.NET Full Stack

Development Program

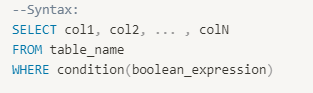
Day9 T-SQL Query

# Outline

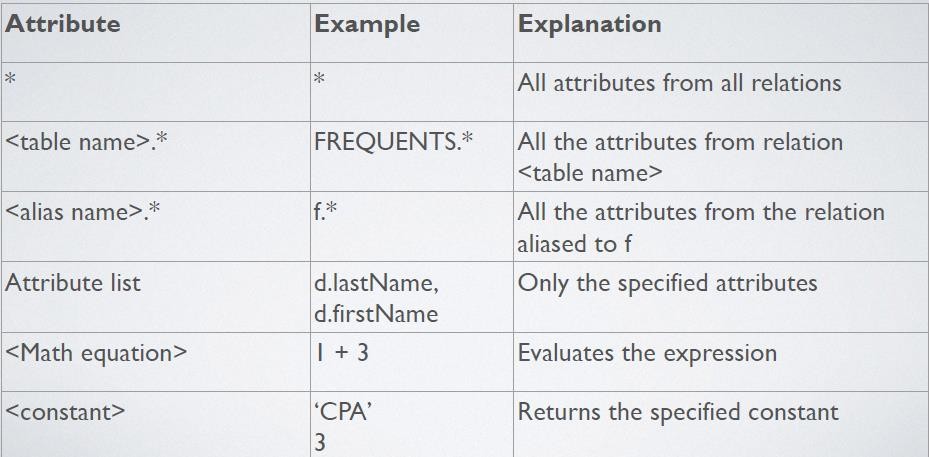
* Basic SQL Query
* Aggregate Function
* Join
* Sub-query & Set Operator
* View
* Variable
* Procedure
* Index

# Select From Where

* The **SELECT** statement is used to select data from a database.
* The **WHERE** clause is used to filter records
* The data returned is stored in a result table, called the result set.
* Example:



# SELECT Clauses

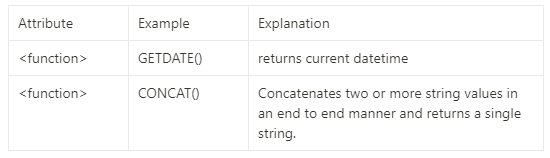


AS

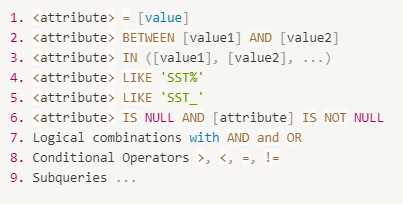
* The AS command is used to create a more meaningful name by renaming a column or table with alias.
* It’s optional and only exists for the duration of the query.



# More SELECT Clauses

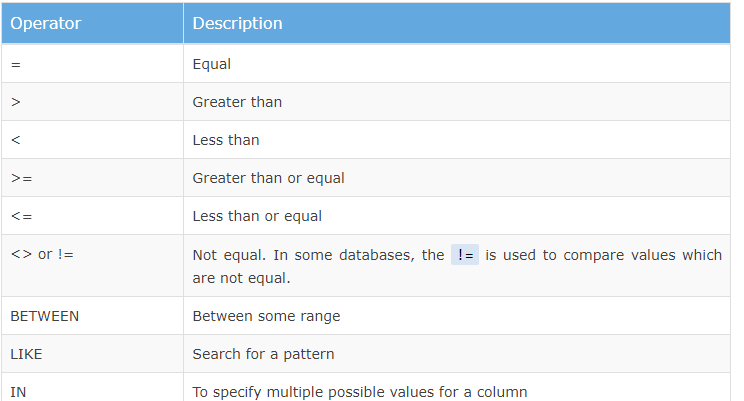
* Functions can be used in SELECT statement
* More build-in functions:
* [https://learn.microsoft.com/en-us/sql/t-sql/functions/functions?view=sql- server-ver15](https://learn.microsoft.com/en-us/sql/t-sql/functions/functions?view=sql-server-ver15)

