12.19.1 Aggregate Function Descriptions

This section describes aggregate functions that operate on sets of values. They are often used with a `GROUP BY` clause to group values into subsets.

Table 12.29 Aggregate Functions

Name	Description
`AVG()`	Return the average value of the argument
`BIT_AND()`	Return bitwise AND
`BIT_OR()`	Return bitwise OR
`BIT_XOR()`	Return bitwise XOR
`COUNT()`	Return a count of the number of rows returned
`COUNT(DISTINCT)`	Return the count of a number of different values
`GROUP_CONCAT()`	Return a concatenated string
`JSON_ARRAYAGG()`	Return result set as a single JSON array
`JSON_OBJECTAGG()`	Return result set as a single JSON object
`MAX()`	Return the maximum value
`MIN()`	Return the minimum value
`STD()`	Return the population standard deviation
`STDDEV()`	Return the population standard deviation
`STDDEV_POP()`	Return the population standard deviation
`STDDEV_SAMP()`	Return the sample standard deviation
`SUM()`	Return the sum
`VAR_POP()`	Return the population standard variance
`VAR_SAMP()`	Return the sample variance
`VARIANCE()`	Return the population standard variance

Unless otherwise stated, aggregate functions ignore `NULL` values.

If you use an aggregate function in a statement containing no `GROUP BY` clause, it is equivalent to grouping on all rows. For more information, see Section 12.19.3, “MySQL Handling of GROUP BY”.

Most aggregate functions can be used as window functions. Those that can be used this way are signified in their syntax description by `[over_clause]`, representing an optional `OVER` clause. ***over_clause*** is described in Section 12.20.2, “Window Function Concepts and Syntax”, which also includes other information about window function usage.

For numeric arguments, the variance and standard deviation functions return a `DOUBLE` value. The `SUM()` and `AVG()` functions return a `DECIMAL` value for exact-value arguments (integer or `DECIMAL`), and a `DOUBLE` value for approximate-value arguments (`FLOAT` or `DOUBLE`).

The `SUM()` and `AVG()` aggregate functions do not work with temporal values. (They convert the values to numbers, losing everything after the first nonnumeric character.) To work around this problem, convert to numeric units, perform the aggregate operation, and convert back to a temporal value. Examples:

```
SELECT SEC_TO_TIME(SUM(TIME_TO_SEC(time_col))) FROM tbl_name;
SELECT FROM_DAYS(SUM(TO_DAYS(date_col))) FROM tbl_name;
```

Functions such as `SUM()` or `AVG()` that expect a numeric argument cast the argument to a number if necessary. For `SET` or `ENUM` values, the cast operation causes the underlying numeric value to be used.

The `BIT_AND()`, `BIT_OR()`, and `BIT_XOR()` aggregate functions perform bit operations. Prior to MySQL 8.0, bit functions and operators required `BIGINT` (64-bit integer) arguments and returned `BIGINT` values, so they had a maximum range of 64 bits. Non-`BIGINT` arguments were converted to `BIGINT` prior to performing the operation and truncation could occur.

In MySQL 8.0, bit functions and operators permit binary string type arguments (`BINARY`, `VARBINARY`, and the `BLOB` types) and return a value of like type, which enables them to take arguments and produce return values larger than 64 bits. For discussion about argument evaluation and result types for bit operations, see the introductory discussion in Section 12.12, “Bit Functions and Operators”.

- `AVG([DISTINCT] expr) [over_clause]`

  Returns the average value of ***expr***. The `DISTINCT` option can be used to return the average of the distinct values of ***expr***.

  If there are no matching rows, `AVG()` returns `NULL`. The function also returns `NULL` if ***expr*** is `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

```
mysql> SELECT student_name, AVG(test_score)
       FROM student
       GROUP BY student_name;
```

- `BIT_AND(`***`expr`***`) [`***`over_clause`***`]`

  Returns the bitwise `AND` of all bits in ***`expr`***.

