SELECT select_list
[ INTO new_table ]
FROM table_source
[ WHERE search_condition ]
[ GROUP BY group_by_expression ]
[ HAVING search_condition ]
[ ORDER BY order_expression [ ASC | DESC ] ]

SELECT Clause

Specifies the columns to be returned by the query.

Syntax

SELECT [ ALL | DISTINCT ]
[ TOP n [ PERCENT ] [ WITH TIES ] ]
< select_list >

< select_list > ::= 

{ *
  | { table_name | view_name | table_alias }.*
  | { column_name | expression | IDENTITYCOL | ROWGUIDCOL }
  [ [ AS ] column_alias ]
  | column_alias = expression
} [ ,...n ]

Arguments

ALL

Specifies that duplicate rows can appear in the result set. ALL is the default.

DISTINCT

Specifies that only unique rows can appear in the result set. Null values are considered equal for the purposes of the DISTINCT keyword.

TOP n [PERCENT]
Specifies that only the first \( n \) rows are to be output from the query result set. \( n \) is an integer between 0 and 4294967295. If PERCENT is also specified, only the first \( n \) percent of the rows are output from the result set. When specified with PERCENT, \( n \) must be an integer between 0 and 100.

If the query includes an ORDER BY clause, the first \( n \) rows (or \( n \) percent of rows) ordered by the ORDER BY clause are output. If the query has no ORDER BY clause, the order of the rows is arbitrary.

**WITH TIES**

Specifies that additional rows be returned from the base result set with the same value in the ORDER BY columns appearing as the last of the TOP \( n \) (PERCENT) rows. TOP ...WITH TIES can only be specified if an ORDER BY clause is specified.

\(< \text{select_list} >\)

The columns to be selected for the result set. The select list is a series of expressions separated by commas.

*   Specifies that all columns from all tables and views in the FROM clause should be returned. The columns are returned by table or view, as specified in the FROM clause, and in the order in which they exist in the table or view.

\(table\_name | view\_name | table\_alias.*\)

Limits the scope of the * to the specified table or view.

\(column\_name\)

Is the name of a column to return. Qualify \(column\_name\) to prevent an ambiguous reference, such as occurs when two tables in the FROM clause have columns with duplicate names. For example, the Customers and Orders tables in the Northwind database both have a column named ColumnID. If the two tables are joined in a query, the customer ID can be specified in the select list as Customers.CustomerID.

\(expression\)

Is a column name, constant, function, any combination of column names, constants, and functions connected by an operator(s), or a subquery.

**IDENTITYCOL**

Returns the identity column. For more information, see IDENTITY (Property), ALTER TABLE, and CREATE TABLE.

If the more than one table in the FROM clause has a column with the IDENTITY property, IDENTITYCOL must be qualified with the specific table name, such as T1.IDENTITYCOL.

**ROWGUIDCOL**

Returns the row global unique identifier column.
If the more than one table in the FROM clause with the ROWGUIDCOL property, ROWGUIDCOL must be qualified with the specific table name, such as T1.ROWGUIDCOL.

**column_alias**

Is an alternative name to replace the column name in the query result set. For example, an alias such as "Quantity", or "Quantity to Date", or "Qty" can be specified for a column named *quantity*.

Aliases are used also to specify names for the results of expressions, for example:

```sql
USE Northwind
SELECT AVG(UnitPrice) AS 'Average Price'
FROM [Order Details]
```

*column_alias* can be used in an ORDER BY clause. However, it cannot be used in a WHERE, GROUP BY, or HAVING clause. If the query expression is part of a DECLARE CURSOR statement, *column_alias* cannot be used in the FOR UPDATE clause.

**INTO Clause**

Creates a new table and inserts the resulting rows from the query into it.

The user executing a SELECT statement with the INTO clause must have CREATE TABLE permission in the destination database. SELECT...INTO cannot be used with the COMPUTE. For more information, see Transactions and Explicit Transactions.

You can use SELECT...INTO to create an identical table definition (different table name) with no data by having a FALSE condition in the WHERE clause.

**Syntax**

```
[ INTO new_table ]
```

**Arguments**

*new_table*

Specifies the name of a new table to be created, based on the columns in the select list and the rows chosen by the WHERE clause. The format of *new_table* is determined by evaluating the expressions in the select list. The columns in *new_table* are created in the order specified by the select list. Each column in *new_table* has the same name, data type, and value as the corresponding expression in the select list.
When a computed column is included in the select list, the corresponding column in the new table is not a computed column. The values in the new column are the values that were computed at the time SELECT...INTO was executed.