# WHERE Clause



WHERE Clause

* Conditional Operators



# Aggregate Functions

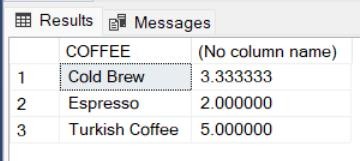
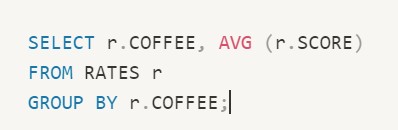
* Aggregate functions are build-in functions that used to perform simple statistics
  + COUNT()
  + AVG()
  + SUM()
  + MAX()
  + MIN()
* The COUNT() function returns the number of rows that matches a specified criterion.
* The AVG() function returns the average value of a numeric column.
* The SUM() function returns the total sum.
* The MIN() and MAX() functions return the smallest and largest values of the selected columns respectively.

# Aggregate Functions

* How does aggregate functions handle Null value?
  + COUNT(\*) gives the total number of records in the table including Null values, there is no difference between these two functions (\*can be replaced by any number in the int capacity)
  + COUNT(column\_name) only considers rows where the column contains a Not-Null value
  + AVG, MIN, MAX, etc. ignore Null values
  + GROUP BY includes a row for null.

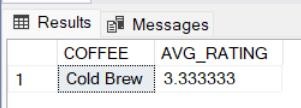
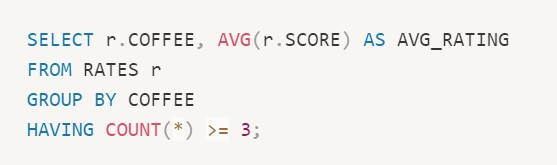
<https://learn.microsoft.com/en-us/sql/t-sql/functions/count-transact-sql?view=sql-server-ver16>

# Group By

* In SQL Server, the GROUP BY clause is used to from the groups of records
* Any non-aggregate columns called in the select statement must be in group by
* Optional

# Having

* Having clause is used to filter out grouping records, it’s like the WHERE clause. The difference is WHERE clause cannot be used with aggregate functions, whereas Having clause can work with
* Having clause must come after GROUP BY clause and before ORDER BY



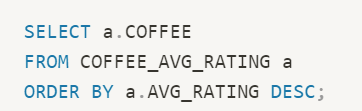
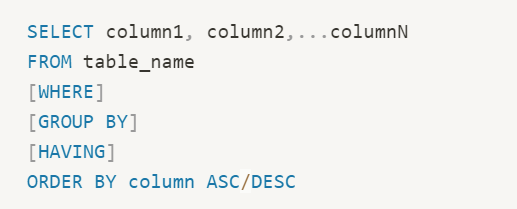
clause

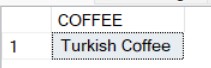
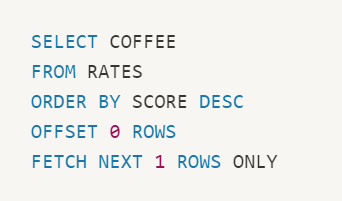
* Optional

# Order By

* The ORDER BY clause is used to sort the result in ascending or descending order.
* ORDER BY clause sorts the result in ascending order by default. To sort the result by descending order, use the DESC keyword.
* The ORDER BY must come after WHERE, GROUPBY, and HAVING clause

if present in the query.





# Offset & Fetch

* The ***OFFSET*** clause specifies the number of rows to skip before starting to

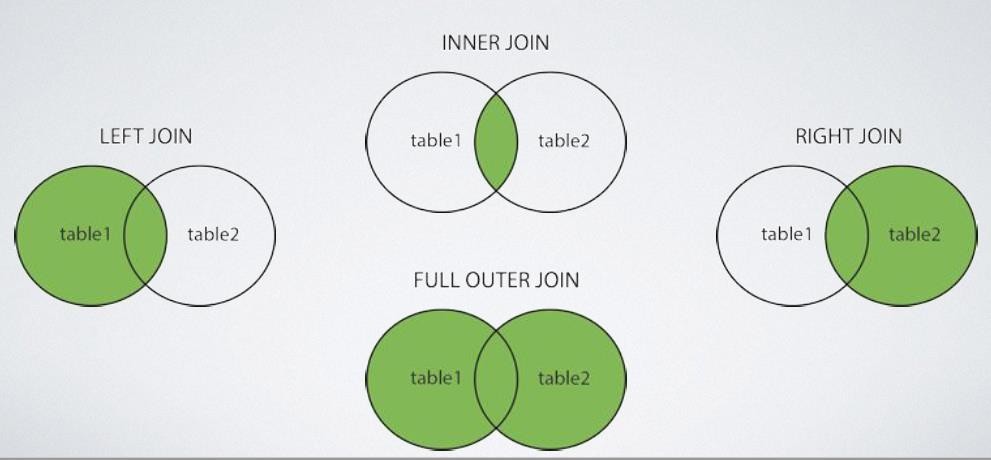
return rows from the query.

