**SQL Library 2.0**

**Using This Document**

* Words in ALL CAPS represent SQL keywords
* Words in *italics* represent user inputs (like names of tables, columns, etc.)
  + *TableName* -> the user replaces with the appropriate table name
  + *ColumnName* -> the user replaces with the appropriate column name
* Comments
  + Some lines have additional comments indicated using two dashes --, like SQL in-line comments
* Use of brackets []
  + Brackets are often used around database, table, and column names
    - This document does NOT use [] around those items
    - SQL Server requires the use of [] around database items (tables, columns) that have a space in the name
  + In this document, [] represent optional items that you don’t have to use
    - Example:
      * SELECT [DISTINCT]
        + In this statement, DISTINCT is not required and is optional
* Workflow
  + When navigating/clicking things, a workflow is represented using arrows
  + A program application name will be in bold
  + Followed by The location to find the item
  + Steps to click will be indicated with arrows
  + When something is on its own line, everything is done in that tab or window
  + Example
    - **Word**
      * File (toolbar) -> Options
        + General (tab)

Set username

Set initials

* + - * + Proofing (tab)

Check -> Hide spelling errors in this document only

Check -> Hide grammar errors in this document only

**Comments**

* In-line comments are made using two dashes --
  + So that means everything from -- now on is a comment
* Block comments
  + Everything in between /\* and \*/ is a comment
  + /\* everything in between the opening and closing symbols is a comment.

this comment can even span multiple lines \*/

**A note on case**

* SQL is case insensitive, which means EMPLID = emplid = EmplID
* Out of convention, all SQL commands are in ALL CAPS in this document

**A note on whitespace in your code**

* SQL ignores whitespace
  + Line breaks, spaces, and indents all have no affect on the code
  + Therefore organize your code so it is readable
* This means that each of the following examples is interpreted exactly the same:
  + Example 1:
    - INSERT INTO Constituents

(ImportID, FirstName, MiddleName, LastName, Employer, SpouseName)

SELECT

EMPLID, first\_name, mid\_name, last\_name, organization, spouse

FROM ConstituentSource

* + Example 2:
    - INSERT INTO Constituents

(

ImportID

, FirstName

, MiddleName

, LastName

, Employer

, SpouseName

)

SELECT

EMPLID

, first\_name

, mid\_name

, last\_name

, organization

, spouse

FROM ConstituentSource

* + Example 3:
    - INSERT INTO Constituents (ImportID, FirstName, MiddleName, LastName, Employer, SpouseName) SELECT EMPLID, first\_name, mid\_name, last\_name, organization, spouse FROM ConstituentSource

**Data Manipulation Language (DML) for Retrieving Information**

* **SELECT**
  + Returns results (a query)
    - The results are essentially a list if only one column is selected
    - The restuls are a table if multiple columns are selected
    - Can include calculations
  + Basic Syntax -- see additional ways to specify columns below
    - SELECT *Column1, Column2*

FROM *TableName*

* + **DISTINCT**
    - Returns only ‘unique’ records
    - This ‘unique-ness’ only applies to the combination of columns listed in the SELECT statements
  + **TOP**
    - SQL Server command
    - Specifies the number of records to return
      * Can do a number -- TOP 10
      * Can do a percentage -- TOP 10 PERCENT
    - Can include WITH TIES to ensure you don’t exclude values that ‘tie’
    - Column list or star (for all columns) goes after the ‘with ties’ if present
    - For MySQL or PostgreSQL, can’t use TOP
      * Instead, use LIMIT *num* at the end of the query after ORDER BY
    - Example using WITH TIES
      * SELECT TOP 5 [WITH TIES] \*

FROM *TableName*

* + - Example using DISTINCT
      * SELECT DISTINCT TOP 5

FROM *TableName*

* + Aliases
    - You can alias any column included in your select statement using the keyword AS
      * This will use the alias name as the column header for the table that is returned from the query
    - It is best practice to provide an alias when using aggregate functions (like sum, avg, etc.)
      * Example:
        + SELECT AVG(Gift\_Amt) AS avg\_gift\_amt
  + Five ways to specify columns
    - SELECT \* -- selects all columns
    - SELECT *ColumnName* -- can select multiple column names, separate each with a comma
    - SELECT *NumericColumn1 + NumericColumn2* AS *CalulatedValue*
      * can do cacluations on numeric data -- best practice to alias the name
    - SELECT *TextColumn1 + ‘ ‘ + TextColumn2* AS *CombinedText*
      * can concatenate strings, again best practice to alias
    - SELECT *FUNCTIONNAME()* AS *AliasName*
      * There are some built-in functions like GETDATE() you can include in a SELECT statement
        + GETDATE() will get the current date (today’s date)
* **FROM**
  + Specifies the data to use for information retrieval
  + Can be a single table
  + Can include **JOIN**s to multiple tables (see section on JOINs below)
  + Can be the results of a SELECT statement (see section on subqueries)
* **WHERE**
  + Condition or conditions used to filter the results of a query
  + Should be a condition that will evaluate ‘True’ or ‘False’ for each record
  + Comparison operators
    - = equals (is the same as)
    - > greater than (after)
    - >= greater than or equal to (at or after)
    - < less than (before)
    - <= less than or equal to (before or at)
    - <> not equal to (not the same)
  + Logical operators -- helpful to use () to prevent ambiguous statements with AND/OR
    - AND all conditions must pass
    - OR one condition must pass
    - NOT opposite
  + IN
    - tests a list of values
    - can supply a list manually
      * WHERE *FieldName* IN (‘value1’, ‘value2’)
    - can use a subquery (see subqueries section for more info)
      * WHERE *FieldName* IN (SELECT *FieldName2* FROM *TableName* WHERE *condition*)
    - can use NOT IN to exclude certain results
  + LIKE
    - used to look for a pattern
    - usually includes wildcard symbols
    - in NOT case sensitive
    - can use NOT LIKE
    - Wildcard Symbols
      * % any string of zero or more characters
      * \_ any single character
      * [ ] a single character specified in the brackets
      * [ - ] a single character within the given range
      * [ ^ ] matches a single character, except what’s listed after the ^
    - Examples:
      * WHERE *SourceField* LIKE ‘A%’
        + matches anything that starts with A
      * WHERE *SourceField* LIKE ‘San%’
        + matches words like ‘San Diego’, ‘San Antonio’, ‘Santana’
      * WHERE *SourceField* LIKE ‘Compu\_er%’
        + matches ‘computer’, ‘compuserve’, ‘computer warehouse’, etc.
      * WHERE *SourceField* LIKE ‘Dev[io]n’
        + matches ‘Devin’ or ‘Devon’
      * WHERE *SourceField* LIKE ‘[A-K]%’
        + matches entries that start with any letter A-K
      * WHERE *SourceField* LIKE ‘N^[K-Y]’
        + matches ‘NC’ and ‘NJ’ but not ‘NV’ or ‘NY’
      * WHERE *SourceField* NOT LIKE ‘[1-9]%’
        + matches anything that doesn’t start with numbers 1-9
  + BETWEEN
    - checks for a value being between two values
      * range is inclusive (like >= instead of >)
    - can use NOT BETWEEN
    - simple case
      * WHERE *FieldName* BETWEEN *value1* AND *value2*
    - using expressions
      * WHERE *FieldName1 - FieldName2* BETWEEN *value1* AND *value2*
      * WHERE *DateField* BETWEEN GetDate() AND GetDate() + 30
        + GetDate() is a function that gets todays date
        + This WHERE statement checks if a *DateField* is between today and 30 days from now
  + NULL
    - Checks for empty/missing values
    - WHERE *SourceField* IS NULL
      * returns records where *SourceField* has null/missing values
    - WHERE *SourceField* IS NOT NULL
      * returns records that do not have null/missing values in *SourceField*
    - IsNull(*SourceField, expression*) function
      * replaces null values in your sourcefield with the specified expression
      * Example
        + IsNull(*SourceField*, ‘’)

