Join query in sql

MySQL supports the following JOIN syntax for the table_references part of SELECT statements and multiple-table DELETE and UPDATE statements: table_references: escaped_table_reference [, escaped_table_reference] ... escaped_table_reference: { table_reference | { OJ table_reference } } table_reference: { table_factor | joined_table } table_factor: { tbl_name [PARTITION (partition_names)] [[AS] alias] [index_hint_list] | [LATERAL] table_subquery [AS] alias [(col_list)] | (table_references) } joined_table: { table_reference {[INNER | CROSS] JOIN | STRAIGHT_JOIN} table_factor [join_specification] | table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_specification | table_reference NATURAL [INNER | {LEFT|RIGHT} [OUTER]] JOIN table_factor } join_specification: { ON search_condition | USING (join_column_list) } join_column_list: column_name [, column_name] ... index_hint_list: index_hint [, index_hint] ... index_hint: { USE {INDEX|KEY} [FOR {JOIN|ORDER BY|GROUP BY}] ([index_list]) | {IGNORE|FORCE} {INDEX|KEY} [FOR {JOIN|ORDER BY|GROUP BY}] (index_list) } index_list: index_name [, index_name] ... A table reference is also known as a join expression. A table reference (when it refers to a partitioned table) may contain a PARTITION clause, including a list of comma-separated partitions, subpartitions, or both. This option follows the name of the table and precedes any alias declaration. The effect of this option is that rows are selected only from the listed partitions or subpartitions. Any partitions or subpartitions not named in the list are ignored. For more information and examples, see Section 24.5, "Partition Selection". The syntax of table_factor is extended in MySQL in comparison with standard SQL. The standard accepts only table_reference, not a list of them inside a pair of parentheses. This is a conservative extension if each comma in a list of table_reference items is considered as equivalent to an inner join. For example: SELECT * FROM t1 LEFT JOIN (t2, t3, t4) ON (t2.a = t1.a AND t3.b = t1.b AND t4.c = t1.c) is equivalent to: SELECT * FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN t4) ON (t2.a = t1.a AND t3.b = t1.b AND t4.c = t1.c) In MySQL, JOIN, CROSS JOIN, and INNER JOIN are syntactic equivalents (they can replace each other). In standard SQL, they are not equivalent. INNER JOIN is used with an ON clause, CROSS JOIN is used otherwise. In general, parentheses can be ignored in join expressions containing only inner join operations. MySQL also supports nested joins. See Section 8.2.1.8, "Nested Join Optimization". Index hints can be specified to affect how the MySQL optimizer makes use of indexes. For more information, see Section 8.9.4, "Index Hints". Optimizer hints and the optimizer_switch system variable are other ways to influence optimizer use of indexes. See Section 8.9.3, "Optimizer Hints", and Section 8.9.2, "Switchable Optimizations". The following list describes general factors to take into account when writing joins: A table reference can be aliased using tbl_name AS alias_name or tbl_name alias_name: SELECT t1.name, t2.salary FROM employee AS t1 INNER JOIN info AS t2 ON t1.name = t2.name; SELECT t1.name, t2.salary FROM employee t1 INNER JOIN info t2 ON t1.name = t2.name; A table_subquery is also known as a derived table or subquery in the FROM clause. See Section 13.2.11.8, "Derived Tables". Such subqueries must include an alias to give the subquery result a table name, and may optionally include a list of table column names in parentheses. A trivial example follows: SELECT * FROM (SELECT 1, 2, 3) AS t1; The maximum number of tables that can be referenced in a single join is 61. This includes a join handled by merging derived tables and views in the FROM clause into the outer query block (see Section 8.2.2.4, "Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization"). INNER JOIN and , (comma) are semantically equivalent in the absence of a join condition: both produce a Cartesian product between the specified tables (that is, each and every row in the first table is joined to each and every row in the second table). However, the precedence of the comma operator is less than that of INNER JOIN, CROSS JOIN, LEFT JOIN, and so on. If you mix comma joins with the other join types when there is a join condition, an error of the form Unknown column 'col_name' in 'on clause' may occur. Information about dealing with this problem is given later in this section. The search_condition used with ON is any conditional expression of the form that can be used in a WHERE clause. Generally, the ON clause serves for conditions that specify how to join tables, and the WHERE clause restricts which rows to include in the result set. If there is no matching row for the right table in the ON or USING part in a LEFT JOIN, a row with all columns set to NULL is used for the right table. You can use this fact to find rows in a table that have no counterpart in another table: SELECT left_tbl.* FROM left_tbl LEFT JOIN right_tbl ON left_tbl.id = right_tbl.id WHERE right_tbl.id IS NULL; This example finds all rows in left_tbl with an id value that is not present in right_tbl (that is, all rows in left_tbl with no corresponding row in right_tbl). See Section 8.2.1.9, "Outer Join Optimization". The USING(join_column_list) clause names a list of columns that must exist in both tables. If tables a and b both contain columns c1, c2, and c3, the following join compares corresponding columns from the two tables: a LEFT JOIN b USING (c1, c2, c3) The NATURAL [LEFT] JOIN of two tables is defined to be semantically equivalent to an INNER JOIN or a LEFT JOIN with a USING clause that names all columns that exist in both tables. RIGHT JOIN works analogously to LEFT JOIN. To keep code portable across databases, it is recommended that you use LEFT JOIN instead of RIGHT JOIN. The { OJ ... } syntax shown in the join syntax description exists only for compatibility with ODBC. The curly braces in the syntax should be written literally; they are not metasyntax as used elsewhere in syntax descriptions. SELECT left_tbl.* FROM { OJ left_tbl LEFT OUTER JOIN right_tbl ON left_tbl.id = right_tbl.id } WHERE right_tbl.id IS NULL; You can use other types of joins within { OJ ... }, such as INNER JOIN or RIGHT OUTER JOIN. This helps with compatibility with some third-party applications, but is not official ODBC syntax. STRAIGHT_JOIN is similar to JOIN, except that the left table is always read before the right table. This can be used for those (few) cases for which the join optimizer processes the tables in a suboptimal order. Some join examples: SELECT * FROM table1, table2; SELECT * FROM table1 INNER JOIN table2 ON table1.id = table2.id; SELECT * FROM table1 LEFT JOIN table2 ON table1.id = table2.id; SELECT * FROM table1 LEFT JOIN table2 USING (id); SELECT * FROM table1 LEFT JOIN table2 ON table1.id = table2.id LEFT JOIN table3 ON table2.id = table3.id; Natural joins and joins with USING, including outer join variants, are processed according to the SQL:2003 standard: Redundant columns of a NATURAL join do not appear. Consider this set of statements: CREATE TABLE t1 (i INT, j INT); CREATE TABLE t2 (k INT, j INT); INSERT INTO t1 VALUES(1, 1); INSERT INTO t2 VALUES(1, 1); SELECT * FROM t1 NATURAL JOIN t2; SELECT * FROM t1 JOIN t2 USING (j); In the first SELECT statement, column j appears in both tables and thus becomes a join column, so, according to standard SQL, it should appear only once in the output, not twice. Similarly, in the second SELECT statement, column j is named in the USING clause and should appear only once in the output, not twice. Thus, the statements produce this output: +------+------+------+ | j | i | k | +------+------+------+ | 1 | 1 | 1 | +------+------+------+ +------+------+------+ | j | i | k | +------+------+------+ | 1 | 1 | 1 | +------+------+------+ Redundant column elimination and column ordering occurs according to standard SQL, producing this display order: First, coalesced common columns of the two joined tables, in the order in which they occur in the first table Second, columns unique to the first table, in order in which they occur in that table Third, columns unique to the second table, in order in which they occur in that table The single result column that replaces two common columns is defined using the coalesce operation. That is, for two t1.a and t2.a the resulting single join column a is defined as a = COALESCE(t1.a, t2.a), where: COALESCE(x, y) = (CASE WHEN x IS NOT NULL THEN x ELSE y END) If the join operation is any other join, the result columns of the join consist of the concatenation of all columns of the joined tables. A consequence of the definition of coalesced columns is that, for outer joins, the coalesced column contains the value of the non-NULL column if one of the two columns is always NULL. If neither or both columns are NULL, both common columns have the same value, so it doesn't matter which one is chosen as the value of the coalesced column. A simple way to interpret this is to consider that a coalesced column of an outer join is represented by the common column of the inner table of a JOIN. Suppose that the tables t1(a, b) and t2(a, c) have the following contents: t1 t2 ---- ---- 1 x 2 z 2 y 3 w Then, for this join, column a contains the values of t1.a: mysql> SELECT * FROM t1 NATURAL LEFT JOIN t2; +------+------+------+ | a | b | c | +------+------+------+ | 1 | x | NULL | | 2 | y | z | +------+------+------+ By contrast, for this join, column a contains the values of t2.a. mysql> SELECT * FROM t1 NATURAL RIGHT JOIN t2; +------+------+------+ | a | c | b | +------+------+------+ | 2 | z | y | | 3 | w | NULL | +------+------+------+ Compare those results to the otherwise equivalent queries with JOIN ... ON: mysql> SELECT * FROM t1 LEFT JOIN t2 ON (t1.a = t2.a); +------+------+------+------+ | a | b | a | c | +------+------+------+------+ | 1 | x | NULL | NULL | | 2 | y | 2 | z | +------+------+------+------+ mysql> SELECT * FROM t1 RIGHT JOIN t2 ON (t1.a = t2.a); +------+------+------+------+ | a | b | a | c | +------+------+------+------+ | 2 | y | 2 | z | | NULL | NULL | 3 | w | +------+------+------+------+ A USING clause can be rewritten as an ON clause that compares corresponding columns. However, although USING and ON are similar, they are not quite the same. Consider