-SHORTCUTS: ctrl + r, ctrl + u, ctrl + l, ctrl + m, ctrl + tab, alt + f1( after selecting a table name. it is shortcut for sp\_help), f4(show properties), ctrl \_ shift + l (lower case. U for upper case), shift + f10 (context menu), f6(iterate through split pane of a single doc), CTRL + SHIFT + R (refresh the cache), ctrl + 1 (sp\_who), ctrl + 2 (sp\_lock), CTRL + F1 (sp\_SQLskills\_helpindex), ---SELECT OBJECT\_NAME(object\_id),\* FROM SYS.dm\_db\_index\_usage\_stats;

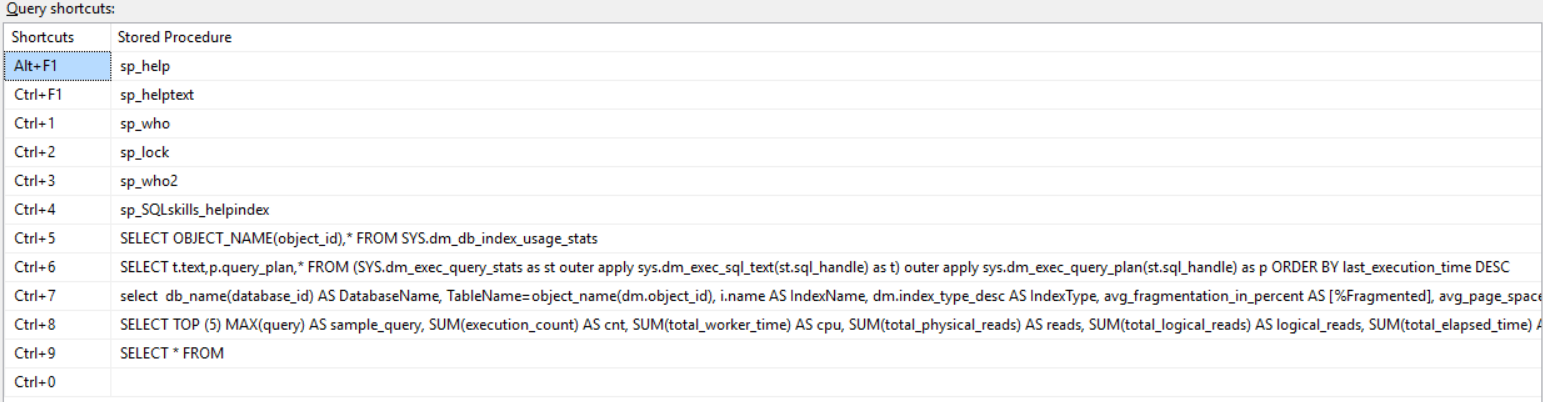
-SELECT t.text,p.query\_plan,\* FROM (SYS.dm\_exec\_query\_stats as st outer apply sys.dm\_exec\_sql\_text(st.sql\_handle) as t) outer apply sys.dm\_exec\_query\_plan(st.sql\_handle) as p ORDER BY last\_execution\_time DESC;

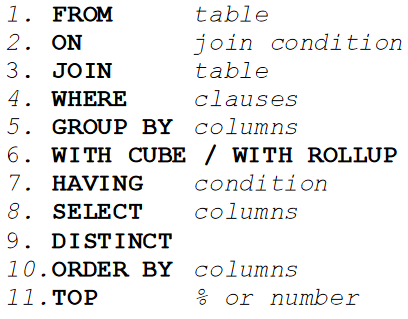
-select db\_name(database\_id) AS DatabaseName, TableName=object\_name(dm.object\_id), i.name AS IndexName, dm.index\_type\_desc AS IndexType, avg\_fragmentation\_in\_percent AS [%Fragmented], avg\_page\_space\_used\_in\_percent AS [%AvgPagePopulation] from sys.dm\_db\_index\_physical\_stats(db\_id(),null,null,null,'sampled') AS dm join sys.indexes AS i on dm.object\_id=i.object\_id and dm.index\_id=i.index\_id order by avg\_fragmentation\_in\_percent desc; --either provide the dbname in db\_id('NatSoil') or through currently selected database(ctrl+U) or through USE statement

