do not have indicator for new data and old data).**

**Creating a table in SQL Server**

Now we create a table named employee using:

**CREATE TABLE [dbo].[Employee]**

**(**

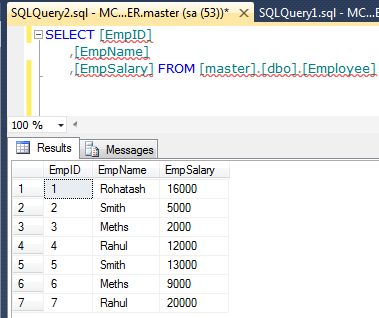
**[EmpID] [int] NULL,**

**[EmpName] [varchar](30) NULL,**

**[EmpSalary] [int] NULL**

**)**

The following is the sample data for the employee table:



## **Method 1: Copy Table using SELECT INTO**

This command only copies a table's schema and its data. The Select into is used to copy a table with data from one database to another database's table. The Select into is also used to create a new table in another database. The general syntax to do that is:

**Syntax**

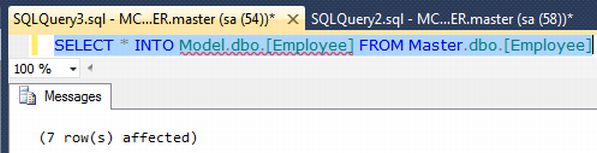
1. **SELECT** \* **INTO** DestinationDB.dbo.tableName **FROM** SourceDB.dbo.SourceTable

**Example**

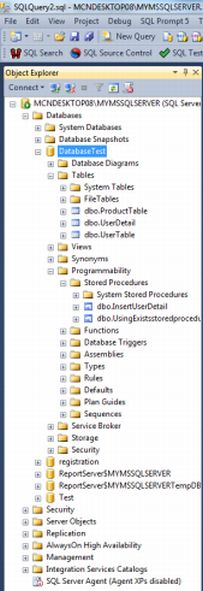
The employee table is defined in the master database. It is also called the source database. Now you want to copy the table with data from the master database to the model database. The following query defines it:

1. **SELECT** \* **INTO** Model.dbo.[Employee] **FROM** Master.dbo.[Employee]

Now hit F5 to execute it.



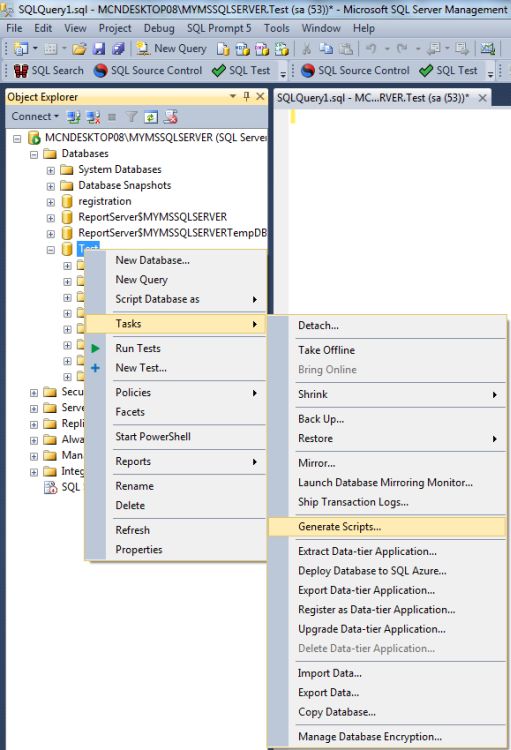
Now press F8 to open the Object Explorer and select the model database to see the employee table.



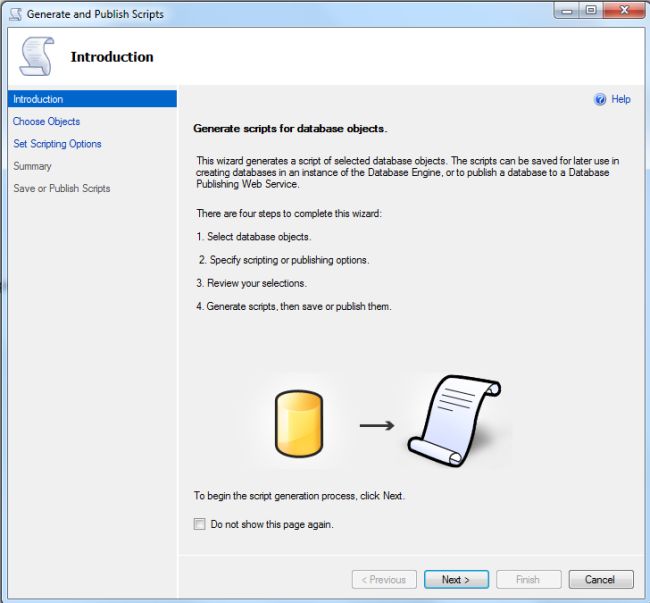
## **Method 2: Generating Script in SQL Server**

If you want to copy all objects, indexes, triggers, constraints etc then do it using "Generate Scripts...". Suppose we have a database named Test. Now right-click on the Test database and select the "Generate Scripts..."option.

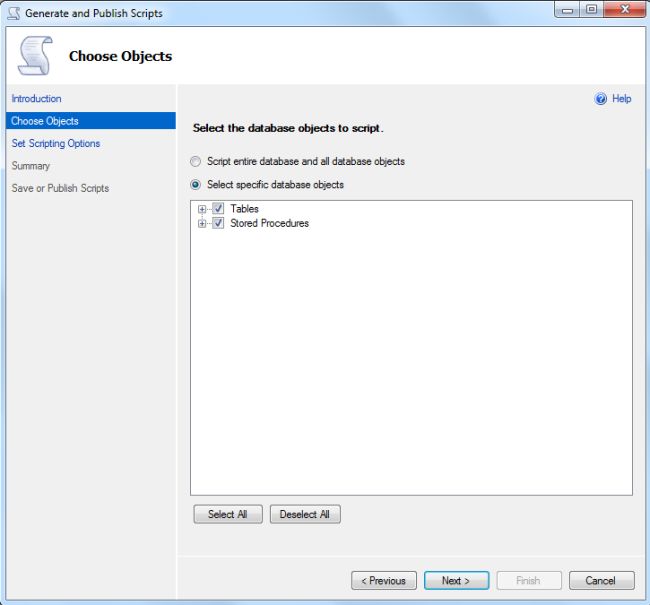
database Name -> "Tasks" -> "Generate Scripts...."