In this release of SQL Server, the select into/bulkcopy database option has no effect on whether you can create a permanent table with SELECT INTO. The amount of logging for certain bulk operations, including SELECT INTO, depends on the recovery model in effect for the database. For more information, see Using Recovery Models.

In previous releases, creating a permanent table with SELECT INTO was allowed only if select into/bulkcopy was set.

select into/bulkcopy is available for backward compatibility purposes, but may not be supported in future releases. Refer to the Recovery Models and Backward Compatibility and ALTER DATABASE topics for more information.
FROM Clause

Specifies the table(s) from which to retrieve rows. The FROM clause is required except when the select list contains only constants, variables, and arithmetic expressions (no column names). For more information, see FROM.

Syntax

[ FROM { <table_source> } [ ,...n ] ]

<table_source> ::= table_name [ [ AS ] table_alias ] [ WITH ( <table_hint> [ ,...n ] ) ]
| view_name [ [ AS ] table_alias ]
| rowset_function [ [ AS ] table_alias ]
| OPENXML
| derived_table [ AS ] table_alias [ ( column_alias [ ,...n ] ) ]
| <joined_table>

<joined_table> ::= <table_source> <join_type> <table_source> ON <search_condition>
| <table_source> CROSS JOIN <table_source>
| <joined_table>

<join_type> ::= [ INNER | { LEFT | RIGHT | FULL } [ OUTER ] ]
| <join_hint> ] JOIN

Arguments

<table_source>

Specifies tables, views, derived tables, and joined tables for the SELECT statement.

table_name [ [ AS ] table_alias ]

Specifies the name of a table and an optional alias.

view_name [ [ AS ] table_alias ]

Specifies the name, a view, and an optional alias.

rowset_function [ [ AS ] table_alias ]

Is the name of a rowset function and an optional alias. For more information about a list of rowset functions, see Rowset Functions.

OPENXML

Provides rowset view over an XML document. For more information see OPENXML.

WITH ( <table_hint> [ ,...n ] )

Specifies one or more table hints. For more information about table hints, see FROM.
derived_table [ [ AS ] table_alias ]
    Is a nested SELECT statement, retrieving rows from the specified database and
table(s).

column_alias
    Is an optional alias to replace a column name in the result set.

< joined_table >

Is a result set that is the product of two or more tables. For example:

```
SELECT *
FROM tab1 LEFT OUTER JOIN tab2 ON tab1.c3 = tab2.c3
    RIGHT OUTER JOIN tab3 LEFT OUTER JOIN tab4
         ON tab3.c1 = tab4.c1
         ON tab2.c3 = tab4.c3
```

For multiple CROSS joins, use parentheses to change the natural order of the joins.

< join_type >

Specifies the type of join operation.

INNER
    Specifies that all matching pairs of rows are returned. Discards unmatched rows
from both tables. This is the default if no join type is specified.

LEFT [ OUTER ]
    Specifies that all rows from the left table not meeting the specified condition are
included in the result set in addition to all rows returned by the inner join. Output
columns from the left table are set to NULL.

RIGHT [ OUTER ]
    Specifies that all rows from the right table not meeting the specified condition are
included in the result set in addition to all rows returned by the inner join. Output
columns from the right table are set to NULL.

FULL [ OUTER ]
    If a row from either the left or right table does not match the selection criteria,
specifies the row be included in the result set, and output columns that correspond
to the other table be set to NULL. This is in addition to all rows usually returned
by the inner join.

< join_hint >
    Specifies a join hint or execution algorithm. If <join_hint> is specified, INNER,
LEFT, RIGHT, or FULL must also be explicitly specified. For more information
about join hints, see FROM.

JOIN
    Indicates that the specified tables or views should be joined.

ON < search_condition >
Specifies the condition on which the join is based. The condition can specify any predicate, although columns and comparison operators are often used. For example:

```sql
SELECT ProductID, Suppliers.SupplierID
FROM Suppliers JOIN Products
    ON (Suppliers.SupplierID = Products.SupplierID)
```

When the condition specifies columns, the columns do not have to have the same name or same data type. However, if the data types are not identical, they must be either compatible or types that Microsoft® SQL Server™ can implicitly convert. If the data types cannot be implicitly converted, the condition must explicitly convert the data type using the CAST function.