-SELECT TOP (5) MAX(query) AS sample\_query, SUM(execution\_count) AS cnt, SUM(total\_worker\_time) AS cpu, SUM(total\_physical\_reads) AS reads, SUM(total\_logical\_reads) AS logical\_reads, SUM(total\_elapsed\_time) AS duration FROM (SELECT QS.\*, SUBSTRING(ST.text, (QS.statement\_start\_offset/2) + 1, ((CASE statement\_end\_offset WHEN -1 THEN DATALENGTH(ST.text) ELSE QS.statement\_end\_offset END - QS.statement\_start\_offset)/2) + 1 ) AS query FROM sys.dm\_exec\_query\_stats AS QS CROSS APPLY sys.dm\_exec\_sql\_text(QS.sql\_handle) AS ST CROSS APPLY sys.dm\_exec\_plan\_attributes(QS.plan\_handle) AS PA WHERE PA.attribute = 'dbid' AND PA.value = DB\_ID('NatSoil')) AS D GROUP BY query\_hash ORDER BY duration DESC; -- TOP queries with highest elapsed time

<https://www.sqlskills.com/blogs/glenn/sql-server-diagnostic-information-queries-for-september-2017/>

<http://bradsruminations.blogspot.com.au/2011/03/shrink-your-databases-regularly.html>





-Check constraint evaluates to ‘TRUE’ with NULL values (usually NULL causes expressions to evaluate to ‘NOT TRUE’). For UNIQUE constraint (and for GROUP BY, ORDER BY, UNION, EXCEPT and INTERSECT), NULL values are treated as equal. Remember UNIQUE constraint allows one NULL value (unlike PRIMARY KEY). If FOREIGN KEY is defined on a NULLable columns, then it can contain more than one NULLs. Aggregate functions like COUNT ignore NULL values.

-if T.col1 is Null’able, then queries with NOT IN need special handling or you should use NOT EXISTS:

SELECT Col1 FROM T2 WHERE Col1 NOT IN (SELECT Col1 FROM T1 WHERE T1.COL1 is NOT NULL)

SELECT Col1 FROM T2 WHERE NOT EXISTS(SELECT \* FROM T1 WHERE T1.COL1 = T2.COL1)

SELECT V FROM (VALUES(1)) AS B(V)

WHERE V NOT IN (SELECT V FROM (VALUES(3),(4),(NULL)) AS D(V));

--1 NOT IN(3,4,NULL) -> NOT (1==3 OR 1==4 OR 1==NULL) -> NOT (FALSE AND FALSE AND UNKNOWN)

--TRUE AND TRUE AND UNKNOWN -> NOT TRUE

SELECT V FROM (VALUES(1)) AS B(V)

WHERE V IN (SELECT V FROM (VALUES(1),(NULL)) AS D(V));

--1 IN(1,NULL) -> (1==1 OR 1==NULL) -> TRUE OR UNKNOWN -> TRUE

-INTERSECT operation can also be handled with INNER JOIN but then you need to handle NULLs

-If a row is inserted but no value is included for a column that allows for null values, the Database Engine supplies the value NULL, unless a DEFAULT definition or object exists. A column defined with the keyword NULL also accepts an explicit entry of NULL from the user, regardless of what data type it is or if it has a default associated with it.

-Left To Right order of eval is followed in FROM clause (not all-at-once). Even the predicates in WHERE clause are executed from left to right??(at least this is what is being suggested in eg. on page 157 “T-SQL Querying ”). All-at-once eg.(no need for temp variable to swap):- UPDATE T1 SET C1=C2, C2=C1;

-we know for matching we should ON clause and for filtering we should use WHERE clause. The subtlety here is that if by mistake if ON is used for filtering purpose and rows being filtered are from preserved side of OUTER JOIN, then the rows that you have filtered out from preserved site are anyways going to be added to the returned result after matching is done. So filtering the rows from preserved side (by using ON clause instead of WHERE clause) does not serve the purpose for OUTER JOIN. For INNER JOIN it does not matter.