* The offset\_row\_count can be a constant, variable, or parameter that is greater or equal to zero.
* The ***FETCH*** clause specifies the number of rows to return after the OFFSET clause has been processed.
* The offset\_row\_count can be a constant or a variable that is greater or equal to one.

# Joins

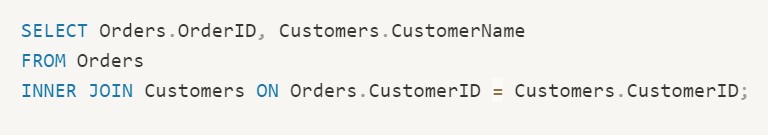
* A JOIN clause is used to combine rows from 2 or more tables, based on a matching column.
* Uses matching data in specified columns to combine or sort data.
* Columns DO NOT have to have the same name.
* Columns DO NOT need to be keys.
* Scope: table to table, table to view, table to synonyms.

# Joins



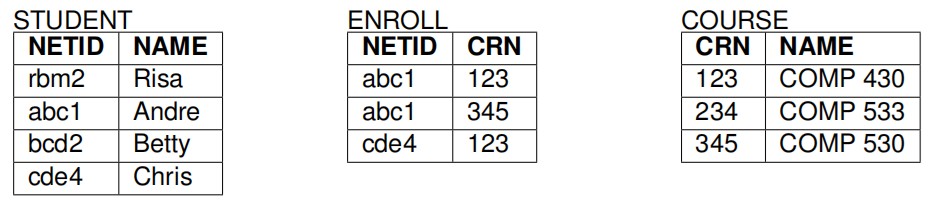
Inner Join

* Inner Join returns records that have matching values in both tables.
* Inner Join selects all rows from both tables as long as there is a match between the columns. If there are records in one table that do not have matches in the other table these records will not be shown.

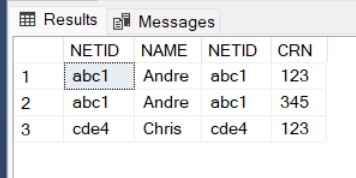
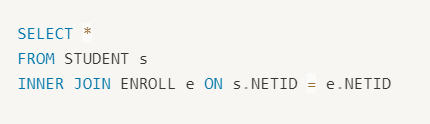


# Inner Join

* Query all the students who enroll in the courses and list their enrolled courses



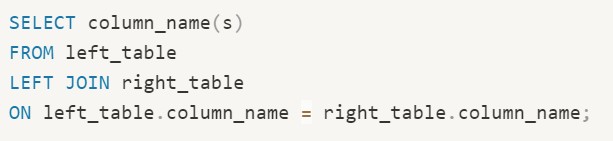
# Inner Join

* Query:

# Left/Right Join

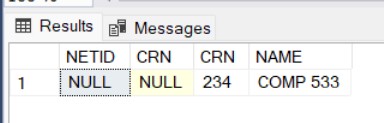
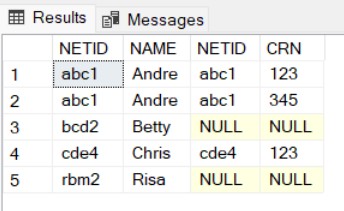
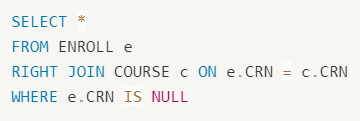
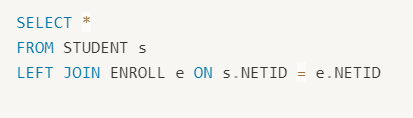
* The LEFT JOIN keyword returns all records from the left table and the matching records from the right table.
* If there is **no matching record** for the left table, **NULL will be assigned in the result set**.
* It’s a good idea to choose one direction (either LEFT or RIGHT) and use it to

maintain consistency.



