/\*

LAB:::

Explore and document:

a. What are Analytical functions. Document Description and examples on any 5 analytical functions

b. What are grouping sets with examples and variations

\*/

**a)** Analytic functions calculate an aggregate value based on a group of rows. Unlike aggregate functions, however, analytic functions can return multiple rows for each group. Use analytic functions to compute moving averages, running totals, percentages or top-N results within a group.

An analytic function generally looks like this:

function\_name ( arguments ) OVER ( [query\_partition\_clause]

[ORDER BY order\_by\_clause [windowing\_clause] ] )

SQL Server supports these analytic functions:

* [CUME\_DIST (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/cume-dist-transact-sql?view=sql-server-ver16)
* [FIRST\_VALUE (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/first-value-transact-sql?view=sql-server-ver16)
* [LAG (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/lag-transact-sql?view=sql-server-ver16)
* [LAST\_VALUE (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/last-value-transact-sql?view=sql-server-ver16)
* [LEAD (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/lead-transact-sql?view=sql-server-ver16)
* [PERCENT\_RANK (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/percent-rank-transact-sql?view=sql-server-ver16)
* [PERCENTILE\_CONT (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/percentile-cont-transact-sql?view=sql-server-ver16)
* [PERCENTILE\_DISC (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/percentile-disc-transact-sql?view=sql-server-ver16)
* SELECT address\_state,
* COUNT(\*)
* FROM student
* GROUP BY address\_state;

2) COUNT(\*) OVER (PARTITION BY address\_state)

3) SELECT first\_name,

last\_name,

address\_state,

COUNT(\*) OVER (PARTITION BY address\_state) AS state\_count

FROM student;

**b)**-> GROUPING SET in standard query language (SQL) can be considered as a sub-clause of GROUP BY clause. For uninitiated, GROUP BY clause is used to group rows having the same values in a column into summary rows. A grouping set is a set or group of columns by which rows with similar values are grouped together. Functionally, it generates a result set similar to the one generated by a UNION ALL of multiple GROUP BY clauses on a single column. Some other sub-clause of GROUP BY clause such as ROLLUP, CUBE etc also produce result sets equivalent to GROUPING SETS.

**C) Transactions and Isolataion levels + Locks**

A transaction is a single unit of work. If a transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database. If a transaction encounters errors and must be cancelled or rolled back, then all of the data modifications are erased.

SQL Server operates in the following transaction modes:

Autocommit transactions  
Each individual statement is a transaction.

Explicit transactions  
Each transaction is explicitly started with the BEGIN TRANSACTION statement and explicitly ended with a COMMIT or ROLLBACK statement.

Implicit transactions  
A new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a COMMIT or ROLLBACK statement.

Batch-scoped transactions  
Applicable only to multiple active result sets (MARS), a Transact-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.

SQL Server provides the following transaction statements:

[BEGIN DISTRIBUTED TRANSACTION](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/begin-distributed-transaction-transact-sql?view=sql-server-ver16)

[ROLLBACK TRANSACTION](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/rollback-transaction-transact-sql?view=sql-server-ver16)

[BEGIN TRANSACTION](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/begin-transaction-transact-sql?view=sql-server-ver16)

[ROLLBACK WORK](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/rollback-work-transact-sql?view=sql-server-ver16)

[COMMIT TRANSACTION](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/commit-transaction-transact-sql?view=sql-server-ver16)

[SAVE TRANSACTION](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/save-transaction-transact-sql?view=sql-server-ver16)

[COMMIT WORK](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/commit-work-transact-sql?view=sql-server-ver16)

Isolation levels come into play when you need to isolate a resource for a transaction and protect that resource from other transactions. The protection is done by obtaining locks. What locks need to be set and how it has to be established for the transaction is determined by SQL Server referring to the Isolation Level that has been set. Lower Isolation Levels allow multiple users to access the resource simultaneously (concurrency) but they may introduce concurrency related problems such as *dirty-reads*and data inaccuracy. Higher Isolation Levels eliminate concurrency related problems and increase the data accuracy but they may introduce *blocking*.

SQL Server clients can control transaction-isolation levels for a connection. To control transaction-isolation level, the SQL Server Native Client OLE DB provider consumer uses:

* DBPROPSET\_SESSION property DBPROP\_SESS\_AUTOCOMMITISOLEVELS for the SQL Server Native Client OLE DB provider default autocommit mode.

The SQL Server Native Client OLE DB provider default for the level is DBPROPVAL\_TI\_READCOMMITTED.