  The result type depends on whether the function argument values are evaluated as binary strings or numbers:

  - Binary-string evaluation occurs when the argument values have a binary string type, and the argument is not a hexadecimal literal, bit literal, or `NULL` literal. Numeric evaluation occurs otherwise, with argument value conversion to unsigned 64-bit integers as necessary.
  - Binary-string evaluation produces a binary string of the same length as the argument values. If argument values have unequal lengths, an `ER_INVALID_BITWISE_OPERANDS_SIZE` error occurs. If the argument size exceeds 511 bytes, an `ER_INVALID_BITWISE_AGGREGATE_OPERANDS_SIZE` error occurs. Numeric evaluation produces an unsigned 64-bit integer.

  If there are no matching rows, `BIT_AND()` returns a neutral value (all bits set to 1) having the same length as the argument values.

  `NULL` values do not affect the result unless all values are `NULL`. In that case, the result is a neutral value having the same length as the argument values.

  For more information discussion about argument evaluation and result types, see the introductory discussion in Section 12.12, “Bit Functions and Operators”.

  If `BIT_AND()` is invoked from within the **mysql** client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

  As of MySQL 8.0.12, this function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

- `BIT_OR(`***`expr`***`) [`***`over_clause`***`]`

  Returns the bitwise `OR` of all bits in ***`expr`***.

  The result type depends on whether the function argument values are evaluated as binary strings or numbers:

- Binary-string evaluation occurs when the argument values have a binary string type, and the argument is not a hexadecimal literal, bit literal, or `NULL` literal. Numeric evaluation occurs otherwise, with argument value conversion to unsigned 64-bit integers as necessary.

- Binary-string evaluation produces a binary string of the same length as the argument values. If argument values have unequal lengths, an `ER_INVALID_BITWISE_OPERANDS_SIZE` error occurs. If the argument size exceeds 511 bytes, an `ER_INVALID_BITWISE_AGGREGATE_OPERANDS_SIZE` error occurs. Numeric evaluation produces an unsigned 64-bit integer.

If there are no matching rows, `BIT_OR()` returns a neutral value (all bits set to 0) having the same length as the argument values.

`NULL` values do not affect the result unless all values are `NULL`. In that case, the result is a neutral value having the same length as the argument values.

For more information discussion about argument evaluation and result types, see the introductory discussion in Section 12.12, “Bit Functions and Operators”.

If `BIT_OR()` is invoked from within the **mysql** client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

As of MySQL 8.0.12, this function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

- `BIT_XOR(`***expr***`) [`***over_clause***`]`

  Returns the bitwise `XOR` of all bits in ***expr***.

  The result type depends on whether the function argument values are evaluated as binary strings or numbers:

  - Binary-string evaluation occurs when the argument values have a binary string type, and the argument is not a hexadecimal literal, bit literal, or `NULL` literal. Numeric evaluation occurs otherwise, with argument value conversion to unsigned 64-bit integers as necessary.

  - Binary-string evaluation produces a binary string of the same length as the argument values. If argument values have unequal lengths, an `ER_INVALID_BITWISE_OPERANDS_SIZE` error occurs. If the argument size exceeds 511 bytes, an `ER_INVALID_BITWISE_AGGREGATE_OPERANDS_SIZE` error occurs. Numeric evaluation produces an unsigned 64-bit integer.

If there are no matching rows, `BIT_XOR()` returns a neutral value (all bits set to 0) having the same length as the argument values.

`NULL` values do not affect the result unless all values are `NULL`. In that case, the result is a neutral value having the same length as the argument values.

For more information discussion about argument evaluation and result types, see the introductory discussion in Section 12.12, “Bit Functions and Operators”.

If `BIT_XOR()` is invoked from within the **mysql** client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

As of MySQL 8.0.12, this function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

- `COUNT(`***expr***`) [`***over_clause***`]`

  Returns a count of the number of non-`NULL` values of ***expr*** in the rows retrieved by a `SELECT` statement. The result is a `BIGINT` value.

