Window functions are allowed in SELECT and ORDER BY clauses(5 and 6):

1. FROM

2. WHERE

3. GROUP BY

4. HAVING

5. SELECT

1. Evaluate expressions

2. DISTINCT

6. ORDER BY

7. TOP/OFFSET-FETCH (what goes first? TOP or DISTINCT in the SELECT clause. DISTINCT goes first but note that TOP is evaluated after it)

TODO: add FPOC indexing strategy

|  |  |
| --- | --- |
| **SQL query solution patterns**  -a. what can be done with LEFT/RIGHT OUTER JOIN can also be done by a correlated scalar subquery in the select list that can then be renamed with an alias. But in-efficient. You repeat the scalar subquery for every value to be fetched. Examples can also be found in section -l. on window funcs.  -b. CROSS JOIN used for generating sequence of values (either using ROW\_NUMBER or table of 0 to 9 values) with TOP for cutoff, mimicking a loop and transposing/pivoting/unpivoting. For some recursive problems, recursive CTE can also be used in place of Loop but remember the recursive depth limit.  CROSS JOIN/CROSS APPLY not only is used to generate Getnums and then CROSS APPLY can be used with GetNums to mimic a loop. CROSS APPLY can also be used (as can the JOINs) to add move alias definition down to FROM clause (see h down below) and to add columns(as well as rows if relationship is 1-to-many) to result set. Solution defining computed columns which build on other computed columns is much more readable using APPLY. Also, CROSS APPLY works per row in the left hand side table. That is why window functions like row\_number don’t give expected result with CROSS APPLY. CASE expression in outer SELECT be replaced with a CROSS APPLY using a SELECT WHERE statement or VALUES clause surrounding the case expression or SELECT CASE.  -c. when performing JOINs, think carefully before using AGGREGATE functions if the relationship is not one-to-one. Row repetition! Tables that form the ‘many’ part of the relationship have to be aggregated before joining.  -d. FULL OUTER JOIN is the UNION of results of LEFT and RIGHT OUTER JOINs.  -e. use EXISTS and NOT EXISTS where ever you IN and NOT IN is being used.  -f. concatenate row values to create a list: recursive CTE, could possibly do with CROSS APPLY(?), XML PATH(‘’) trick and string\_agg (sql server 2017). CROSS APPLY would not work as it is applied per row(from left hand side) and concat with cross apply would need all the previous concatenations. Using a variable did not work in CROSS APPLied CONCAT as I could not find a way to store the result of each concatenation in that var.  -g. split a list of comma separated numbers(number array) into individual values: CROSS APPLY to walk the string splitting it or use string\_split(sql server 2016). Note that splitting combined with concatenating will solve many problems related to strings.  -h. CROSS APPLY can also be used to move the computed or hardcoded alias from the SELECT phase to the FROM phase. Since FROM is the very first phase, all subsequent phases then can use the alias. Can do this with CTE but if we keep on building on the computed alias, we need multiple CTE built on top of each other. But CROSS APPLY being a table operator, makes the result available to subsequent CROSS APPLY operators in the same FROM clause.  -m. Use JOINs/APPLY/’corelatedsubquery in SELECT list’ whenever you have to reference prior/previous or post/next rows. Can also be done with lag or lead WINDOW funcs. If you want to include the edge cases in LAG/LEAD, you would have to use OUTER APPLY/OUTER JOIN.LAG/LEAD take an ORDER BY argument where as for JOIN/APPLY we could use NON-EQUI JOIN on the column deciding the order along with aggregate funcs MIN/MAX on that column to give us an order without explicitly specifying ORDER BY. Note that with LEFT OUTER JOIN, we would have to use GROUP BY on columns from left had side table(see that use in (i) running total using INNER JOIN section as well).  -n. When you want to get distinct combinations(not permutation) of two columns(order does not matter), then CROSS JOIN(or a INNER JOIN) with => or <= on the filtering/joining column. In case of same values for some columns, use DISTINCT. As template rule, always use DINSTINCT with <=.  -o. gaps and island: can we solve them without window funcs?variations of problems:  One numeric column, one date column solved using lead, lag and dense\_rank/row\_number funcs. One date column gap variation involves finding identifying islands while ignoring gaps upto a certain size. Problems with 2 columns(startdate and enddate) might look like gaps and island problems but are not exactly same as we need to find a column which gives us the sort order. Finding islands is easy but gaps hard??  Now note that for both the types of problems, you are trying to refer to either the row before or row after(depending on how dates are sorted) to work out gaps or continuation. Refer -m, -a and -i.  The 2 columns gaps and islands is also correlated with -y. ‘packing intervals’. You just have to use max(lag) for startOfRangeMarker condition.  -s. recursive CTE: remember anchor member is not the base condition but just a starting row. The recursive member of CTE has to either return a empty result set either by itself or by adding a WHERE predicate. We can either move up or bottom of the tree by using appropriate JOIN condition in recursive member.  -v. sorted hierarchy: tree structure of employees and their managers. Employees under a manager need to be sorted in some order.  -w. conditional aggregates: calculate running total which resets to 0 if current value is negative and then starts again from there.  -x. Concurrent sessions: you might think it is similar to gaps and islands for 2 date problems but it is not. It usually involves datetime values and you check a particular starttime falls between how many start and end times. Have to work how it is different from island | -i. return rows in random order each time query is executed. Without ORDER BY, the result is set is not guaranteed to be in any order (as it is a set) but still the result rows are in the same random order (again that is not guaranteed). Use NEWID(). We can use a number in ORDER BY to refer a column by its position number or we an use col name or we can use a built-in function. SYSDATETIME would not produce random ordering. NEWID() not producing random ordering inside of OVER clause with ROW\_NUMBER()?  -j. u can unroll the ‘GROUP BY’ rowset inside the ‘OVER’ clause.  -k. TOP N per GROUP. Use ROW\_NUMBER() for low density case and CROSS APPLY with TOP for high density case. Pattern is that since both TOP and OFFSET-FETCH can’t work with GROUP(you would lose the detail), you either use GROUP(aka PARTITION) with ROW\_NUMBER or make use of CROSS APPLY with TOP(N)(or OFFSET-FETCH) with the left hand side providing you the GROUPed column.  Use POC(partitioning, ordering and covering) index strategy.  Partitioning: custid  Ordering: orderdate desc, ordered desc  Covering: empid  For e.g.: 10,000,000 rows in orders table. Index row size is 4+ 8 + 8+4=24 bytes. Roughly 8000/24 = 333 rows per page. So about 10,000,000/333 = 30030 pages in the leaf level of the POC index.  Low density case – 1,000,000 cust X 10 orders, seeks ~3,000,000 random reads(seek), scans ~30,000 seq reads(scan)  We do not want a plan that does seek per customer as then we would have 3 X 1,000,000 random reads where 3 would be the number of levels in the b-tree with 10,000,000 rows. Not good. A scan would have used just 30030 sequential reads (30030 is the total number of pages). So we want to use a scan in this case.  High density case – 10 cust X 1,000,000 orders, seeks ~30 random reads(seek), scans ~30030 seq reads(scan)  We want a plan that does seek per cust as then we would have 3 X 10 random reads where 3 would be the number of levels in the b-tree with 10,000,000 rows. Very good as compared to a scan which would have used just 30030 sequential reads (30030 is the total number of pages). So we want to use a seek in this case.  Another way of looking at density is that a column with high selectivity(uniqueness) will have low density and a col with low selectivity will have high density. Density = 1/(no. of distinct values in a col). The lower the col density, the more suitable it is for nci use.  -l. WINDOW functions allow to mix details with aggregates. The same can be done with scalar subqueries in the SELECT list. The other technique to do the same is use aggregates separately and JOINing/APPLYing the results (good performance).  ROW\_NUMBER: window func or scalar subquery or LEFT OUTER JOIN or CROSS APPLY. Except window func, the other 2 require UNIQUE column for counting the rows before a particular row. Does a solution exist for case when no UNIQUE col exist? Either you can APPLY NEWID() to each row, save result to temp table and then using it as unique column OR APPLY GetNums. APPLYing GetNums is inefficient as it does Cartesian product.  Rank and Dense Rank: what has been said for Row\_number goes for rank as well. But dense\_rank functionality can be replicated using scalar subquery (and I think with JOINs) if we use COUNT(DISTINCT e1.DEPTNO).  Running total: window function or scalar subquery or CROSS APPLY or INNER JOIN(<= join) with GROUPing on unique columns from on the 1 side of 1..\*relationship. Easier thing is just add all the LEFT hand side columns to GROUP BY clause.  If the SQL server version does not have window funcs and the table has a lot of rows, then instead of scalar subquery or INNER JOIN, it is better to use a cursor because it scales linearly while other solutions scale n2 . Even better than TSQL cursor is CLR based solution.  Lead: window function or scalar subquery or CROSS APPLY or LEFT OUTER JOIN(haven’t tried it)  NTILE: window function or using CROSS APPLY with OFFSET-FETCH or row\_number with mod operator(tiles are not the same as given by NTILE)  -p. TOP can not only be implemented with OFFSET-FETCH but also with WHERE clause.  -q. where we do not have a grouping column, aggregate funcs such as MAX, MIN can be used to collapse the rows given we have only 1 non-null value in each column. Another way to think about it is that aggregate functions is alternate way to a WHERE filtering clause or a CASE expession for removing NULL values.  -r. sparse reports make use of row\_number or RANK funcs.  -t. NULL is a value. So a result with a NULL value is different from a empty result set. Empty result set can be generated using a FALSE predicate. But both empty result set and a result with a NULL value can be verified using IS NULL predicate. Why would a empty result set  -u. GROUP BY can used in some cases for DISTINCT and vice versa but sometimes it is convenient to use DISTINCT  -y. packing intervals: You might have to add a partition column for windows but the island logic will work but with a twitst. The twist is that island assumes no one date range completely encompasses the other date range and dates are sort of chained together and we can find where the chain breaks. What that means is that the end date of succeeding task would would always be greater than the max of all the previous dates in that island. But not in a packing interval.  `  Blues starts first and then yellow and then red. The issue is yellow finishes before the start of red and thus it creates what seems like a gap and the chain is broken. If we do not want to consider this as a gap but as an island(denoted by boundaries of blue), then we have to take max of lag values to compare for rangemarkers.  **Islands looks for unbroken chain links whereas in packing interval we are looking for real ‘islands’.** |

