-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)

-Choose your clustering (index) key before you define a primary key. Do not let the primary key define the clustering key for you. If clustering key is different from primary key, primary key can then be non-clustered.

-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)

-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** in case base table does not have clustered index or **Key** 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??)**