-ORDER BY clause has access to both the input and output virtual tables of SELECT clause (think of it in this way that any expression that could have been used in a particular SELECT is available for use in ORDER BY). But if DISTINCT is used in SELECT, then only the output virtual table of SELECT is available for use in ORDER BY. But why this discrepancy? Because then which of the non-distinct rows would provide the value for the ORDER BY clause. Now you can try to an aggregate function on that column but remember that DISTINCT is not grouping rows. SO can’t. If you want this to be not the case, use GROUP BY instead of DISTINCT and then you can at least use aggregate funcs on columns not in SELECT list.

-ORDER BY is the last statement but if TOP or OFFSET-FETCH are used, then they are the last statements.

-ORDER BY goes against the SET theory as it orders the results (if ORDER BY is used for presentation purpose). DISTINCT makes the result confirm to SET theory as it removes the duplicates. ORDER BY can’t be used in any table expression (CTE, view, inline TVF or derived table) unless accompanied by TOP or OFFSET-FETCH (used with TO  
P/OFFSET-FETCH, ORDER BY’s purpose is not presentation ordering).

-select \* from (values(1),(1)) as tbl(a) union select \* from (values(2),(3)) as tbl2(a)

Returns a

-----------

1

2

3

So UNION, EXCEPT and INTERSECT return a distinct row set even if the duplication is in one table! ~~They are SET operators and it is as if they had an implicit DISTINCT clause in both queries as well as on the combined result set~~. The combined result set would only have DISTINCT values, no matter what the case.

- Pages belong exclusively to an index or table. But extents do not. Mixed extents are used for pages of objects with less than 8 pages. Once an objects has 8 pages, then all **future allocations** are made from uniform extents.

-size of page is 8KB. Now a data row with fixed with columns has to fit in a page as it can’t span multiple pages. So the maximum size of the row cannot exceed 8k. So you cannot have 10 CHAR(2000) columns in a table. But 10 CHAR(2000) columns in a table is allowed as the overflow will be stored off page.

- SQL Server enforces a primary key using a clustered index unless we specify NONCLUSTERED keyword explicitly or a clustered index already exists on the table. So choose your clustering (index) key before you define a primary key. Do not let the primary key define the clustering key for you. UNIQUE constraint is enforced using NONCLUSTERED index, unless you specify keyword CLUSTERED. With a CLUSTERED index a table is organized as a B-Tree, otherwise it is organized as a heap.

NONCLUSTERED index itself is also B-Tree, just not the table.

-B-Tree is a special case of balanced tree. A balanced tree is a tree where all the leaf nodes are at the same distance from the root node. A leaf node here is a page with the data rows in key order(data rows only in case of clustered index. RID or Key in case of nonclustered). Index order scan vs allocation order scan. RID is of the form (FileID:PageID:SlotID) and is an 8 byte int(2bytes:4bytes:2bytes).

-joins can processed using order based, hash based or loop based algorithms. Grouping/aggregating and distinct can be processed using an order based or hash based algo. Presentation ordering can only use order based algo. If right index exists, then no need for explicit SORT(expensive).

~~- Logical operation can be Table Scan or Index Scan. Physical operation can be Allocation Order Scan(heap scan) or Index Order Scan. For TS logical operation~~

-When the plan shows Table Scan operator, storage engine has to use the allocation order scan (heap scan). When the plan shows an index scan operator (clustered or unclustered) with property ORDERED ‘True’, the storage engine can use only Index Order Scan. When the plan shows an index scan operator with ORDERED ‘False’, there are 2 options to scan the data: allocation order scan or index order scan (I would think that these 2 options exist only for Clustered index as it has both an allocation order and index order. Does nonclustered index also support allocation order scan (heap scan) as it does not contain the table data?). Logical fragmentation vs physical fragmentation? I think physical frag. is inside a page whereas logical frag. means pages are out of order(due to splits)? Splits occur only in indexes, not in heap. Why?

- Rows in a heap seldom move, and when they do move, they leave a forwarding address at the old location. The rows of a clustered index, however, can move; that is, they can be relocated to another page during data modification or index reorganization (But I think in case of heap as well, the data is stored in pages. Just that pages in an index behave differently and data rows do not leave a forwarding address. Think about it as we could have used RID lookups for clustered indexes as well instead of Key lookup but RID does not permanently identify a row in case of clustered indexes).