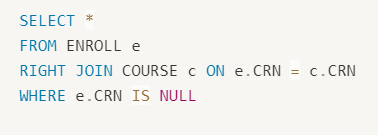
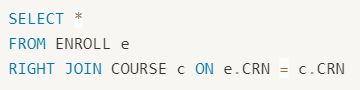
# Left Join

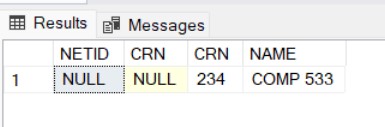
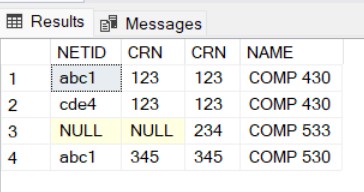
* Query all students and show the enrollment process



# Right Join

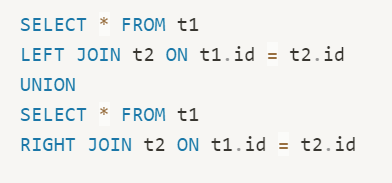
* Query all the enrollment status in course





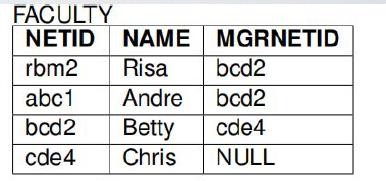
# Full Outer Join

* Used to match up tuples from different relations
* Includes all the relations from both sides
* If there is no matching tuple, shows NULL



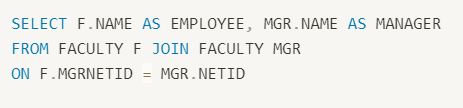
# Self Join

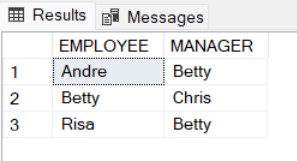
* SELF JOIN is used when a JOIN is used on the same table.



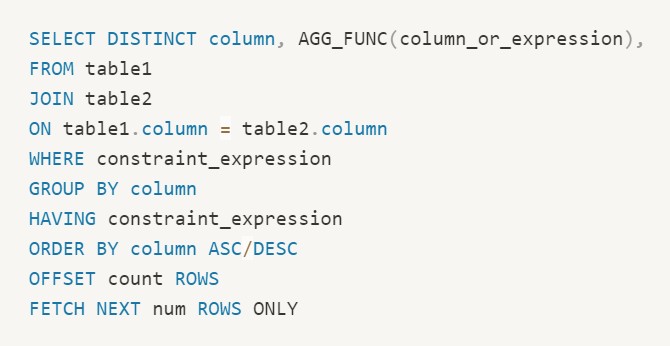
# Self Join

* Query





# Complete Query Example



Set Operations

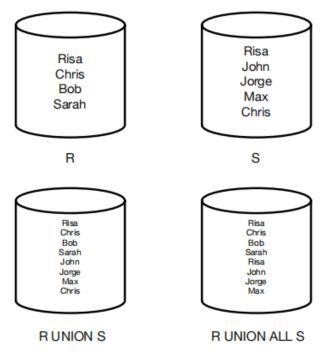
* Results are unordered. It could be useful to perform operations on these:

***Union Intersection Difference***

* Different RDBMS provide different levels of support

# UNION and UNION ALL

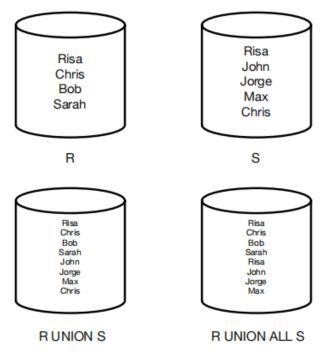
* UNION- eliminates duplicates



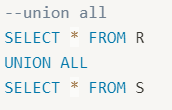
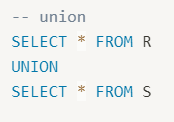
* UNION ALL- does NOT eliminate duplicates
* Uses the column names from the first result set
* ***Data types*** must match
* ***Number of attributes*** must match

# UNION and UNION ALL Example

* R(RName)