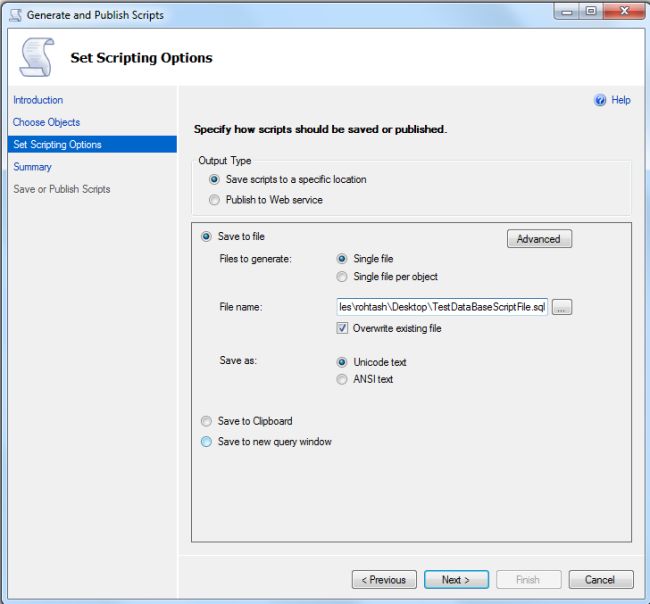
Now click on "Generate Scripts...". The Generate Scripts wizard will be opened.



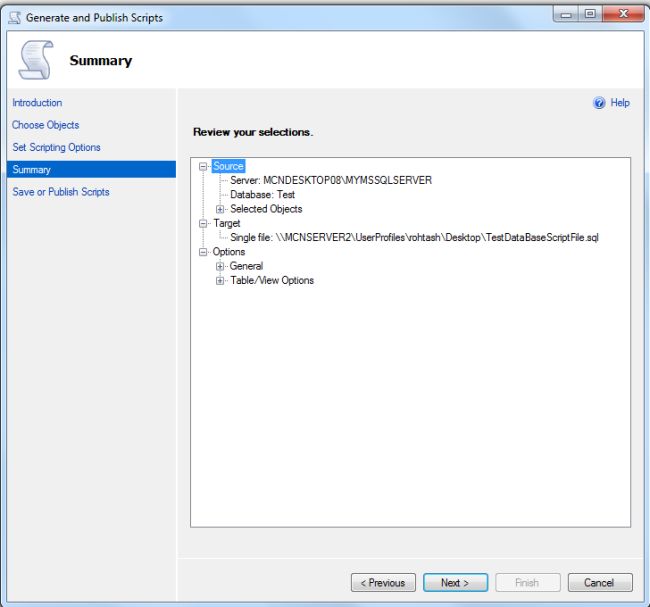
Now click on the "Next" Button and select tables and Stored Procedures.



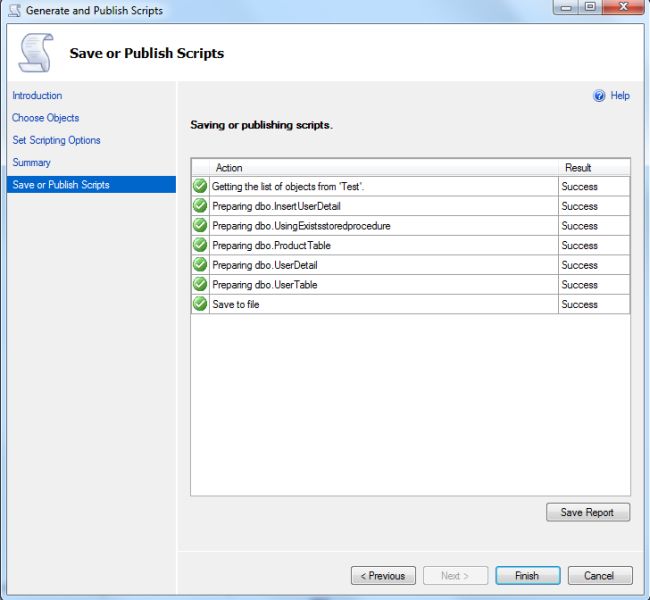
Now click on the **"Next"** Button and provide the proper name with path of the file.



Now click on the **"Next"** Button and review your source and target location.



Now click on the **"Next"** Button.



Now finally click on the "Finish" button.

The script file has been generated for the Test database. To see the generated script file, select the location of the file in your computer.

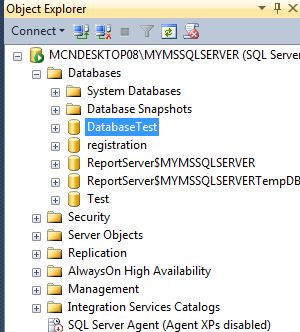


## **Creating a Database in SQL Server**

These are the following steps to create a new database:

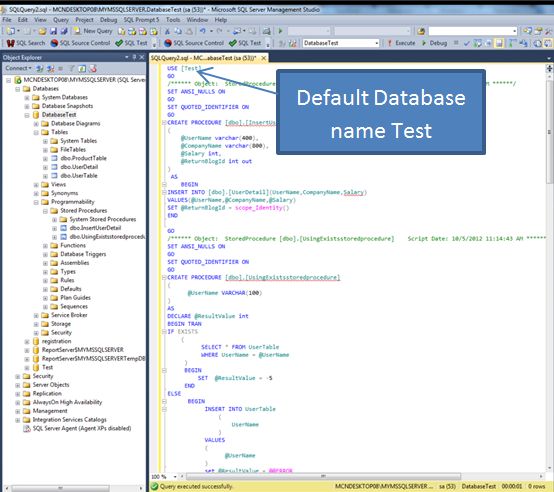
* Press F8 to open the Object Browser in SQL Server Management Studio and expend
* Database -> right-click-> select New database
* This would open the "**New database**" window
* Now enter a database name to create a database
* Now click on the OK button to create the database. The new database will be shown in the Object Explorer

Now the database, named DatabaseTest, has been created.