Examples:

|  |  |
| --- | --- |
| -b. CROSS APPLY takes an table or an table expression(which includes DTE, CTE, VIEW and TVF) and applies it to every row from LHS. When it is non-corelated, it behaves similar to CROSS JOIN. When it is used with co-related subquery/DTE, it behaves similar to INNER JOIN in that you get matching rows. When it is used with a co-related TVF, then that functionality can't be replicated either with CROSS JOIN or INNER JOIN. **Co-related tvf funcs do not work with CROSS JOINs and INNER JOINs**. OUTER APPLY provides functionality of LEFT OUTER JOIN.  CROSS APPLY is similar to CROSS JOIN as both do a Cartesian product and when the right hand side result set is empty, the corresponding row from left hand side is also removed from query result. CROSS APPLY is similar to INNER JOIN that it can match rows(or think of it as filter after cartesian).  select \*  from Sales.Customers as c  --works  --cross apply GetNums(1,1) as b  --cross join GetNums(1,1) as b  --inner join GetNums(1,1) as b on 1=1  --inner join ( select \* from GetNums(1,1))as b(n) on 1=1  --inner join Sales.Customers as s on s.custid = c.custid  --cross apply (select \* from Sales.Customers as s where s.custid = c.custid) as b  --cross apply GetNums(c.custid,c.custid+1)  --does not work  --cross join GetNums(c.custid,c.custid+1) as b  --inner join GetNums(c.custid,c.custid+1) as b on 1=1  --inner join ( select \* from GetNums(c.custid,c.custid+1))as b(n) on 1=1  ;with c as  (select \*  from (values(0),(1)) as b(n)  )  select c1.n as c1n, c2.n as c2n  from c as c1 cross apply (select \* from c where c.n = c1.n) as c2  --error  ;with c as  (select \*  from (values(0),(1)) as b(n)  )  select \*  from c as c1 cross join (select \* from c where c.n = c1.n) as c2  SELECT \*  FROM (VALUES(1)) AS A(N)  CROSS APPLY (SELECT \* FROM (VALUES(1)) AS C(NN) WHERE C.NN = A.N) AS B(NN)  --error  SELECT \*  FROM (VALUES(1)) AS A(N)  CROSS JOIN (SELECT \* FROM (VALUES(1)) AS C(NN) WHERE C.NN = A.N) AS B(NN)  --note the use of SELECT-WHERE without FROM clause. It could be rewritten as CASE which could then be used with SELECT or in inside VALUES clause. The ordDate value can be moved into a VALUES clause to enable use of a FROM clause. See below  select v.\*, n, [verifiedDate], nn  from (values(1,'19890101'), (2,'19890102'), (3,'19890103')) as v(id,ordDate)  cross apply GetNums(1,3) as gn  outer apply (select dateadd(day,1,ordDate) as [verifiedDate] where n=2 or n=3) as a1([verifiedDate])  outer apply (values(case when n=2 then 22 end)) as a3(nn)  --extraction of CASE expression from outer SELECT to be used with CROSS APPLY.  select \*, case when n=1 then 'aaho' else 'jaande' end as nn  from (values(1),(2)) as a(n)  select \*  from (values(1),(2)) as a(n)  cross apply (values(case when a.n=1 then 'aaho' else 'jaande' end )) as b(nn)  select \*  from (values(1),(2)) as a(n)  cross apply (select case when a.n=1 then 'aaho' else 'jaande' end as nn) as b(nn)  --use of SELECT-WHERE without FROM clause. this and the following 3 examples can be used to move  --case expression from outer select to OUTER APPLied subquery expression  select dateadd(day,1,GETDATE()) as [verifiedDate] where 2=2 or 3=3  --values clause being used to provide base data  select DATEADD(day,1,dt)  from (values(GETDATE())) as b(dt)  where 1=1 or 3=3  --case being used in select  select case when (1=1 or 3=3) then dateadd(day,1,GETDATE()) end  --case being used in values clause  select \*  from (values(case when (1=1 or 3=3) then dateadd(day,1,GETDATE()) end)) as b(dt)  -O.  CREATE TABLE dbo.T1(col1 INT NOT NULL CONSTRAINT PK\_T1 PRIMARY KEY);  GO  INSERT INTO dbo.T1(col1) VALUES(1),(2),(3),(7),(8),(9),(11),(15),(16),(17),(28);  --gaps  ;with cte as  (  select col1 as curr, lead(col1) over(order by col1) as nxt  from t1  )  select curr +1 as rangeFrom, nxt -1 as rangeTo  from cte  where nxt - curr > 1  --islands..note did not use row\_number/rank as col1 might not be unique  ;with cte as  (  select col1 as curr, col1 - dense\_rank () over(order by col1) as grp  from t1  )  select MIN(curr) as rangeFrom, max(curr) as rangeTo  from cte  group by grp  --drop table t1  --CREATE TABLE dbo.T1(shipdate date);  --GO  --INSERT INTO dbo.T1(shipDate) VALUES('20070101'),('20070102'),('20070102'),('20070103')  ,('20070106'),('20070107'),('20070109'),('20070109');  --gaps with dates  ;with cte as  (  select shipDate as curr, lead(shipDate) over(order by shipDate) as nxt  from t1  )  select dateadd(day,1,curr) as rangeFrom, dateadd(day, -1, nxt) as rangeTo  from cte  where datediff(day, curr, nxt) > 1  --islands with dates..note did not use row\_number/rank as col1 might not be unique  ;with cte as  (  select shipDate as curr, dateadd(day, -1 \* dense\_rank() over(order by shipDate) , shipDate )as grp  from t1  )  select MIN(curr) as rangeFrom, max(curr) as rangeTo  from cte  group by grp  --drop table t1;  --CREATE TABLE dbo.T1(projID int,startDate date, endDate date);  --INSERT INTO dbo.T1(projID, startDate, endDate)  -- VALUES(1,'20070101','20070103')  -- ,(2,'20070102','20070104')  -- ,(3,'20070104','20070105')  -- ,(4,'20070106','20070107')  -- ,(5,'20070109','20070109')  -- ,(6,'20070109','20070110');  --not exactly islands...gives us range of consecutive projects  SELECT t1.\*  FROM T1  inner JOIN T1 AS T2 ON T1.projID != T2.projID and (T1.endDate = T2.startDate or T1.startDate = T2.endDate)  --islands..stand alone rows will be included as well as that is also an island. Define a new sort column. Here I am using endDate for sorting but down below I have the same worked out with startDate sorted logic which seems a bit more comprehensible to me.  ;with base as  (  select \*, row\_number() over(order by endDate) as sortColumn  from T1  ),  cte as  (  SELECT t1.projID, t1.startDate, t1.endDate, t1.sortColumn, case when (select max(endDate) from base as s where s.sortColumn < t1.sortColumn) >= T1.startDate then 0 else 1 end  as flag  FROM base as T1--scalar subquery left could also be moved to outer join or cross apply  ),  cte2 as  (  select c.projID, c.startDate, c.endDate, (select sum(b.flag) from cte as b where b.sortColumn <= c.sortColumn) as grp  from cte as c  )  select MIN(startDate) as rangeFrom, max(endDate) as rangeTo  from cte2  group by grp  --this time using window function to define the grp  ;with base as  (  select \*, row\_number() over(order by endDate) as sortColumn  from T1  ),  cte as  (  SELECT t1.projID, t1.startDate, t1.endDate, t1.sortColumn, case when (select max(endDate) from base as s where s.sortColumn < t1.sortColumn) >= T1.startDate then 0 else 1 end  as flag  FROM base as T1--scalar subquery left could also be moved to outer join or cross apply)  ,cte2 as  (  select c.projID, c.startDate, c.endDate, sum(c.flag) over(order by sortColumn) as grp  from cte as c  )  select MIN(startDate) as rangeFrom, max(endDate) as rangeTo  from cte2  group by grp  --this time using lag instead of scalar subquery.  ;with base as  (  select \*, row\_number() over(order by endDate) as sortColumn  from T1  ),  cte as  (  SELECT t1.projID, t1.startDate, t1.endDate, t1.sortColumn  ,case when lag(T1.endDate) over (order by T1.sortColumn) >= T1.startDate then 0 else 1 end  as flag  FROM base as T1  ),  cte2 as  (  select c.projID, c.startDate, c.endDate, sum(c.flag) over(order by sortColumn) as grp  from cte as c  )  select MIN(startDate) as rangeFrom, max(endDate) as rangeTo  from cte2  group by grp  --sort by startDate logic for islands  ;with ordered as  (  select \*, ROW\_NUMBER() over(order by startdate, enddate) as rn  from t1  )  ,previousVals as  (  select \*, lag(enddate) over(order by startdate, enddate) as lg  from ordered  )  ,startOfRangeCol as  (  select \*  , case when DATEDIFF(day,lg,startdate)< 1 then 0 else 1 end as startOfRangeMarker  , DATEDIFF(day,lg,startdate) as dtDiff  from previousVals  )  ,sumOfstartOfRangeColToBeUsedAsGrouping as  (  select \*, sum(startOfRangeMarker) over (order by startdate, enddate rows between unbounded preceding and current row) as SumOfRangeMarker  from startOfRangeCol  )  select min(startDate), max(enddate)  from sumOfstartOfRangeColToBeUsedAsGrouping  group by SumOfRangeMarker  --once you have islands, you can easily work out the gaps. Just order the islands by startDate and use lead for startDate along with current endDate to work out gaps  ;with ordered as  (  select \*, ROW\_NUMBER() over(order by startdate, enddate) as rn  from t1  )  ,previousVals as  (  select \*, lag(enddate) over(order by startdate, enddate) as lg  from ordered  )  ,startOfRangeCol as  (  select \*  , case when DATEDIFF(day,lg,startdate)< 1 then 0 else 1 end as startOfRangeMarker  , DATEDIFF(day,lg,startdate) as dtDiff  from previousVals  )  ,sumOfstartOfRangeColToBeUsedAsGrouping as  (  select \*, sum(startOfRangeMarker) over (order by startdate, enddate) as SumOfRangeMarker  from startOfRangeCol  )  ,islands as  (  select min(startDate) as rangeStart, max(enddate) as rangeEnd  from sumOfstartOfRangeColToBeUsedAsGrouping  group by SumOfRangeMarker  )  select rangeEnd as gapStart, lead(rangeStart) over(order by rangeStart) as gapEnd  from islands | -b. Reference names for column needed for pivoting are: categorical that becomes new column names, data column whose values are spread under the new columns and the ‘for’ column. Now the ‘for’ column is not needed and you can do with only 2 columns but in that case the there should only be one data column value for each category. Make it a rule to add the ‘for’ column when it is missing.  PIVOT to spread the values even when the categorical data to be used for spreading does not exist as such but can be inferred. PIVOT should not be confused with Transpose.  SELECT D, [0] AS DEPTNO, [1] AS [NAME]  FROM (SELECT CTE2.D, CTE2.value, CASE WHEN ISNUMERIC(CTE2.value) = 0 THEN 1 ELSE 0 END AS IsNumber FROM CTE2) AS BASE  PIVOT(MAX(BASE.VALUE) FOR IsNumber IN ([1],[0])) AS PVT  To UNPIVOT multiple sets of columns, can’t use UNPIVOT operator. Use CROSS APPLY:  --custid, qty2012, qty2013, qty2013, val2012, val2013, val2014  --becomes custid, orderyear, qty, val  select s.custid, a.\*  from dbo.sales as s  cross apply (values(2012,qty2012, val2012),  (2013,qty2013, val2013),  (2014,qty2014, val2014)) as a(orderyear, qty, val)  CASE + GROUP BY can be used instead of PIVOT:  --use pivot to display calendar.  select week, [Monday], [Tuesday], [Wednesday], [Thursday], [Friday], [Saturday], [Sunday]  from  (select week, currentDate, weekdayName  from (values('20070201','20070301')) as v(startOfMonth,startOfNextMonth)  cross apply (select \* from GetNums(0,DATEDIFF(day,startOfMonth,startOfNextMonth))) as gn(n)  cross apply (values(DATEADD(day,gn.n,startOfMonth))) as a3(currentDate)  cross apply (values(DATENAME(dw,currentDate))) as a4(weekdayName)  cross apply (values(DATENAME(ISO\_WEEK,currentDate))) as a5(week)  ) as base  pivot(max(currentDate) for weekdayName in ([Monday], [Tuesday], [Wednesday], [Thursday], [Friday], [Saturday], [Sunday])) as pvt  --use CASE with GROUP BY to display calendar  select a5.week,  max(case when weekdayName='Monday' then currentDate end) as 'Monday',  max(case when weekdayName='Tuesday' then currentDate end) as 'Tuesday',  max(case when weekdayName='Wednesday' then currentDate end) as 'Wednesday',  max(case when weekdayName='Thursday' then currentDate end) as 'Thursday',  max(case when weekdayName='Friday' then currentDate end) as 'Friday',  max(case when weekdayName='Saturday' then currentDate end) as 'Saturday',  max(case when weekdayName='Sunday' then currentDate end) as 'Sunday'  from (values('20070201','20070301')) as v(startOfMonth,startOfNextMonth)  cross apply (select \* from GetNums(0,DATEDIFF(day,startOfMonth,startOfNextMonth))) as gn(n)  cross apply (values(DATEADD(day,gn.n,startOfMonth))) as a3(currentDate)  cross apply (values(DATENAME(dw,currentDate))) as a4(weekdayName)  cross apply (values(DATENAME(ISO\_WEEK,currentDate))) as a5(week)  group by a5.week  --you do not need the ‘for’ column but it is a good idea to add it as explained below(it will handle more than data value for a category)  select \*  into #base  from (values(10,3), (20,5), (30,6)) as v(value,cnt)  select [3] as [cnt3], [5] as [cnt5], [6] as [cnt6]  from #base  pivot(max(value) for cnt in ([3], [5], [6])) as pvt  select max(case when cnt=3 then value end) as cnt3, max(case when cnt=5 then value end) as cnt5, max(case when cnt=6 then value end) as cnt6  from #base  --you have 2 columns and have more than one data value for a category column(either unique or non-unique values)  --then you have to add differentiating 'for' column. Add row\_number  select \*  into #base  from (values('clerk','david'), ('clerk', 'david'), ('manager', 'david'), ('manager', 'bai'), ('manager', 'tom')) as v(job,ename)  --won't work  select [clerk], [manager]  from #base  pivot(max(ename) for job in ([clerk], [manager])) as pvt  --this will make it work  select \*, ROW\_NUMBER() over (partition by job order by (select null)) as rn  into #base1  from (values('clerk','david'), ('clerk', 