For more information about search conditions and predicates, see [Search Condition](#).

**CROSS JOIN**

Specifies the cross-product of two tables. Returns the same rows as if the tables to be joined were simply listed in the FROM clause and no WHERE clause was specified. For example, both of these queries return a result set that is a cross join of all the rows in $T_1$ and $T_2$:

```sql
SELECT * FROM T1, T2
SELECT * FROM T1 CROSS JOIN T2
```
WHERE Clause

Specifies a search condition to restrict the rows returned.

Syntax

[ WHERE < search_condition > | < old_outer_join > ]

< old_outer_join > ::= 
  column_name { * = | = * } column_name

Arguments

< search_condition >

Restricts the rows returned in the result set through the use of predicates. There is no limit to the number of predicates that can be included in a search condition. For more information about search conditions and predicates, see Search Condition.

< old_outer_join >

Specifies an outer join using the nonstandard product-specific syntax and the WHERE clause. The *= operator is used to specify a left outer join and the =* operator is used to specify a right outer join.

This example specifies a left outer join in which the rows from Tab1, that do not meet the specified condition, are included in the result set:

```sql
SELECT Tab1.name, Tab2.id
FROM Tab1, Tab2
WHERE Tab1.id *= Tab2.id
```

Note Using this syntax for outer joins is discouraged because of the potential for ambiguous interpretation and because it is nonstandard. Instead, specify joins in the FROM clause.

It is possible to specify outer joins by using join operators in the FROM clause or by using the non-standard *= and =* operators in the WHERE clause. The two methods cannot both be used in the same statement.

GROUP BY Clause

Specifies the groups into which output rows are to be placed and, if aggregate functions are included in the SELECT clause <select list>, calculates a summary value for each group. When GROUP BY is specified, either each column in any non-aggregate
expression in the select list should be included in the GROUP BY list, or the GROUP BY expression must match exactly the select list expression.

**Note** If the ORDER BY clause is not specified, groups returned using the GROUP BY clause are not in any particular order. It is recommended that you always use the ORDER BY clause to specify a particular ordering of the data.

**Syntax**

```
[ GROUP BY [ ALL ] group_by_expression [ ,...n ]
   [ WITH { CUBE | ROLLUP } ] ]
```

**Arguments**

**ALL**

Includes all groups and result sets, even those that do not have any rows that meet the search condition specified in the WHERE clause. When ALL is specified, null values are returned for the summary columns of groups that do not meet the search condition. You cannot specify ALL with the CUBE or ROLLUP operators.

GROUP BY ALL is not supported in queries that access remote tables if there is also a WHERE clause in the query.

**group_by_expression**

Is an expression on which grouping is performed. `group_by_expression` is also known as a grouping column. `group_by expression` can be a column or a nonaggregate expression that references a column. A column alias that is defined in the select list cannot be used to specify a grouping column.

**Note** Columns of type **text**, **ntext**, and **image** cannot be used in `group_by_expression`.

For GROUP BY clauses that do not contain CUBE or ROLLUP, the number of `group_by_expression` items is limited by the GROUP BY column sizes, the aggregated columns, and the aggregate values involved in the query. This limit originates from the limit of 8,060 bytes on the intermediate work table that is needed to hold intermediate query results. A maximum of 10 grouping expressions is permitted when CUBE or ROLLUP is specified.

**CUBE**
Specifies that in addition to the usual rows provided by GROUP BY, summary rows are introduced into the result set. A GROUP BY summary row is returned for every possible combination of group and subgroup in the result set. A GROUP BY summary row is displayed as NULL in the result, but is used to indicate all values. Use the GROUPING function to determine whether null values in the result set are GROUP BY summary values.

The number of summary rows in the result set is determined by the number of columns included in the GROUP BY clause. Each operand (column) in the GROUP BY clause is bound under the grouping NULL and grouping is applied to all other operands (columns). Because CUBE returns every possible combination of group and subgroup, the number of rows is the same, regardless of the order in which the grouping columns are specified.

ROLLUP

Specifies that in addition to the usual rows provided by GROUP BY, summary rows are introduced into the result set. Groups are summarized in a hierarchical order, from the lowest level in the group to the highest. The group hierarchy is determined by the order in which the grouping columns are specified. Changing the order of the grouping columns can affect the number of rows produced in the result set.