replaces null values with blank ‘’ values

useful when there are already blank values in the data that aren’t NULL values

* + Example:
    - SELECT ImportID, Gift\_Amt

FROM Gifts

WHERE Gift\_Amt > 100

* + - * Returns every single ImportID/Gift\_Amt combination that has a Gift\_Amt larger than 100
* **GROUP BY**
  + Needed when using an **aggregate function** (see section below for more info) and SELECT-ing additional columns
    - You must ‘GROUP BY’ every additional column
  + Example when GROUP BY is not needed
    - SELECT AVG(Gift\_Amt)

FROM Gifts

* + - * Returns a single number that is the average of all Gift\_Amt’s
  + Example when GROUP BY is needed
    - SELECT ImportID, AVG(Gift\_Amt)

FROM Gifts

GROUP BY ImportID

* + - * Returns each ImportID along with its AVG(Gift\_Amt)
  + Example with two extra columns using GROUP BY
    - SELECT ImportID, Year, AVG(Gift\_Amt)

FROM Gifts

GROUP BY ImportID, Year

* + - * Returns each ImportID and Year combination along with its AVG(Gift\_Amt)
* **HAVING**
  + Works a lot like WHERE, but used whenever you have to ‘GROUP BY’
  + Example:
    - SELECT ImportID, AVG(Gift\_Amt)

FROM Gifts

GROUP BY ImportID

HAVING AVG(Gift\_Amt) > 100

* + - * Returns every ImportID with its AVG(Gift\_Amt) if the AVG(Gift\_Amt) is larger than 100
* **ORDER BY**
  + Sorts the data, can be ascending (ASC) or descending (DESC)
  + Default ORDER BY is ASC
  + Example:
    - ORDER BY *ColumnName* DESC
    - ORDER BY *ColumnName*
      * default sort is ASC, so you don’t need to code that
  + Sorting numeric data
    - may need to ORDER BY CAST(*FieldName* AS INT) if numeric data is stored as text
  + Can sort by any column, even if it’s not in the SELECT statement!
    - that is unless you use DISTINCT, then you can only order by columns in the SELECT statement
  + Advanced sorting
    - you can ORDER BY an alias
    - you can ORDER BY an expression
      * ORDER BY *Column1 + Column2*
    - you can ORDER BY column position
      * ORDER BY 1, 2
      * not easy to tell what you’re sorting by, so probably not so useful most of the time
  + OFFSET, FETCH, and NEXT
    - Retrieving a subset of records
    - Usually nested in the ORDER BY statement
    - If you only want the first few records, you can use the TOP statement mentioned above
    - Examples
      * ORDER BY *ColumnName*

OFFSET 10 ROWS

FETCH NEXT 10 ROWS ONLY;

Only retrieves rows 11-20

* Sequence of statements
  + The statements above must be coded in the proper order for a query to work
  + The order is: -- remember that [] denote optional entries
    - SELECT [DISTINCT] [TOP # [WITH TIES]]

FROM

[WHERE]

[GROUP BY]

[HAVING]

[ORDER BY]

**Case Statements**

* Two different syntaxes
* Syntax 1:
  + CASE WHEN *condition* THEN *result\_action* END AS *FieldNameAlias*
  + Can combine several WHEN/THEN statements
    - CASE
      * WHEN … THEN…
      * WHEN … THEN…
      * WHEN … THEN…
    - END AS *FieldNameAlias*
* Syntax 2: -- when you’re checking for different values within the same field
  + CASE *FieldName*
    - WHEN *value1* THEN *result\_action1*
    - WHEN *value2* THEN *result\_action2*
  + END AS *FieldNameAlias*

**IIF and CHOOSE Functions (SQL Server 2012 and up)**

* IIF
  + Checks a conditional expression, then sets a value if true and a different value if false
  + Syntax:
    - IIF(*conditional\_expression, true\_value, false\_value*)
  + this syntax evaluates your *condition*, the returns *true\_value* when the condition is met and *false­\_value* when the condition is not met
* CHOOSE
  + replaces the value of the index with the supplied information
  + Syntax:
    - CHOOSE(*FieldName, value1, value2, value3, value4…*)
  + This syntax goes through each row checking the value in *FieldName*
    - if the value in *FieldName* is 1 for a particular row, then *value1* is returned
    - if the value in *FleldName* is 4 for a particular row, then *value4* is returned and so on
  + values contained in *FieldName* column must all be integers and range from 1 to the number of values in the supplied list

**Working with NULL values**

* COALESCE(*expression1[, expression2…]*)
  + returns the first NON-NULL expression among its arguments
  + can use it exactly like IsNull() or can provide multiple expressions, in which the first to not return NULL is returned
  + Example:
    - COALESCE(DATE1, DATE2, DATE3, ‘1/1/1900’)
  + This example with try to return DATE1, if it’s null it goes to DATE2, if that’s null DATE3, if that’s null, it will finally return ‘1/1/1900’
* IsNull(*expression, replacement*) -- SQL Server only
  + Returns the *expression* unless it’s null, then it returns its *replacement*