the following two queries: a LEFT JOIN b USING (c1, c2, c3) a LEFT JOIN b ON a.c1 = b.c1 AND a.c2 = b.c2 AND a.c3 = b.c3 With respect to determining which rows satisfy the join condition, both joins are semantically identical. With respect to determining which columns to display for SELECT * expansion, the two joins are not semantically identical. The USING join selects the coalesced value of corresponding columns, whereas the ON join selects all columns from all tables. For the USING join, SELECT * selects these values: COALESCE(a.c1, b.c1), COALESCE(a.c2, b.c2), COALESCE(a.c3, b.c3) For the ON join, SELECT * selects these values: a.c1, a.c2, a.c3, b.c1, b.c2, b.c3 With an inner join, COALESCE(a.c1, b.c1) is the same as either a.c1 or b.c1 because both columns have the same value. With an outer join (such as LEFT JOIN), one of the two columns can be NULL. That column is omitted from the result. An ON clause can refer only to its operands. Example: CREATE TABLE t1 (i1 INT); CREATE TABLE t2 (i2 INT); CREATE TABLE t3 (i3 INT); SELECT * FROM t1 JOIN t2 ON (i1 = i3) JOIN t3; The statement fails with an Unknown column 'i3' in 'on clause' error because i3 is a column in t3, which is not an operand of the ON clause. To enable the join to be processed, rewrite the statement as follows: SELECT * FROM t1 JOIN t2 JOIN t3 ON (i1 = i3); JOIN has higher precedence than the comma operator (,), so the join expression t1, t2 JOIN t3 is interpreted as (t1, (t2 JOIN t3)), not as ((t1, t2) JOIN t3). This affects statements that use an ON clause because that clause can refer only to columns in the operands of the join, and the precedence affects interpretation of what those operands are. Example: CREATE TABLE t1 (i1 INT, j1 INT); CREATE TABLE t2 (i2 INT, j2 INT); CREATE TABLE t3 (i3 INT, j3 INT); INSERT INTO t1 VALUES(1, 1); INSERT INTO t2 VALUES(1, 1); INSERT INTO t3 VALUES(1, 1); SELECT * FROM t1, t2 JOIN t3 ON (t1.i1 = t3.i3); The JOIN takes precedence over the comma operator, so the operands for the ON clause are t2 and t3. Because t1.i1 is not a column in either of the operands, the result is an Unknown column 't1.i1' in 'on clause' error. To enable the join to be processed, use either of these strategies: Group the first two tables explicitly with parentheses so that the operands for the ON clause are (t1, t2) and t3: SELECT * FROM (t1, t2) JOIN t3 ON (t1.i1 = t3.i3); Avoid the use of the comma operator and use JOIN instead: SELECT * FROM t1 JOIN t2 JOIN t3 ON (t1.i1 = t3.i3); The same precedence interpretation also applies to statements that mix the comma operator with INNER JOIN, CROSS JOIN, LEFT JOIN, and RIGHT JOIN, all of which have higher precedence than the comma operator. A MySQL extension compared to the SQL:2003 standard is that MySQL permits you to qualify the common (coalesced) columns of NATURAL or USING joins, whereas the standard disallows that. Page 2 SELECT ... UNION [ALL | DISTINCT] SELECT ... [UNION [ALL | DISTINCT] SELECT ...] UNION combines the result from multiple SELECT statements into a single result set. Example: mysql> SELECT 1, 2; +---+---+ | 1 | 2 | +---+---+ | 1 | 2 | +---+---+ mysql> SELECT 'a', 'b'; +---+---+ | a | b | +---+---+ | a | b | +---+---+ mysql> SELECT 1, 2 UNION SELECT 'a', 'b'; +---+---+ | 1 | 2 | +---+---+ | 1 | 2 | | a | b | +---+---+ The column names for a UNION result set are taken from the column names of the first SELECT statement. Selected columns listed in corresponding positions of each SELECT statement should have the same data type. For example, the first column selected by the first statement should have the same type as the first column selected by the other statements. If the data types of corresponding SELECT columns do not match, the types and lengths of the columns in the UNION result take into account the values retrieved by all the SELECT statements. For example, consider the following, where the column length is not constrained to the length of the value from the first SELECT: mysql> SELECT REPEAT('a',1) UNION SELECT REPEAT('b',20); +----------------------+ | REPEAT('a',1) | +----------------------+ | a | | bbbbbbbbbbbbbbbbbbbb | +----------------------+ TABLE in Unions Beginning with MySQL 8.0.19, you can also use a TABLE statement or VALUES statement in a UNION wherever you can employ the equivalent SELECT statement. Assume that tables t1 and t2 are created and populated as shown here: CREATE TABLE t1 (x INT, y INT); INSERT INTO t1 VALUES ROW(4,-2),ROW(5,9); CREATE TABLE t2 (a INT, b INT); INSERT INTO t2 VALUES ROW(1,2),ROW(3,4); The preceding being the case, and disregarding the column names in the output of the queries beginning with VALUES, all of the following UNION queries yield the same result: SELECT * FROM t1 UNION SELECT * FROM t2; TABLE t1 UNION SELECT * FROM t2; VALUES ROW(4,-2), ROW(5,9) UNION SELECT * FROM t2; SELECT * FROM t1 UNION TABLE t2; TABLE t1 UNION TABLE t2; VALUES ROW(4,-2), ROW(5,9) UNION TABLE t2; SELECT * FROM t1 UNION VALUES ROW(4,-2),ROW(5,9); TABLE t1 UNION VALUES ROW(4,-2),ROW(5,9); VALUES ROW(4,-2), ROW(5,9) UNION VALUES ROW(4,-2),ROW(5,9); To force the column names to be the same, wrap the VALUES on the left hand side in a SELECT and use aliases, like this: SELECT * FROM (VALUES ROW(4,-2), ROW(5, 9)) AS t(x,y) UNION TABLE t2; SELECT * FROM (VALUES ROW(4,-2), ROW(5,9)) AS t(x,y) UNION VALUES ROW(4,-2),ROW(5,9); UNION DISTINCT and UNION ALL By default, duplicate rows are removed from UNION results. The optional DISTINCT keyword has the same effect but makes it explicit. With the optional ALL keyword, duplicate-row removal does not occur and the result includes all matching rows from all the SELECT statements. You can mix UNION ALL and UNION DISTINCT in the same query. Mixed UNION types are treated such that a DISTINCT union overrides any ALL union to its left. A DISTINCT union can be produced explicitly by using UNION DISTINCT or implicitly by using UNION with no following DISTINCT or ALL keyword. In MySQL 8.0.19 and later, UNION ALL and UNION DISTINCT work the same way when one or more TABLE statements are used in the union. ORDER BY and LIMIT in Unions To apply an ORDER BY or LIMIT clause to an individual SELECT, parenthesize the SELECT and place the clause inside the parentheses: (SELECT a FROM t1 WHERE a=10 AND B=1 ORDER BY a LIMIT 10) UNION (SELECT a FROM t2 WHERE a=11 AND B=2 ORDER BY a LIMIT 10); Use of ORDER BY for individual SELECT statements implies nothing about the order in which the rows appear in the final result because UNION by default produces an unordered set of rows. Therefore, ORDER BY in this context typically is used in conjunction with LIMIT, to determine the subset of the selected rows to retrieve for the SELECT, even though it does not necessarily affect the order of those rows in the final UNION result. If ORDER BY appears without LIMIT in a SELECT, it is optimized away because it has no effect in any case. To use an ORDER BY or LIMIT clause to sort or limit the entire UNION result, parenthesize the individual SELECT statements and place the ORDER BY or LIMIT after the last one: (SELECT a FROM t1 WHERE a=10 AND B=1) UNION (SELECT a FROM t2 WHERE a=11 AND B=2) ORDER BY a LIMIT 10; A statement without parentheses is equivalent to one parenthesized as just shown. Beginning with MySQL 8.0.19, you can use ORDER BY and LIMIT with TABLE in unions in the same way as just shown, bearing in mind that TABLE does not support a WHERE clause. This kind of ORDER BY cannot use column references that include a table name (that is, names in tbl_name.col_name format). Instead, provide a column alias in the first SELECT statement and refer to the alias in the ORDER BY. (Alternatively, refer to the column in the ORDER BY using its column position. However, use of column positions is deprecated.) Also, if a column to be sorted is aliased, the ORDER BY clause must refer to the alias, not the column name. The first of the following statements is permitted, but the second fails with an Unknown column 'a' in 'order clause' error: (SELECT a AS b FROM t) UNION (SELECT ...) ORDER BY b; (SELECT a AS b FROM t) UNION (SELECT ...) ORDER BY a; To cause rows in a UNION result to consist of the sets of rows retrieved by each SELECT one after the other, select an additional column in each SELECT to use as a sort column and add an ORDER BY that sorts on that column following the last SELECT: (SELECT 1 AS sort_col, col1a, col1b, ... FROM t1) UNION (SELECT 2, col2a, col2b, ... FROM t2) ORDER BY sort_col; To additionally maintain sort order within individual SELECT results, add a secondary column to the ORDER BY clause: (SELECT 1 AS sort_col, col1a, col1b, ... FROM t1) UNION (SELECT 2, col2a, col2b, ... FROM t2) ORDER BY sort_col, col1a; Use of an additional column also enables you to determine which SELECT each row comes from. Extra columns can provide other identifying information as well, such as a string that indicates a table name. UNION Restrictions In a UNION, the SELECT statements are normal select statements, but with the following restrictions: HIGH_PRIORITY in the first SELECT has no effect. HIGH_PRIORITY in any subsequent SELECT produces a syntax error. Only the last SELECT statement can use an INTO clause. However, the entire UNION result is written to the INTO output destination. As of MySQL 8.0.20, these two UNION variants containing INTO are deprecated and you should expect support for them to be removed in a future version of MySQL: In the trailing query block of a query expression, use of INTO before FROM produces a warning. Example: ... UNION SELECT * INTO OUTFILE 'file_name' FROM table_name; In a parenthesized trailing block of a query expression, use of INTO (regardless of its position relative to FROM) produces a warning. Example: ... UNION (SELECT * INTO OUTFILE 'file_name' FROM table_name); Those variants are deprecated because they are confusing, as if they collect information from the named table rather than the entire query expression (the UNION). UNION queries with an aggregate function in an ORDER BY clause are rejected with an ER_AGGREGATE_ORDER_FOR_UNION error. Example: SELECT 1 AS foo UNION SELECT 2 ORDER BY MAX(1); UNION Handling in MySQL 8.0 Compared to MySQL 5.7 In MySQL 8.0, the parser rules for SELECT and UNION were refactored to be more consistent (the same SELECT syntax applies uniformly in each such context) and reduce