  If there are no matching rows, `COUNT()` returns `0`. `COUNT(NULL)` returns 0.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

```
mysql> SELECT student.student_name,COUNT(*)
       FROM student,course
       WHERE student.student_id=course.student_id
       GROUP BY student_name;
```

`COUNT(*)` is somewhat different in that it returns a count of the number of rows retrieved, whether or not they contain `NULL` values.

For transactional storage engines such as `InnoDB`, storing an exact row count is problematic. Multiple transactions may be occurring at the same time, each of which may affect the count.

`InnoDB` does not keep an internal count of rows in a table because concurrent transactions might “see” different numbers of rows at the same time. Consequently, `SELECT COUNT(*)` statements only count rows visible to the current transaction.

As of MySQL 8.0.13, `SELECT COUNT(*) FROM tbl_name` query performance for `InnoDB` tables is optimized for single-threaded workloads if there are no extra clauses such as `WHERE` or `GROUP BY`.

`InnoDB` processes `SELECT COUNT(*)` statements by traversing the smallest available secondary index unless an index or optimizer hint directs the optimizer to use a different index. If a secondary index is not present, `InnoDB` processes `SELECT COUNT(*)` statements by scanning the clustered index.

Processing `SELECT COUNT(*)` statements takes some time if index records are not entirely in the buffer pool. For a faster count, create a counter table and let your application update it according to the inserts and deletes it does. However, this method may not scale well in situations where thousands of concurrent transactions are initiating updates to the same counter table. If an approximate row count is sufficient, use `SHOW TABLE STATUS`.

`InnoDB` handles `SELECT COUNT(*)` and `SELECT COUNT(1)` operations in the same way. There is no performance difference.

For `MyISAM` tables, `COUNT(*)` is optimized to return very quickly if the `SELECT` retrieves from one table, no other columns are retrieved, and there is no `WHERE` clause. For example:

```
mysql> SELECT COUNT(*) FROM student;
```

This optimization only applies to `MyISAM` tables, because an exact row count is stored for this storage engine and can be accessed very quickly. `COUNT(1)` is only subject to the same optimization if the first column is defined as `NOT NULL`.

- `COUNT(DISTINCT expr,[expr...])`

  Returns a count of the number of rows with different non-`NULL` *`expr`* values.

  If there are no matching rows, `COUNT(DISTINCT)` returns `0`.

  ```
  mysql> SELECT COUNT(DISTINCT results) FROM student;
  ```

  In MySQL, you can obtain the number of distinct expression combinations that do not contain `NULL` by giving a list of expressions. In standard SQL, you would have to do a concatenation of all expressions inside `COUNT(DISTINCT ...)`.

- `GROUP_CONCAT(expr)`

This function returns a string result with the concatenated non-`NULL` values from a group. It returns `NULL` if there are no non-`NULL` values. The full syntax is as follows:

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
             [ORDER BY {unsigned_integer | col_name | expr}
                 [ASC | DESC] [,col_name ...]]
             [SEPARATOR str_val])
```

```
mysql> SELECT student_name,
         GROUP_CONCAT(test_score)
       FROM student
       GROUP BY student_name;
```

Or:

```
mysql> SELECT student_name,
         GROUP_CONCAT(DISTINCT test_score
                      ORDER BY test_score DESC SEPARATOR ' ')
       FROM student
       GROUP BY student_name;
```

In MySQL, you can get the concatenated values of expression combinations. To eliminate duplicate values, use the `DISTINCT` clause. To sort values in the result, use the `ORDER BY` clause. To sort in reverse order, add the `DESC` (descending) keyword to the name of the column you are sorting by in the `ORDER BY` clause. The default is ascending order; this may be specified explicitly using the `ASC` keyword. The default separator between values in a group is comma (`,`). To specify a separator explicitly, use `SEPARATOR` followed by the string literal value that should be inserted between group values. To eliminate the separator altogether, specify `SEPARATOR ''`.

The result is truncated to the maximum length that is given by the `group_concat_max_len` system variable, which has a default value of 1024. The value can be set higher, although the effective maximum length of the return value is constrained by the value of `max_allowed_packet`. The syntax to change the value of `group_concat_max_len` at runtime is as follows, where ***`val`*** is an unsigned integer:

```
SET [GLOBAL | SESSION] group_concat_max_len = val;
```

The return value is a nonbinary or binary string, depending on whether the arguments are nonbinary or binary strings. The result type is `TEXT` or `BLOB` unless `group_concat_max_len` is less than or equal to 512, in which case the result type is `VARCHAR` or `VARBINARY`.

If `GROUP_CONCAT()` is invoked from within the **mysql** client, binary string results display using hexadecimal notation, depending on the value of the `--binary-as-hex`. For more information about that option, see Section 4.5.1, "mysql — The MySQL Command-Line Client".

See also `CONCAT()` and `CONCAT_WS()`: Section 12.8, "String Functions and Operators".

- `JSON_ARRAYAGG(`***`col_or_expr`***`) [`***`over_clause`***`]`

  Aggregates a result set as a single `JSON` array whose elements consist of the rows. The order of elements in this array is undefined. The function acts on a column or an expression that evaluates to a single value. Returns `NULL` if the result contains no rows, or in the event of an error. If ***`col_or_expr`*** is `NULL`, the function returns an array of JSON `[null]` elements.

  As of MySQL 8.0.14, this function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 12.20.2, "Window Function Concepts and Syntax".

```
mysql> SELECT o_id, attribute, value FROM t3;
+------+-----------+-------+
| o_id | attribute | value |
+------+-----------+-------+
|    2 | color     | red   |
|    2 | fabric    | silk  |
|    3 | color     | green |
|    3 | shape     | square|
+------+-----------+-------+
4 rows in set (0.00 sec)

mysql> SELECT o_id, JSON_ARRAYAGG(attribute) AS attributes
    -> FROM t3 GROUP BY o_id;
+------+---------------------+
| o_id | attributes          |
+------+---------------------+
|    2 | ["color", "fabric"] |
|    3 | ["color", "shape"]  |
+------+---------------------+
2 rows in set (0.00 sec)
```

- `JSON_OBJECTAGG(`***`key`***`, `***`value`***`) [`***`over_clause`***`]`

Takes two column names or expressions as arguments, the first of these being used as a key and the second as a value, and returns a JSON object containing key-value pairs. Returns `NULL` if the result contains no rows, or in the event of an error. An error occurs if any key name is `NULL` or the number of arguments is not equal to 2.

As of MySQL 8.0.14, this function executes as a window function if ***`over_clause`*** is present. ***`over_clause`*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

```
mysql> SELECT o_id, attribute, value FROM t3;
+------+-----------+-------+
| o_id | attribute | value |
+------+-----------+-------+
|    2 | color     | red   |
|    2 | fabric    | silk  |
|    3 | color     | green |
|    3 | shape     | square|
+------+-----------+-------+
4 rows in set (0.00 sec)

mysql> SELECT o_id, JSON_OBJECTAGG(attribute, value)
    -> FROM t3 GROUP BY o_id;
+------+---------------------------------------+
| o_id | JSON_OBJECTAGG(attribute, value)      |
+------+---------------------------------------+
|    2 | {"color": "red", "fabric": "silk"}    |
|    3 | {"color": "green", "shape": "square"} |
+------+---------------------------------------+
2 rows in set (0.00 sec)
```

Duplicate key handling. When the result of this function is normalized, values having duplicate keys are discarded. In keeping with the MySQL `JSON` data type specification that does not permit duplicate keys, only the last value encountered is used with that key in the returned object (“last duplicate key wins”). This means that the result of using this function on columns from a `SELECT` can depend on the order in which the rows are returned, which is not guaranteed.

When used as a window function, if there are duplicate keys within a frame, only the last value for the key is present in the result. The value for the key from the last row in the frame is deterministic if the `ORDER BY` specification guarantees that the values have a specific order. If not, the resulting value of the key is nondeterministic.

Consider the following:

```
mysql> CREATE TABLE t(c VARCHAR(10), i INT);
Query OK, 0 rows affected (0.33 sec)

mysql> INSERT INTO t VALUES ('key', 3), ('key', 4), ('key', 5);
Query OK, 3 rows affected (0.10 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT c, i FROM t;
+------+------+
| c    | i    |
+------+------+
| key  |    3 |
| key  |    4 |
| key  |    5 |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT JSON_OBJECTAGG(c, i) FROM t;
+----------------------+
| JSON_OBJECTAGG(c, i) |
+----------------------+
| {"key": 5}           |
+----------------------+
1 row in set (0.00 sec)

mysql> DELETE FROM t;
Query OK, 3 rows affected (0.08 sec)

mysql> INSERT INTO t VALUES ('key', 3), ('key', 5), ('key', 4);
Query OK, 3 rows affected (0.06 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT c, i FROM t;
+------+------+
| c    | i    |
+------+------+
| key  |    3 |
| key  |    5 |
| key  |    4 |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT JSON_OBJECTAGG(c, i) FROM t;
+----------------------+
| JSON_OBJECTAGG(c, i) |
+----------------------+
| {"key": 4}           |
```

```
+----------------------+
1 row in set (0.00 sec)
```

The key chosen from the last query is nondeterministic. If the query does not use `GROUP BY` (which usually imposes its own ordering regardless) and you prefer a particular key ordering, you can invoke `JSON_OBJECTAGG()` as a window function by including an `OVER` clause with an `ORDER BY` specification to impose a particular order on frame rows. The following examples show what happens with and without `ORDER BY` for a few different frame specifications.

Without `ORDER BY`, the frame is the entire partition:

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER () AS json_object FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 4}  |
| {"key": 4}  |
| {"key": 4}  |
+-------------+
```

With `ORDER BY`, where the frame is the default of `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` (in both ascending and descending order):

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i) AS json_object FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 3}  |
| {"key": 4}  |
| {"key": 5}  |
+-------------+
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i DESC) AS json_object FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 5}  |
| {"key": 4}  |
| {"key": 3}  |
+-------------+
```

With `ORDER BY` and an explicit frame of the entire partition:

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i
            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
        AS json_object
       FROM t;
+-------------+
| json_object |
+-------------+
| {"key": 5}  |
| {"key": 5}  |
| {"key": 5}  |
+-------------+
```

To return a particular key value (such as the smallest or largest), include a `LIMIT` clause in the appropriate query. For example:

```
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i) AS json_object FROM t LIMIT 1;
+-------------+
| json_object |
+-------------+
| {"key": 3}  |
+-------------+
mysql> SELECT JSON_OBJECTAGG(c, i)
       OVER (ORDER BY i DESC) AS json_object FROM t LIMIT 1;
+-------------+
| json_object |
+-------------+
| {"key": 5}  |
+-------------+
```

See Normalization, Merging, and Autowrapping of JSON Values, for additional information and examples.

- `MAX([DISTINCT] expr) [over_clause]`

  Returns the maximum value of *`expr`*. `MAX()` may take a string argument; in such cases, it returns the maximum string value. See Section 8.3.1, “How MySQL Uses Indexes”. The `DISTINCT` keyword can be used to find the maximum of the distinct values of *`expr`*, however, this produces the same result as omitting `DISTINCT`.

If there are no matching rows, or if ***expr*** is `NULL`, `MAX()` returns `NULL`.