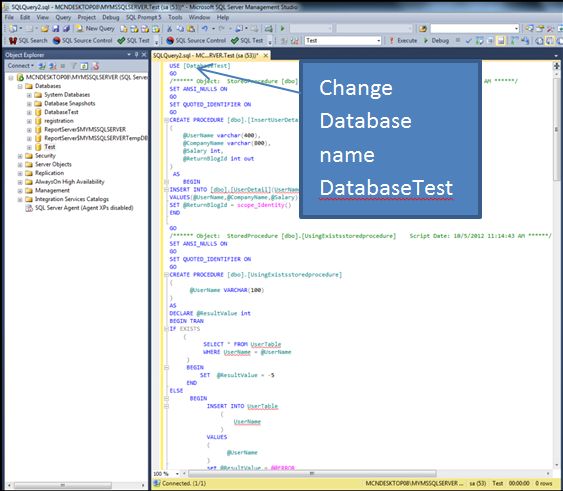


## **Copy Database Schema and Data to Other Database**

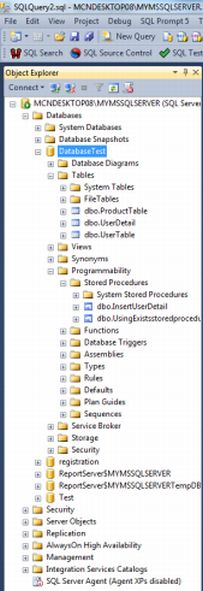
Now right-click on the script file and open it in Notepad and copy all the data and paste it in the query window in SQL Server. It will look as in the following:



Now only change the database name test to DatabaseTest.



Now press F5 to execute the script and expand the databaseTest to see the schema and data.



**Project 4:-**

**Task 1 :-**

**Understand the concept of dimension tables in data modelling. Learn the importance and schema structure of DimDate table (date dimension table) in modelling and implement a stored procedure code to load 25 years(from today’s date) date data and its computed date fields in date dimension table.**

I build calendar tables all the time, for a variety of business applications, and have come up with a few ways to handle certain details. Sharing them here will hopefully prevent you from re-inventing any wheels when populating your own tables.

One of the biggest objections I hear to calendar tables is that people don't want to create a table. I can't stress enough how cheap a table can be in terms of size and memory usage, especially as underlying storage continues to be larger and faster, compared to using all kinds of functions to determine date-related information in every single query. Twenty or thirty years of dates stored in a table takes a few MBs at most, even less with compression, and if you use them often enough, they'll always be in memory.

I also always explicitly set things like DATEFORMAT, DATEFIRST, and LANGUAGE to avoid ambiguity, default to U.S. English for week starts and for month and day names, and assume that quarters for the fiscal year align with the calendar year. You may need to change some of these specifics depending on your display language, your fiscal year, and other factors.

This is a one-time population, so I'm not worried about speed, even though this specific CTE approach is no slouch. I like to materialize all of the columns to disk, rather than rely on computed columns, since the table becomes read-only after initial population. So I'm going to do a lot of those calculations during the initial series of [CTEs](https://www.sentryone.com/blog/aaronbertrand/backtobasics-ctes). To start, I'll show the output of each CTE one at a time.

You can change some of these details to experiment on your own. In this example, I'm going to populate the date dimension table with data spanning 30 years, starting from 2010-01-01.

First, we have a recursive CTE that returns a sequence representing the number of days between our start date (2010-01-01) and 30 years later less a day (2039-12-31):

**-- prevent set or regional settings from interfering with**

**-- interpretation of dates / literals**

**SET DATEFIRST 7, -- 1 = Monday, 7 = Sunday**

**DATEFORMAT mdy,**

**LANGUAGE US\_ENGLISH;**

**-- assume the above is here in all subsequent code blocks.**

**DECLARE @StartDate date = '20100101';**

**DECLARE @CutoffDate date = DATEADD(DAY, -1, DATEADD(YEAR, 30, @StartDate));**

**;WITH seq(n) AS**

**(**

**SELECT 0 UNION ALL SELECT n + 1 FROM seq**

**WHERE n < DATEDIFF(DAY, @StartDate, @CutoffDate)**

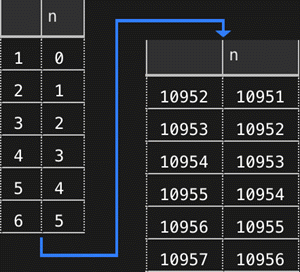
**)**

**SELECT n FROM seq**

**ORDER BY n**

**OPTION (MAXRECURSION 0);**

This returns the following list of numbers:



Next, we can add a second CTE that translates those numbers into all the dates in our range:

**DECLARE @StartDate date = '20100101';**

**DECLARE @CutoffDate date = DATEADD(DAY, -1, DATEADD(YEAR, 30, @StartDate));**

**;WITH seq(n) AS**

**(**

**SELECT 0 UNION ALL SELECT n + 1 FROM seq**

**WHERE n < DATEDIFF(DAY, @StartDate, @CutoffDate)**

**),d(d) AS**

**(**

**SELECT DATEADD(DAY, n, @StartDate) FROM seq**

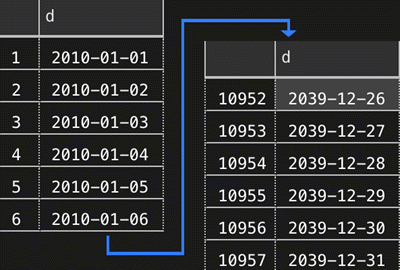
**)**

**SELECT d FROM d**

**ORDER BY d**

**OPTION (MAXRECURSION 0);**

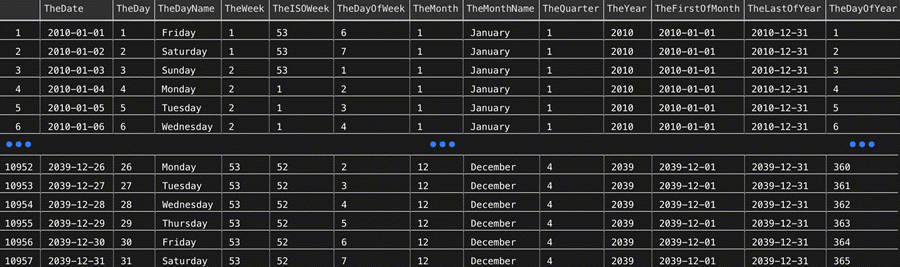
Which returns the following range of dates:



Now, we can start extending those dates with information commonly vital to calendar tables / date dimensions. Many are bits of information you can extract from the date, but it's more convenient to have them readily available in a view or table than it is to have every query calculate them inline. I'm working a little backward here, but I'm going to create an intermediate CTE to extract exactly once some computations I'll later have to make multiple times. This query:

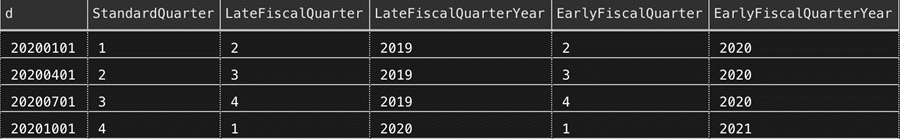
**DECLARE @StartDate date = '20100101';DECLARE @CutoffDate date = DATEADD(DAY, -1, DATEADD(YEAR, 30, @StartDate));;WITH seq(n) AS ( SELECT 0 UNION ALL SELECT n + 1 FROM seq WHERE n < DATEDIFF(DAY, @StartDate, @CutoffDate)),d(d) AS ( SELECT DATEADD(DAY, n, @StartDate) FROM seq),src AS( SELECT TheDate = CONVERT(date, d), TheDay = DATEPART(DAY, d), TheDayName = DATENAME(WEEKDAY, d), TheWeek = DATEPART(WEEK, d), TheISOWeek = DATEPART(ISO\_WEEK, d), TheDayOfWeek = DATEPART(WEEKDAY, d), TheMonth = DATEPART(MONTH, d), TheMonthName = DATENAME(MONTH, d), TheQuarter = DATEPART(Quarter, d), TheYear = DATEPART(YEAR, d), TheFirstOfMonth = DATEFROMPARTS(YEAR(d), MONTH(d), 1), TheLastOfYear = DATEFROMPARTS(YEAR(d), 12, 31), TheDayOfYear = DATEPART(DAYOFYEAR, d) FROM d)SELECT \* FROM src ORDER BY TheDate OPTION (MAXRECURSION 0);**

Yields this data:



If you wanted your fiscal year aligned differently, you could change the year and quarter calculations, or add additional columns. Let's say your fiscal year starts October 1st, then depending on whether that's 9 months late or 3 months early, you could just substitute d for a DATEADD expression:

**;WITH q AS (SELECT d FROM ( VALUES('20200101'), ('20200401'), ('20200701'), ('20201001') ) AS d(d))SELECT d, StandardQuarter = DATEPART(QUARTER, d), LateFiscalQuarter = DATEPART(QUARTER, DATEADD(MONTH, -9, d)), LateFiscalQuarterYear = YEAR(DATEADD(MONTH, -9, d)), EarlyFiscalQuarter = DATEPART(QUARTER, DATEADD(MONTH, 3, d)), EarlyFiscalQuarterYear = YEAR(DATEADD(MONTH, 3, d))FROM q;**



Whatever my source data is, I can build on those parts and get much more detail about each date:

**DECLARE @StartDate date = '20100101';DECLARE @CutoffDate date = DATEADD(DAY, -1, DATEADD(YEAR, 30, @StartDate));;WITH seq(n) AS ( SELECT 0 UNION ALL SELECT n + 1 FROM seq WHERE n < DATEDIFF(DAY, @StartDate, @CutoffDate)),d(d) AS ( SELECT DATEADD(DAY, n, @StartDate) FROM seq),src AS( SELECT TheDate = CONVERT(date, d), TheDay = DATEPART(DAY, d), TheDayName = DATENAME(WEEKDAY, d), TheWeek = DATEPART(WEEK, d), TheISOWeek = DATEPART(ISO\_WEEK, d), TheDayOfWeek = DATEPART(WEEKDAY, d), TheMonth = DATEPART(MONTH, d), TheMonthName = DATENAME(MONTH, d), TheQuarter = DATEPART(Quarter, d), TheYear = DATEPART(YEAR, d), TheFirstOfMonth = DATEFROMPARTS(YEAR(d), MONTH(d), 1), TheLastOfYear = DATEFROMPARTS(YEAR(d), 12, 31), TheDayOfYear = DATEPART(DAYOFYEAR, d) FROM d),dim AS( SELECT TheDate, TheDay, TheDaySuffix = CONVERT(char(2), CASE WHEN TheDay / 10 = 1 THEN 'th' ELSE CASE RIGHT(TheDay, 1) WHEN '1' THEN 'st' WHEN '2' THEN 'nd' WHEN '3' THEN 'rd' ELSE 'th' END END), TheDayName, TheDayOfWeek, TheDayOfWeekInMonth = CONVERT(tinyint, ROW\_NUMBER() OVER (PARTITION BY TheFirstOfMonth, TheDayOfWeek ORDER BY TheDate)), TheDayOfYear, IsWeekend = CASE WHEN TheDayOfWeek IN (CASE @@DATEFIRST WHEN 1 THEN 6 WHEN 7 THEN 1 END,7) THEN 1 ELSE 0 END, TheWeek, TheISOweek, TheFirstOfWeek = DATEADD(DAY, 1 - TheDayOfWeek, TheDate), TheLastOfWeek = DATEADD(DAY, 6, DATEADD(DAY, 1 - TheDayOfWeek, TheDate)), TheWeekOfMonth = CONVERT(tinyint, DENSE\_RANK() OVER (PARTITION BY TheYear, TheMonth ORDER BY TheWeek)), TheMonth, TheMonthName, TheFirstOfMonth, TheLastOfMonth = MAX(TheDate) OVER (PARTITION BY TheYear, TheMonth), TheFirstOfNextMonth = DATEADD(MONTH, 1, TheFirstOfMonth), TheLastOfNextMonth = DATEADD(DAY, -1, DATEADD(MONTH, 2, TheFirstOfMonth)), TheQuarter, TheFirstOfQuarter = MIN(TheDate) OVER (PARTITION BY TheYear, TheQuarter), TheLastOfQuarter = MAX(TheDate) OVER (PARTITION BY TheYear, TheQuarter), TheYear, TheISOYear = TheYear - CASE WHEN TheMonth = 1 AND TheISOWeek > 51 THEN 1 WHEN TheMonth = 12 AND TheISOWeek = 1 THEN -1 ELSE 0 END, TheFirstOfYear = DATEFROMPARTS(TheYear, 1, 1), TheLastOfYear, IsLeapYear = CONVERT(bit, CASE WHEN (TheYear % 400 = 0) OR (TheYear % 4 = 0 AND TheYear % 100 <> 0) THEN 1 ELSE 0 END), Has53Weeks = CASE WHEN DATEPART(WEEK, TheLastOfYear) = 53 THEN 1 ELSE 0 END, Has53ISOWeeks = CASE WHEN DATEPART(ISO\_WEEK, TheLastOfYear) = 53 THEN 1 ELSE 0 END, MMYYYY = CONVERT(char(2), CONVERT(char(8), TheDate, 101)) + CONVERT(char(4), TheYear), Style101 = CONVERT(char(10), TheDate, 101), Style103 = CONVERT(char(10), TheDate, 103), Style112 = CONVERT(char(8), TheDate, 112), Style120 = CONVERT(char(10), TheDate, 120) FROM src)SELECT \* FROM dim ORDER BY TheDate OPTION (MAXRECURSION 0);**