duplication. Compared to MySQL 5.7, several user-visible effects resulted from this work, which may require rewriting of certain statements: NATURAL JOIN permits an optional INNER keyword (NATURAL INNER JOIN), in compliance with standard SQL. Right-deep joins without parentheses are permitted (for example, ... JOIN ... JOIN ... ON ... ON), in compliance with standard SQL. STRAIGHT_JOIN now permits a USING clause, similar to other inner joins. The parser accepts parentheses around query expressions. For example, (SELECT ... UNION SELECT ...) is permitted. See also Section 13.2.10.4, "Parenthesized Query Expressions". The parser better conforms to the documented permitted placement of the SQL_CACHE and SQL_NO_CACHE query modifiers. Left-hand nesting of unions, previously permitted only in subqueries, is now permitted in top-level statements. For example, this statement is now accepted as valid: (SELECT 1 UNION SELECT 1) UNION SELECT 1; Locking clauses (FOR UPDATE, LOCK IN SHARE MODE) are allowed only in non-UNION queries. This means that parentheses must be used for SELECT statements containing locking clauses. This statement is no longer accepted as valid: SELECT 1 FOR UPDATE UNION SELECT 1 FOR UPDATE; Instead, write the statement like this: (SELECT 1 FOR UPDATE) UNION (SELECT 1 FOR UPDATE); Page 3 13.2.10.4 Parenthesized Query Expressions parenthesized_query_expression: (query_expression [order_by_clause] [limit_clause]) [order_by_clause] [limit_clause] [into_clause] query_expression: query_block [UNION query_block [UNION query_block ...]] [order_by_clause] [limit_clause] [into_clause] query_block: SELECT ... (see Section 13.2.10, "SELECT Statement") order_by_clause: ORDER BY as for SELECT (see Section 13.2.10, "SELECT Statement") limit_clause: LIMIT as for SELECT (see Section 13.2.10, "SELECT Statement") into_clause: INTO as for SELECT (see Section 13.2.10, "SELECT Statement") MySQL 8.0.22 and higher supports parenthesized query expressions according to the preceding syntax. At its simplest, a parenthesized query expression contains a single SELECT and no following optional clauses: (SELECT 1); (SELECT * FROM INFORMATION_SCHEMA.SCHEMATA WHERE SCHEMA_NAME = 'mysql'); A parenthesized query expression can also contain a UNION comprising multiple SELECT statements, and end with any or all of the optional clauses: mysql> (SELECT 1 AS result UNION SELECT 2); +--------+ | result | +--------+ | 1 | | 2 | +--------+ mysql> (SELECT 1 AS result UNION SELECT 2) LIMIT 1; +--------+ | result | +--------+ | 1 | +--------+ mysql> (SELECT 1 AS result UNION SELECT 2) LIMIT 1 OFFSET 1; +--------+ | result | +--------+ | 2 | +--------+ mysql> (SELECT 1 AS result UNION SELECT 2) ORDER BY result DESC LIMIT 1; +--------+ | result | +--------+ | 2 | +--------+ mysql> (SELECT 1 AS result UNION SELECT 2) ORDER BY result DESC LIMIT 1 OFFSET 1; +--------+ | result | +--------+ | 1 | +--------+ mysql> (SELECT 1 AS result UNION SELECT 3 UNION SELECT 2) ORDER BY result LIMIT 1 OFFSET 1 INTO @var; mysql> SELECT @var; +------+ | @var | +------+ | 2 | +------+ Parenthesized query expressions are also used as query expressions, so a query expression, usually composed of query blocks, may also consist of parenthesized query expressions: (SELECT * FROM t1 ORDER BY a) UNION (SELECT * FROM t2 ORDER BY b) ORDER BY z; Query blocks may have trailing ORDER BY and LIMIT clauses, which are applied before the outer UNION and ORDER BY and LIMIT. You cannot have a query block with a trailing ORDER BY or LIMIT, without wrapping it in parentheses, but parentheses may be used for enforcement in various ways: To enforce LIMIT on each query block: (SELECT 1 LIMIT 1) UNION (SELECT 2 LIMIT 1); To enforce LIMIT on both query blocks and the entire query expression: (SELECT 1 LIMIT 1) UNION (SELECT 2 LIMIT 1) LIMIT 1; To enforce LIMIT on the entire query expression (with no parentheses): SELECT 1 UNION SELECT 2 LIMIT 1; Hybrid enforcement: LIMIT on the first query block and on the entire query expression: (SELECT 1 LIMIT 1) UNION SELECT 2 LIMIT 1; The syntax described in this section is subject to certain restrictions: If ORDER BY occurs within a parenthesized query expression and also is applied in the outer query, the results are undefined and may change in a future version of MySQL. The same is true if LIMIT occurs within a parenthesized query expression and also is applied in the outer query. A trailing INTO clause for a query expression is not permitted if there is another INTO clause inside parentheses. Parenthesized query expressions do not permit multiple levels of ORDER BY or LIMIT operations. For example: mysql> (SELECT 'a' UNION SELECT 'b' LIMIT 1) LIMIT 2; ERROR 1235 (42000): This version of MySQL doesn't yet support 'parenthesized query expression with more than one external level of ORDER/LIMIT operations' Page 4 A subquery is a SELECT statement within another statement. All subquery forms and operations that the SQL standard requires are supported, as well as a few features that are MySQL-specific. Here is an example of a subquery: SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2); In this example, SELECT * FROM t1 ... is the outer query (or outer statement), and (SELECT column1 FROM t2) is the subquery. We say that the subquery is nested within the outer query, and in fact it is possible to nest subqueries within other subqueries, to a considerable depth. A subquery must always appear within parentheses. The main advantages of subqueries are: They allow queries that are structured so that it is possible to isolate each part of a statement. They provide alternative ways to perform operations that would otherwise require complex joins and unions. Many people find subqueries more readable than complex joins or unions. Indeed, it was the innovation of subqueries that gave people the original idea of calling the early SQL "Structured Query Language." Here is an example statement that shows the major points about subquery syntax as specified by the SQL standard and supported in MySQL: DELETE FROM t1 WHERE s11 > ANY (SELECT COUNT(*) /* no hint */ FROM t2 WHERE NOT EXISTS (SELECT * FROM t3 WHERE ROW(5*t2.s1,77)= (SELECT 50,11*s1 FROM t4 UNION SELECT 50,77 FROM (SELECT * FROM t5) AS t5))); A subquery can return a scalar (a single value), a single row, a single column, or a table (one or more rows of one or more columns). These are called scalar, column, row, and table subqueries. Subqueries that return a particular kind of result often can be used only in certain contexts, as described in the following sections. There are few restrictions on the type of statements in which subqueries can be used. A subquery can contain many of the keywords or clauses that an ordinary SELECT can contain: DISTINCT, GROUP BY, ORDER BY, LIMIT, joins, index hints, UNION constructs, comments, functions, and so on. Beginning with MySQL 8.0.19, TABLE and VALUES statements can be used in subqueries. Subqueries using VALUES are generally more verbose versions of subqueries that can be rewritten more compactly using set notation, or with SELECT or TABLE syntax; assuming that table ts is created using the statement CREATE TABLE ts VALUES ROW(2), ROW(4), ROW(6), the statements shown here are all equivalent: SELECT * FROM tt WHERE b > ANY (VALUES ROW(2), ROW(4), ROW(6)); SELECT * FROM tt WHERE b > ANY (2, 4, 6); SELECT * FROM tt WHERE b > ANY (SELECT * FROM ts); SELECT * FROM tt WHERE b > ANY (TABLE ts); Examples of TABLE subqueries are shown in the sections that follow. A subquery's outer statement can be any one of: SELECT, INSERT, UPDATE, DELETE, SET, or DO. For information about how the optimizer handles subqueries, see Section 8.2.2, "Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions". For a discussion of restrictions on subquery use, including performance issues for certain forms of subquery syntax, see Section 13.2.11.12, "Restrictions on Subqueries". Page 5 13.2.11.1 The Subquery as Scalar Operand In its simplest form, a subquery is a scalar subquery that returns a single value. A scalar subquery is a simple operand, and you can use it almost anywhere a single column value or literal is legal, and you can expect it to have those characteristics that all operands have: a data type, a length, an indication that it can be NULL, and so on. For example: CREATE TABLE t1 (s1 INT, s2 CHAR(5) NOT NULL); INSERT INTO t1 VALUES(100, 'abcde'); SELECT (SELECT s2 FROM t1); The subquery in this SELECT returns a single value ('abcde') that has a data type of CHAR, a length of 5, a character set and collation equal to the defaults in effect at CREATE TABLE time, and an indication that the value in the column can be NULL. Nullability of the value selected by a scalar subquery is not copied because if the subquery result is empty, the result is NULL. For the subquery just shown, if t1 were empty, the result would be NULL even though s2 is NOT NULL. There are a few contexts in which a scalar subquery cannot be used. If a statement permits only a literal value, you cannot use a subquery. For example, LIMIT requires literal integer arguments, and LOAD DATA requires a literal string file name. You cannot use subqueries to supply these values. When you see examples in the following sections that contain the rather spartan construct (SELECT column1 FROM t1), imagine that your own code contains much more diverse and complex constructions. Suppose that we make two tables: CREATE TABLE t1 (s1 INT); INSERT INTO t1 VALUES (1); CREATE TABLE t2 (s1 INT); INSERT INTO t2 VALUES (2); Then perform a SELECT: SELECT (SELECT s1 FROM t2) FROM t1; The result is 2 because there is a row in t2 containing a column s1 that has a value of 2. In MySQL 8.0.19 and later, the preceding query can also be written like this, using TABLE: SELECT (TABLE t2) FROM t1; A scalar subquery can be part of an expression, but remember the parentheses, even if the subquery is an operand that provides an argument for a function. For example: SELECT UPPER((SELECT s1 FROM t1)) FROM t2; The same result can be obtained in MySQL 8.0.19 and later using SELECT UPPER((TABLE t1)) FROM t2. Page 6 13.2.11.2 Comparisons Using Subqueries The most common use of a subquery is in the form: non_subquery_operand comparison_operator (subquery) Where comparison_operator is one of these operators: = > < >= <= <> != <=> ANY (SELECT s1 FROM t2); Suppose that there is a row in table t1 