This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

```
mysql> SELECT student_name, MIN(test_score), MAX(test_score)
       FROM student
       GROUP BY student_name;
```

For `MAX()`, MySQL currently compares `ENUM` and `SET` columns by their string value rather than by the string's relative position in the set. This differs from how `ORDER BY` compares them.

- `MIN([DISTINCT] ***expr***) [***over_clause***]`

  Returns the minimum value of ***expr***. `MIN()` may take a string argument; in such cases, it returns the minimum string value. See Section 8.3.1, “How MySQL Uses Indexes”. The `DISTINCT` keyword can be used to find the minimum of the distinct values of ***expr***, however, this produces the same result as omitting `DISTINCT`.

  If there are no matching rows, or if ***expr*** is `NULL`, `MIN()` returns `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

  ```
  mysql> SELECT student_name, MIN(test_score), MAX(test_score)
         FROM student
         GROUP BY student_name;
  ```

  For `MIN()`, MySQL currently compares `ENUM` and `SET` columns by their string value rather than by the string's relative position in the set. This differs from how `ORDER BY` compares them.

- `STD(***expr***) [***over_clause***]`

  Returns the population standard deviation of ***expr***. `STD()` is a synonym for the standard SQL function `STDDEV_POP()`, provided as a MySQL extension.

  If there are no matching rows, or if ***expr*** is `NULL`, `STD()` returns `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

- `STDDEV(expr) [over_clause]`

  Returns the population standard deviation of *`expr`*. `STDDEV()` is a synonym for the standard SQL function `STDDEV_POP()`, provided for compatibility with Oracle.

  If there are no matching rows, or if *`expr`* is `NULL`, `STDDEV()` returns `NULL`.

  This function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

- `STDDEV_POP(expr) [over_clause]`

  Returns the population standard deviation of *`expr`* (the square root of `VAR_POP()`). You can also use `STD()` or `STDDEV()`, which are equivalent but not standard SQL.

  If there are no matching rows, or if *`expr`* is `NULL`, `STDDEV_POP()` returns `NULL`.

  This function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

- `STDDEV_SAMP(expr) [over_clause]`

  Returns the sample standard deviation of *`expr`* (the square root of `VAR_SAMP()`.

  If there are no matching rows, or if *`expr`* is `NULL`, `STDDEV_SAMP()` returns `NULL`.

  This function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

- `SUM([DISTINCT] expr) [over_clause]`

  Returns the sum of *`expr`*. If the return set has no rows, `SUM()` returns `NULL`. The `DISTINCT` keyword can be used to sum only the distinct values of *`expr`*.

  If there are no matching rows, or if *`expr`* is `NULL`, `SUM()` returns `NULL`.

  This function executes as a window function if *`over_clause`* is present. *`over_clause`* is as described in Section 12.20.2, “Window Function Concepts and Syntax”; it cannot be used with `DISTINCT`.

- `VAR_POP(expr) [over_clause]`

  Returns the population standard variance of *`expr`*. It considers rows as the whole population, not as a sample, so it has the number of rows as the denominator. You can also use `VARIANCE()`, which is

equivalent but is not standard SQL.

If there are no matching rows, or if ***expr*** is `NULL`, `VAR_POP()` returns `NULL`.

This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

- `VAR_SAMP(`***expr***`) [`***over_clause***`]`

  Returns the sample variance of ***expr***. That is, the denominator is the number of rows minus one.

  If there are no matching rows, or if ***expr*** is `NULL`, `VAR_SAMP()` returns `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”.

- `VARIANCE(`***expr***`) [`***over_clause***`]`

  Returns the population standard variance of ***expr***. `VARIANCE()` is a synonym for the standard SQL function `VAR_POP()`, provided as a MySQL extension.

  If there are no matching rows, or if ***expr*** is `NULL`, `VARIANCE()` returns `NULL`.

  This function executes as a window function if ***over_clause*** is present. ***over_clause*** is as described in Section 12.20.2, “Window Function Concepts and Syntax”.