|  |  |
| --- | --- |
| **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.  -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 be used in place of Loop.  CROSS JOIN/CROSS APPLY not only is used to generate Getnums and then CROSS APPLY can be used with GetNums to mimic a loop.  -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 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)  -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)  -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. | -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().  -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 with ROW\_NUMBER or make use of CROSS APPLY with TOP(N) 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, scans ~30,000 seq reads  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, scans ~30030 seq reads  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.  -i. WINDOW function 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).  Running total: window function or scalar subquery or INNER JOIN(<= join) with GROUPing |

Examples:

|  |  |
| --- | --- |
| CROSS APPLY is deemed to be equal 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. But CROSS JOIN can use co-related subqueries which CROSS JOIN can’t and in that case it behaves as INNER JOIN as you can match rows.  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) | PIVOT to spread the values even when the categorical data to be used for spreading does not exist as such but can be inferred  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) |
| **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  SELECT TOP(500000) \*, CAST(CONCAT(A.A,B.B,C.C,D.D,E.E,F.F) AS INT) + 1 AS NUM  FROM (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS A(A)  CROSS JOIN (SELECT \* FROM (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS CP(A)) AS B(B)  CROSS JOIN (SELECT \* FROM (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS CP(A)) AS C(C)  CROSS JOIN (SELECT \* FROM (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS CP(A)) AS D(D)  CROSS JOIN (SELECT \* FROM (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS CP(A)) AS E(E)  CROSS JOIN (SELECT \* FROM (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS CP(A)) AS F(F)  ORDER BY NUM  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 | f.  --CONCATENATE COLUMN VALUES TO A LIST USING RECURSIVE CTE  --slow but handles special XML characters  ;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: 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', ',')) | 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)  i.  select top 2\*  from EMP  order by (select null)  select top 2\*  from EMP  order by (2)  select top 2\*  from EMP  order by (NEWID())  j.  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  from emp  group by DEPTNO, JOB |
| -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.  --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  --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 | --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. 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 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  For median, either use PERCENTILE\_DISC or try to use row\_number to allocate numbers to each group entry and then finding the middle row(it still uses a window function) or use CROSS APPLY to mimic row\_number function and then select middle row or what itzik says  SELECT distinct actid, COUNT(val) OVER(PARTITION BY actid) AS [TOTALCOUNT],  PERCENTILE\_DISC(0.5) WITHIN GROUP (ORDER BY val) OVER (PARTITION BY actid) AS MEDIAN  FROM dbo.Transactions as t |

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 |  |
|  |  |
|  |  |