containing (10). The expression is TRUE if table t2 contains (21,14,7) because there is a value 7 in t2 that is less than 10. The expression is FALSE if table t2 contains (20,10), or if table t2 is empty. The expression is unknown (that is, NULL) if table t2 contains (NULL,NULL,NULL). When used with a subquery, the word IN is an alias for = ANY. Thus, these two statements are the same: SELECT s1 FROM t1 WHERE s1 = ANY (SELECT s1 FROM t2); SELECT s1 FROM t1 WHERE s1 IN (SELECT s1 FROM t2); IN and = ANY are not synonyms when used with an expression list. IN can take an expression list, but = ANY cannot. See Section 12.4.2, "Comparison Functions and Operators". NOT IN is not an alias for <> ANY, but for <> ALL. See Section 13.2.11.4, "Subqueries with ALL". The word SOME is an alias for ANY. Thus, these two statements are the same: SELECT s1 FROM t1 WHERE s1 <> ANY (SELECT s1 FROM t2); SELECT s1 FROM t1 WHERE s1 <> SOME (SELECT s1 FROM t2); Use of the word SOME is rare, but this example shows why it might be useful. To most people, the English phrase "a is not equal to any b" means "there is no b which is equal to a," but that is not what is meant by the SQL syntax. The syntax means "there is some b to which a is not equal." Using <> SOME instead helps ensure that everyone understands the true meaning of the query. Beginning with MySQL 8.0.19, you can use TABLE in a scalar IN, ANY, or SOME subquery provided the table contains only a single column. If t2 has only one column, the statements shown previously in this section can be written as shown here, in each case substituting TABLE t2 for SELECT s1 FROM t2: SELECT s1 FROM t1 WHERE s1 > ANY (TABLE t2); SELECT s1 FROM t1 WHERE s1 = ANY (TABLE t2); SELECT s1 FROM t1 WHERE s1 IN (TABLE t2); SELECT s1 FROM t1 WHERE s1 <> ANY (TABLE t2); SELECT s1 FROM t1 WHERE s1 <> SOME (TABLE t2); Page 8 13.2.11.4 Subqueries with ALL Syntax: operand comparison_operator ALL (subquery) The word ALL, which must follow a comparison operator, means "return TRUE if the comparison is TRUE for ALL of the values in the column that the subquery returns." For example: SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2); Suppose that there is a row in table t1 containing (10). The expression is TRUE if table t2 contains (-5,0,+5) because 10 is greater than all three values in t2. The expression is FALSE if table t2 contains (12,6,NULL,-100) because there is a single value 12 in table t2 that is greater than 10. The expression is unknown (that is, NULL) if table t2 contains (0,NULL,1). Finally, the expression is TRUE if table t2 is empty. So, the following expression is TRUE when table t2 is empty: SELECT * FROM t1 WHERE 1 > ALL (SELECT s1 FROM t2); But this expression is NULL when table t2 is empty: SELECT * FROM t1 WHERE 1 > (SELECT s1 FROM t2); In addition, the following expression is NULL when table t2 is empty: SELECT * FROM t1 WHERE 1 > ALL (SELECT MAX(s1) FROM t2); In general, tables containing NULL values and empty tables are "edge cases." When writing subqueries, always consider whether you have taken those two possibilities into account. NOT IN is an alias for <> ALL. Thus, these two statements are the same: SELECT s1 FROM t1 WHERE s1 <> ALL (SELECT s1 FROM t2); SELECT s1 FROM t1 WHERE s1 NOT IN (SELECT s1 FROM t2); MySQL 8.0.19 supports the TABLE statement. As with IN, ANY, and SOME, you can use TABLE with ALL and NOT IN provided that the following two conditions are met: The table in the subquery contains only one column The subquery does not depend on a column expression For example, assuming that table t2 consists of a single column, the last two statements shown previously can be written using TABLE t2 like this: SELECT s1 FROM t1 WHERE s1 <> ALL (TABLE t2); SELECT s1 FROM t1 WHERE s1 NOT IN (TABLE t2); A query such as SELECT * FROM t1 WHERE 1 > ALL (SELECT MAX(s1) FROM t2); cannot be written using TABLE t2 because the subquery depends on a column expression. Page 9 Scalar or column subqueries return a single value or a column of values. A row subquery is a subquery variant that returns a single row and can thus return more than one column value. Legal operators for row subquery comparisons are: = > < >= <= <> != <=> 1; Result: +------+------+------+ | sb1 | sb2 | sb3 | +------+------+------+ | 2 | 2 | 4 | +------+------+------+ Here is another example: Suppose that you want to know the average of a set of sums for a grouped table. This does not work: SELECT AVG(SUM(column1)) FROM t1 GROUP BY column1; However, this query provides the desired information: SELECT AVG(sum_column1) FROM (SELECT SUM(column1) AS sum_column1 FROM t1 GROUP BY column1) AS t1; Notice that the column name used within the subquery (sum_column1) is recognized in the outer query. The column names for a derived table come from its select list: mysql> SELECT * FROM (SELECT 1, 2, 3, 4) AS dt; +---+---+---+---+ | 1 | 2 | 3 | 4 | +---+---+---+---+ | 1 | 2 | 3 | 4 | +---+---+---+---+ To provide column names explicitly, follow the derived table name with a parenthesized list of column names: mysql> SELECT * FROM (SELECT 1, 2, 3, 4) AS dt (a, b, c, d); +---+---+---+---+ | a | b | c | d | +---+---+---+---+ | 1 | 2 | 3 | 4 | +---+---+---+---+ A derived table can return a scalar, column, row, or table. Derived tables are subject to these restrictions: A derived table cannot contain references to other tables of the same SELECT (use a LATERAL derived table for that; see Section 13.2.11.9, "Lateral Derived Tables"). Prior to MySQL 8.0.14, a derived table cannot contain outer references. This is a MySQL restriction that is lifted in MySQL 8.0.14, not a restriction of the SQL standard. For example, the derived table dt in the following query contains a reference t1.b to the table t1 in the outer query: SELECT * FROM t1 WHERE t1.d > (SELECT AVG(dt.a) FROM (SELECT SUM(t2.a) AS a FROM t2 WHERE t2.b = t1.b GROUP BY t2.c) dt WHERE dt.a > 10); The query is valid in MySQL 8.0.14 and higher. Before 8.0.14, it produces an error: Unknown column 't1.b' in 'where clause' The optimizer determines information about derived tables in such a way that EXPLAIN does not need to materialize them. See Section 8.2.2.4, "Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization". It is possible under certain circumstances that using EXPLAIN SELECT modifies table data. This can occur if the outer query accesses any tables and an inner query invokes a stored function that changes one or more rows of a table. Suppose that there are two tables t1 and t2 in database d1, and a stored function f1 that modifies t2, created as shown here: CREATE DATABASE d1; USE d1; CREATE TABLE t1 (c1 INT); CREATE TABLE t2 (c1 INT); CREATE FUNCTION f1(p1 INT) RETURNS INT BEGIN INSERT INTO t2 VALUES (p1); RETURN p1; END; Referencing the function directly in an EXPLAIN SELECT has no effect on t2, as shown here: mysql> SELECT * FROM t2; Empty set (0.02 sec) mysql> EXPLAIN SELECT f1(5)\G *************************** 1. row *************************** id: 1 select_type: SIMPLE table: NULL partitions: NULL type: NULL possible_keys: NULL key: NULL key_len: NULL ref: NULL rows: NULL filtered: NULL Extra: No tables used 1 row in set (0.01 sec) mysql> SELECT * FROM t2; Empty set (0.01 sec) This is because the SELECT statement did not reference any tables, as can be seen in the table and Extra columns of the output. This is also true of the following nested SELECT: mysql> EXPLAIN SELECT NOW() AS a1, (SELECT f1(5)) AS a2\G *************************** 1. row *************************** id: 1 select_type: PRIMARY table: NULL type: NULL possible_keys: NULL key: NULL key_len: NULL ref: NULL rows: NULL filtered: NULL Extra: No tables used 1 row in set, 1 warning (0.00 sec) mysql> SHOW WARNINGS; +-------+------+--+ | Level | Code | Message | +-------+------+--+ | Note | 1249 | Select 2 was reduced during optimization | +-------+------+--+ 1 row in set (0.00 sec) mysql> SELECT * FROM t2; Empty set (0.00 sec) However, if the outer SELECT references any tables, the optimizer executes the statement in the subquery as well, with the result that t2 is modified: mysql> EXPLAIN SELECT * FROM t1 AS a1, (SELECT f1(5)) AS a2\G *************************** 1. row *************************** id: 1 select_type: PRIMARY table: partitions: NULL type: system possible_keys: NULL key: NULL key_len: NULL ref: NULL rows: 1 filtered: 100.00 Extra: NULL *************************** 2. row *************************** id: 1 select_type: PRIMARY table: a1 partitions: NULL type: ALL possible_keys: NULL key: NULL key_len: NULL ref: NULL rows: 1 filtered: 100.00 Extra: NULL *************************** 3. row *************************** id: 2 select_type: DERIVED table: NULL partitions: NULL type: NULL possible_keys: NULL key: NULL key_len: NULL ref: NULL rows: NULL filtered: NULL Extra: No tables used 3 rows in set (0.00 sec) mysql> SELECT * FROM t2; +------+ | c1 | +------+ | 5 | +------+ 1 row in set (0.00 sec) This also means that an EXPLAIN SELECT statement such as the one shown here may take a long time to execute because the BENCHMARK() function is executed once for each row in t1: EXPLAIN SELECT * FROM t1 AS a1, (SELECT BENCHMARK(1000000, MD5(NOW()))); The derived table optimization can also be employed with many correlated (scalar) subqueries (MySQL 8.0.24 and later). For more information and examples, see Section 13.2.11.7, "Correlated Subqueries". Page 13 13.2.11.9 Lateral Derived Tables A derived table cannot normally refer to (depend on) columns of preceding tables in the same FROM clause. As of MySQL 8.0.14, a derived table may be defined as a lateral derived table to specify that such references are permitted. Nonlateral derived tables are specified using the syntax discussed in Section 13.2.11.8, "Derived Tables". The syntax for a lateral derived table is the same as for a nonlateral derived table except that the keyword LATERAL is specified before the derived table specification. The LATERAL keyword must precede each table to be used as a lateral derived table. Lateral derived tables are subject to these restrictions: A lateral derived table can occur only in a FROM clause, either in a list of tables separated with commas or in a join specification (JOIN, INNER JOIN, CROSS JOIN, LEFT [OUTER] JOIN, or RIGHT [OUTER] JOIN). If a lateral derived table is in the right operand of a join clause and contains a reference to the