**Aggregate Functions**

* AVG(*ColumnName*)
  + returns the average value
* SUM(*ColumnName*)
  + returns the sum
* MIN(*ColumnName*)
  + returns the minimum value (or the first value)
* MAX(*ColumnName*)
  + returns the maximum value (or the last value)
* COUNT(*ColumnName*)
  + counts the number of non-null values
* COUNT(\*)
  + counts the total number of rows
* ALIAS these calculations
  + Otherwise your column name will either be ‘Column1’ or similar
  + Column name could also be the full calculation
  + Either way, it’s bad, so alias all of your calculations
* Use of DISTINCT
  + you can add DISTINCT in front of all of these to only evaluate on distinct values for that field
  + Example:
    - COUNT(DISTINCT [ConsID])
* **GROUP BY**
  + You have to GROUP BY any columns that you include that aren’t aggregate functions
  + If you don’t include any other columns and do not use GROUP BY, then you get a single value for each aggregate function in the SELECT statement
* **HAVING**
  + HAVING works like WHERE for aggregate functions
  + HAVING *full\_function* *condition*
    - must code the *full\_function*, can’t use aliases here
    - *condition* uses comparison operators like =, >, etc.
* **SQL Server Extensions for Summarizing Data**
  + **ROLLUP**
    - does not work if you use DISTINCT within an aggregate function
    - adds one or more summary rows to a result set that uses GROUP BY/aggregates
      * adds a final summary row at the end
      * adds other summary rows at the end of each GROUP from GROUP BY
        + if you ORDER BY, you should choose DESC, because NULL comes before any value, so ordering ASC would put your summary row at the top of each group
    - columns that can’t be summarized are assigned NULL values
    - Syntax:
      * GROUP BY *ColumnName* WITH ROLLUP
    - alt. syntax (SQL Server 2008 and later)
      * GROUP BY ROLLUP(*ColumnName*)
  + **CUBE**
    - similar to ROLLUP in that it adds summary row(s)
      * but adds a row for each GROUPING
    - Syntax:
      * GROUP BY *ColumnName* WITH CUBE
    - can include multiple column names to group by / summarize
  + **GROUPING SETS**
    - similar to CUBE, but it ***only*** returns the summary rows
    - returns summary rows for each specified group
    - Syntax:
      * GROUP BY GROUPING SETS (*ColumnName*)
    - can use multiple column names
  + **Ranking Functions**
    - ROW\_NUMBER()
      * Adds a ‘row number’ to each row for the result set
      * Can provide an ORDER BY clause to ‘sort’ the data before assigning numbers
        + Change add ‘DESC’ to the ORDER BY clause to invert the sort
      * Can provide a PARTITION BY clause to only number within subgroups
      * Example to number the entire result set:
        + SELECT

ROW\_NUMBER() OVER(ORDER BY *column\_name*) as *Row\_Num*, *column\_name*

FROM *TableName*;

this example will number each row with 1 assigned to the first record as sorted by the ORDER BY statement

* + - * Example to number rows and restart between different sets
        + SELECT

ROW\_NUMBER() OVER(PARTITION BY *[EMPLID]* ORDER BY *[SDTNT\_DEGR]*)

as *Row\_Num*, [EMPLID], [STDNT\_DEGR]

FROM *TableName;*

this example will number each row using [EMPLID] as a group, so it restarts numbering with each new EMPLID

* + - RANK()
      * Works similarly to ROW\_NUMBER() but will rank based on the supplied ORDER BY *column\_name*
      * ties will be assigned the same rank (ex. 1, 1, 1, 4, 4, 6…)
      * works with PARTITION BY
    - DENSE\_RANK()
      * Works like rank, except no values are skipped (ex. 1, 1, 1, 2, 2, 3…)
    - NTILE(*integer\_expression*)
      * Syntax
        + SELECT *column\_name,*

NTILE(*INT*) OVER (ORDER BY *column\_name*) as [Tile*INT*]

FROM *TableName;*

NTILE works by breaking up the number of rows returned in to *INT* number of groups. The number assigned in the [Tile*INT*] column is the number of the group that row falls into

it’s weird, I know

* + - * If the rows don’t divide evenly, then later rows may have 1 less row than earlier groups
        + the first group will get the extra number first, then the next group, and so on
  + **Analytic Functions (SQL Server 2012 and up)**
    - Work on ordered data sets
      * where the first and last values have meaning (like max/min values)
      * Some of these are weird, so look up if you need them
    - FIRST\_VALUE
      * returns the first value from an ordered set
      * returns any single value, so can concat the ‘first value’ of multiple fields into one
      * Syntax:
        + FIRST\_VALUE (*expression*) OVER (PARTITION BY *grouping\_column* ORDER BY *sort\_column*) as [*col\_alias*]
        + *expression* can be a single column or an expression that returns one value
    - LAST\_VALUE
      * works exactly like FIRST\_VALUE except that the last value is returned
      * same syntax as FIRST\_VALUE except you need to specify the RANGE clause
        + RANGE clause goes at the end of the PARTITION statement in ()
        + RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

this will return only the last value, rather than the last value for each group, and use this instead of PARTITION BY

* + - LEAD
      * let you refer to values in other rows of the result set
    - LAG
      * let you refer to values in other rows of the result set
    - PERCENT\_RANK
    - CUME\_DIST
    - PERCENTILE\_CONT
    - PERCENTILE\_DISC

**Return a Row That Includes Summarized Data (from aggregate functions)**

* OVER / PARTITION BY allows you to return a row that also includes summary data (aggregate functions)
* Syntax:
  + SELECT *Column1…, aggr\_function* OVER (*[PARTITION BY ColumnName] [ORDER BY ColumnName]*) as ALIAS
* Example:

SELECT DISTINCT [EMPLID]

,[GIFT\_NO]

,SUM(CAST([GIFT\_AMT] as DECIMAL)) OVER (PARTITION BY[GIFT\_NO]) as [Gift\_Total]

FROM [TRAINING\_RIC\_Src].[Src].[GIFT\_DETAIL\_FIXED]

ORDER BY [EMPLID], [GIFT\_NO]

* This example would return each [EMPLID]/[Gift\_No] with the sum total for that gift number
  + useful because there are some Gift\_No that are there multiple times
  + would give the sum total for each gift
* OVER/PARTITION BY is also used with ROW\_NUMBER(), RANK(), DENSE\_RANK(), and NTILE() functions above