This adds supplemental information about any given date, such as the first of period / last of period the date falls within, whether it is a leap year, a few popular string formats, and some specific ISO 8601 specifics (I'll talk more about those in another tip). You may only want some of these columns, and you may want others, too. When you're happy with the output, you can change this line:

**SELECT \* FROM dim**

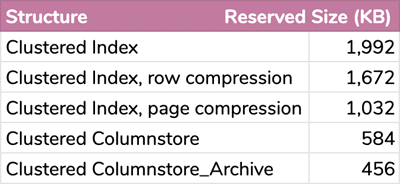
To this:

**SELECT \* INTO dbo.DateDimension FROM dim**

Then you can add a clustered primary key (and any other indexes you want to have handy):

**CREATE UNIQUE CLUSTERED INDEX PK\_DateDimension ON dbo.DateDimension(TheDate);**

To give an idea of how much space this table really takes, even with all those columns that you probably don't need, the max is about 2MB with a regular clustered index defined on the TheDate column, all the way down to 500KB for a clustered columnstore index compressed with COLUMNSTORE\_ARCHIVE (not necessarily something you should do, depending on the workload that will work against this table, but since it is effectively read only, the DML overhead isn't really a consideration):



Next, we need to talk about holidays, one of the primary seasons you need to use a calendar table instead of relying on built-in date/time functions. In the original version of this tip, I added an IsHoliday column, but as a comment rightly pointed out, this set is probably best held in a separate table:

**CREATE TABLE dbo.HolidayDimension( TheDate date NOT NULL, HolidayText nvarchar(255) NOT NULL, CONSTRAINT FK\_DateDimension FOREIGN KEY(TheDate) REFERENCES dbo.DateDimension(TheDate));CREATE CLUSTERED INDEX CIX\_HolidayDimension ON dbo.HolidayDimension(TheDate);GO**

This allows you to have more than one holiday for any given date, and in fact allows for multiple entire calendars each with their own set of holidays (imagine an additional column specifying the CalendarID).

Populating the holiday dimension table can be complex. Since I am in the United States, I'm going to deal with statutory holidays here; of course, if you live in another country, you'll need to use different logic. You'll also need to add your own company's holidays manually, but hopefully if you have things that are deterministic, like bank holidays, Boxing Day, or the third Monday of July is your annual off-site arm-wrestling tournament, you should be able to do most of that without much work by following the same sort of pattern I use below. You may also have to add some logic if your company observes weekend holidays on the previous or following weekday, which gets even more complex if those happen to collide with other company- or industry-specific non-business days. We can add most of the traditional holidays with a single pass and rather simple criteria:

**;WITH x AS ( SELECT TheDate, TheFirstOfYear, TheDayOfWeekInMonth, TheMonth, TheDayName, TheDay, TheLastDayOfWeekInMonth = ROW\_NUMBER() OVER ( PARTITION BY TheFirstOfMonth, TheDayOfWeek ORDER BY TheDate DESC ) FROM dbo.DateDimension),s AS( SELECT TheDate, HolidayText = CASE WHEN (TheDate = TheFirstOfYear) THEN 'New Year''s Day' WHEN (TheDayOfWeekInMonth = 3 AND TheMonth = 1 AND TheDayName = 'Monday') THEN 'Martin Luther King Day' -- (3rd Monday in January) WHEN (TheDayOfWeekInMonth = 3 AND TheMonth = 2 AND TheDayName = 'Monday') THEN 'President''s Day' -- (3rd Monday in February) WHEN (TheLastDayOfWeekInMonth = 1 AND TheMonth = 5 AND TheDayName = 'Monday') THEN 'Memorial Day' -- (last Monday in May) WHEN (TheMonth = 7 AND TheDay = 4) THEN 'Independence Day' -- (July 4th) WHEN (TheDayOfWeekInMonth = 1 AND TheMonth = 9 AND TheDayName = 'Monday') THEN 'Labour Day' -- (first Monday in September) WHEN (TheDayOfWeekInMonth = 2 AND TheMonth = 10 AND TheDayName = 'Monday') THEN 'Columbus Day' -- Columbus Day (second Monday in October) WHEN (TheMonth = 11 AND TheDay = 11) THEN 'Veterans'' Day' -- (November 11th) WHEN (TheDayOfWeekInMonth = 4 AND TheMonth = 11 AND TheDayName = 'Thursday') THEN 'Thanksgiving Day' -- (Thanksgiving Day ()fourth Thursday in November) WHEN (TheMonth = 12 AND TheDay = 25) THEN 'Christmas Day' END FROM x WHERE (TheDate = TheFirstOfYear) OR (TheDayOfWeekInMonth = 3 AND TheMonth = 1 AND TheDayName = 'Monday') OR (TheDayOfWeekInMonth = 3 AND TheMonth = 2 AND TheDayName = 'Monday') OR (TheLastDayOfWeekInMonth = 1 AND TheMonth = 5 AND TheDayName = 'Monday') OR (TheMonth = 7 AND TheDay = 4) OR (TheDayOfWeekInMonth = 1 AND TheMonth = 9 AND TheDayName = 'Monday') OR (TheDayOfWeekInMonth = 2 AND TheMonth = 10 AND TheDayName = 'Monday') OR (TheMonth = 11 AND TheDay = 11) OR (TheDayOfWeekInMonth = 4 AND TheMonth = 11 AND TheDayName = 'Thursday') OR (TheMonth = 12 AND TheDay = 25))INSERT dbo.HolidayDimension(TheDate, HolidayText)SELECT TheDate, HolidayText FROM s UNION ALL SELECT DATEADD(DAY, 1, TheDate), 'Black Friday' FROM s WHERE HolidayText = 'Thanksgiving Day'ORDER BY TheDate;**

Black Friday is a little trickier, because it's the Friday after the fourth Thursday in November. Usually that makes it the fourth Friday, but several times a century it is actually the fifth Friday, so the UNION ALL above just grabs the day after each Thanksgiving Day.