left operand, the join operation must be an INNER JOIN, CROSS JOIN, or LEFT [OUTER] JOIN. If the table is in the left operand and contains a reference to the right operand, the join operation must be an INNER JOIN, CROSS JOIN, or RIGHT [OUTER] JOIN. If a lateral derived table references an aggregate function, the function's aggregation query cannot be the one that owns the FROM clause in which the lateral derived table occurs. Per the SQL standard, a table function has an implicit LATERAL, so it behaves as in MySQL 8.0 versions prior to 8.0.14. However, per the standard, the LATERAL word is not allowed before JSON_TABLE(), even though it is implicit. The following discussion shows how lateral derived tables make possible certain SQL operations that cannot be done with nonlateral derived tables or that require less-efficient workarounds. Suppose that we want to solve this problem: Given a table of people in a sales force (where each row describes a member of the sales force), and a table of all sales (where each row describes a sale: salesperson, customer, amount, date), determine the size and customer of the largest sale for each salesperson. This problem can be approached two ways. First approach to solving the problem: For each salesperson, calculate the maximum sale size, and also find the customer who provided this maximum. In MySQL, that can be done like this: SELECT salesperson.name, -- find maximum sale size for this salesperson (SELECT MAX(amount) AS amount FROM all_sales WHERE all_sales.salesperson_id = salesperson.id) AS amount, -- find customer for this maximum size (SELECT customer_name FROM all_sales WHERE all_sales.salesperson_id = salesperson.id AND all_sales.amount = -- find maximum size, again (SELECT MAX(amount) AS amount FROM all_sales WHERE all_sales.salesperson_id = salesperson.id)) AS customer_name FROM salesperson; That query is inefficient because it calculates the maximum size twice per salesperson (once in the first subquery and once in the second). We can try to achieve an efficiency gain by calculating the maximum once per salesperson and "caching" it in a derived table, as shown by this modified query: SELECT salesperson.name, max_sale.amount, max_sale_customer.customer_name FROM salesperson, -- calculate maximum size, cache it in transient derived table max_sale (SELECT MAX(amount) AS amount FROM all_sales WHERE all_sales.salesperson_id = salesperson.id) AS max_sale, -- find customer, reusing cached maximum size (SELECT customer_name FROM all_sales WHERE all_sales.salesperson_id = salesperson.id AND all_sales.amount = -- the cached maximum size max_sale.amount) AS max_sale_customer; However, the query is illegal in SQL-92 because derived tables cannot depend on other tables in the same FROM clause. Derived tables must be constant over the query's duration, not contain references to columns of other FROM clause tables. As written, the query produces this error: ERROR 1054 (42S22): Unknown column 'salesperson.id' in 'where clause' In SQL:1999, the query becomes legal if the derived tables are preceded by the LATERAL keyword (which means "this derived table depends on previous tables on its left side"): SELECT salesperson.name, max_sale.amount, max_sale_customer.customer_name FROM salesperson, -- calculate maximum size, cache it in transient derived table max_sale LATERAL (SELECT MAX(amount) AS amount FROM all_sales WHERE all_sales.salesperson_id = salesperson.id) AS max_sale, -- find customer, reusing cached maximum size LATERAL (SELECT customer_name FROM all_sales WHERE all_sales.salesperson_id = salesperson.id AND all_sales.amount = -- the cached maximum size max_sale.amount) AS max_sale_customer; A lateral derived table need not be constant and is brought up to date each time a new row from a preceding table on which it depends is processed by the top query. Second approach to solving the problem: A different solution could be used if a subquery in the SELECT list could return multiple columns: SELECT salesperson.name, -- find maximum size and customer at same time (SELECT amount, customer_name FROM all_sales WHERE all_sales.salesperson_id = salesperson.id ORDER BY amount DESC LIMIT 1) FROM salesperson; That is efficient but illegal. It does not work because such subqueries can return only a single column: ERROR 1241 (21000): Operand should contain 1 column(s) One attempt at rewriting the query is to select multiple columns from a derived table: SELECT salesperson.name, max_sale.amount, max_sale.customer_name FROM salesperson, -- find maximum size and customer at same time (SELECT amount, customer_name FROM all_sales WHERE all_sales.salesperson_id = salesperson.id ORDER BY amount DESC LIMIT 1) AS max_sale; However, that also does not work. The derived table is dependent on the salesperson table and thus fails without LATERAL: ERROR 1054 (42S22): Unknown column 'salesperson.id' in 'where clause' Adding the LATERAL keyword makes the query legal: SELECT salesperson.name, max_sale.amount, max_sale.customer_name FROM salesperson, -- find maximum size and customer at same time LATERAL (SELECT amount, customer_name FROM all_sales WHERE all_sales.salesperson_id = salesperson.id ORDER BY amount DESC LIMIT 1) AS max_sale; In short, LATERAL is the efficient solution to all drawbacks in the two approaches just discussed. Page 14 13.2.11.10 Subquery Errors There are some errors that apply only to subqueries. This section describes them. Unsupported subquery syntax: ERROR 1235 (ER_NOT_SUPPORTED_YET) SQLSTATE = 42000 Message = "This version of MySQL doesn't yet support 'LIMIT & IN/ALL/ANY/SOME subquery'" This means that MySQL does not support statements like the following: SELECT * FROM t1 WHERE s1 IN (SELECT s2 FROM t2 ORDER BY s1 LIMIT 1) Incorrect number of columns from subquery: ERROR 1241 (ER_OPERAND_COL) SQLSTATE = 21000 Message = "Operand should contain 1 column(s)" This error occurs in cases like this: SELECT (SELECT column1, column2 FROM t2) FROM t1; You may use a subquery that returns multiple columns, if the purpose is row comparison. In other contexts, the subquery must be a scalar operand. See Section 13.2.11.5, "Row Subqueries". Incorrect number of rows from subquery: ERROR 1242 (ER_SUBSELECT_NO_1_ROW) SQLSTATE = 21000 Message = "Subquery returns more than 1 row" This error occurs for statements where the subquery must return at most one row but returns multiple rows. Consider the following example: SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2); If SELECT column1 FROM t2 returns just one row, the previous query works. If the subquery returns more than one row, error 1242 occurs. In that case, the query should be rewritten as: SELECT * FROM t1 WHERE column1 = ANY (SELECT column1 FROM t2); Incorrectly used table in subquery: Error 1093 (ER_UPDATE_TABLE_USED) SQLSTATE = HY000 Message = "You can't specify target table 'x' for update in FROM clause" This error occurs in cases such as the following, which attempts to modify a table and select from the same table in the subquery: UPDATE t1 SET column2 = (SELECT MAX(column1) FROM t1); You can use a common table expression or derived table to work around this. See Section 13.2.11.12, "Restrictions on Subqueries". In MySQL 8.0.19 and later, all of the errors described in this section also apply when using TABLE in subqueries. For transactional storage engines, the failure of a subquery causes the entire statement to fail. For nontransactional storage engines, data modifications made before the error was encountered are preserved. Page 15 13.2.11.11 Optimizing Subqueries Development is ongoing, so no optimization tip is reliable for the long term. The following list provides some interesting tricks that you might want to play with. See also Section 8.2.2, "Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions". Move clauses from outside to inside the subquery. For example, use this query: SELECT * FROM t1 WHERE s1 IN (SELECT s1 FROM t1 UNION ALL SELECT s1 FROM t2); Instead of this query: SELECT * FROM t1 WHERE s1 IN (SELECT s1 FROM t1) OR s1 IN (SELECT s1 FROM t2); For another example, use this query: SELECT (SELECT column1 + 5 FROM t1) FROM t2; Instead of this query: SELECT (SELECT column1 FROM t1) + 5 FROM t2; Page 16 13.2.11.12 Restrictions on Subqueries In general, you cannot modify a table and select from the same table in a subquery. For example, this limitation applies to statements of the following forms: DELETE FROM t WHERE ... (SELECT ... FROM t ...); UPDATE t ... WHERE col = (SELECT ... FROM t ...); {INSERT|REPLACE} INTO t (SELECT ... FROM t ...); Exception: The preceding prohibition does not apply if for the modified table you are using a derived table and that derived table is materialized rather than merged into the outer query. (See Section 8.2.2.4, "Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization".) Example: UPDATE t ... WHERE col = (SELECT * FROM (SELECT ... FROM t...) AS dt ...); Here the result from the derived table is materialized as a temporary table, so the relevant rows in t have already been selected by the time the update to t takes place. In general, you may be able to influence the optimizer to materialize a derived table by adding a NO_MERGE optimizer hint. See Section 8.9.3, "Optimizer Hints". Row comparison operations are only partially supported: For expr [NOT] IN subquery, expr can be an n-tuple (specified using row constructor syntax) and the subquery can return rows of n-tuples. The permitted syntax is therefore more specifically expressed as row_constructor [NOT] IN table_subquery For expr op {ALL|ANY|SOME} subquery, expr must be a scalar value and the subquery must be a column subquery; it cannot return multiple-column rows. In other words, for a subquery that returns rows of n-tuples, this is supported: (expr_1, ..., expr_n) [NOT] IN table_subquery But this is not supported: (expr_1, ..., expr_n) op {ALL|ANY|SOME} subquery The reason for supporting row comparisons for IN but not for the others is that IN is implemented by rewriting it as a sequence of = comparisons and AND operations. This approach cannot be used for ALL, ANY, or SOME. Prior to MySQL 8.0.14, subqueries in the FROM clause cannot be correlated subqueries. They are materialized in whole (evaluated to produce a result set) during query execution, so they cannot be evaluated per row of the outer query. The optimizer delays materialization