**JOIN**

* JOINs allow use of more than one table in FROM statements
* JOINs can also allow you to create more interesting results by doing a self join
* Four main types of JOINs
  + INNER JOIN
  + LEFT JOIN -- a type of outer join
  + RIGHT JOIN -- a type of outer join
  + FULL JOIN -- a type of outer join
* **INNER JOIN**
  + only returns results that meet the ON condition
  + ‘INNER’ keyword is not necessary, but can provide clarity
    - INNER JOIN *SecondTable* is the same as JOIN *SecondTable*
  + Basic Syntax
    - FROM *FirstTable* *f*

INNER JOIN *SecondTable s*

ON *f.Column1 = s.Column2*

* + Use of aliases
    - It is good practice to alias table names, so you don’t have to type them out, but you can also use the entire table name
    - You can use the AS keyword between the name and alias, but it’s not required
    - Example:
      * FROM *FirstTable*

INNER JOIN *SecondTable*

ON *FirstTable.Column1 = SecondTable.Column2*

* + - If you use an alias, you must use it on all columns that appear in both tables
  + Often the ON statement is linking the PRIMARY KEY from one table to a FOREIGN KEY in another table
    - but this is not always the case, and you don’t always have to use ‘=’ to create a join!
  + Another Syntax for INNER JOINS
    - probably not going to use this, but you may see it in older code or when maintaining code
    - FROM *FirstTable f, SecondTable s*

WHERE *f.column1 = s.column2*

* **LEFT JOIN**
  + An outer join
  + Includes all of the records from the first table
    - adds data you specify from the second table
    - any records that do not relate to the joined table will have NULL values for joined fields
  + Syntax:
    - FROM *FirstTable f*

LEFT JOIN *SecondTable s*

ON *f.ColumnName = s.ColumnName*

* **RIGHT JOIN**
  + An outer join
  + Less common than LEFT JOIN
  + Includes all of the records from the second table
    - adds data you specify from the first table
    - any records that do not relate to the joined table will have NULL values for joined fields
  + Syntax:
    - FROM *FirstTable f*

RIGHT JOIN *SecondTable s*

ON *f.ColumnName = s.ColumnName*

* **FULL JOIN**
  + Returns every record from both tables
  + NULL values will appear for any fields from unmatched records
  + Syntax:
    - FROM *FirstTable f*

FULL JOIN *SecondTable s*

ON *f.ColumnName = s.ColumnName*

* Complex JOIN conditions
  + You can specify more than one JOIN condition (Compound JOIN)
    - FROM *FirstTable f*

INNER JOIN *SecondTable s*

ON *f.column1 = s.column2* AND *f.column3 = s.column4*

* + You can also use other comparisons besides the = sign
    - Simple JOIN example
      * FROM *FirstTable f*

INNER JOIN *SecondTable s*

ON *f.column1 <= s.column2*

* + - Compound JOIN example
      * FROM *FirstTable f*

INNER JOIN *SecondTable s*

ON *f.column1 = s.column2* AND *f.column3 > s.column4*

* + Some of these results could also be accomplished using WHERE statements
    - There is more than one way to return a result set in many cases
* SELF JOINS
  + Sometimes a self join can help produce desired results
  + You join a table with itself, and you must use aliases to distinguish between column names
  + Example:
    - SELECT DISTINCT o1.[Name], o1.[City], o1.[State]

FROM [Organizations] o1

INNER JOIN [Organizations] o2

ON (o1.[City] = o2.[City])

AND (o1.[State] = o2.[State])

AND (o1.[Name] <> o2.[Name])

ORDER BY o1.[State], o1.[City]

* + - * This example will return all orgs that are in the same city and state as another org
      * An org that is the only one in that city, state will not be returned (INNER JOIN)
      * No org will be listed twice (DISTINCT)
      * An org can’t match itself (o1.[Name] <> o2.[Name])
        + without this statement, every org would match itself in the self joined table and be returned
* Joining Multiple Tables
  + Looking up values that are stored in multiple tables, joining each to the first table
    - Joining a -> b, a -> c, a -> d
    - FROM *FirstTable f*

JOIN *SecondTable s*

ON *f.ColumnName = s.ColumnName*

JOIN *ThirdTable t*

ON *f.ColumnName2 = t.ColumnName*

* + Joining through multiple levels
    - Joining a -> b, b -> c, c -> d
      * these joins must be done in order
    - FROM *FirstTable f*

JOIN *SecondTable s*

ON *f.ColumnName = s.ColumnName*

JOIN *ThirdTable t*

ON *s.ColumnName2 = t.ColumnName*

* Other JOINs
  + CROSS JOIN
    - Will distribute each row from the first table with each row from the second table
    - No matching or ON condition
    - If table1 has 5 rows and table2 has 9 rows, the query result would have 45 rows (all combinations)
    - Syntax:
      * FROM *FirstTable f* CROSS JOIN *SecondTable s*
* **UNION**
  + works sort of like pasting together two queries
  + records returned by both queries will only be returned once
    - use **UNION ALL** to negate this function, resulting in duplicate records that appear in both SELECT statements
  + Both SELECT statements must return the same number of columns
    - The columns must be of the same data type
      * order determines this -- the first columns must match, the second cols must match…
  + Your SELECT statements can use the same table or different tables, as long as the number of columns and datatypes match
    - Your column name displayed for the output will be those from the first SELECT statement
    - It’s common to use aliases when necessary for this reason, and they must be in the first SELECT
  + Syntax:
    - SELECT…

FROM…

WHERE…

**UNION**

SELECT…

FROM…

WHERE…

ORDER BY…

* **EXCEPT**
  + uses a second query to exclude records
  + records that are returned by the second SELECT statement will NOT appear in the result set
  + Syntax:
    - SELECT…

FROM…

WHERE…

**EXCEPT**

SELECT…

FROM…

WHERE…

ORDER BY…

* **INTERSECT**
  + returns only the records that are selected by both queries
    - works much like an INNER JOIN
  + same restrictions as UNION/EXCEPT with number of columns and datatypes
  + Syntax:
    - SELECT…