**Important** Distinct aggregates, for example, AVG(DISTINCT column_name), COUNT(DISTINCT column_name), and SUM(DISTINCT column_name), are not supported when using CUBE or ROLLUP. If used, SQL Server returns an error message and cancels the query.
GROUP BY Clause

Specifies the groups into which output rows are to be placed and, if aggregate functions are included in the SELECT clause <select list>, calculates a summary value for each group. When GROUP BY is specified, either each column in any non-aggregate expression in the select list should be included in the GROUP BY list, or the GROUP BY expression must match exactly the select list expression.

**Note** If the ORDER BY clause is not specified, groups returned using the GROUP BY clause are not in any particular order. It is recommended that you always use the ORDER BY clause to specify a particular ordering of the data.

**Syntax**

```
[ GROUP BY [ ALL ] group_by_expression [ ,...n ]
    [ WITH { CUBE | ROLLUP } ] ]
```

**Arguments**

**ALL**

Includes all groups and result sets, even those that do not have any rows that meet the search condition specified in the WHERE clause. When ALL is specified, null values are returned for the summary columns of groups that do not meet the search condition. You cannot specify ALL with the CUBE or ROLLUP operators.

GROUP BY ALL is not supported in queries that access remote tables if there is also a WHERE clause in the query.

**group_by_expression**

Is an expression on which grouping is performed. *group_by_expression* is also known as a grouping column. *group_by_expression* can be a column or a non-aggregate expression that references a column. A column alias that is defined in the select list cannot be used to specify a grouping column.

**Note** Columns of type text, ntext, and image cannot be used in *group_by_expression*.

For GROUP BY clauses that do not contain CUBE or ROLLUP, the number of *group_by_expression* items is limited by the GROUP BY column sizes, the aggregated columns, and the aggregate values involved in the query. This limit originates from the limit of 8,060 bytes on the intermediate work table that is needed to hold intermediate
query results. A maximum of 10 grouping expressions is permitted when CUBE or ROLLUP is specified.

CUBE

Specifies that in addition to the usual rows provided by GROUP BY, summary rows are introduced into the result set. A GROUP BY summary row is returned for every possible combination of group and subgroup in the result set. A GROUP BY summary row is displayed as NULL in the result, but is used to indicate all values. Use the GROUPING function to determine whether null values in the result set are GROUP BY summary values.

The number of summary rows in the result set is determined by the number of columns included in the GROUP BY clause. Each operand (column) in the GROUP BY clause is bound under the grouping NULL and grouping is applied to all other operands (columns). Because CUBE returns every possible combination of group and subgroup, the number of rows is the same, regardless of the order in which the grouping columns are specified.

ROLLUP

Specifies that in addition to the usual rows provided by GROUP BY, summary rows are introduced into the result set. Groups are summarized in a hierarchical order, from the lowest level in the group to the highest. The group hierarchy is determined by the order in which the grouping columns are specified. Changing the order of the grouping columns can affect the number of rows produced in the result set.

Important  Distinct aggregates, for example, AVG(DISTINCT column_name), COUNT(DISTINCT column_name), and SUM(DISTINCT column_name), are not supported when using CUBE or ROLLUP. If used, SQL Server returns an error message and cancels the query.
HAVING Clause

Specifies a search condition for a group or an aggregate. HAVING is usually used with the GROUP BY clause. When GROUP BY is not used, HAVING behaves like a WHERE clause.

Syntax

[ HAVING < search_condition > ]

Arguments

< search_condition >

Specifies the search condition for the group or the aggregate to meet. When HAVING is used with GROUP BY ALL, the HAVING clause overrides ALL. For more information, see Search Condition.

The text, image, and ntext data types cannot be used in a HAVING clause.

Note Using the HAVING clause in the SELECT statement does not affect the way the CUBE operator groups the result set and returns summary aggregate rows.
UNION Operator

Combines the results of two or more queries into a single result set consisting of all the rows belonging to all queries in the union. This is different from using joins that combine columns from two tables.

Two basic rules for combining the result sets of two queries with UNION are:

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

Syntax

```
{ < query specification > | ( < query expression > ) }
UNION [ ALL ]
< query specification | ( < query expression > )
[ UNION [ ALL ] < query specification | ( < query expression > )
[ ...n ] ]
```

Arguments

`< query_specification > | ( < query_expression > )`

Is a query specification or query expression that returns data to be combined with the data from another query specification or query expression. The definitions of the columns that are part of a UNION operation do not have to be identical, but they must be compatible through implicit conversion.