until the result is needed, which may permit materialization to be avoided. See Section 8.2.2.4, "Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization". MySQL does not support LIMIT in subqueries for certain subquery operators: mysql> SELECT * FROM t1 WHERE s1 IN (SELECT s2 FROM t2 ORDER BY s1 LIMIT 1); ERROR 1235 (42000): This version of MySQL doesn't yet support 'LIMIT & IN/ALL/ANY/SOME subquery' See Section 13.2.11.10, "Subquery Errors". MySQL permits a subquery to refer to a stored function that has data-modifying side effects such as inserting rows into a table. For example, if f() inserts rows, the following query can modify data: SELECT ... WHERE x IN (SELECT f() ...); This behavior is an extension to the SQL standard. In MySQL, it can produce nondeterministic results because f() might be executed a different number of times for different executions of a given query depending on how the optimizer chooses to handle it. For statement-based or mixed-format replication, one implication of this indeterminism is that such a query can produce different results on the source and its replicas. Page 17 TABLE is a DML statement introduced in MySQL 8.0.19 which returns rows and columns of the named table. TABLE table_name [ORDER BY column_name] [LIMIT number [OFFSET number]] The TABLE statement in some ways acts like SELECT. Given the existance of a table named t, the following two statements produce identical output: TABLE t; SELECT * FROM t; You can order and limit the number of rows produced by TABLE using ORDER BY and LIMIT clauses, respectively. These function identically to the same clauses when used with SELECT (including an optional OFFSET clause with LIMIT), as you can see here: mysql> TABLE t; +----+----+ | a | b | +----+----+ | 1 | 2 | | 6 | 7 | | 9 | 5 | | 10 | -4 | | 11 | -1 | | 13 | 3 | | 14 | 6 | +----+----+ 7 rows in set (0.00 sec) mysql> TABLE t ORDER BY b; +----+----+ | a | b | +----+----+ | 10 | -4 | | 11 | -1 | | 1 | 2 | | 13 | 3 | | 9 | 5 | | 14 | 6 | | 6 | 7 | +----+----+ 7 rows in set (0.00 sec) mysql> TABLE t LIMIT 3; +---+---+ | a | b | +---+---+ | 1 | 2 | | 6 | 7 | | 9 | 5 | +---+---+ 3 rows in set (0.00 sec) mysql> TABLE t ORDER BY b LIMIT 3; +----+----+ | a | b | +----+----+ | 10 | -4 | | 11 | -1 | | 1 | 2 | +----+----+ 3 rows in set (0.00 sec) mysql> TABLE t ORDER BY b LIMIT 3 OFFSET 2; +----+----+ | a | b | +----+----+ | 1 | 2 | | 13 | 3 | | 9 | 5 | +----+----+ 3 rows in set (0.00 sec) TABLE differs from SELECT in two key respects: TABLE always displays all columns of the table. TABLE does not allow for any arbitrary filtering of rows; that is, TABLE does not support any WHERE clause. For limiting which table columns are returned, filtering rows beyond what can be accomplished using ORDER BY and LIMIT, or both, use SELECT. TABLE can be used with temporary tables. TABLE can also be used in place of SELECT in a number of other constructs, including those listed here: With UNION, as shown here: mysql> TABLE t1; +---+----+ | a | b | +---+----+ | 2 | 10 | | 5 | 3 | | 7 | 8 | +---+----+ 3 rows in set (0.00 sec) mysql> TABLE t2; +---+---+ | a | b | +---+---+ | 1 | 2 | | 3 | 4 | | 6 | 7 | +---+---+ 3 rows in set (0.00 sec) mysql> TABLE t1 UNION TABLE t2; +---+----+ | a | b | +---+----+ | 2 | 10 | | 5 | 3 | | 7 | 8 | | 1 | 2 | | 3 | 4 | | 6 | 7 | +---+----+ 6 rows in set (0.00 sec) The UNION just shown is equivalent to the following statement: mysql> SELECT * FROM t1 UNION SELECT * FROM t2; +---+----+ | a | b | +---+----+ | 2 | 10 | | 5 | 3 | | 7 | 8 | | 1 | 2 | | 3 | 4 | | 6 | 7 | +---+----+ 6 rows in set (0.00 sec) TABLE can also be used together in unions with SELECT statements, VALUES statements, or both. See Section 13.2.10.3, "UNION Clause". With INTO to populate user variables, and with INTO OUTFILE or INTO DUMPFILE to write table data to a file. See Section 13.2.10.1, "SELECT ... INTO Statement", for more specific information and examples. In many cases where you can employ subqueries. Given any table t1 with a column named a, and a second table t2 having a single column, statements such as the following are possible: SELECT * FROM t1 WHERE a IN (TABLE t2); Assuming that the single column of table ts is named x, the preceding is equivalent to each of the statements shown here (and produces exactly the same result in either case): SELECT * FROM t1 WHERE a IN (SELECT x FROM t2); SELECT * FROM t1 WHERE a IN (SELECT * FROM t2); See Section 13.2.11, "Subqueries", for more information. With INSERT and REPLACE statements, where you would otherwise use SELECT *. See Section 13.2.6.1, "INSERT ... SELECT Statement", for more information and examples. TABLE can also be used in many cases in place of the SELECT in CREATE TABLE ... SELECT or CREATE VIEW ... SELECT. See the descriptions of these statements for more information and examples. Page 18 UPDATE is a DML statement that modifies rows in a table. An UPDATE statement can start with a WITH clause to define common table expressions accessible within the UPDATE. See Section 13.2.15, "WITH (Common Table Expressions)". Single-table syntax: UPDATE [LOW_PRIORITY] [IGNORE] table_reference SET assignment_list [WHERE where_condition] [ORDER BY ...] [LIMIT row_count] value: {expr | DEFAULT} assignment: col_name = value assignment_list: assignment [, assignment] ... Multiple-table syntax: UPDATE [LOW_PRIORITY] [IGNORE] table_references SET assignment_list [WHERE where_condition] For the single-table syntax, the UPDATE statement updates columns of existing rows in the named table with new values. The SET clause indicates which columns to modify and the values they should be given. Each value can be given as an expression, or the keyword DEFAULT to set a column explicitly to its default value. The WHERE clause, if given, specifies the conditions that identify which rows to update. With no WHERE clause, all rows are updated. If the ORDER BY clause is specified, the rows are updated in the order that is specified. The LIMIT clause places a limit on the number of rows that can be updated. For the multiple-table syntax, UPDATE updates rows in each table named in table_references that satisfy the conditions. Each matching row is updated once, even if it matches the conditions multiple times. For multiple-table syntax, ORDER BY and LIMIT cannot be used. For partitioned tables, both the single-single and multiple-table forms of this statement support the use of a PARTITION clause as part of a table reference. This option takes a list of one or more partitions or subpartitions (or both). Only the partitions (or subpartitions) listed are checked for matches, and a row that is not in any of these partitions or subpartitions is not updated, whether it satisfies the where_condition or not. Note Unlike the case when using PARTITION with an INSERT or REPLACE statement, an otherwise valid UPDATE ... PARTITION statement is considered successful even if no rows in the listed partitions (or subpartitions) match the where_condition. For more information and examples, see Section 24.5, "Partition Selection". where_condition is an expression that evaluates to true for each row to be updated. For expression syntax, see Section 9.5, "Expressions". table_references and where_condition are specified as described in Section 13.2.10, "SELECT Statement". You need the UPDATE privilege only for columns referenced in an UPDATE that are actually updated. You need only the SELECT privilege for any columns that are read but not modified. The UPDATE statement supports the following modifiers: With the LOW_PRIORITY modifier, execution of the UPDATE is delayed until no other clients are reading from the table. This affects only storage engines that use only table-level locking (such as MyISAM, MEMORY, and MERGE). With the IGNORE modifier, the update statement does not abort even if errors occur during the update. Rows for which duplicate-key conflicts occur on a unique key value are not updated. Rows updated to values that would cause data conversion errors are updated to the closest valid values instead. For more information, see The Effect of IGNORE on Statement Execution. UPDATE IGNORE statements, including those having an ORDER BY clause, are flagged as unsafe for statement-based replication. (This is because the order in which the rows are updated determines which rows are ignored.) Such statements produce a warning in the error log when using statement-based mode and are written to the binary log using the row-based format when using MIXED mode. (Bug #11758262, Bug #50439) See Section 17.2.1.3, "Determination of Safe and Unsafe Statements in Binary Logging", for more information. If you access a column from the table to be updated in an expression, UPDATE uses the current value of the column. For example, the following statement sets col1 to one more than its current value: UPDATE t1 SET col1 = col1 + 1; The second assignment in the following statement sets col2 to the current (updated) col1 value, not the original col1 value. The result is that col1 and col2 have the same value. This behavior differs from standard SQL. UPDATE t1 SET col1 = col1 + 1, col2 = col1; Single-table UPDATE assignments are generally evaluated from left to right. For multiple-table updates, there is no guarantee that assignments are carried out in any particular order. If you set a column to the value it currently has, MySQL notices this and does not update it. If you update a column that has been declared NOT NULL by setting to NULL, an error occurs if strict SQL mode is enabled; otherwise, the column is set to the implicit default value for the column data type and the warning count is incremented. The implicit default value is 0 for numeric types, the empty string ('') for string types, and the "zero" value for date and time types. See Section 11.6, "Data Type Default Values". If a generated column is updated explicitly, the only permitted value is DEFAULT. For information about generated columns, see Section 13.1.20.8, "CREATE TABLE and Generated Columns". UPDATE returns the number of rows that were actually changed. The mysql_info() C API function returns the number of rows that were matched and updated and the number of warnings that occurred during the UPDATE. You can use LIMIT row_count to restrict the scope of the UPDATE. A LIMIT clause is a rows-matched restriction. The statement stops as soon as it has found row_count