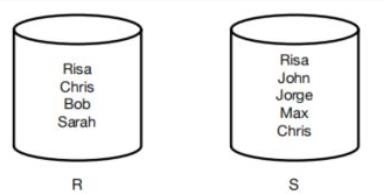


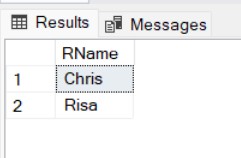
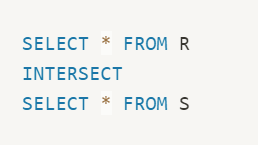
* S(SName)



# Intersection

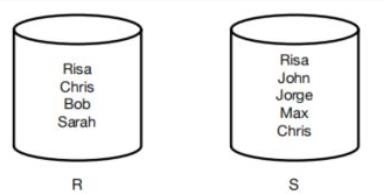
* Intersection Implemented via ***INTERSECT***

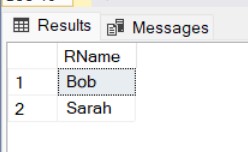




# Difference

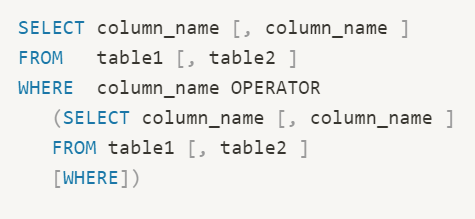
* Difference Implemented via ***EXCEPT***
  + Display the values in the first select statement MINUS any values found in the second select

statement



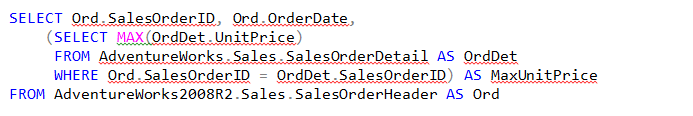
# Subquery

* A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, DELETE statement or inside another subquery.
* Subqueries must be written in parentheses.
* Subqueries must include the SELECT clause and the FROM clause.



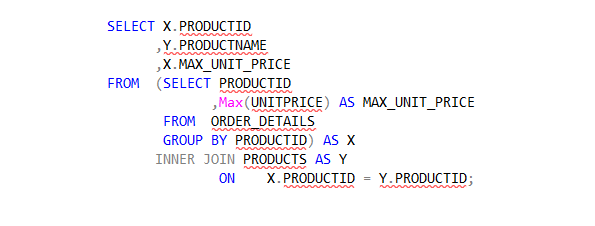
# Where to Put a Subquery

* SELECT clause
  + Can be used to retrieve values in a select clause, but only if they return a single result



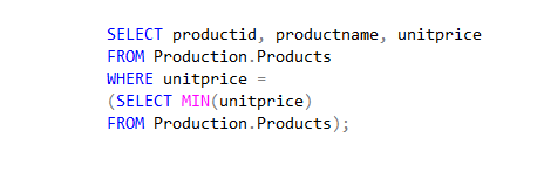
# Where to Put a Subquery

* FROM clause
  + Can be used to return an entire table but **must have an alias**
  + Derived Table



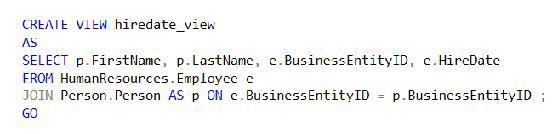
# Where to Put a Subquery

* WHERE clause
  + Most common use
  + Used to filter results based on another table



# View

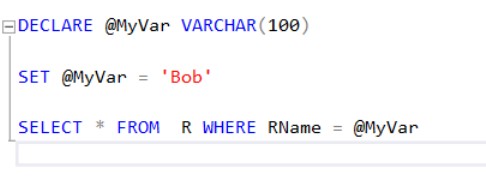
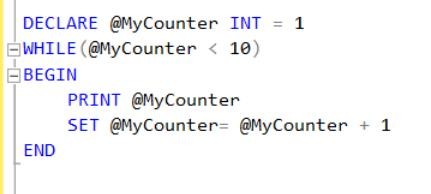
* A view is often seen as a **virtual table**
* It displays data that you choose, but does not actually hold any data
* Good for security since you can prevent showing extra data
* DML operations just happen on the table.
  + you can modify data on view level, and the source data will be updated as well.