The table shows the rules for comparing the data types and options of corresponding (ith) columns.

<table>
<thead>
<tr>
<th>Data type of ith column</th>
<th>Data type of ith column of results table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not data type-compatible (data conversion not handled implicitly by Microsoft® SQL Server™).</td>
<td>Error returned by SQL Server.</td>
</tr>
<tr>
<td>Both fixed-length char with lengths L1 and L2.</td>
<td>Fixed-length char with length equal to the greater of L1 and L2.</td>
</tr>
<tr>
<td>Both fixed-length binary with lengths L1 and L2.</td>
<td>Fixed-length binary with length equal to the greater of L1 and L2.</td>
</tr>
<tr>
<td>Either or both variable-length char.</td>
<td>Variable-length char with length equal to the maximum of the lengths specified for the ith columns.</td>
</tr>
<tr>
<td>Either or both variable-length binary.</td>
<td>Variable-length binary with length equal to</td>
</tr>
<tr>
<td>Both numeric data types (for example, smallint, int, float, money).</td>
<td>the maximum of the lengths specified for the ( i^{th} ) columns.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Data type equal to the maximum precision of the two columns. For example, if the ( i^{th} ) column of table A is of type int and the ( i^{th} ) column of table B is of type float, then the data type of the ( i^{th} ) column of the results table is float because float is more precise than int.</td>
<td></td>
</tr>
<tr>
<td>Both columns' descriptions specify NOT NULL.</td>
<td>Specifies NOT NULL.</td>
</tr>
</tbody>
</table>

**UNION**

Specifies that multiple result sets are to be combined and returned as a single result set.

**ALL**

Incorporates all rows into the results, including duplicates. If not specified, duplicate rows are removed.

**ORDER BY Clause**

Specifies the sort for the result set. The ORDER BY clause is invalid in views, inline functions, derived tables, and subqueries, unless TOP is also specified.

**Syntax**

```
[ ORDER BY { order_by_expression [ ASC | DESC ] } [ ,...n ] ]
```

**Arguments**

`order_by_expression`

Specifies a column on which to sort. A sort column can be specified as a name or column alias (which can be qualified by the table or view name), an expression, or a nonnegative integer representing the position of the name, alias, or expression in select list.

Multiple sort columns can be specified. The sequence of the sort columns in the ORDER BY clause defines the organization of the sorted result set.

The ORDER BY clause can include items not appearing in the select list. However, if SELECT DISTINCT is specified, or if the SELECT statement contains a UNION operator, the sort columns must appear in the select list.

Furthermore, when the SELECT statement includes a UNION operator, the column names or column aliases must be those specified in the first select list.

**Note** `ntext, text, or image` columns cannot be used in an ORDER BY clause.

**ASC**

Specifies that the values in the specified column should be sorted in ascending order, from lowest value to highest value.

**DESC**

Specifies that the values in the specified column should be sorted in descending order, from highest value to lowest value.

Null values are treated as the lowest possible values.

There is no limit to the number of items in the ORDER BY clause. However, there is a limit of 8,060 bytes for the row size of intermediate worktables needed for sort operations. This limits the total size of columns specified in an ORDER BY clause.
ORDER BY Clause

Specifies the sort for the result set. The ORDER BY clause is invalid in views, inline functions, derived tables, and subqueries, unless TOP is also specified.

Syntax

\[ \text{[ ORDER BY } \{ \text{order\_by\_expression [ ASC | DESC ] } \} \ [,...n]\] \]

Arguments

order\_by\_expression

Specifies a column on which to sort. A sort column can be specified as a name or column alias (which can be qualified by the table or view name), an expression, or a nonnegative integer representing the position of the name, alias, or expression in select list.

Multiple sort columns can be specified. The sequence of the sort columns in the ORDER BY clause defines the organization of the sorted result set.

The ORDER BY clause can include items not appearing in the select list. However, if SELECT DISTINCT is specified, or if the SELECT statement contains a UNION operator, the sort columns must appear in the select list.

Furthermore, when the SELECT statement includes a UNION operator, the column names or column aliases must be those specified in the first select list.

Note ntext, text, or image columns cannot be used in an ORDER BY clause.

ASC

Specifies that the values in the specified column should be sorted in ascending order, from lowest value to highest value.

DESC

Specifies that the values in the specified column should be sorted in descending order, from highest value to lowest value.

Null values are treated as the lowest possible values.