**And then there's Easter**. This has always been a complicated problem; [the rules for calculating the exact date are so convoluted](https://en.wikipedia.org/wiki/Easter#Date), I suspect most people can only mark those dates where they have physical calendars they can look at to confirm. If your company doesn't recognize Easter, you can skip ahead; if it does, you can use the following function, which will return the Easter holiday dates for any given year:

**CREATE FUNCTION dbo.GetEasterHolidays(@TheYear INT) RETURNS TABLEWITH SCHEMABINDINGAS RETURN ( WITH x AS ( SELECT TheDate = DATEFROMPARTS(@TheYear, [Month], [Day]) FROM (SELECT [Month], [Day] = DaysToSunday + 28 - (31 \* ([Month] / 4)) FROM (SELECT [Month] = 3 + (DaysToSunday + 40) / 44, DaysToSunday FROM (SELECT DaysToSunday = paschal - ((@TheYear + (@TheYear / 4) + paschal - 13) % 7) FROM (SELECT paschal = epact - (epact / 28) FROM (SELECT epact = (24 + 19 \* (@TheYear % 19)) % 30) AS epact) AS paschal) AS dts) AS m) AS d ) SELECT TheDate, HolidayText = 'Easter Sunday' FROM x UNION ALL SELECT DATEADD(DAY, -2, TheDate), 'Good Friday' FROM x UNION ALL SELECT DATEADD(DAY, 1, TheDate), 'Easter Monday' FROM x);GO**

(You can adjust the function easily, depending on whether they recognize just Easter Sunday or also Good Friday and/or Easter Monday. There is also another tip [here](https://www.mssqltips.com/sqlservertip/1537/tsql-function-to-determine-holidays-in-sql-server/) that will show you how to determine the date for Mardi Gras, given the date for Easter.)

Now, to use that function to add the Easter holidays to the HolidayDimension table:

**INSERT dbo.HolidayDimension(TheDate, HolidayText) SELECT d.TheDate, h.HolidayText FROM dbo.DateDimension AS d CROSS APPLY dbo.GetEasterHolidays(d.TheYear) AS h WHERE d.TheDate = h.TheDate;**

Finally, you can create a view that bridges these two tables (or multiple views):

**CREATE VIEW dbo.TheCalendarAS SELECT d.TheDate, d.TheDay, d.TheDaySuffix, d.TheDayName, d.TheDayOfWeek, d.TheDayOfWeekInMonth, d.TheDayOfYear, d.IsWeekend, d.TheWeek, d.TheISOweek, d.TheFirstOfWeek, d.TheLastOfWeek, d.TheWeekOfMonth, d.TheMonth, d.TheMonthName, d.TheFirstOfMonth, d.TheLastOfMonth, d.TheFirstOfNextMonth, d.TheLastOfNextMonth, d.TheQuarter, d.TheFirstOfQuarter, d.TheLastOfQuarter, d.TheYear, d.TheISOYear, d.TheFirstOfYear, d.TheLastOfYear, d.IsLeapYear, d.Has53Weeks, d.Has53ISOWeeks, d.MMYYYY, d.Style101, d.Style103, d.Style112, d.Style120, IsHoliday = CASE WHEN h.TheDate IS NOT NULL THEN 1 ELSE 0 END, h.HolidayText FROM dbo.DateDimension AS d LEFT OUTER JOIN dbo.HolidayDimension AS h ON d.TheDate = h.TheDate;**

And now you have a functional calendar view you can use for all of your reporting or business needs.

## **Summary**

Creating a dimension or calendar table for business dates and fiscal periods might seem intimidating at first, but once you have a solid methodology in line, it can be very worthwhile. There are many ways to do this; some will subscribe to the idea that many of these date-related facts can be derived at query time, or at least be non-persisted computed columns. You will have to decide if the values are calculated often enough to justify the additional space on disk and in the buffer pool.

If you are using Enterprise Edition on SQL Server 2014 or above, you could consider using In-Memory OLTP, and possibly even a non-durable table that you rebuild using a startup procedure. Or on any version or edition, you could put the calendar table into its own filegroup (or database), and mark it as read-only after initial population (this won't force the table to stay in memory all the time, but it will reduce other types of contention).

##### **Next Steps**

* Build a persisted calendar table to help with reporting queries, business logic, and gathering additional facts about given dates.

**Task 2:-**

# What are Slowly Changing Dimensions?

**Slowly Changing Dimensions (SCD)** - dimensions that change slowly over time, rather than changing on regular schedule, time-base. In Data Warehouse there is a need to track changes in dimension attributes in order to report historical data. In other words, implementing one of the SCD types should enable users assigning proper dimension's attribute value for given date. Example of such dimensions could be: customer, geography, employee.

There are many approaches how to deal with SCD. The most popular are:

* **Type 0** - The passive method
* **Type 1** - Overwriting the old value
* **Type 2** - Creating a new additional record
* **Type 3** - Adding a new column
* **Type 4** - Using historical table
* **Type 6** - Combine approaches of types 1,2,3 (1+2+3=6)

**Type 0** - The passive method. In this method no special action is performed upon dimensional changes. Some dimension data can remain the same as it was first time inserted, others may be overwritten.

**Type 1** - Overwriting the old value. In this method no history of dimension changes is kept in the database. The old dimension value is simply overwritten be the new one. This type is easy to maintain and is often use for data which changes are caused by processing corrections(e.g. removal special characters, correcting spelling errors).

Before the change:

|  |  |  |
| --- | --- | --- |
| **Customer\_ID** | **Customer\_Name** | **Customer\_Type** |
| 1 | Cust\_1 | Corporate |

After the change:

|  |  |  |
| --- | --- | --- |
| **Customer\_ID** | **Customer\_Name** | **Customer\_Type** |
| 1 | Cust\_1 | Retail |

**Type 2** - Creating a new additional record. In this methodology all history of dimension changes is kept in the database. You capture attribute change by adding a new row with a new surrogate key to the dimension table. Both the prior and new rows contain as attributes the natural key(or other durable identifier). Also 'effective date' and 'current indicator' columns are used in this method. There could be only one record with current indicator set to 'Y'. For 'effective date' columns, i.e. start\_date and end\_date, the end\_date for current record usually is set to value 9999-12-31. Introducing changes to the dimensional model in type 2 could be very expensive database operation so it is not recommended to use it in dimensions where a new attribute could be added in the future.