'david'), ('manager', 'david'), ('manager', 'bai'), ('manager', 'tom')) as v(job,ename)  select [clerk], [manager]  from #base1  pivot(max(ename) for job in ([clerk], [manager])) as pvt  select max(case when job='clerk' then ename end) as [clerk], max(case when job='manager' then ename end) as [manager]  from #base1  group by rn  --this shows 1) creation of a categorical column(groupId) and then 2) since each category has more than one  --values, adding a row\_number for each category(by using partition by on the category column) and then  --3) pivoting either by using PIVOT or GROUP BY-MAX-CASE  with cte1 as  (select \*, concat(ENAME,'(', cast(sal as varchar), ')') as name  , DENSE\_RANK() over(order by sal desc) as dr  from EMP  ),  cte2 as  (select \*, case when dr<=3 then 1 when dr>3 and dr<=6 then 2 else 3 end as groupId  from cte1  ),  cte3 as  (  select \*, row\_number() over(PARTITION BY groupId order by sal desc) as rn from cte2  )  select \* into #base from cte3  select MAX(case when groupid = 1 then name end) as [1], MAX(case when groupid = 2 then name end) as [2]  , MAX(case when groupid = 3 then name end) as [3]  from #base  group by rn  select [1], [2], [3]  from (select b.name, b.rn, b.groupId from #base as b) AS B  pivot (max(name) for groupid in ([1],[2],[3])) as pvt  -p.  ;with cte as  (select \*, ROW\_NUMBER() over(order by sal) as rn  from emp  )  select \*  from cte  order by rn  offset 0 rows fetch next 5 rows only;  ;with cte as  (select \*, ROW\_NUMBER() over(order by sal) as rn  from emp  )  select top(5) \*  from cte  order by rn  ;with cte as  (select \*, ROW\_NUMBER() over(order by sal) as rn  from emp  )  select \*  from cte  where rn between 1 and 5  -n.  ;with v as  (select \*  from (values(10,20), (20,10), (30,40), (80,130), (130,80), (5,5), (5,5), (1,3) ,(1,1)) as b(a,b)  )  select distinct v1.\*  from v as v1  inner join v as v2 on ((v1.a=v2.b and v2.a=v1.b ) and v1.a <= v1.b )  or not exists(select 1 from v as t where t.a = v1.b and t.b=v1.a)  -x. For green, 3 conc. sessions. For yellow, 2. For blue, 1.  Traditional set based solution. Inefficient for large partition sizes  ;WITH sessions as  (  select \*  from (VALUES (2, 'app1', 'user1', 'host1', '20120212 08:30', '20120212 10:30'),  (3, 'app1', 'user2', 'host1', '20120212 08:30', '20120212 08:45'),(5, 'app1', 'user3', 'host2', '20120212 09:00', '20120212 09:30'), (7, 'app1', 'user4', 'host2', '20120212 09:15', '20120212 10:30'), (11, 'app1', 'user5', 'host3', '20120212 09:15', '20120212 09:30'), (13, 'app1', 'user6', 'host3', '20120212 10:30', '20120212 14:30'), (17, 'app1', 'user7', 'host4', '20120212 10:45', '20120212 11:30'), (19, 'app1', 'user8', 'host4', '20120212 11:00', '20120212 12:30'), (23, 'app2', 'user8', 'host1', '20120212 08:30', '20120212 08:45'), (29, 'app2', 'user7', 'host1', '20120212 09:00', '20120212 09:30'), (31, 'app2', 'user6', 'host2', '20120212 11:45', '20120212 12:00'), (37, 'app2', 'user5', 'host2', '20120212 12:30', '20120212 14:00'), (41, 'app2', 'user4', 'host3', '20120212 12:45', '20120212 13:30'), (43, 'app2', 'user3', 'host3', '20120212 13:00', '20120212 14:00'), (47, 'app2', 'user2', 'host4', '20120212 14:00', '20120212 16:30'), (53, 'app2', 'user1', 'host4', '20120212 15:30', '20120212 17:00'))  as s(keycol, app, usr, host, starttime, endtime)  )  ,Counts AS  (  SELECT app, starttime, endtime, (SELECT COUNT(\*) FROM Sessions AS S WHERE S.app = P.app AND S.starttime <= P.starttime AND S.endtime > P.starttime) AS concurrent  FROM Sessions AS P  )  SELECT app, MAX(concurrent) AS mx  FROM Counts  GROUP BY app;  --organize start and end events as chronological sequence and then computes a running total using the event 'type' where +1 represents a start event and -1 represents an end. Now the same thing can be done with a cursor as well which would be a bit slower but not as slow as the traditional set based solution given above. In essence what we are doing here is doing a running total and then taking a max of it. And using cursors was one of the efficient solutions before sql srv 2012 when Window aggregate funcs were introduced.  ;with cte as  (  SELECT app, starttime AS ts, +1 AS type  FROM dbo.Sessions    UNION ALL    SELECT app, endtime, -1  FROM dbo.Sessions  )  ,runningTot as  (  select \*, sum(type) over(partition by app order by ts, type type rows between unbounded preceding and current row) as concurrent  from cte  )  select app, max(rnSum) AS mx  from runningTot  group by app;  --cursor based sol  declare @app as varchar(10), @t as datetime, @type as int, @prevapp as varchar(10);  declare @runningSum as int = 0 ;  declare @maxSessions as int = 0;  declare @tbl as table(app varchar(10), concSessions int);  declare cur cursor fast\_forward for  SELECT app, starttime AS ts, +1 AS type  FROM dbo.Sessions    UNION ALL    SELECT app, endtime, -1  FROM dbo.Sessions  ORDER BY app, ts, type;  open cur;  fetch next from cur into  @app, @t, @type;  set @prevapp = @app;  while @@FETCH\_STATUS = 0  Begin  if @prevapp<>@app  Begin  insert into @tbl values(@prevapp, @maxSessions);  set @maxSessions = 0;  set @runningSum = 0;  set @prevapp = @app;  end    set @runningSum = @runningSum + @type;  if @runningSum > @maxSessions  set @maxSessions = @runningSum;    fetch next from cur into  @app, @t, @type;  end  if @prevapp is not null  insert into @tbl values(@prevapp, @maxSessions);  close cur;  deallocate cur;  select \* from @tbl; |
| -b. **Loop examples using recursive CTE and CROSS APPLY**:  --USING CROSS APPLY WITH GETNUMS  DECLARE @STR AS VARCHAR(10) = 'KINGS';  SELECT SUBSTRING(@STR, S.N, 1) AS S  FROM (VALUES(1)) AS DUMMY(D)  CROSS APPLY (SELECT \* FROM GetNums(1, LEN(@STR))) AS S(N)  --USING RECURSIVE CTE  DECLARE @STARTPOS AS INT = 1;  ;WITH CTER  AS(  SELECT SUBSTRING(@STR, @STARTPOS, 1) AS S, @STARTPOS AS L  UNION ALL  SELECT SUBSTRING(@STR, L + 1 , 1) AS S, L + 1  FROM CTER AS R  WHERE SUBSTRING(@STR, L + 1 , 1) <> ''  )  SELECT \*  FROM CTER  ;with c as  (select \*  from (values(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) as b(n)  )  ,nums as  (select CONCAT(c4.n,c3.n,c2.n,c1.n,c0.n) as str, cast(CONCAT(c4.n,c3.n,c2.n,c1.n,c0.n) as int) + 1 as number  from c as c0 cross join c as c1 cross join c as c2 cross join c as c3 cross join c as c4  )  select \*  from nums  order by number    WITH  L0 AS (SELECT 0 AS C FROM (VALUES(1),(1)) AS D(C)),--2  L1 AS (SELECT 0 AS C FROM L0 AS A CROSS JOIN L0 AS B ),--4  L2 AS (SELECT 0 AS C FROM L1 AS A CROSS JOIN L1 AS B ),--16  L3 AS (SELECT 0 AS C FROM L2 AS A CROSS