rows that satisfy the WHERE clause, whether or not they actually were changed. If an UPDATE statement includes an ORDER BY clause, the rows are updated in the order specified by the clause. This can be useful in certain situations that might otherwise result in an error. Suppose that a table t contains a column id that has a unique index. The following statement could fail with a duplicate-key error, depending on the order in which rows are updated: UPDATE t SET id = id + 1; For example, if the table contains 1 and 2 in the id column and 1 is updated to 2 before 2 is updated to 3, an error occurs. To avoid this problem, add an ORDER BY clause to cause the rows with larger id values to be updated before those with smaller values: UPDATE t SET id = id + 1 ORDER BY id DESC; You can also perform UPDATE operations covering multiple tables. However, you cannot use ORDER BY or LIMIT with a multiple-table UPDATE. The table_references clause lists the tables involved in the join. Its syntax is described in Section 13.2.10.2, "JOIN Clause". Here is an example: UPDATE items,month SET items.price=month.price WHERE items.id=month.id; The preceding example shows an inner join that uses the comma operator, but multiple-table UPDATE statements can use any type of join permitted in SELECT statements, such as LEFT JOIN. If you use a multiple-table UPDATE statement involving InnoDB tables for which there are foreign key constraints, the MySQL optimizer might process tables in an order that differs from that of their parent/child relationship. In this case, the statement fails and rolls back. Instead, update a single table and rely on the ON UPDATE capabilities that InnoDB provides to cause the other tables to be modified accordingly. See Section 13.1.20.5, "FOREIGN KEY Constraints". You cannot update a table and select directly from the same table in a subquery. You can work around this by using a multi-table update in which one of the tables is derived from the table that you actually wish to update, and referring to the derived table using an alias. Suppose you wish to update a table named items which is defined using the statement shown here: CREATE TABLE items (id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY, wholesale DECIMAL(6,2) NOT NULL DEFAULT 0.00, retail DECIMAL(6,2) NOT NULL DEFAULT 0.00, quantity BIGINT NOT NULL DEFAULT 0); To reduce the retail price of any items for which the markup is 30% or greater and of which you have fewer than one hundred in stock, you might try to use an UPDATE statement such as the one following, which uses a subquery in the WHERE clause. As shown here, this statement does not work: mysql> UPDATE items > SET retail = retail * 0.9 > WHERE id IN > (SELECT id FROM items > WHERE retail / wholesale >= 1.3 AND quantity > 100); ERROR 1093 (HY000): You can't specify target table 'items' for update in FROM clause Instead, you can employ a multi-table update in which the subquery is moved into the list of tables to be updated, using an alias to reference it in the outermost WHERE clause, like this: UPDATE items, (SELECT id FROM items WHERE id IN (SELECT id FROM items WHERE retail / wholesale >= 1.3 AND quantity < 100)) AS discounted SET items.retail = items.retail * 0.9 WHERE items.id = discounted.id; Because the optimizer tries by default to merge the derived table discounted into the outermost query block, this works only if you force materialization of the derived table. You can do this by setting the derived_merge flag of the optimizer_switch system variable to off before running the update, or by using the NO_MERGE optimizer hint, as shown here: UPDATE /*+ NO_MERGE(discounted) */ items, (SELECT id FROM items WHERE retail / wholesale >= 1.3 AND quantity < 100) AS discounted SET items.retail = items.retail * 0.9 WHERE items.id = discounted.id; The advantage of using the optimizer hint in such a case is that it applies only within the query block where it is used, so that it is not necessary to change the value of optimizer_switch again after executing the UPDATE. Another possibility is to rewrite the subquery so that it does not use IN or EXISTS, like this: UPDATE items, (SELECT id, retail / wholesale AS markup, quantity FROM items) AS discounted SET items.retail = items.retail * 0.9 WHERE discounted.markup >= 1.3 AND discounted.quantity < 100 AND items.id = discounted.id; In this case, the subquery is materialized by default rather than merged, so it is not necessary to disable merging of the derived table. Page 19 VALUES is a DML statement introduced in MySQL 8.0.19 which returns a set of one or more rows as a table. In other words, it is a table value constructor which also functions as a standalone SQL statement. VALUES row_constructor_list [ORDER BY column_designator] [LIMIT BY number] row_constructor_list: ROW(value_list)[, ROW(value_list)][, ...] value_list: value[, value][, ...] column_designator: column_index The VALUES statement consists of the VALUES keyword followed by a list of one or more row constructors, separated by commas. A row constructor consists of the ROW() row constructor clause with a value list of one or more scalar values enclosed in the parentheses. A value can be a literal of any MySQL data type or an expression that resolves to a scalar value. ROW() cannot be empty (but each of the supplied scalar values can be NULL). Each ROW() in the same VALUES statement must have the same number of values in its value list. The DEFAULT keyword is not supported by VALUES and causes a syntax error, except when it is used to supply values in an INSERT statement. The output of VALUES is a table: mysql> VALUES ROW(1,-2,3), ROW(5,7,9), ROW(4,6,8); +----------+----------+----------+ | column_0 | column_1 | column_2 | +----------+----------+----------+ | 1 | -2 | 3 | | 5 | 7 | 9 | | 4 | 6 | 8 | +----------+----------+----------+ 3 rows in set (0.00 sec) The columns of the table output from VALUES have the implicitly named columns column_0, column_1, column_2, and so on, always beginning with 0. This fact can be used to order the rows by column using an optional ORDER BY clause in the same way that this clause works with a SELECT statement, as shown here: mysql> VALUES ROW(1,-2,3), ROW(5,7,9), ROW(4,6,8) ORDER BY column_1; +----------+----------+----------+ | column_0 | column_1 | column_2 | +----------+----------+----------+ | 1 | -2 | 3 | | 4 | 6 | 8 | | 5 | 7 | 9 | +----------+----------+----------+ 3 rows in set (0.00 sec) The VALUES statement also supports a LIMIT clause for limiting the number of rows in the output. The VALUES statement is permissive regarding data types of column values; you can mix types within the same column, as shown here: mysql> VALUES ROW("q", 42, '2019-12-18'), -> ROW(23, "abc", 98.6), -> ROW(27.0002, "Mary Smith", '{"a": 10, "b": 25}'); +----------+------------+--------------------+ | column_0 | column_1 | column_2 | +----------+------------+--------------------+ | q | 42 | 2019-12-18 | | 23 | abc | 98.6 | | 27.0002 | Mary Smith | {"a": 10, "b": 25} | +----------+------------+--------------------+ 3 rows in set (0.00 sec) Important VALUES with one or more instances of ROW() acts as a table value constructor; although it can be used to supply values in an INSERT or REPLACE statement, do not confuse it with the VALUES keyword that is also used for this purpose. You should also not confuse it with the VALUES() function that refers to column values in INSERT ... ON DUPLICATE KEY UPDATE. You should also bear in mind that ROW() is a row value constructor (see Section 13.2.11.5, "Row Subqueries", whereas VALUES ROW() is a table value constructor; the two cannot be used interchangeably. VALUES can be used in many cases where you could employ SELECT, including those listed here: With UNION, as shown here: mysql> SELECT 1,2 UNION SELECT 10,15; +----+----+ | 1 | 2 | +----+----+ | 1 | 2 | | 10 | 15 | +----+----+ 2 rows in set (0.00 sec) mysql> VALUES ROW(1,2) UNION VALUES ROW(10,15); +----------+----------+ | column_0 | column_1 | +----------+----------+ | 1 | 2 | | 10 | 15 | +----------+----------+ 2 rows in set (0.00 sec) It is also possible in this fashion to union together constructed tables having more than one row, like this: mysql> VALUES ROW(1,2), ROW(3,4), ROW(5,6) > UNION VALUES ROW(10,15),ROW(20,25); +----------+----------+ | column_0 | column_1 | +----------+----------+ | 1 | 2 | | 3 | 4 | | 5 | 6 | | 10 | 15 | | 20 | 25 | +----------+----------+ 5 rows in set (0.00 sec) You can also (and it is usually preferable to) omit UNION altogether in such cases and use a single VALUES statement, like this: mysql> VALUES ROW(1,2), ROW(3,4), ROW(5,6), ROW(10,15), ROW(20,25); +----------+----------+ | column_0 | column_1 | +----------+----------+ | 1 | 2 | | 3 | 4 | | 5 | 6 | | 10 | 15 | | 20 | 25 | +----------+----------+ VALUES can also be used in unions with SELECT statements, TABLE statements, or both. The constructed tables in the UNION must contain the same number of columns, just as if you were using SELECT. See Section 13.2.10.3, “UNION Clause”, for further examples. In joins. See Section 13.2.10.2, “JOIN Clause”, for more information and examples. In place of VALUES() in an INSERT or REPLACE statement, in which case its semantics differ slightly from what is described here. See Section 13.2.6, “INSERT Statement”, for details. In place of the source table in CREATE TABLE ... SELECT and CREATE VIEW ... SELECT. See the descriptions of these statements for more information and examples. Page 20 13.2.15 WITH (Common Table Expressions) A common table expression (CTE) is a named temporary result set that exists within the scope of a single statement and that can be referred to later within that statement, possibly multiple times. The following discussion describes how to write statements that use CTEs. For information about CTE optimization, see Section 8.2.2.4, “Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization”. Additional Resources These articles contain additional information about using CTEs in MySQL, including many examples: To specify common table expressions, use a WITH clause that has one or more comma-separated subclauses. Each subclause provides a subquery that produces a result set, and associates a name with the subquery. The following example defines CTEs named cte1 and cte2 in the WITH clause, and refers to them in the top-level SELECT that follows the