-Each level of pages in an index pyramid structure(except the top page) is maintained as a linked list.

-Statistics (histogram) are used for estimating the cardinality (of an operator?) and cardinality in turn influences the selection if on execution plan(underestimation of cardinality causes index seek and loopkups to be used over a scan, serial over a parallel plan, for sort and hash opertions there might be spills to tempdb due to small memory grant, for aggregates, joins and distinct use of order based algo over hash-based ones. Remember for joins, nested loops is used when cardinality is small, else merge or hash based algo are used).

-

-parameter sniffing involves only parameters. Similar sniffing does not work on variables. So use of variables do not lead to a skewed execution plan but would select a non-optimal one (based on unknown value for variable).

-NOT IN, NOT EXISTS and EXCEPT can be used as alternate solutions in some problems. But NOT IN if working with NULLABLE columns would give wrong result. EXCEPT would return you the DISTINCT set and only NOT EXISTS would return the data as is.

-if you have a Null column and want to make the column unique, then remember that UINIQUE constraint treats 2 NULLs as equals. Thus you are only allowed 1 NULL value in the column. The way around it is UNIQUE filtered index. So what you do is you create a UNIQUE filtered index on the NULLable column with the predicate being COL is NOT NULL. CREATE UNIQUE NONCLUSTERED INDEX idx\_col1\_notnull ON dbo.T1(col1) WHERE col1 IS NOT NULL;

-iterative (row based) solution uses either cursor or TOP(1) queries. Even when you have to implement an iterative solution (say a cursor based), it is faster to use CLR based solution instead of T-SQL based.

-Although SQL is set based but internally the operations are still row based (the actual underlying implementation in some low level language). So it is rows which move between operators in an execution plan. It is not individual rows but an abstraction called CXPacket which is filled up with rows and then moved around. When data has to be moved from one thread to another (as in when a parallel portions starts or ends), it would be moved around in CXPackets.

-one should prefer to use RECOMPILE option at a statement level(OPTION(RECOMPILE)) inside the stored procedure instead of at stored procedure level(WITH RECOMPILE).

-some solutions using joins can be replaced with subqueries. Similarly some aggregate solutions can be replaced with TOP clauses.

-If you are using a CTE or a derived table in a query more than once, then perhaps u might want to store the result of the CTE or derived table in a temporary table or table variable instead to avoid repeating the work.

-recursive CTE performance is very bad. Worse than loop based solutions.

-Inline TVFs are great for logic reusability without performance penalties. But same cannot be said for UDFs!

- select \*

from (values(1)) as a(d) cross apply (select \* from (values(12)) as aa(s)) as b

select \*

from (values(1),(2)) as a(d) cross apply (select \* from (values(12),(13)) as aa(s)) as b

select \*

from (values(1)) as a(d) cross apply (select 1 as s where 1=0) as b

-Generating sample data:

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

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

-in addition to what is mentioned below, CROSS JOIN can also be used to increase the number of rows (replicate number of times the total number of rows in CROSS JOIN table)

-- can’t refer to alias for filtering purpose. Could use table expressions. But could also use CROSS APPLY as shown below. It is also a type of correlated CROSS APPLY where we are referring only to columns from right side. The gist of it is that you do not need right side table for a CROSS APPLY

SELECT custid, orderyear,

CASE orderyear

WHEN 2013 THEN [2013]

WHEN 2014 THEN [2014]

WHEN 2015 THEN [2015]

END AS val

FROM dbo.PvtOrders

CROSS JOIN (VALUES(2013),(2014),(2015)) AS Y(orderyear);

-- Removing NULLs

SELECT custid, orderyear, val

FROM dbo.PvtOrders

CROSS JOIN (VALUES(2013),(2014),(2015)) AS Y(orderyear)

CROSS APPLY (VALUES(CASE orderyear

WHEN 2013 THEN [2013]

WHEN 2014 THEN [2014]

WHEN 2015 THEN [2015]

END)) AS A(val)

WHERE val IS NOT NULL;