JOIN L2 AS B ),--256  L4 AS (SELECT 0 AS C FROM L3 AS A CROSS JOIN L3 AS B ),--65536  L5 AS (SELECT 0 AS C FROM L4 AS A CROSS JOIN L4 AS B ),--4294967296  NUMS AS (SELECT ROW\_NUMBER() OVER(ORDER BY (SELECT NULL)) AS ROWNUM FROM L5)  SELECT TOP(500000 - 0 + 1) 0 + ROWNUM -1 AS N  FROM NUMS  ORDER BY ROWNUM  -h.  select n, a1.newnumber, newnumber2  from (values(1), (3), (4)) as b(n)  cross apply (select n+10) as a1(newnumber)  cross apply (select newnumber+10) as a2(newnumber2)  select n, a1.newnumber, newnumber2  from (values(1), (3), (4)) as b(n)  cross apply (values(n+10)) as a1(newnumber)  cross apply (values(newnumber+10)) as a2(newnumber2)  -j.The expressions used inside window funcs should either be aggregate or part of Group By list. Note the nested SUM below:  select AVG(sal), ROW\_NUMBER() over(partition by deptno order by sum(sal)) as rn  --,ROW\_NUMBER() over(partition by deptno order by sal) as rn1 --can't do  --, lag(sal) over(partition by deptno order by sum(sal)) as previousRow --can't do this  --, lag(sum(sal)) over(partition by deptno order by sum(sal)) as previousRow – this would work  from emp  group by DEPTNO, JOB  ;with cte as  (  select 'act' + cast(act.n as varchar) as act,  'trn' + cast(trn.n as varchar) as trn,  trn.n as num  from GetNums(1,4) as act  cross apply GetNums(1,2) as trn  )  select act,trn, SUM(num), SUM(SUM(num)) over(partition by act), lag(SUM(num)) over(partition by act order by sum(num))  from cte  group by act,trn | -f. what we are looking for is a aggregation function(like SUM, COUNT, et.) which will concatenate all the values in a column used in conjunction with GROUP By or on the full column set. STRING\_AGG does it but before that we either had to do the concatenation piecemeal one column value at a time using concat or use pseudo aggregation functions like STUFF for same  --CONCATENATE COLUMN VALUES TO A LIST USING RECURSIVE CTE  --slow but handles special XML characters  ;with base as  (  select \* from (values(23),(24),(25),(26),(27),(28)) as b(val)  )  ,cte as  (  select val, ROW\_NUMBER() over(order by val) as rn from base  )  ,RecursiveCTE as  (  select cast(val as varchar(max)) as list, rn from cte  where rn=1  union all  select CONCAT(r.list,',',c.val) as list, c.rn from RecursiveCTE as r  inner join cte as c on c.rn = r.rn + 1  )  select top(1) \* from RecursiveCTE  order by rn desc;  ;WITH CTE AS  (  SELECT \*  FROM (VALUES('LAPTOP', 'DELL<'), ('LAPTOP', 'HP'), ('LAPTOP', 'COMPAQ'), ('MAC', 'APPLE')) AS P(PR, SUPP)  )  ,CTE1 AS  (  SELECT \*, ROW\_NUMBER() OVER(PARTITION BY PR ORDER BY SUPP) AS RN  FROM CTE  ),  RECURSIVECTE AS  (  SELECT PR, CAST(CONCAT(SUPP, '') AS VARCHAR) AS LIST, RN  FROM CTE1  WHERE RN = 1  UNION ALL  SELECT R.PR, CAST(CONCAT(R.LIST, ',' + C.SUPP) AS VARCHAR) AS LIST , C.RN  FROM RECURSIVECTE AS R  INNER JOIN CTE1 AS C ON C.RN = R.RN + 1 AND R.PR = C.PR  )  , REVERSEORDER AS  (  SELECT \*, ROW\_NUMBER() OVER(PARTITION BY PR ORDER BY RN DESC) AS REVERSEORDERING  FROM RECURSIVECTE  )  SELECT \*  FROM REVERSEORDER  WHERE REVERSEORDERING = 1  --CONCATENATE COLUMN VALUES TO A LIST USING XML PATH. Note how STUFF is being used as a aggregate function with GROUP BY. STRING\_AGG below does that aggregation.  --Note: SPECIAL XML CHARACTERS LIKE ‘<’ WILL BE ENCODED. Workaround given in T-SQL querying.  ;WITH CTE AS  (  SELECT \*  FROM (VALUES('LAPTOP', 'DELL<'), ('LAPTOP', 'HP'), ('LAPTOP', 'COMPAQ'), ('MAC', 'APPLE')) AS P(PR, SUPP)  )  SELECT A.PR, STUFF((SELECT ',' + B.SUPP FROM CTE AS B WHERE B.PR = A.PR ORDER BY SUPP FOR XML PATH ('')), 1, 1, '')  FROM CTE AS A  GROUP BY PR;  --REQUIRES SQL SERVER 2017. Handles special XML characters  ;WITH CTE AS  (  SELECT \*  FROM (VALUES('LAPTOP', 'DELL<'), ('LAPTOP', 'HP'), ('LAPTOP', 'COMPAQ'), ('MAC', 'APPLE')) AS P(PR, SUPP)  )  SELECT A.PR, STRING\_AGG(SUPP, ',') WITHIN GROUP (ORDER BY SUPP ASC) AS List  FROM CTE AS A  GROUP BY PR; |
| -g.  --string split to individual entries  --using cross apply (atleast sql server 2005)  select id, arr, n as indexpos, SUBSTRING(arr, n, CHARINDEX(',', arr + ',', n) - n) as element  from (values('a','6,55,2'), ('b','5,44,6'), ('c','21')) as base(id,arr)  cross apply (select n from GetNums(1,len(base.arr)) as num where num.n = CHARINDEX(',', ',' + base.arr, num.n) ) as a  --generic string split function using cross apply(to be used before sql2016)  go  drop function if exists my\_string\_split  go  create function my\_string\_split(@str as varchar(max), @sep as char(1))  returns table  as  return  select SUBSTRING(@str, n, CHARINDEX(@sep, @str + @sep, n) - n) as element  from (select n from GetNums(1,len(@str)) as num where num.n = CHARINDEX(@sep, @sep + @str, num.n) ) as a  go  select id, arr, sp.element  from (values('a','6,55,2'), ('b','5,44,6'), ('c','21')) as base(id,arr)  cross apply my\_string\_split(base.arr, ',') as sp    --string\_split in sql2016  select \*  from (values(1), (3), (4)) as b(n)  where b.n in (select \* from string\_split('1,2,3', ',')) | -i.  select top 2\*  from EMP  order by (select null)  select top 2\*  from EMP  order by (select 1)  select top 2\*  from EMP  order by (2) –ORDERING BY COLUMN INDEX, NOT A CONSTANT  select top 2\*  from EMP  order by (NEWID())  --newid() does not seem to work with row\_number to give a random ordering for each execution  select \*, ROW\_NUMBER() over(order by (select null)) as rn  from EMP  select \*, ROW\_NUMBER() over(order by (select 1)) as rn  from EMP  select \*, ROW\_NUMBER() over(order by (select sysdatetime())) as rn  from EMP  select \*, ROW\_NUMBER() over(order by (select newid())) as rn  from EMP |