# Variable

* A Transact-SQL local variable is an object that can hold a single data value of a specific type. Variables in *batches* and *scripts* are typically used:
  + As a counter either to count the number of times a loop is performed or to control how many times the loop is performed.
  + To hold a data value to be tested by a control-of-flow statement.
  + To save a data value to be returned by a ***stored procedure*** return code or function return value.
* User-Defined Variables are displayed with an "@" symbol
* System Variables are displayed with an "@@" symbol

# Variable Example

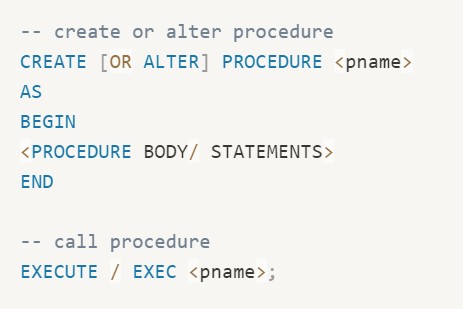


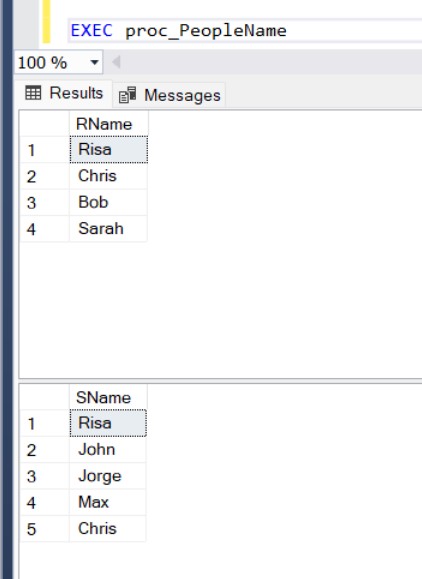
Stored Procedure

* In SQL Server, a stored procedure is a set of T-SQL statements that are compiled and stored in the database. The stored procedure accepts input and output parameters, executes the SQL statements, and returns a result set if any.
* System procedures: System procedures are included with SQL Server and are physically stored in the internal, hidden Resource database and logically appear in the sys schema of all the databases. The system-stored procedures start with the sp\_ prefix.

[https://learn.microsoft.com/en-us/sql/relational-databases/system-stored- procedures/system-stored-procedures-transact-sql?view=sql-server-ver16](https://learn.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/system-stored-procedures-transact-sql?view=sql-server-ver16)

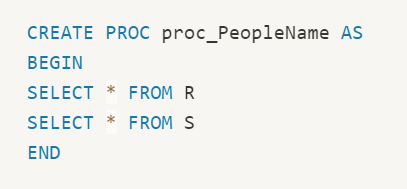
# User Defined Stored Procedure

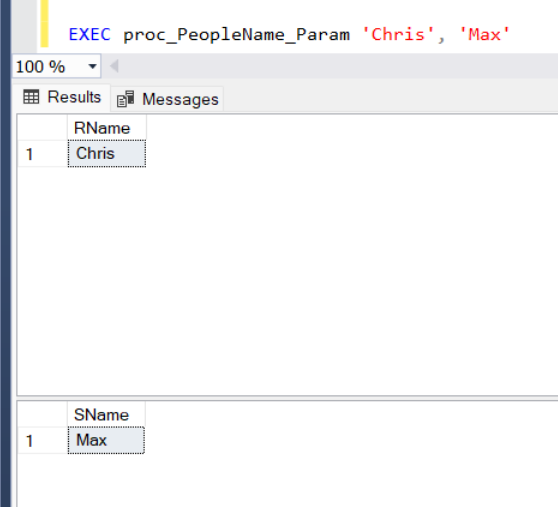
* User Defined Stored Procedures are just Stored Procedures, but created by the user
* Contains statements including calling other stored procedures
* Can have different Input and Output Parameters
* Must be recompiled after time or changes



# User Defined Stored Procedure

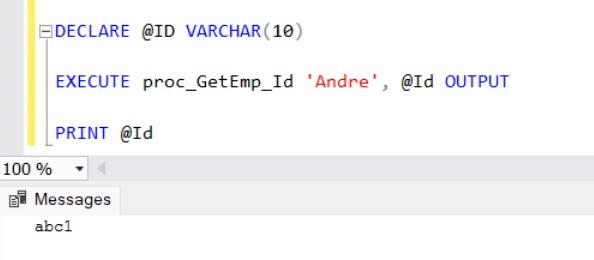
* (basic)
* A stored procedure can have zero or more INPUT and OUTPUT parameters.