- CROSS JOIN or CROSS APPLY , both being table operators, can be used to define alias for computed columns which can then be used to in all logical phases after the ‘FROM’ phases(trick can be used in place of using table expressions). Think about them as way for adding columns to the underlying data source right at the beginning of query processing. Nice little trick! Remember that CROSS JOIN and CROSS APPLY are same but with 1 difference: CROSS APPLY can be correlated (it can refer to outer or left tables’ columns. But ~~in case~~ instead of correlated CROSS APPY, u can also use INNER JOIN with a table expression/subquery with an ON clause ~~instead~~?). Other than that both will return empty set if JOINed or APPLYied with an empty right side table expression. To get the Left table rows regardless, u need to use OUTER APPLY

SELECT orderid

FROM Sales.OrderValues

CROSS JOIN (SELECT 1 FROM (values(2),(3)) as a(n) WHERE 1=0) AS AGGS(n)

SELECT orderid

FROM Sales.OrderValues

CROSS APPLY (SELECT 1 FROM (values(2),(3)) as a(n) WHERE 1=0) AS AGGS(n)

SELECT orderid

FROM Sales.OrderValues

OUTER APPLY (SELECT 1 FROM (values(2),(3)) as a(n) WHERE 1=0) AS AGGS(n)

- The following 2 queries should have returned same empty result set but they don’t as the table expression is not returning empty result set which seems weird behaviour:

SELECT ordered,val/SUMVAL AS PCT, val-AVGVAL AS DIFF

FROM Sales.OrderValues

CROSS JOIN (SELECT SUM(val) AS SUMVAL, AVG(val) AS AVGVAL FROM Sales.OrderValues WHERE 1=0) AS AGGS

SELECT orderid,val/SUMVAL AS PCT,val-AVGVAL AS DIFF

FROM Sales.OrderValues

CROSS APPLY (SELECT SUM(val) AS SUMVAL, AVG(val) AS AVGVAL FROM Sales.OrderValues WHERE 1=0) AS AGGS

What I mean is that result of this query should be empty set but it is not. Whereas the query down below returns empty set. Something weird going with the aggregate functions?

SELECT SUM(val) AS SUMVAL, AVG(val) AS AVGVAL FROM Sales.OrderValues WHERE 1=0

SELECT 1, orderid, SUM(val) AS SUMVAL FROM Sales.OrderValues WHERE 1=0 GROUP BY orderid

-For the INNER JOIN, WHERE and ON clause serve the purpose of matching but not for OUTER JOIN clause where ON is for matching and WHERE for filtering.

- Classic way to identify only non-matches is to use a outer join and filter for nulls from the non-preserved side(on primary key column). Return all customers how placed no orders. I think u could also use NOT EXISTS for it. Maybe query can be written using other clauses as well.

-Nested loops only can be used for non-equi joins and CROSS JOINS. Merge and Hash based join processing requires Equi joins.

-If you are using left outer join and then again inner join(or right outer join) on fields from the non-preserved side of the outer join, you are losing the outer join values that did not have a match and hence your left outer join becomes an inner join effectively.

-so for the above scenario, it is better to have the inner joins preceding a RIGHT OUTER JOIN. Do not use make all the joins above as LEFT OUTER JOINS as they change the meaning of the query. So depending on the columns you are selecting using SELECT, the result might not be what you expect (I have to verify this). An even better solution to the above scenario is use parentheses to fix the scope. So you start with LEFT OUTER JOIN and then have the INNER JOINs in parentheses. The only doubt I have here is that after the end parentheses, when we use the ON clause, how did we refer to the alias defined for a table in the parentheses. You are also not allowed to give an outer alias to the query in the parentheses. So it is not quite a table expression/subquery. Even the parentheses are not required if you just move ON clause to being the last one (that is after you have written the query using parentheses, you can drop them as long as you keep the overall structure)

-if you do a JOIN but only return columns from one table, then it is called a Semi JOIN (use either INNER JOIN or EXISTS clause). Anti-Semi JOIN is when you return columns from one table if matching rows cannot be found in the related table (use either LEFT OUTER JOIN or RIGHT OUTER JOIN or NOT EXISTS). Note: The EXISTS clause returned DISTINCT by default. Why?? The reason is not that EXISTS is applying a implicit DISTINCT but that INNER JOIN is causing repetition of Customer data if a customer has place more than one order. Also note that there is no duplication of customer data for customer who did not place an order in the LEFT OUTER JOIN!