| -k. The first case uses scan and the second uses seek. Note that ROW\_NUMBER() used a scan and that is what we want. Having the index in place saves us sorting. You can generate custom data using utility presented at the end(don’t forget to create FPOC index) to test different density cases.  --low density case  --create index idx\_my on Sales.Orders(custid, orderdate desc, orderid desc ) include(empid)  ;with cte as  (  select custid, orderdate, orderid, empid, ROW\_NUMBER() over(partition by custid order by orderdate desc, orderid desc ) as rn  from Sales.Orders  )  select custid, orderdate, orderid, empid  from cte  where rn <=3  --high density case. This is also showcasing the parallel apply technique which could improve performance(page112 win funcs book)  --create index idx\_my on Sales.Orders(custid, orderdate desc, orderid desc ) include(empid)  select so.custid, a.\*  from sales.Customers as so  cross apply (select top(3) orderdate, orderid, empid from sales.orders as so1 where so1.custid = so.custid order by orderdate desc, orderid desc) as a  -m.  select e.HIREDATE as currEmpHireDate, Prev.hiredate  from EMP as e  outer apply (select max(e1.HIREDATE) as prevEmpHireDate from emp as e1 where e.HIREDATE> e1.HIREDATE) as Prev(hiredate)  select e.HIREDATE as currEmpHireDate, Max(e1.HIREDATE)  from EMP as e  left outer join EMP as e1 on e.HIREDATE > e1.HIREDATE  group by e.EMPNO, e.HIREDATE  -t.  --EXISTS treats NULL as a value and empty result set as no values aka empty  --selects 10 rows  select top 10 \*  from Transactions  where exists(select null)  --selects empty result set. 0 rows  select top 10 \*  from Transactions  where exists(select \* from Transactions where 1=0)  --IS NULL treats both null as well as empty column as NULL!  --selects 10 rows  select top 10 \*  from Transactions as t  where (select null) is null  --selects 10 rows. This is unexpected  select top 10 \*  from Transactions as t  where (select t1.actid from Transactions as t1 where 1=0) is null  --this would throw error as other than for EXISTS clause, the subquery should return only one value  select top 10 \*  from Transactions as t  where (select t1.actid from Transactions as t1 where 1=1) is null  --empty result set also amounts to TRUE when 'IS NULL' predicate is used.  select case when (select e1.EMPNO as empno from emp as e1 where 0 = 1) is null then 1 else 0 end  --result set with a NULL value amounts to TRUE when 'IS NULL' predicate is used. That is understanable  select case when (select null as empno) is null then 1 else 0 end  -u.  --have to use CTE to use GROUP BY to solve this problem as columns are not available and you can’t use window function with CROSS APPLY to make them available as CROSS APPLY works per row of data from LHS table  select distinct job, COUNT(EMPNO) over(partition by job) as numemps, cast(100.0 \* sum(SAL) over(partition by job)/sum(SAL) over(order by (select null)) as int) as pctofsal  from EMP  --group by JOB, numemps, pctofsal  -v.  --for the leading key for sorting, either we can use 1)mgrid or 2)level or 3)mgrid.mgrid. The problem with using 1 or 2 is that employees at the same level are sorted before their underlings and that is not a tree structure. For underlings to be sorted right after their manager, we need to use the manager's manager. So we have to go one level up for that. And for that we need a self join. since the join is already being done in recursive CTE, we just use that and create the sorting key and adding that as a column of info for each row. Had the sorting key info been available from each row itself(for eg, lvl or mgrid along with rn), we could have just use 'ORDER BY lvl, rn' or 'ORDER BY mgrid, rn'. But since it is not we have to create a new concatenated column storing the sorting order like CONCAT(mgr.mgrid, rn). Note that 'ORDER BY lvl, rn' is equivalent to 'ORDER BY CONCAT(lvl, rn)'. Remember that CONCAT gives you a varchar output and varchars(or chars) are sorted using dictionary sort. Dictionary sort is as if we are sorting by individual elements for CONCAT in the same order. How does it compare with binary collation sort?  ;with base as  (  select \*, ROW\_NUMBER() over(partition by mgrid order by empname) as rn  from (values (1, NULL, 'David' , $10000.00),  (2, 1, 'Eitan' , $7000.00),  (3, 1, 'Ina' , $7500.00),  (4, 2, 'Seraph' , $5000.00),  (5, 2, 'Jiru' , $5500.00),  (6, 2, 'Steve' , $4500.00),  (7, 3, 'Aaron' , $5000.00),  (8, 5, 'Lilach' , $3500.00),  (9, 7, 'Rita' , $3000.00),  (10, 5, 'Sean' , $3000.00),  (11, 7, 'Gabriel', $3000.00),  (12, 9, 'Emilia' , $2000.00),  (13, 9, 'Michael', $2000.00),  (14, 9, 'Didi' , $1500.00)  ) as employees(empid, mgrid, empname, salary)  )  ,c as  ( select \*, 0 as lvl, cast(mgrid as varchar) as srtOrder  from base as b  where b.mgrid is null  union all  select b1.\*, c.lvl + 1 as lvl, cast(concat(srtOrder, b1.rn) as varchar) as srtOrder  from base as b1 inner join c  on b1.mgrid = c.empid  )  select c.empid, c.mgrid, concat(REPLICATE(' | ',lvl), empname) as ename  ,srtOrder  from c  order by srtOrder  -w.  ;with base as  (  select \*  from (values(1, 10),  (4, -15),  (5, 5),  (6, -10),  (8, -15),  (10, 20),  (17, 10),  (18, -10),  (20, -30),  (31, 20)) as b(ordcol, datacol)  )  ,rTotal as  (  select \*, sum(datacol) over (order by ordcol rows between unbounded preceding and current row) as runningTotal  from base as b  )  ,minRTotal as  (  select \*, min(runningTotal) over(order by ordcol rows between unbounded preceding and current row) as minRunningTotal  from rTotal  )  select \*, runningTotal - (case when minRunningTotal <0 then minRunningTotal else 0 end) as runningTotalReset  from minRTotal  -y.  ;with ordered as  (  select \*, ROW\_NUMBER() over(partition by username order by starttime, endtime) as rn  from Sessions  )  ,previousVals as  (  select \*, lag(endtime) over(partition by username order by starttime, endtime) as lg  from ordered  )  ,startOfRangeCol\_max as  (  select \*  , case when starttime <= max(lg) over(partition by username order by starttime, endtime rows between unbounded preceding and current row) then 0 else 1 end as startOfRangeMarker  from previousVals  )  ,sumOfstartOfRangeColToBeUsedAsGrouping as  (  select \*, sum(startOfRangeMarker) over (partition by username order by starttime, endtime) as SumOfRangeMarker  from startOfRangeCol\_max  )  select username, min(starttime), max(endtime)  from sumOfstartOfRangeColToBeUsedAsGrouping  group by username, SumOfRangeMarker  order by username | -l.  select \*, ROW\_NUMBER() over(order by (select null)) as rn  from EMP  --assume a unique column is present.  select \*, (select count(e1.EMPNO)+1 from emp as e1 where e.EMPNO > e1.EMPNO)  from EMP as e  --assumes unique column present to provide ordering.  select e.\*, COUNT(e1.EMPNO) + 1  from EMP as e  left outer join emp as e1 on e.EMPNO>e1.EMPNO  group by e.EMPNO, e.ENAME, e.JOB, e.MGR, e.HIREDATE, e.SAL, e.COMM, e.DEPTNO  --assume a unique column is present.  select \*  from EMP as e  cross apply (select count(e1.EMPNO)+1 from emp as e1 where e.EMPNO > e1.EMPNO) as b(rn)  --if no unique column is present. then does not work as shown below  select \*, (select count(e1.DEPTNO)+1 from emp as e1 where e.DEPTNO > e1.DEPTNO)  from EMP as e  --no unique column present. What if no unique column is present to give number of rows before a particular row  (like we specify SELECT NULL in ORDER by for ROW\_NUmber).  