FROM…

WHERE…

**INTERSECT**

SELECT…

FROM…

WHERE…

ORDER BY…

**Subqueries**

* A subquery is a SELECT statement embedded within another SQL statement
* Subqueries can return…
  + a single value (like an aggregate function)
  + a list of values (like returning a single column)
  + a result set (like a table)
* There are 4 places you can do a subquery
  + WHERE -- as a search condition
  + HAVING -- as a search condition
  + FROM -- produces a table that is now the source of your main SELECT query
  + SELECT -- specifies what will appear in a column
* General syntax for a subquery
  + the same as a regular SELECT statement except…
    - typically doesn’t include GROUP BY or HAVING
    - can’t include ORDER BY unless you use the TOP *num* clause
* **WHERE** subqueries
  + most common type of subquery
  + let’s you check a condition involving the results of a query
    - could include an aggregate function -- ex. everything greater than the average
    - could provide a list to check against -- ex. WHERE *Field* IN (SELECT….)
  + Notes
    - If you use any arithmetic comparisons (>, >=, <, <=, =, or <>) then your subquery must return a single value or you must use either ALL or ANY keyword
      * WHERE *SourceField >* ALL (SELECT…)
        + Your source field must be greater than every value returned to be ‘True’
      * WHERE *SourceField =* ANY (SELECT…)
        + Your source field must = any of the returned values to return ‘True’
        + You may see SOME used in place of ANY in some situations, they are the same
    - If you use IN or NOT IN, then your subquery can return a list of values (like a column)
      * must only SELECT one column
      * WHERE *SourceField* IN (SELECT *ColumnName* FROM… WHERE…)
  + Example:
    - WHERE [ID] NOT IN (SELECT [ID] FROM *DifferentTableName* WHERE *[ColumnName] = value*)
    - this example will exclude all records for which the ID in question has a column in another table that has a specific value
  + Correlated WHERE subquery
    - runs for each row -- and would eat a lot of memory and take a long time to run
    - Example:
      * SELECT *IDFieldName, NumericFieldName*

FROM *SourceTable* AS *s2*

WHERE *NumericFieldName >* (SELECT AVG(*NumericFieldName*)

FROM *SourceTable* AS *s2*

WHERE *s2.IDFieldName = s1.IDFieldName*)

* + - This example would have to evaluate the condition for each row to see if the value in *NumericFieldName* for that record is greater than the average for all records
  + **EXISTS**
    - Can use this to see if the subquery returns a result set
      * No result set is actually returned, it’s just a T/F evaluation
      * Because of this, you usually SELECT \* in the subquery because who cares?
    - It works similarly to using IN and NOT IN with a WHERE subquery, but this one would execute quicker, because it skips evaluation if no result set is returned
    - Example:
      * WHERE EXISTS (SELECT *\** FROM… WHERE…)
    - This example if set up correctly, could work the same as WHERE *SourceField* IN (SELECT…)
    - Using EXISTS can make correlated subqueries perform faster
      * EXISTS is usually only used with correlated subqueries, because other methods like JOINs are preferred for other circumstances
* **HAVING** subqueries
  + not that common
  + Example:
    - GROUP BY …

HAVING AVG(*FieldName*) > (SELECT AVG(*FieldName*) FROM …)

* **FROM** subqueries
  + very useful, because they let you create the ‘table’ to use in your FROM statement
  + MUST assign aliases to any calculated values in a FROM subquery (should do this anyway tho)
  + Example:
    - SELECT *(Column\_list)*

FROM (SELECT *(Column\_list)* FROM *TableName* WHERE *Conditions*)

* + The table that is created by the second SELECT statement will be the table that is used in the first SELECT statement
    - This lets you filter a table or generate aggregate columns, concatenations, etc., and then run a SELECT
    - It’s basically creating a temporary table that is only stored in RAM while the query runs
  + Can JOIN to a derived table produced from a subquery
    - Example:
      * SELECT (*ColumnList*)

FROM *TableName*

INNER JOIN

(SELECT … FROM *TableName*…)

* + - This example not only joins to a derived table, but the derived table is produced from the original table, allowing you to use aggregate functions, TOP keyword, and other conditions
* **SELECT** subqueries
  + Used instead of a column specification in a SELECT statement
    - So it must return a single value for each row
  + Commonly used with aggregate functions that return a single value, like AVG, MAX, MIN, SUM
  + Example:
    - SELECT DISTINCT

[EMPLID]

, [STDNT\_DEGR]

, (SELECT CAST(MAX([STDNT\_DEGR]) AS INT) FROM <RIC\_Src>.[Src].[DEGREES] dg WHERE dg.[EMPLID] = d.[EMPLID]) AS [MaxStdntDegr]

, CASE WHEN

(SELECT CAST(MAX([STDNT\_DEGR]) AS INT) FROM <Src>.[Src].[DEGREES] dg WHERE dg.[EMPLID] = d.[EMPLID]) = [STDNT\_DEGR]

THEN ‘True’ ELSE ‘False’ END as [ESRPrimAlum]

FROM <RIC\_Src>.[Src].[DEGREES] d

* + This example would display the maximum student degree value [MaxStdntDegr] along with each record’s actual student degree value [STDNT\_DEGR], and use the CASE statement with the second subquery to set [ESRPrimAlum] ‘True’ only for records with the max [STDNT\_DEGR]
    - Saves some major time coding, and works really well! Thanks Aisha!
* Complex subquery structure available p. 205 Murach’s SQL Server 2016

**Common Table Expressions (CTE)**

* Feature introduced with SQL Server 2005 helping to create derived tables for complex queries, instead of using subqueries
* Uses the WITH keyword to specify the derived table(s) to use in the query
* Basic Syntax:
  + WITH *DerivedTableName* AS

(SELECT ……..)

SELECT ….. FROM *DerivedTableName*

* + You can code multiple derived tables separating each with a comma, and no comma after the last one
    - WITH *DerivedTableName1* AS

(SELECT *yada yada* FROM *yada yada*),

*DerivedTableName2* AS

(SELECT *yada yada* FROM *yada yada*)

SELECT *yada yada*

FROM *DerivedTableName1* *t1*

INNER JOIN

*DerivedTableName2 t2*

ON *t1.ColumnName = t2.ColumnName*

* Can code recursive CTE queries that loop through data, and seem really complex
  + see p. 210 Murach SQL Server 2016 for details

**Data Manipulation Language (DML) for Modifying or Adding Information**

* **INSERT**
  + Adds data to existing columns in a table
  + Basic Syntax
    - INSERT INTO *TargetTable*

(*ColumnName1, ColumnName2, ColumnName3*)

VALUES

(*Value1, Value2, Value3*);

* + - * The order here matters: the values will be inserted into the columns in the same order they are listed -- no matter what the names are
  + Combining a SELECT with an INSERT
    - Use to retrieve data, then add it to a different table
    - Syntax
      * INSERT INTO *TargetTable*

(*ColumnName1, ColumnName2, ColumnName3, ColumnName4*)

SELECT

(*OtherColumn1, OtherColumn2, OtherColumn3, OtherColumn4*)

FROM *SourceTable*;

* **UPDATE**
  + Changes the data in a table
    - Can change values based on cacluations when the calculation is performed in the *value* part
  + Usually combined with a **WHERE** clause to ensure only the desired records are affected
  + Change the value of a field
    - UPDATE *TableName*