# User Defined Stored Procedure

* (SP w/ input parameters)
* Each parameter is assigned a name, and a data type, if no other statement follows, then this parameter is treated as an INPUT parameter



# User Defined Stored Procedure

* (SP w/ both INPUT and OUTPUT)
* Stored procedures can return a value to the calling program if the parameter is specified as OUTPUT.

# Query Execution

Step 1:

* Parser check query syntax
* Break query to token --> (intermediate files)

Step 2:

* Query Optimizer creates the best possible execution plan based on current resource utilization

Step 3:

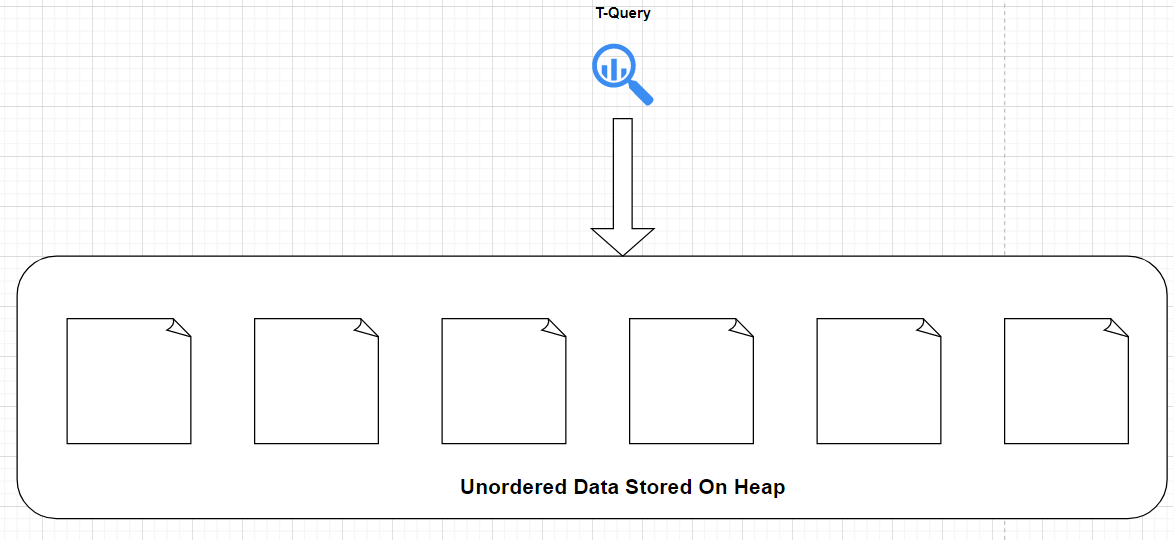
* DB engine --> Run the query

# Index

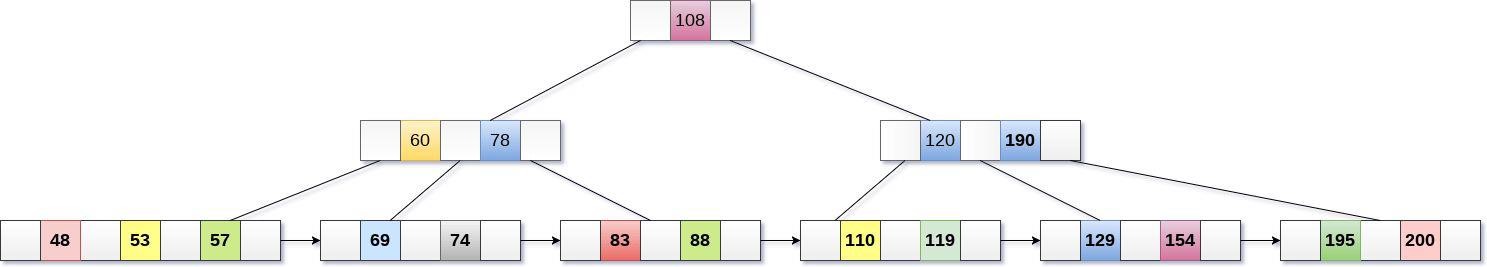
* It's used to sort and **optimize data fetch time**
* Operate similar to index in a book
* When created, an index will create a dynamic Balance Tree (B+ Tree)
* Primary key creates Clustered Index, unique key creates Non-Clustered Index
* Tables without a Clustered Index are called **HEAP** Tables
* Keys ≠ Indexes

# Index

* If a table has no index at all, when new data is inserted, they are added wherever there is free space, and in no particular order.