See accounts table in utilities section below. cartesian product. so not efficient.  **Wrong** as it assumes fname and lname is same for all rows so that grouping works.  ~~select a.fname, a.lname, rn~~  ~~from accounts as a with(nolock)~~  ~~cross apply (select count(\*) from accounts) as a1(cnt)~~  ~~cross apply (select \* from GetNums(1,cnt)) as a2(rn)~~  ~~group by a.fname, a.lname, rn~~  --could also use recursive CTE. 2 problems for large tables: very slow and 32767 recursive depth limit  **Wrong** as it assumes fname and lname is same for all rows so that grouping works.  ~~;WITH CTE AS~~  ~~(~~  ~~SELECT a.fname, a.lname, (SELECT COUNT(\*) FROM accounts) AS CNT , 1 AS RN~~  ~~FROM accounts AS a with(nolock)~~  ~~UNION ALL~~  ~~SELECT fname, lname, CNT, RN + 1~~  ~~FROM CTE~~  ~~WHERE RN < CNT~~  ~~)~~  ~~SELECT fname,lname,rn~~  ~~FROM CTE~~  ~~group by fname,lname,rn~~  ~~--OPTION (MAXRECURSION 1000000); can't use this limit as maximum allowed is 32767~~  ~~OPTION (MAXRECURSION 32767);~~  --another option is to add a new unique column to table, save the result as we need to reference it again either using a scalar subquery or joins. Note: if you tried to not use the temp table instead by using a CTE, you won’t get desired result as CTE would be inlined and the CTE uqery would be executed more than once hence generating new uids for each execution.  select \*  into #temp  from accounts as e  cross apply (select newid()) as a(id)  --for 10000 rows, the relative cost of scalar sub query was 0% as compared to the left outer join and outer apply plans given below. quickest  select \*, (select count(\*) +1 from #temp as t1 where t.id > t1.id) as rn  from #temp as t  order by rn  --75%. Costliest  select t.fname, t.lname, count(t1.id)+1 as rn  from #temp as t  left outer join #temp as t1 on t.id > t1.id  group by t.fname, t.lname, t.id  order by rn  --25%. 2nd best. It should have been the quickest?  select \*  from #temp as t  cross apply (select count(\*) +1 from #temp as t1 where t.id > t1.id) as b(rn)  order by rn  Running total  --running total with window function. Most efficient  select empid, ordermonth, qty, SUM(qty) over(partition by empid order by ordermonth rows between unbounded preceding and current row) as runqty  from dbo.EmpOrders  --running total with scalar sub query. 3rd worst perfromance  select empid, ordermonth, qty, (select sum(qty) from dbo.EmpOrders as e1 where e1.empid=e.empid and e1.ordermonth <= e.ordermonth) as runqty  from dbo.EmpOrders as e  --running total with cross apply. 3rd worst perfromance  select empid, ordermonth, qty, runqty  from dbo.EmpOrders as e  cross apply (select sum(qty) from dbo.EmpOrders as e1 where e1.empid=e.empid and e1.ordermonth <= e.ordermonth) as b(runqty)  --running total with scalar inner join. 2nd most efficient  select e.empid, e.ordermonth, e.qty, sum(e1.qty) as runqty  from dbo.EmpOrders as e  inner join dbo.EmpOrders as e1 on e1.empid=e.empid and e1.ordermonth <= e.ordermonth  group by e.empid, e.ordermonth, e.qty  order by e.empid, e.ordermonth  Median:could be discrete or continous  SELECT distinct actid,  PERCENTILE\_DISC(0.5) WITHIN GROUP (ORDER BY val) OVER (PARTITION BY actid) AS MEDIAN  FROM dbo.Transactions as t  order by actid  --index PK\_Transactions not beneficial.  --create index idx\_median on dbo.Transactions(actid,val); this POC index is required for performance otherwise same performance as median using window function  --create index idx\_my on dbo.Transactions(actid) include(val);this is also not beneficial as we are orderding by val.  ;with cte  as  (select distinct t.actid, cnt, skipRows, fetchRows  from Transactions as t  cross apply (select count(\*) from Transactions as t1 where t1.actid = t.actid) as b1(cnt)  cross apply (select (cnt-1)/2) as b2(skipRows)  cross apply (select 2 - (cnt%2)) as b3(fetchRows)  )  select c.actid, avg(1.0 \* b4.val)  from cte as c  cross apply( select t1.val  from Transactions as t1  where c.actid = t1.actid  order by val  offset skipRows rows fetch next fetchRows rows only) as b4  group by actid  order by actid  ;with c as  (  SELECT actid, COUNT(VAL) AS cnt, (COUNT(\*)-1)/2 AS skipRows, 2-count(\*)%2 as fetchRows  FROM Transactions AS T  GROUP BY actid  )  select c.actid, avg(1.0 \* a.val) as median  from c  cross apply(select \* from dbo.Transactions as t  where t.actid = c.actid  order by val  offset c.skipRows rows fetch next c.fetchRows rows only) as a  group by c.actid  order by actid  --lead window func. What happens in case of ties: is the next row with same hiredate returned or does it skip the same value rows?  select DEPTNO, ENAME, SAL, HIREDATE, lead(sal) over(partition by deptno order by hiredate) as salOfEmpHiredNextInDept  from EMP  --scalar subquery for lead window func. MIN() to ensure one value is returned in case of ties.  select DEPTNO, ENAME, SAL, HIREDATE, (select min(e2.SAL) from emp as e2 where e2.DEPTNO = e.DEPTNO and e2.HIREDATE = (select min(e1.HIREDATE) from emp as e1 where e.deptno = e1.DEPTNO and e1.HIREDATE > e.HIREDATE)) as salOfEmpHiredNextInDept  from EMP as e  order by DEPTNO  --with cross apply. U could also do it with left outer join  select DEPTNO, ENAME, SAL, HIREDATE  from EMP as e  cross apply (select min(e2.SAL) from emp as e2 where e2.DEPTNO = e.DEPTNO and e2.HIREDATE = (select min(e1.HIREDATE) from emp as e1 where e.deptno = e1.DEPTNO and e1.HIREDATE > e.HIREDATE)) as b(salOfEmpHiredNextInDept)  order by DEPTNO  --create 5 buckets with n rows  select \*, ntile(5) over(order by empno)  from emp as e  --creates buckets with 5 rows each...different from NTILE  select \*, (ROW\_NUMBER() over(order by empno)% 5) + 1 AS NTILE  from emp as e  ORDER BY empno  --same solution as above but demonstrates the cyclic property of modulus % and thus is more readable. Also used for Median.  select \*, 5 - ROW\_NUMBER() over(order by empno)%5 AS NTILE  from emp as e  ORDER BY empno  --NTILE equivalent  ;with cte as  (  select distinct totalRows, numOfRowsInTile, numOfTiles, n as tileNumber  from Transactions  cross apply(select 4) as b0(numOfTiles)  cross apply (select count(\*) from Transactions ) as b(totalRows)  cross apply(select ceiling(1.0 \* totalRows/4) from Transactions) as b1(numOfRowsInTile)  cross apply GetNums(0,numOfTiles-1) as g  )  select TiledDataSet.\*  from cte as c  cross apply (select \*, c.tileNumber as ntile  from Transactions as t  order by (select null)  offset c.tileNumber \* cast(c.numOfRowsInTile as int) rows fetch next cast(c.numOfRowsInTile as int) rows only) as TiledDataSet |