SET *ColumnName* = *value*

WHERE *condition*

* + Combining subqueries with updates

UPDATE [TRAINING\_RIC\_Trg].[dbo].[Organization\_Relationship]

SET [ORIsprimary] = 'True'

-- SELECT [temp1].[ImportID], [temp1].[ORImpID], org.[ORIsprimary]

FROM (SELECT [ImportID], MAX([ORImpID]) AS [ORImpID]

FROM [TRAINING\_RIC\_Trg].[dbo].[Organization\_Relationship]

GROUP BY [ImportID]) AS [temp1]

INNER JOIN [TRAINING\_RIC\_Trg].[dbo].[Organization\_Relationship] org

ON [temp1].[ORImpID] = org.[ORImpID]

* + This subquery sets ORIsprimary to true, only for the max sequence\_nbr, for each emplid
    - It assumes that the ORImpID’s for the different records for each constituent differ only in that the sequence number is appended to the end, so the max ORImpID correlates with the max sequence\_nbr for each ImportID
    - You could just use the sequence nbr, and change the join condition to [ImportID] = [EMPLID] AND [NOTES] = [SEQUENCE\_NBR], as long as you stored [SEQUENCE\_NBR] in [NOTES]
* **DELETE**
  + Deletes record(s) from a table
  + Usually combined with **WHERE** statements so only desired records are deleted
  + You can omit the ‘FROM’ keyword here, and people often do
  + Delete records that meet a certain condition
    - DELETE FROM *TableName*

WHERE *condition*

* + Delete every record from a table
    - DELETE FROM *TableName*
  + Can use subqueries and JOINs here, with syntax similar to what is described above
    - Example:
      * DELETE FROM *TableName1*

FROM *TableName1 t1*

INNER JOIN

(SELECT *Columns*

FROM *TableName2*

WHERE *conditions*

GROUP BY *column*) AS *TempTable t2*

ON *t1.column = t2.column*

WHERE *conditions*;

**Merge Two Tables**

* MERGE works on two tables, a source and a target
  + You can choose to INSERT, UPDATE, DELETE, or do nothing when a match between source/target is found
  + WHEN MATCHED
    - this is the case when the ON condition finds a match from the source in the target
  + WHEN NOT MATCHED
    - this is the case when the ON condition does not find a match from the source in the target
  + Can add AND statements to the WHEN MATCHED, and WHEN NOT MATCHED clauses
* Basic Syntax:
  + MERGE INTO *TargetTable t*

USING *SourceTable s*

ON *t.ID = s.ID* -- would usually use a PRIMARY KEY here

WHEN MATCHED

[AND *optional conditional statement*]

[AND *additional optional conditional statement*]

THEN

[*actions such as* UPDATE, INSERT, or DELETE]

WHEN NOT MATCHED

THEN

[*actions such as* INSERT…]

; -- must end a MERGE statement with a semicolon or you will break SQL

* Example to combine two tables of constituents (Constituents1 and Constituents2) into one table:
  + MERGE INTO [Constituents1] c1

USING [Constituents2] c2

ON c1.[EMPLID] = c2.[EMPLID]

WHEN MATCHED

AND c1.[Name] = c2.[Name]

AND c1.[Address1] = c2.[Address1]

THEN UPDATE SET

c1.[EMPLID] = c2.[EMPLID] -- this essentially does nothing since they’re matched

WHEN NOT MATCHED THEN

INSERT ([EMPLID], [NAME], [ADDRESS1], …….)

VALUES (c2.[EMPLID], c2.[NAME], c2.[ADDRESS1]…)

;

**Converting Data Types**

* CAST vs CONVERT
  + Both do the same thing
  + Use CAST unless you want to specify a format
  + Use CONVERT when you want to specify a format (like for dates)
* CAST
  + Syntax:
    - CAST(*SourceValue* AS *datatype*)
      * your *SourceValue* could be a single value or a column name
* CONVERT
  + Syntax:
    - CONVERT(*datatype, SourceValue[, style]*) -- *style* code is optional
      * must lookup *style* codes, because they are simply a number representing a style
* TRY\_CONVERT
  + this will return a NULL value rather than an error if the type conversion fails
  + Syntax:
    - TRY\_CONVERT(*datatype, SourceValue[, style]*) -- *style* code is optional
* Additional Conversion Functions -- probably use CAST/CONVERT/TRY\_CONVERT most of the time
  + STR(*float[, length[, decimal]]*)
    - converts a float to a decimal with the number of digits specified after the *decimal* (optional) point, before converting to a string with a maximum specified *length* (optional)
  + CHAR(*integer*)
    - converts an INT between 0-255 to its ASCII char equivalent
  + ASCII(*string*)
    - converts the first char only in a *string* to its ASCII code
  + NCHAR(*integer*)
    - converts the Unicode code represented by an *integer* 0-65535 to its char equivalent
  + UNICODE(*string*)
    - converts the first char only in a *string* to its equivalent Unicode code
* Common ASCII codes using the Additional Functions above
  + Char(9) -- tab
  + Char(10) -- line feed
  + Char(13) -- carriage return

**Working with String Data**

* This section includes information about working with data as strings
  + ‘String’ is programming language for text
  + Strings are coded in single quotes
* Concatenating
  + Combining strings into one string
  + Done using the ‘+’ operator
  + Will throw an exception (and return no result) if ANY value you are trying to concatenate is not a string
    - Must convert other datatypes
    - Example
      * CAST(*NumberField* as varchar) -- will convert numbers in this field to strings
  + Adding a single apostrophe
    - You need to concatenate it into the data, and surround it by single quotes
    - Example
      * To get the value -> Chuck’s
      * ‘Chuck’ + ‘’’ + ‘s’
* **Functions for working with strings**
  + LEN(*string*)
    - returns the number of chars in the string, leading spaces included, trailing spaces excluded
  + LTRIM(*string*)
    - returns the string with any leading spaces removed
    - helps to remove space chars when parsing field contents, like names
  + RTRIM(*string*)
    - returns the string with any trailing spaces removed
    - helps to remove space chars when parsing field contents, like names
  + LEFT(*string, length*)
    - returns the specified number of chars starting at the beginning of the string
  + RIGHT(*string, length*)
    - returns the specified number of chars starting at the end of the string
  + SUBSTRING(*string, start, length*)
    - returns the specified number of chars (*length*) beginning at the *start* position
    - useful when combined with CHARINDEX and LEN
  + REPLACE(*string, find, replace*)
    - returns the *string* with all occurrences of *find* replaced with *replace*
  + REVERSE(*string*)
    - returns the string with the chars in the reverse order
  + CHARINDEX(*find*, *string[, start]*)
    - returns an integer for the position of the first occurrences of the *find* string in the *string* supplied, can supply an optional *start* position or it begins at the beginning by default
    - returns 0 if the *find* string is not found
  + PATINDEX(*find, string*)
    - returns an integer for the position of the first occurrence of *find*, and returns 0 if it isn’t found
    - *find* can include wildcard chars
      * if a wildcard is the first char in the *find* string, the value returned is the position of the first non-wildcard char
  + CONCAT(*string1, string2[, string3…]*)
    - Not compatible before SQL Server 2012
    - returns a string that contains a concatenation of the specified values
    - values are implicitly converted to strings, and NULL values are cast as empty strings
  + LOWER(*string*)
    - returns the string with all chars converted to lowercase
  + UPPER(*string*)
    - returns the string with all chars converted to uppercase
  + SPACE(*integer*)
    - returns a string with the specified number of blank space chars