Before the change:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Customer\_ID** | **Customer\_Name** | **Customer\_Type** | **Start\_Date** | **End\_Date** | **Current\_Flag** |
| 1 | Cust\_1 | Corporate | 22-07-2010 | 31-12-9999 | Y |

After the change:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Customer\_ID** | **Customer\_Name** | **Customer\_Type** | **Start\_Date** | **End\_Date** | **Current\_Flag** |
| 1 | Cust\_1 | Corporate | 22-07-2010 | 17-05-2012 | N |
| 2 | Cust\_1 | Retail | 18-05-2012 | 31-12-9999 | Y |

**Type 3** - Adding a new column. In this type usually only the current and previous value of dimension is kept in the database. The new value is loaded into 'current/new' column and the old one into 'old/previous' column. Generally speaking the history is limited to the number of column created for storing historical data. This is the least commonly needed technique.

Before the change:

|  |  |  |  |
| --- | --- | --- | --- |
| **Customer\_ID** | **Customer\_Name** | **Current\_Type** | **Previous\_Type** |
| 1 | Cust\_1 | Corporate | Corporate |

After the change:

|  |  |  |  |
| --- | --- | --- | --- |
| **Customer\_ID** | **Customer\_Name** | **Current\_Type** | **Previous\_Type** |
| 1 | Cust\_1 | Retail | Corporate |

**Type 4** - Using historical table. In this method a separate historical table is used to track all dimension's attribute historical changes for each of the dimension. The 'main' dimension table keeps only the current data e.g. customer and customer\_history tables.

Current table:

|  |  |  |
| --- | --- | --- |
| **Customer\_ID** | **Customer\_Name** | **Customer\_Type** |
| 1 | Cust\_1 | Corporate |

Historical table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Customer\_ID** | **Customer\_Name** | **Customer\_Type** | **Start\_Date** | **End\_Date** |
| 1 | Cust\_1 | Retail | 01-01-2010 | 21-07-2010 |
| 1 | Cust\_1 | Oher | 22-07-2010 | 17-05-2012 |
| 1 | Cust\_1 | Corporate | 18-05-2012 | 31-12-9999 |

**Type 6** - Combine approaches of types 1,2,3 (1+2+3=6). In this type we have in dimension table such additional columns as:

* current\_type - for keeping current value of the attribute. All history records for given item of attribute have the same current value.
* historical\_type - for keeping historical value of the attribute. All history records for given item of attribute could have different values.
* start\_date - for keeping start date of 'effective date' of attribute's history.
* end\_date - for keeping end date of 'effective date' of attribute's history.
* current\_flag - for keeping information about the most recent record.

In this method to capture attribute change we add a new record as in type 2. The current\_type information is overwritten with the new one as in type 1. We store the history in a historical\_column as in type 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Customer\_ID** | **Customer\_Name** | **Current\_Type** | **Historical\_Type** | **Start\_Date** | **End\_Date** | **Current\_Flag** |
| 1 | Cust\_1 | Corporate | Retail | 01-01-2010 | 21-07-2010 | N |
| 2 | Cust\_1 | Corporate | Other | 22-07-2010 | 17-05-2012 | N |
| 3 | Cust\_1 | Corporate | Corporate | 18-05-2012 | 31-12-9999 | Y |

**Task 4:-**

**Create a stored procedure to implement SCD-type2 logic for a sample dimension table(take it any table).**

In Code Sample 1 below, we will create our staging table and our slowly changing dimension table.

**--=============================================================================-- Code Sample 1 --=============================================================================-- Create the staging table for the type two slowly changing dimension table datacreate table dbo.tblStaging( SourceSystemID int not null, Attribute1 varchar(128) not null constraint DF\_tblStaging\_Attribute1 default 'N/A', Attribute2 varchar(128) not null constraint DF\_tblStaging\_Attribute2 default 'N/A', Attribute3 int not null constraint DF\_tblStaging\_Attribute3 default -1, DimensionCheckSum int not null constraint DF\_tblStaging\_DimensionCheckSum default -1, LastUpdated datetime not null constraint DF\_tblStaging\_LastUpdated default getdate(), UpdatedBy varchar(50) not null constraint DF\_tblStaging\_UpdatedBy default suser\_sname())-- Create the type two slowly changing dimension tablecreate table dbo.tblDimSCDType2Example( SurrogateKey int not null identity(1,1) PRIMARY KEY, SourceSystemID int not null, Attribute1 varchar(128) not null constraint DF\_tblDimSCDType2Example\_Attribute1 default 'N/A', Attribute2 varchar(128) not null constraint DF\_tblDimSCDType2Example\_Attribute2 default 'N/A', Attribute3 int not null constraint DF\_tblDimSCDType2Example\_Attribute3 default -1, DimensionCheckSum int not null constraint DF\_tblDimSCDType2Example\_DimensionCheckSum default -1, EffectiveDate date not null constraint DF\_tblDimSCDType2Example\_EffectiveDate default getdate(), EndDate date not null constraint DF\_tblDimSCDType2Example\_EndDate default '12/31/9999', CurrentRecord char(1) not null constraint DF\_tblDimSCDType2Example\_CurrentRecord default 'Y', LastUpdated datetime not null constraint DF\_tblDimSCDType2Example\_LastUpdated default getdate(), UpdatedBy varchar(50) not null constraint DF\_tblDimSCDType2Example\_UpdatedBy default suser\_sname())**

**Task 5:-**

**Load the following configuration table in your database. Create a dynamic stored procedure that will work over the following configuration table(existing in your DB) to create new tables in database, if the status is ‘New’( along with the primary keys, Clustered indexes, Non- clustered indexes) or add columns for tables with status ‘Old’ with Alter command( which should run only once, even if there are multiple columns listed to be added in your ‘Old’ status tables.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **New Column** | **Data Types** | **Table** | **Table Type** | **Table Status** | **If\_New\_Table\_then\_PrimaryKey** | **Any\_Index(CLI)** | **Any\_Index(NCLI)** |
| Camp\_id | nvarchar(100) | Dim  Campaign | Dim | New | Yes | Yes |  |
| Total | float | Orders | Fact | Old |  |  | Yes |
| Camp\_type | nvarchar(100) | Dim  Campaign | Dim | New |  |  |  |
| Department | nvarchar(100) | Dim  Employees | Dim | Old |  |  |  |

The SQL CREATE INDEX statement is used to create clustered as well as non-clustered indexes in SQL Server. An index in a database is very similar to an index in a book. A book index may have a list of topics discussed in a book in alphabetical order. Therefore, if you want to search for any specific topic, you simply go to the index, find the page number of the topic, and go to that specific page number. Database indexes are similar and come handy. Particularly, if you have a huge number of records in your database, indexes can speed up the query execution process. There are two major types of indexes in SQL Server: clustered indexes and non-clustered indexes.