Utilities:

|  |  |
| --- | --- |
| -Generate some values  CREATE TABLE dbo.Accounts  (  actid INT NOT NULL CONSTRAINT PK\_Accounts PRIMARY KEY  );  CREATE TABLE dbo.Transactions  (  actid INT NOT NULL,  tranid INT NOT NULL,  val MONEY NOT NULL,  CONSTRAINT PK\_Transactions PRIMARY KEY(actid, tranid)  );  DECLARE  @num\_partitions AS INT = 100,  @rows\_per\_partition AS INT = 20000;  INSERT INTO dbo.Accounts WITH (TABLOCK) (actid)  SELECT NP.n  FROM TSQLV3.dbo.GetNums(1, @num\_partitions) AS NP;  INSERT INTO dbo.Transactions WITH (TABLOCK) (actid, tranid, val)  SELECT NP.n, RPP.n,  (ABS(CHECKSUM(NEWID())%2)\*2-1) \* (1 + ABS(CHECKSUM(NEWID())%5))  FROM TSQLV3.dbo.GetNums(1, @num\_partitions) AS NP  CROSS JOIN TSQLV3.dbo.GetNums(1, @rows\_per\_partition) AS RPP;  GO | -creating dummy data  --using GO to add a lot of rows repeatedly would be very slow as it is looping one row at a time  declare @accounts as table( fname VARCHAR(20), lname VARCHAR(20))  insert into @accounts  values('Fred', 'Flintstone');  go 10  Create Table dbo.T1 (Col1 int Identity, Col2 varchar(20) Default 'Capisce');  GO  Insert dbo.T1 Default Values  Go 5 -- Repeat the batch above five times. DO NOT use a semi-colon to terminate the statement.  --use the with(tablock) option for inserting large amounts of data. And remember truncate as well  create table accounts( fname VARCHAR(20), lname VARCHAR(20));  insert into accounts with(TABLOCK)(fname,lname)  select fname,lname  from (values('Fred', 'Flintstone')) as t(fname,lname)  cross apply GetNums(1,10) as a1; |
|  |  |

Keyword list: relation, relational theory(set theory + predicate logic), relation/set un-ordered unique/distinct list of elements/entities, deterministic/un-deterministic, rowset/columnset, presentation ordering, ties, table operators(per row and left to right order of execution/precedence which can be changed by using . E.g. apply), predicate, all at once, I-TVF(no begin-end blocks. Just a single return statement which gets inlined into the outer query) vs multi statement TVF(returns result via table var. no stats) vs scalar UDF(impedance to set based plan(not entirely true as the set based plan internally would be implement by operators which return rows one by one)).

Query,hash,proccache,lexer,parser>expression tree>query optimizer>physical execution plan(s)>lowest cost>stats for cardinality

Page(header,payload,slot array/row-offset array),buffer pool, logical read/write vs physical read/write, dirty pages, lazy writer, checkpoint, LRU, extent, mixed extent + SGAM, uniform extent + GAM