# CLUSTERED INDEX

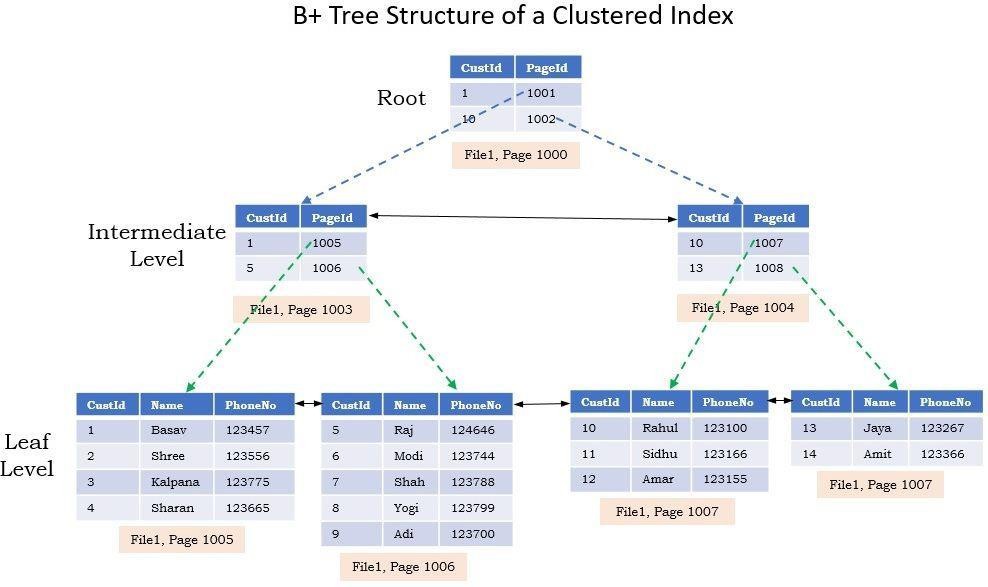
* Composed of 3 main levels
  + Root Level
  + Intermediate Level
  + Leaf Page Level
* Each Node is about 8KB in size
  + 8060B for data
  + 132B for pointers
  + 8192B in Total

Each Index created will have a Balance tree structure to be used, but the type of Index will determine how data is stored in a Balance Tree

Clustered Indexes will store data in Leaf Pages and sort them based on the Key values of the column you choose. Non-Clustered Indexes will **NOT** store data in the Leaf Pages, instead they’ll point to the rows they’re referencing

# CLUSTERED INDEX

* A clustered index will physically move the data

from the table into it’s Balance Tree

* The data is now matching physically and logically
* Data is sorted based on ascending order for the column chosen, this becomes the clustering key
* This is why there can only be 1 Clustered
* Index on a table, data can only be physicall sorted and stored once

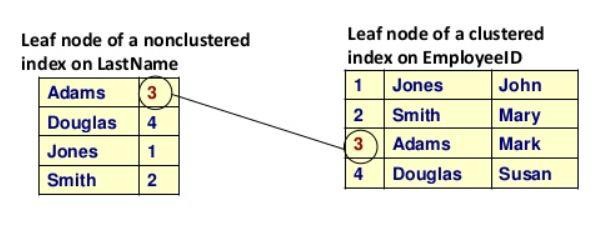
y

# NON-CLUSTERED INDEX

* Since Non-Clustered Indexes do not physically move or store data, there can be many on a single table.
  + Currently up to 999 different Indexes
* A Non-Clustered Index on a table with a Clustered Index must now grab data

from the B-Tree of the CI.

* So data will come up through the Root of the CI and fall into the Leaf Pages of the NCI



# Questions

* How does Index improve the performance?

Find Index vs. Table Scan

* Will Index always improve the performance?

Maintain the index

Questions?