In this article, you will see what the clustered and non-clustered indexes are, what are the differences between the two types and how they can be created via SQL CREATE INDEX statement. So let’s begin without any further ado.

## **Creating dummy data**

The following script creates a dummy database named **BookStore** with one table i.e. Books. The Books table has four columns: **id**, **name**, **category**, and **price**:

**CREATE Database BookStore;**

**GO**

**USE BookStore;**

**CREATE TABLE Books**

**(**

**id INT PRIMARY KEY NOT NULL,**

**name VARCHAR(50) NOT NULL,**

**category VARCHAR(50) NOT NULL,**

**price INT NOT NULL**

**)**

Let’s now add some dummy records in the Books table:

**USE BookStore**

**INSERT INTO Books**

**VALUES**

**(1, 'Book1', 'Cat1', 1800),**

**(2, 'Book2', 'Cat2', 1500),**

**(3, 'Book3', 'Cat3', 2000),**

**(4, 'Book4', 'Cat4', 1300),**

**(5, 'Book5', 'Cat5', 1500),**

**(6, 'Book6', 'Cat6', 5000),**

**(7, 'Book7', 'Cat7', 8000),**

**(8, 'Book8', 'Cat8', 5000),**

**(9, 'Book9', 'Cat9', 5400),**

**(10, 'Book10', 'Cat10', 3200)**

The above script adds 10 dummy records in the Books table.

## **Clustered indexes**

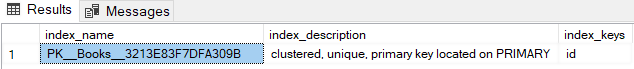
Clustered indexes define the way records are physically sorted in a database table. A clustered index is very similar to the table of contents of a book. In the table of contents, you can see how the book has been physically sorted. Either the topics are sorted chapter wise according to their relevance or they can be sorted alphabetically.

There can be only one way in which records can be physically sorted on a disk. For example, records can either be sorted by their ids or they can be sorted by the alphabetical order of some string column or any other criteria. However, you cannot have records physically sorted by ids as well as names. Hence, there can be only one clustered index for a database table. A database table has one clustered index by default on the primary key column. To see the default index, you can use the **sp\_helpindex** stored procedure as shown below:

**USE BookStore**

**EXECUTE sp\_helpindex Books**

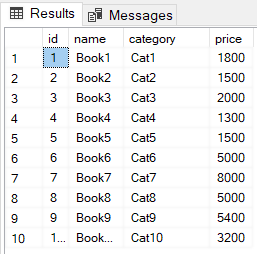
Here is the output:



You can see the clustered index name and the column on which the clustered index has been created by default.

To see the records arranged by default clustered index, simply execute the SELECT statement to select all the records from the books table:

**SELECT \* FROM Books**



You can see that the records have been sorted by default clustered index for the primary key column i.e. id.

To create a clustered index in SQL Server, you can modify SQL CREATE INDEX. Here is the syntax:

**CREATE CLUSTERED INDEX <index\_name>**

**ON <table\_name>(<column\_name> ASC/DESC)**

Let’s now create a custom clustered index that physically sorts the record in the Books table in the ascending order of the price. Since there can be only one clustered index, we first need to remove the default clustered index created via the primary key constraint. To remove the default clustered index, you simply have to remove the primary key constraint from the table that contains the default clustered index. Look at the following script:

**USE BookStore**

**ALTER TABLE Books**

**DROP CONSTRAINT PK\_\_Books\_\_3213E83F7DFA309B**

**GO**

Now we can create a new clustered index via SQL CREATE INDEX statement as shown below:

**USE BookStore**

**CREATE CLUSTERED INDEX IX\_tblBook\_Price**

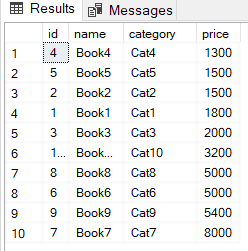
**ON Books(price ASC)**

In the script above, we create a clustered index named **IX\_tblBook\_Price**. This clustered index physically sorts all the records in the Books table by the ascending order of the price.

Let’s now select all the records from the Books table to see if they have been sorted in the ascending order of their prices:

**SELECT \* FROM Books**

Here is the output:



From the output, you can see that records have actually been sorted by the increasing amount of price.

## **Non-clustered indexes**

A non-clustered index is an index that doesn’t physically sort the database records. Rather, a non-clustered index is stored at a separate location from the actual database table. It is the non-clustered index which is actually similar to an index of a book. A book index is stored at a separate location, while the actual content of the book is separately located.

The SQL CREATE INDEX query can be modified as follows to create a non-clustered index:

**CREATE NONCLUSTERED INDEX <index\_name>**

**ON <table\_name>(<column\_name> ASC/DESC)**

Let’s create a simple non-clustered index that sorts the records in the Books table by name. You can modify the SQL CREATE INDEX query as follows:

**use BookStore**

**CREATE NONCLUSTERED INDEX IX\_tblBook\_Name**

**ON Books(name ASC)**

As I said earlier, the non-clustered index is stored at a location which is different from the location of the actual table, the non-clustered index that we created will look like this:

|  |  |
| --- | --- |
| **Name** | **Record Address** |
| Book1 | Record address |
| Book2 | Record address |
| Book3 | Record address |
| Book4 | Record address |
| Book5 | Record address |
| Book6 | Record address |
| Book7 | Record address |
| Book8 | Record address |
| Book9 | Record address |
| Book10 | Record address |

Now if a user searches for the name, id, and price of a specific book, the database will first search the book’s name in the non-clustered index. Once the book name is searched, the id and price of the book are searched from the actual table using the record address of the record in the actual table.

## **Conclusion**

The article covers how to use SQL CREATE INDEX statement to create a clustered as well as a non-clustered index. The article also shows the main differences between the two types of clustered indexes with the help of examples.