There is no limit to the number of items in the ORDER BY clause. However, there is a limit of 8,060 bytes for the row size of intermediate worktables needed for sort operations. This limits the total size of columns specified in an ORDER BY clause.
**COMPUTE Clause**

Generates totals that appear as additional summary columns at the end of the result set. When used with BY, the COMPUTE clause generates control-breaks and subtotals in the result set. You can specify COMPUTE BY and COMPUTE in the same query.

**Syntax**

```
[ COMPUTE
  { { AVG | COUNT | MAX | MIN | STDEV | STDEVP
    | VAR | VARP | SUM }
    ( expression ) } [ ,...n ]
  [ BY expression [ ,...n ] ]
]
```

**Arguments**

AVG | COUNT | MAX | MIN | STDEV | STDEVP | VAR | VARP | SUM

Specifies the aggregation to be performed. These row aggregate functions are used with the COMPUTE clause.

<table>
<thead>
<tr>
<th>Row aggregate function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>Average of the values in the numeric expression</td>
</tr>
<tr>
<td>COUNT</td>
<td>Number of selected rows</td>
</tr>
<tr>
<td>MAX</td>
<td>Highest value in the expression</td>
</tr>
<tr>
<td>MIN</td>
<td>Lowest value in the expression</td>
</tr>
<tr>
<td>STDEV</td>
<td>Statistical standard deviation for all values in the expression</td>
</tr>
<tr>
<td>STDEVP</td>
<td>Statistical standard deviation for the population for all values in the expression</td>
</tr>
<tr>
<td>SUM</td>
<td>Total of the values in the numeric expression</td>
</tr>
<tr>
<td>VAR</td>
<td>Statistical variance for all values in the expression</td>
</tr>
<tr>
<td>VARP</td>
<td>Statistical variance for the population for all values in the expression</td>
</tr>
</tbody>
</table>

There is no equivalent to COUNT(*). To find the summary information produced by GROUP BY and COUNT(*), use a COMPUTE clause without BY.

These functions ignore null values.
The DISTINCT keyword is not allowed with row aggregate functions when they are specified with the COMPUTE clause.

When you add or average integer data, SQL Server treats the result as an `int` value, even if the data type of the column is `smallint` or `tinyint`. For more information about the return types of added or average data, see `SUM` and `AVG`.

**Note** To reduce the possibility of overflow errors in ODBC and DB-Library programs, make all variable declarations for the results of averages or sums the data type `int`.

( expression )

An expression, such as the name of a column on which the calculation is performed. `expression` must appear in the select list and must be specified exactly the same as one of the expressions in the select list. A column alias specified in the select list cannot be used within `expression`.

**Note** `ntext`, `text`, or `image` data types cannot be specified in a COMPUTE or COMPUTE BY clause.

**BY expression**

Generates control-breaks and subtotals in the result set. `expression` is an exact copy of an `order_by_expression` in the associated ORDER BY clause. Typically, this is a column name or column alias. Multiple expressions can be specified. Listing multiple expressions after BY breaks a group into subgroups and applies the aggregate function at each level of grouping.

If you use COMPUTE BY, you must also use an ORDER BY clause. The expressions must be identical to or a subset of those listed after ORDER BY, and must be in the same sequence. For example, if the ORDER BY clause is:

```
ORDER BY a, b, c
```

The COMPUTE clause can be any (or all) of these:

```
COMPUTE BY a, b, c
COMPUTE BY a, b
COMPUTE BY a
```

**Note** In a SELECT statement with a COMPUTE clause, the order of columns in the select list overrides the order of the aggregate functions in the COMPUTE clause. ODBC and DB-Library programmers must be aware of this order requirement to put the aggregate function results in the correct place.
You cannot use COMPUTE in a SELECT INTO statement because statements including COMPUTE generate tables and their summary results are not stored in the database. Therefore, any calculations produced by COMPUTE do not appear in the new table created with the SELECT INTO statement.

You cannot use the COMPUTE clause when the SELECT statement is part of a DECLARE CURSOR statement.
FOR Clause

FOR clause is used to specify either the BROWSE or the XML option (BROWSE and XML are unrelated options).

Syntax

```sql
[ FOR { BROWSE | XML { RAW | AUTO | EXPLICIT } }
   [ , XMLDATA ]
   [ , ELEMENTS ]
   [ , BINARY BASE64 ]
]
```

Arguments

BROWSE

Specifies that updates be allowed while viewing the data in a DB-Library browse mode cursor. A table can be browsed in an application if the table includes a time-stamped column (defined with the `timestamp` data type), the table has a unique index, and the FOR BROWSE option is at the end of the SELECT statement(s) sent to SQL Server. For more information, see [Browse Mode](#).