**Working with Numeric Data**

* Artithmetic Operators
  + \* multiplication
  + / division
  + % modulus (returns only the remainder from division)
  + + addition
  + - subtraction
* Performing arithmetic on two INT datatypes will return an INT value
  + even if you’re doing division, and the result should be a decimal
  + you neec to CAST one of the values to a ‘decimal’ datatype first, to return a decimal result
* Sorting numeric data
  + may need to ORDER BY CAST(*FieldName* AS INT) if numeric data is stored as text
* **Functions for working with numeric data**
  + ROUND(*number, length[, function]*)
    - returns the number rounded to the number of decimal places specified
      * zeros will still be there after rounding for some silly reason
    - positive *length* goes to the right of the decimal, negative *length* goes to the left of the decimal
    - to truncate rather than round, code 1 for the function
  + ISNUMERIC(*expression*)
    - returns 1 if ‘True’ and 0 if ‘False’
  + ABS(*number*)
    - returns the absolute value of the number
  + CEILING(*number*)
    - rounds up to the nearest INT
  + FLOOR(*number*)
    - rounds down to the nearest INT
  + SQUARE(*float*)
    - returns the square of a floating-point number
    - must cast to float if not a float
  + SQRT(*float*)
    - returns the square root of a float
    - must cast to float if not a float
  + RAND([*integer*])
    - returns a random float between 0-1
    - if *integer* is specified, it provides a starting value for the function, else the function will return the same value each time it’s called within the same query

**Working with dates and times**

* **Functions for working with dates and times**
  + GETDATE()
    - returns a datetime value for the current local date/time based on the system clock
  + GETUTCDATE()
    - returns a datetime value for the current UTC date/time based on the system clock and time zone setting
  + SYSDATETIME()
    - returns a datetime2(7) value for the current local date/time based on the system clock
  + SYSUTCDATETIME()
    - returns a datetime2(7) value for the UTC date/time based on the system clock and time zone setting
  + SYSDATETIMEOFFSET()
    - returns a datetimeoffset(7) value for the current UTC date/time based on the system clock and time zone with a time zone offset that is not adjusted for daylight savings
  + DAY(*date*)
    - returns the day of the month as an INT
  + MONTH(*date*)
    - returns the month as an INT
  + YEAR(*date*)
    - returns the 4-digit year as an INT
  + DATENAME(*datepart, date*)
    - returns the part of the *date* specified by *datepart* as a char string
  + DATEPART(*datepart, date*)
    - returns the part of the date specified by a *datepart* string as an integer
  + DATEADD(*datepart, number, date*)
    - returns the date that results from adding the specified *number* of *datepart* units to the *date*
  + DATEDIFF(*datepart, startdate, enddate*)
    - returns the number of *datepart* units between the specified *startdate* and *enddate*
  + TODATETIMEOFFSET(*datetime2, tzoffset*)
    - returns a datetimeoffset value that results from adding the specified *time zone offset* to the specified *datetime2* value
  + SWITCHOFFSET(*datetimeoffset, tzoffset*)
    - returns a datetimeoffset value that results from switching the time zone offset for the specified *datetimeoffset* value to the *specified offset*
  + EOMONTH(*startdate[, months]*)
    - returns a date for the last day of the month specified by *startdate*
    - if *months* is specified, the number of *months* is added to the start date before the calculation
  + DATEFROMPARTS(*year, month, day*)
    - returns a date for the specified *year, month,* and *day*
  + ISDATE(*expression*)
    - returns 1 if the *expression* is a valid datetime else 0
* Date part values (use for the *datepart* argument with the functions above)
  + year
    - yy -- two digit year
    - yyyy -- four digit year
  + quarter
    - qq -- two digit quarter
    - q -- one digit quarter
  + month
    - mm -- two digits
    - m -- one digit
  + dayofyear
    - dy
    - y
  + day
    - dd
    - d
  + week
    - wk
    - ww
  + weekday
    - dw
  + hour
    - hh
  + minute
    - mi
    - n
  + second
    - ss
    - s
  + millisecond
    - ms
  + microsecond
    - mcs
  + nanosecond
    - ns
  + tzoffset
    - tz

**Data Definition Language (DDL)**

* **CREATE**
  + **DATABASE**
    - CREATE DATABASE *DataBaseName*;
  + **TABLE**
    - CREATE TABLE *TableName*

(

*ColumnName1 datatype attributes,*

*ColumnName2 datatype attributes,*

*ColumnName3 datatype attributes*

);

* + - Example:
      * CREATE TABLE Constituents

(

EMPLID nvarchar(max) PRIMARY KEY

ORG\_ID bigint NOT NULL REFERENCES Orgs(OrgID)

);

* + **INDEX**
* **ALTER**
  + **TABLE**
    - Syntax
      * ALTER TABLE *TableName*

*actions*

* + - Need to ALTER TABLE to add or delete columns
      * ADD *ColumnName*
        + will add a column
      * DROP *ColumnName*
        + will delete a column
    - Example:
      * ALTER TABLE Constituents