* The isoLevel parameter of the **ITransactionLocal::StartTransaction** method for local manual-commit transactions.
* The isoLevel parameter of the **ITransactionDispenser::BeginTransaction** method for MS DTC-coordinated distributed transactions.

SQL Server allows read-only access at the dirty read isolation level. All other levels restrict concurrency by applying locks to SQL Server objects. As the client requires greater concurrency levels, SQL Server applies greater restrictions on concurrent access to data. To maintain the highest level of concurrent access to data, the SQL Server Native Client OLE DB provider consumer should intelligently control its requests for specific concurrency levels.

The **SQLServer:Locks** object in Microsoft SQL Server provides information about SQL Server locks on individual resource types. Locks are held on SQL Server resources, such as rows read or modified during a transaction, to prevent concurrent use of resources by different transactions. For example, if an exclusive (X) lock is held on a row within a table by a transaction, no other transaction can modify that row until the lock is released. Minimizing locks increases concurrency, which can improve performance. Multiple instances of the **Locks** object can be monitored at the same time, with each instance representing a lock on a resource type.

This table describes the SQL Server **Locks** counters.

| **SQL Server Locks counters** | **Description** |
| --- | --- |
| **Average Wait Time (ms)** | Average amount of wait time (in milliseconds) for each lock request that resulted in a wait. |
| **Average Wait Time Base** | For internal use only. |
| **Lock Requests/sec** | Number of new locks and lock conversions per second requested from the lock manager. |
| **Lock Timeouts (timeout > 0)/sec** | Number of lock requests per second that timed out, but excluding requests for NOWAIT locks. |
| **Lock Timeouts/sec** | Number of lock requests per second that timed out, including requests for NOWAIT locks. |
| **Lock Wait Time (ms)** | Total wait time (in milliseconds) for locks in the last second. |
| **Lock Waits/sec** | Number of lock requests per second that required the caller to wait. |
| **Number of Deadlocks/sec** | Number of lock requests per second that resulted in a deadlock. |

SQL Server can lock these resources.

| **Item** | **Description** |
| --- | --- |
| **\_Total** | Information for all locks. |
| **AllocUnit** | A lock on an allocation unit. |
| **Application** | A lock on an application-specified resource. |
| **Database** | A lock on a database, including all objects in the database. |
| **Extent** | A lock on a contiguous group of 8 pages. |
| **File** | A lock on a database file. |
| **Heap/B-tree** | Heap or B-tree. A lock on a heap of data pages, or on the B-tree structure of an index. |
| **Key** | A lock on a row in an index. |
| **Metadata** | A lock on a piece of catalog information, also called metadata. |
| **Object** | A lock on table, stored procedure, view, etc, including all data and indexes. The object can be anything that has an entry in **sys.all\_objects**. |
| **OIB** | Lock resource for online index build locks, specifically for a online index build LOB tracking table. |
| **Page** | A lock on an 8-kilobyte (KB) page in a database. |
| **RID** | Row ID. A lock on a single row in a heap. |
| **RowGroup** | Lock resource for a columnstore index rowgroup. |
| **Xact** | Lock resource for a transactions. |

**d. how do we use inserted and deleted tables outside of trigger (\*\*Hint: Use OUTPUT clause)**

**e. What are SET operations in SQL (UNION,UNION ALL, INTERSECT, MINUS)**

Concatenates the results of two queries into a single result set. You control whether the result set includes duplicate rows:

* **UNION ALL** - Includes duplicates.
* **UNION** - Excludes duplicates.

A **UNION** operation is different from a [**JOIN**](https://docs.microsoft.com/en-us/sql/t-sql/queries/from-transact-sql?view=sql-server-ver16):

* A **UNION** concatenates result sets from two queries. But a **UNION** does not create individual rows from columns gathered from two tables.
* A **JOIN** compares columns from two tables, to create result rows composed of columns from two tables.

The following are basic rules for combining the result sets of two queries by using **UNION**:

* The number and the order of the columns must be the same in all queries.
* The data types must be compatible.

**INTERSECT:** Returns the intersection of two input sets, optionally retaining duplicates.

The **Intersect** function returns the intersection of two sets. By default, the function removes duplicates from both sets prior to intersecting the sets. The two sets specified must have the same dimensionality.

The optional **ALL** flag retains duplicates. If **ALL** is specified, the **Intersect** function intersects nonduplicated elements as usual, and also intersects each duplicate in the first set that has a matching duplicate in the second set. The two sets specified must have the same dimensionality.

The **SQL MINUS operator** is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement. Each SELECT statement will define a dataset. The MINUS operator will retrieve all records from the first dataset and then remove from the results all records from the second dataset.