-- semi join

SELECT DISTINCT C.custid, C.companyname

FROM Sales.Customers AS C

INNER JOIN Sales.Orders AS O

ON O.custid = C.custid;

SELECT custid, companyname

FROM Sales.Customers AS C

WHERE EXISTS(SELECT \*

FROM Sales.Orders AS O

WHERE O.custid = C.custid);

-- Left anti semi join

SELECT C.custid, C.companyname

FROM Sales.Customers AS C

LEFT OUTER JOIN Sales.Orders AS O

ON O.custid = C.custid

WHERE O.orderid IS NULL;

SELECT custid, companyname

FROM Sales.Customers AS C

WHERE NOT EXISTS(SELECT \*

FROM Sales.Orders AS O

WHERE O.custid = C.custid);

-there are some performance deficiencies with using window aggregate functions without a frame, The issue being the query optimizer is not smart enough in these cases. Window aggregate functions with a frame, window ranking functions and window offset functions should be fine.

-window or partition is like a group. A frame is a further filter on the partition.

-GO 3 repeats the batch queries 3 times!

-Create an empty table with by just copying the columns from a source table. The use of ISNULL is just for removing the identity property:

SELECT ISNULL(orderid, 0) AS orderid, orderdate, empid, custid

INTO dbo.Orders

FROM TSQLV3.Sales.Orders WHERE 1 = 2;

-using SYSDATETIMEOFFSET() is the best way to capture date and time.

-Transactions have CHECKPOINT and SAVE TRAN

-You use the SET TRANSACTION ISOLATION LEVEL READ COMMITTED statement when you want to ensure that statements cannot read data altered by other transactions when those transactions have not yet been committed. When you set the READ COMMITTED transaction level, data can be altered by other transactions between individual statements within the current transaction.

You use the SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED statement when you want to allow statements to read rows altered by other transactions before those transactions have been committed.

You use the SET TRANSACTION ISOLATION LEVEL SNAPSHOT statement when you want to ensure that data read by any statement in a transaction will be transactionally consistent with how that data existed at the start of the transaction. SNAPSHOT transactions do not block other transactions from writing data.

You use the SET TRANSACTION ISOLATION LEVEL SERIALIZABLE statement when you want to ensure that Transact-SQL statements cannot read data that has been altered but not committed, that other transactions cannot modify data read by the current transaction until the transaction commits, and other transactions cannot insert rows with key values that would fall into the range read by statements in the current transaction until that transaction completes.

-when we play with isolation levels, we are playing with the ‘I’ from the ACID properties of transactions.

-using BEGIN TRAN and ROLLBACK TRAN with every SSMS window is a good practise for production databases.

-sqlskills blog

-Either avoid using names with TRANs OR just name the TRANs with BEGIN without naming the COMMITs or ROLLBACKs. Reason being transactions can be nested and if the inner transaction is doing a named ROLLBACK, it will throw an error as for nested transactions, any ROLLBACK (at any level) is a full rollback and thus an attempt is made to associate the name with the outermost TRAN name and that won't match and thus fails. You can now ask “why bother with naming them at all?”. By naming at least the BEGIN TRAN section, you get the benefit of the TRAN being named in the DMOs and thus can help in debugging deadlocks etc. (tip from Gail Shaw)

-parameter sniffing. if your stored procedure has been compiled and is using an execution plan that is the result of the it being called the first time using some parameter values, that execution plan might not be the best. For example, passing a p1 parameter value of 0 for the first call could result in a filtered indexed not being if it exists only for values 1 to 5. One option is use Option RECOMPILE for the stored proc which will result the execution plan not being cached. Or you could have a plan guide associated with the procedure asking it to use the execution plan that would have been created as if the parameter passed was in the range from 1 to 5. Another way is to use [dynamic SQL](https://youtu.be/gGMoHrIlHVY?t=872).

-do not use UDFs in persisted computed columns anywhere in sql server as sql server can’t generate parallel plans then.