**Note** It is not possible to use the `<lock_hint>` HOLDLOCK in a SELECT statement that includes the FOR BROWSE option.

The FOR BROWSE option cannot appear in SELECT statements joined by the UNION operator.

XML

Specifies that the results of a query are to be returned as an XML document. One of these XML modes must be specified: RAW, AUTO, EXPLICIT. For more information about XML data and SQL Server, see [Retrieving XML Documents Using FOR XML](#).

RAW

Takes the query result and transforms each row in the result set into an XML element with a generic identifier `<row />` as the element tag. For more information, see [Using RAW Mode](#).

AUTO

Returns query results in a simple, nested XML tree. Each table in the FROM clause, for which at least one column is listed in the SELECT clause, is represented as an XML
element. The columns listed in the SELECT clause are mapped to the appropriate element attributes. For more information, see Using AUTO Mode.

EXPLICIT

Specifies that the shape of the resulting XML tree is defined explicitly. Using this mode, queries must be written in a particular way so that additional information about the desired nesting is specified explicitly. For more information, see Using EXPLICIT Mode.

XMLDATA

Returns the schema, but does not add the root element to the result. If XMLDATA is specified, it is appended to the document.

ELEMENTS

 Specifies that the columns are returned as subelements. Otherwise, they are mapped to XML attributes.

BINARY BASE64

Specifies that the query returns the binary data in binary base64-encoded format. In retrieving binary data using RAW and EXPLICIT mode, this option must be specified. This is the default in AUTO mode.
OPTION Clause

Specifies that the indicated query hint should be used throughout the entire query. Each query hint can be specified only once, although multiple query hints are permitted. Only one OPTION clause may be specified with the statement. The query hint affects all operators in the statement. If a UNION is involved in the main query, only the last query involving a UNION operator can have the OPTION clause. If one or more query hints causes the query optimizer to not generate a valid plan, error 8622 is produced.

Caution Because the query optimizer usually selects the best execution plan for a query, it is recommended that <join_hint>, <query_hint>, and <table_hint> be used only as a last resort by experienced database administrators.

Syntax

[ OPTION ( <query_hint> [ ,...n ] ) ]

<query_hint> ::= 
  { { HASH | ORDER } GROUP 
  | { CONCAT | HASH | MERGE } UNION 
  | { LOOP | MERGE | HASH } JOIN 
  | FAST number_rows 
  | FORCE ORDER 
  | MAXDOP number 
  | ROBUST PLAN 
  | KEEP PLAN 
  | KEEPFIXED PLAN 
  | EXPAND VIEWS 
  }

Arguments

{ HASH | ORDER } GROUP

Specifies that aggregations described in the GROUP BY, DISTINCT, or COMPUTE clause of the query should use hashing or ordering.

{ MERGE | HASH | CONCAT } UNION

Specifies that all UNION operations are performed by merging, hashing, or concatenating UNION sets. If more than one UNION hint is specified, the query optimizer selects the least expensive strategy from those hints specified.

{ LOOP | MERGE | HASH } JOIN
Specifies that all join operations are performed by loop join, merge join, or hash join in the whole query. If more than one join hint is specified, the optimizer selects the least expensive join strategy from the allowed ones.

If, in the same query, a join hint is also specified for a specific pair of tables, this join hint takes precedence in the joining of the two tables although the query hints still must be honored. Thus, the join hint for the pair of tables may only restrict the selection of allowed join methods in the query hint. See Hints for details.

**FAST number_rows**

Specifies that the query is optimized for fast retrieval of the first `number_rows` (a nonnegative integer). After the first `number_rows` are returned, the query continues execution and produces its full result set.

**FORCE ORDER**

Specifies that the join order indicated by the query syntax is preserved during query optimization.

**MAXDOP number**

Overrides the max degree of parallelism configuration option (of `sp_configure`) only for the query specifying this option. All semantic rules used with max degree of parallelism configuration option are applicable when using the MAXDOP query hint. For more information, see max degree of parallelism Option.