ADD Row\_Num IDENTITY

* + **INDEX**
* **DROP**
  + **DATABASE**
  + **TABLE**
  + **INDEX**

**Relational Databases – General Info**

* Are comprised of tables
  + Tables consist columns and rows
  + Each column holds a certain type of data
    - Often called ‘fields’
  + Each row is an instance (client, organization, order, gift, etc.)
    - Often called ‘records’
* Primary Keys
  + Are not required for all tables, but are very useful for most tables
  + Must be unique
* Indexes
  + Are not required
  + Must also be unique
  + Provide a quick way to query a table
* Constraints (additional)
  + NOT NULL
    - can’t contain a null value
  + PRIMARY KEY
    - unique, not null
  + REFERENCES
    - creates a foreign key that links to another table
  + UNIQUE
    - must contain a unique value
  + CHECK
    - use to apply a value constraint
    - CHECK (*col\_name* *condition*)
      * *condition* could be ‘> *value*’ etc.
* Relationships between tables
  + Table must link using data from one or more columns/fields
  + Example:
    - Table1 is a list of organizations, each having a unique primary key called ID
    - Table2 is a list of people, who each work at an organization
    - Instead of storing all of the organization info in both table1 and table2, you can store all the info in table1, and just use the ID in table 2

**SQL Server (2016)**

**Acronyms**

* SQL Server Management Studio (SSMS)

**SQL Server Data Types**

* bit
  + single character with value of 0 or 1 representing ‘True’ or ‘False’
* int, bigint, smallint, tinyint
  + integer (whole number) values of various sizes (characters allowed)
* money, smallmoney
  + monetary values accurate to four decimal places
* decimal, numeric
  + decimal values (accurate to the least significant digit)
  + can set optional **p**recision (total chars) and **s**cale (nums after the decimal)
    - decimal(p, s) and numeric(p, s)
* float, real
  + floating point values that approximate a decimal value
* dates/times
  + datetime, smalldatetime
    - dates/times
    - smalldatetime is sufficient for most things unless you need to go before 1900, after 2079, or include milliseconds
  + date
    - no time, includes month, day, and year for years 0000-9999
  + time
    - time only, no date 0 - 23:59:59.9999999
    - optional parameter time(n) specifies how many decimal points for the sub/seconds
  + datetime2
    - like datetime, but with more decimal places available with optional datetime2(n)
  + datetimeoffset(n)
    - like datetime2, but includes an optional timezone offset of -14 to 14
* char, varchar
  + string of letters, symbols, and numbers in ASCII char set
  + adding (max) as an optional parameter lets you save oodles of data in this field
* nchar, nvarchar
  + string of letters, symbols, and numbers in the Unicode char set
  + these types use twice as much storage space as the ASCII type equivalents
  + adding (max) as an optional parameter lets you save oodles of data in this field
* varbinary
  + adding (max) as an optional parameter lets you save oodles of data in this field

**Attach a Database**

* **SSMS**
  + Object Explorer (side window)
    - Right click ‘Databases’ folder -> Attach
      * Add…
        + Navigate to the .mdf file of the database you want to attach
      * OK

**Detach a Database**

* **SSMS**
  + Object Explorer (side window)
    - Right click the folder for the database you want to detach -> Tasks -> Detach
      * OK

**Backup a Database**

* **SSMS**
  + Object Explorer (side window)
    - Right click the folder for the database you want to restore -> Tasks -> Backup
      * General (tab)
        + Backup type: Full
        + Check the destination and set the path

Add… the path you want

Remove a path you don’t want

* + - * OK

**Resore a Database**

* **SSMS**
  + Object Explorer (side window)
    - Right click the folder for the database you want to restore -> Tasks -> Restore -> Database
      * From Device
        + General (Tab)

Add…

Choose the .bak file for the backup you want

Check ‘Restore’

* + - * + Options (Tab)

Check -> Overwrite the existing database

* + - * + OK

**View the Database Diagrams**

* **SSMS**
  + Expand the folder for the database you want to view
    - Expand the ‘Database Diagrams’ folder
    - ‘Yes’ to create required support objects if that window appears

**Stored Procedures**

* Set of one or more SQL statements that are stored together in a database
* Creating a stored procedure
  + CREATE PROCEDURE *procedureName* *@inputParameters* *datatype* AS

*SQL statements…*

* Executing a stored procedure
  + EXEC *procedureName* *‘inputParameter’*;

**Triggers**

* Special type of stored procedure that’s executed automatically when an INSERT, UPDATE, or DELETE operation is executed or when DDL statement is executed
* Utilization
  + Most often used to validate data before a row is added/updated
  + Maintain relationships
  + Provide info about changes to the definition of a database

**User Defined Functions (UDF)**

* Special type of procedure

**Scripts (T-SQL)**

**Script Basics**

* Allow execution of multiple ‘batches’ of code
* A batch ends with a GO command
* Helps to separate batches that must be executed in a specific order or by themselves
* Good practice to provide script documentation in /\* block comments \*/
* Individual statements should end with a semicolon ‘;’

**Statements that must be in their own batch**

* CREATE VIEW
* CREATE TRIGGER
* CREATE PROCEDURE
* CREATE SCHEMA
* CREATE FUNCTION

**T-SQL Statements for Script Processing**

* IF…ELSE
  + control flow based on evaluation of a condition
* BEGIN…END
  + define a statement block
* WHILE
  + repeats statements while a condition is true
* BREAK
  + exits the innermost while loop
* CONTINUE
  + returns to the beginning of a while loop
* TRY…CATCH
  + controls flow of execution when an error occurs
* GOTO
  + unconditionally changes the flow of execution
* RETURN
  + exits unconditionally
* USE
  + changes the database context to the specified database
* PRINT
  + returns a message
* DECLARE
  + declares a local variable
* SET
  + sets the value of a local variable or session variable
* EXEC
  + executes a dynamic SQL statement or stored procedure

**Variables**

* Variables must be declared, and can be different types
  + scalar (single value) variables
  + table variables
    - store entire tables
* Declaring a scalar variable
  + DECLARE @*variable\_name1* *data\_type* [, @*variable\_name2* *data\_type*…]
    - use ‘@’ before your variables
    - include a data type for each variable
    - separate multiple variable declarations with commas
    - if declared in a batch, this is a local variable, only accessible in that batch
* Setting the value of a scalar variable
  + SET @*variable\_name1 = expression*
  + Alt syntax (perhaps better when setting values for multiple variables)
    - SELECT @*variable\_name1* = *column\_specification1*[, @*variable\_name2 = col\_spec2*…]
* Declaring a table variable
  + DECLARE @*table\_name* TABLE

(

*col1\_name data\_type [attributes]*

*[, col2\_name data\_type [attributes]…]*

)

* + Using a table variable
    - can use within the following areas in a script:
      * SELECT
        + except a SELECT INTO statement
      * INSERT
      * UPDATE
      * DELETE

**Using Variables in Scripts**

* Can’t use a variable in place of SQL keywords or object names (like a table name)
  + however you can use dynamic SQL for this