- Any SQL operator that benefits from sequenced data can benefit from an index. This includes ORDER BY, GROUP BY, DISTINCT, UNION (not UNION ALL), and JOIN…ON. What about HAVING?

-Columns that are in a nonclustered index, but are not part of the index key, are called included columns. They are not sorted in sequence like key columns.

-when u include a column in a non-clustered index, you let know that you want to include the column in the SELECT clause and not the WHEN clause or any other clause that requires an index. By including a column in the non-clustered index, you prevent a lookup (**RID** lookup in case base table does not have clustered index or **Key** lookup in case base table does have clustered index) into the underlying table. Basically, you would not be searching on that column. Whereas a composite index is made up of two or more columns and all of them could be(or are all of them) used for searching or any other operation requiring an index. **Doubt**: if I have composite index on (firstname, lastname), then does it only serves queries which use both these columns for searching or can this index be used for searching also in queries using any one of the said columns. If only firstname was used for searching, then the index could be used (if the search is selective enough in case of non covered queries such that some lookups are required. In terms of covered queries, it would definitely be used). If lastname was used for search, then index would not be used and a table scan would be performed.

-covered index: when all the info needed is given by index itself without having to go through the lookups. Either we can use INCLUDE for covering an index(in cases where the extra cols are not used for index seek but only for display) or have a composite index. Clustered indexes are covering indexes by default (as the clustered index is the data table itself. So no lookup involved)

-8KB page \* 8 becomes extent.

-FOR XML RAW takes output rows as emits them as ‘row’ elements with attribute name value pairs corresponding to the actual output. You can provide a root element for RAW option as well as change the output to element centric: FOR XML RAW, ELEMENTS, ROOT('CustomersOrders');

-FOR XML AUTO knows the about the joins taking place in the query. So it gives output in a nested structure and renames the ‘row’ output to the name of the table it corresponds to. Now because it uses nesting, it prevents duplication. If you had Customers and Orders, it would have the Customers in the parent element and Orders as it’s child elements. You can provide a root element for AUTO option as well as change the output to element centric: FOR XML AUTO, ELEMENTS, ROOT('CustomersOrders');

-for both AUTO and RAW options, ORDER clause is very important. FOR XML and AUTO statement should follow the ORDER clause and the ORDER clause itself list the columns in correct nesting order. Failure to provide ORDER BY clause or providing wrong order in it will result in xml document with dispersed output.

-Schema of the document can also be returned embedded in the xml itself: FOR XML AUTO, ELEMENTS, ROOT('CustomersOrders'), XMLSCHEMA('TK461-CustomersOrders');

-FOR XML PATH can be used to manually fiddle with elements and attributes of the XML using XPATH expressions. Using FOR XML PATH option, the document becomes element centric by default (so no need for ELEMENTS option) and every column becomes an element. So you have to prefix ‘@’ to the column alias in the SELECT list to make it an attribute. Inline schema can’t be generated for it as shown above.

-- FOR XML PATH

SELECT Customer.custid AS [@custid], Customer.companyname AS [companyname]

FROM Sales.Customers AS Customer WHERE Customer.custid <= 2

ORDER BY Customer.custid

FOR XML PATH ('Customer'), ROOT('Customers');

GO

-- 1.

SELECT Customer.custid AS [@custid], Customer.companyname AS [@companyname],

(SELECT [Order].orderid AS [@orderid], [Order].orderdate AS [@orderdate]

FROM Sales.Orders AS [Order]

WHERE Customer.custid = [Order].custid

AND [Order].orderid %2 = 0

ORDER BY [Order].orderid

FOR XML PATH('Order'), TYPE)

FROM Sales.Customers AS Customer

WHERE Customer.custid <= 2

ORDER BY Customer.custid

FOR XML PATH('Customer');

GO

-shred xml to tables using OPENXML rowset function.

-xpath(expressions) vs xquery(query language. More functionality). Xquery is available on XML data type variables



**Smaller table should be on top in nested joins (efficient to parallelize??)**

For case-sensitive dictionary sort, characters abcxyzABCXYZ sort as AaBbCcXxYyZz where binary sort will give u ABCXYZabcxyz