**ROBUST PLAN**

Forces the query optimizer to attempt a plan that works for the maximum potential row size, possibly at the expense of performance. When the query is processed, intermediate tables and operators may need to store and process rows that are wider than any of the input rows. The rows may be so wide that, in some cases, the particular operator cannot process the row. If this happens, SQL Server produces an error during query execution. By using ROBUST PLAN, you instruct the query optimizer not to consider any query plans that may encounter this problem.

**KEEP PLAN**

Forces the query optimizer to relax the estimated recompile threshold for a query. The estimated recompile threshold is the point at which a query is automatically recompiled when the estimated number of indexed column changes (update, delete, or insert) have been made to a table. Specifying KEEP PLAN ensures that a query will not be recompiled as frequently when there are multiple updates to a table.

**KEEPFIXED PLAN**
Forces the query optimizer not to recompile a query due to changes in statistics or to the indexed column (update, delete, or insert). Specifying KEEPFIXED PLAN ensures that a query will be recompiled only if the schema of the underlying tables is changed or sp_recompile is executed against those tables.

**EXPAND VIEWS**

Specifies that the indexed views are expanded and the query optimizer will not consider any indexed view as a substitute for any part of the query. (A view is expanded when the view name is replaced by the view definition in the query text.) This query hint virtually disallows direct use of indexed views and indexes on indexed views in the query plan.

The indexed view is not expanded only if the view is directly referenced in the SELECT part of the query and WITH (NOEXPAND) or WITH (NOEXPAND, INDEX( index_val [ ,...n ] ) ) is specified. For more information about the query hint WITH (NOEXPAND), see FROM.

Only the views in the SELECT portion of statements, including those in INSERT, UPDATE, and DELETE statements are affected by the hint.

**Remarks**

The order of the clauses in the SELECT statement is significant. Any of the optional clauses can be omitted, but when used, they must appear in the appropriate order.

SELECT statements are allowed in user-defined functions only if the select lists of these statements contain expressions that assign values to variables that are local to the functions.

A table variable, in its scope, may be accessed like a regular table and thus may be used as a table source in a SELECT statement.

A four-part name constructed with the OPENDATASOURCE function as the server-name part may be used as a table source in all places a table name can appear in SELECT statements.

Some syntax restrictions apply to SELECT statements involving remote tables. For information, see External Data and Transact-SQL.

The length returned for text or ntext columns included in the select list defaults to the smallest of the actual size of the text, the default TEXTSIZE session setting, or the hard-coded application limit. To change the length of returned text for the session, use the SET statement. By default, the limit on the length of text data returned with a SELECT statement is 4,000 bytes.
SQL Server raises exception 511 and rolls back the current executing statement if either of these occur:

- The SELECT statement produces a result row or an intermediate work table row exceeding 8,060 bytes.
- The DELETE, INSERT, or UPDATE statement attempts action on a row exceeding 8,060 bytes.

In SQL Server, an error occurs if no column name is given to a column created by a SELECT INTO or CREATE VIEW statement.

**Selecting Identity Columns**

When selecting an existing identity column into a new table, the new column inherits the IDENTITY property, unless one of the following conditions is true:

- The SELECT statement contains a join, GROUP BY clause, or aggregate function.
- Multiple SELECT statements are joined with UNION.
- The identity column is listed more than once in the select list.
- The identity column is part of an expression.

If any of these conditions is true, the column is created NOT NULL instead of inheriting the IDENTITY property. All rules and restrictions for the identity columns apply to the new table.

**Old-Style Outer Joins**

Earlier versions of SQL Server supported the definition of outer joins that used the *=` and =* operators in the WHERE clause. SQL Server version 7.0 supports the SQL-92 standard, which provides join operators in the FROM clause. It is recommended that queries be rewritten to use the SQL-92 syntax.

**Processing Order of WHERE, GROUP BY, and HAVING Clauses**

This list shows the processing order for a SELECT statement with a WHERE clause, a GROUP BY clause, and a HAVING clause:

1. The WHERE clause excludes rows not meeting its search condition.
2. The GROUP BY clause collects the selected rows into one group for each unique value in the GROUP BY clause.
3. Aggregate functions specified in the select list calculate summary values for each group.

4. The HAVING clause further excludes rows not meeting its search condition.

Permissions

SELECT permissions default to members of the sysadmin fixed server role, the db_owner and db_datareader fixed database roles, and the table owner. Members of the sysadmin, db_owner, and db_securityadmin roles, and the table owner can transfer permissions to other users.

If the INTO clause is used to create a permanent table, the user must have CREATE TABLE permission in the destination database.