WITH clause: WITH cte1 AS (SELECT a, b FROM table1), cte2 AS (SELECT c, d FROM table2) SELECT b, d FROM cte1 JOIN cte2 WHERE cte1.a = cte2.c; In the statement containing the WITH clause, each CTE name can be referenced to access the corresponding CTE result set. A CTE name can be referenced in other CTEs, enabling CTEs to be defined based on other CTEs. A CTE can refer to itself to define a recursive CTE. Common applications of recursive CTEs include series generation and traversal of hierarchical or tree-structured data. Common table expressions are an optional part of the syntax for DML statements. They are defined using a WITH clause: with_clause: WITH [RECURSIVE] cte_name [(col_name [, col_name] ...)] AS (subquery) [, cte_name [(col_name [, col_name] ...)] AS (subquery)] ... cte_name names a single common table expression and can be used as a table reference in the statement containing the WITH clause. The subquery part of AS (subquery) is called the “subquery of the CTE” and is what produces the CTE result set. The parentheses following AS are required. A common table expression is recursive if its subquery refers to its own name. The RECURSIVE keyword must be included if any CTE in the WITH clause is recursive. For more information, see Recursive Common Table Expressions. Determination of column names for a given CTE occurs as follows: If a parenthesized list of names follows the CTE name, those names are the column names: WITH cte (col1, col2) AS (SELECT 1, 2 UNION ALL SELECT 3, 4) SELECT col1, col2 FROM cte; The number of names in the list must be the same as the number of columns in the result set. Otherwise, the column names come from the select list of the first SELECT within the AS (subquery) part: WITH cte AS (SELECT 1 AS col1, 2 AS col2 UNION ALL SELECT 3, 4) SELECT col1, col2 FROM cte; A WITH clause is permitted in these contexts: At the beginning of SELECT, UPDATE, and DELETE statements. WITH ... SELECT ... WITH ... UPDATE ... WITH ... DELETE ... At the beginning of subqueries (including derived table subqueries): SELECT ... WHERE id IN (WITH ... SELECT ...) ... SELECT * FROM (WITH ... SELECT ...) AS dt ... Immediately preceding SELECT for statements that include a SELECT statement: INSERT ... WITH ... SELECT ... REPLACE ... WITH ... SELECT ... CREATE TABLE ... WITH ... SELECT ... CREATE VIEW ... WITH ... SELECT ... DECLARE CURSOR ... WITH ... SELECT ... EXPLAIN ... WITH ... SELECT ... Only one WITH clause is permitted at the same level. WITH followed by WITH at the same level is not permitted, so this is illegal: WITH cte1 AS (...) WITH cte2 AS (...) SELECT ... To make the statement legal, use a single WITH clause that separates the subclauses by a comma: WITH cte1 AS (...), cte2 AS (...) SELECT ... However, a statement can contain multiple WITH clauses if they occur at different levels: WITH cte1 AS (SELECT 1) SELECT * FROM (WITH cte2 AS (SELECT 2) SELECT * FROM cte2 JOIN cte1) AS dt; A WITH clause can define one or more common table expressions, but each CTE name must be unique to the clause. This is illegal: WITH cte1 AS (...), cte1 AS (...) SELECT ... To make the statement legal, define the CTEs with unique names: WITH cte1 AS (...), cte2 AS (...) SELECT ... A CTE can refer to itself or to other CTEs: A self-referencing CTE is recursive. A CTE can refer to CTEs defined earlier in the same WITH clause, but not those defined later. This constraint rules out mutually-recursive CTEs, where cte1 references cte2 and cte2 references cte1. One of those references must be to a CTE defined later, which is not permitted. A CTE in a given query block can refer to CTEs defined in query blocks at a more outer level, but not CTEs defined in query blocks at a more inner level. For resolving references to objects with the same names, derived tables hide CTEs; and CTEs hide base tables, TEMPORARY tables, and views. Name resolution occurs by searching for objects in the same query block, then proceeding to outer blocks in turn while no object with the name is found. Like derived tables, a CTE cannot contain outer references prior to MySQL 8.0.14. This is a MySQL restriction that is lifted in MySQL 8.0.14, not a restriction of the SQL standard. For additional syntax considerations specific to recursive CTEs, see Recursive Common Table Expressions. Recursive Common Table Expressions A recursive common table expression is one having a subquery that refers to its own name. For example: WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte WHERE n < 5) SELECT * FROM cte; When executed, the statement produces this result, a single column containing a simple linear sequence: +------+ | n | +------+ | 1 | | 2 | | 3 | | 4 | | 5 | +------+ A recursive CTE has this structure: The WITH clause must begin with WITH RECURSIVE if any CTE in the WITH clause refers to itself. (If no CTE refers to itself, RECURSIVE is permitted but not required.) If you forget RECURSIVE for a recursive CTE, this error is a likely result: ERROR 1146 (42S02): Table 'cte_name' doesn't exist The recursive CTE subquery has two parts, separated by UNION [ALL] or UNION DISTINCT: SELECT ... -- return initial row set UNION ALL SELECT ... -- return additional row sets The first SELECT produces the initial row or rows for the CTE and does not refer to the CTE name. The second SELECT produces additional rows and recurses by referring to the CTE name in its FROM clause. Recursion ends when this part produces no new rows. Thus, a recursive CTE consists of a nonrecursive SELECT part followed by a recursive SELECT part. Each SELECT part can itself be a union of multiple SELECT statements. The types of the CTE result columns are inferred from the column types of the nonrecursive SELECT part only, and the columns are all nullable. For type determination, the recursive SELECT part is ignored. If the nonrecursive and recursive parts are separated by UNION DISTINCT, duplicate rows are eliminated. This is useful for queries that perform transitive closures, to avoid infinite loops. Each iteration of the recursive part operates only on the rows produced by the previous iteration. If the recursive part has multiple query blocks, iterations of each query block are scheduled in unspecified order, and each query block operates on rows that have been produced either by its previous iteration or by other query blocks since that previous iteration's end. The recursive CTE subquery shown earlier has this nonrecursive part that retrieves a single row to produce the initial row set: SELECT 1 The CTE subquery also has this recursive part: SELECT n + 1 FROM cte WHERE n < 5 At each iteration, that SELECT produces a row with a new value one greater than the value of n from the previous row set. The first iteration operates on the initial row set (1) and produces 1+1=2; the second iteration operates on the first iteration's row set (2) and produces 2+1=3; and so forth. This continues until recursion ends, which occurs when n is no longer less than 5. If the recursive part of a CTE produces wider values for a column than the nonrecursive part, it may be necessary to widen the column in the nonrecursive part to avoid data truncation. Consider this statement: WITH RECURSIVE cte AS (SELECT 1 AS n, 'abc' AS str UNION ALL SELECT n + 1, CONCAT(str, str) FROM cte WHERE n < 3) SELECT * FROM cte; In nonstrict SQL mode, the statement produces this output: +------+------+ | n | str | +------+------+ | 1 | abc | | 2 | abc | | 3 | abc | +------+------+ The str column values are all 'abc' because the nonrecursive SELECT determines the column widths. Consequently, the wider str values produced by the recursive SELECT are truncated. In strict SQL mode, the statement produces an error: ERROR 1406 (22001): Data too long for column 'str' at row 1 To address this issue, so that the statement does not produce truncation or errors, use CAST() in the nonrecursive SELECT to make the str column wider: WITH RECURSIVE cte AS (SELECT 1 AS n, CAST('abc' AS CHAR(20)) AS str UNION ALL SELECT n + 1, CONCAT(str, str) FROM cte WHERE n < 3) SELECT * FROM cte; Now the statement produces this result, without truncation: +------+--------------+ | n | str | +------+--------------+ | 1 | abc | | 2 | abcabc | | 3 | abcabcabcabc | +------+--------------+ Columns are accessed by name, not position, which means that columns in the recursive part can access columns in the nonrecursive part that have a different position, as this CTE illustrates: WITH RECURSIVE cte AS (SELECT 1 AS n, 1 AS p, -1 AS q UNION ALL SELECT n + 1, q * 2, p * 2 FROM cte WHERE n < 5) SELECT * FROM cte; Because p in one row is derived from q in the previous row, and vice versa, the positive and negative values swap positions in each successive row of the output: +------+------+------+ | n | p | q | +------+------+------+ | 1 | 1 | -1 | | 2 | -2 | 2 | | 3 | 4 | -4 | | 4 | -8 | 8 | | 5 | 16 | -16 | +------+------+------+ Some syntax constraints apply within recursive CTE subqueries: The recursive SELECT part must not contain these constructs: Aggregate functions such as SUM() Window functions GROUP BY ORDER BY DISTINCT Prior to MySQL 8.0.19, the recursive SELECT part of a recursive CTE also could not use a LIMIT clause. This restriction is lifted in MySQL 8.0.19, and LIMIT is now supported in such cases, along with an optional OFFSET clause. The effect on the result set is the same as when using LIMIT in the outermost SELECT, but is also more efficient, since using it with the recursive SELECT stops the generation of rows as soon as the requested number of them has been produced. These constraints do not apply to the nonrecursive SELECT part of a recursive CTE. The prohibition on DISTINCT applies only to UNION members; UNION DISTINCT is permitted. The recursive SELECT part must reference the CTE only once and only in its FROM clause, not in any subquery. It can reference tables other than the CTE and join them with the CTE. If used in a join like this, the CTE must not be on the right side of a LEFT JOIN. These constraints come from the SQL standard, other than the MySQL-specific exclusions of ORDER BY, LIMIT (MySQL 8.0.18 and earlier), and DISTINCT. For recursive CTEs, EXPLAIN output rows for recursive SELECT parts display Recursive in the Extra column. Cost estimates displayed by EXPLAIN represent cost per iteration, which might differ considerably from total cost. The optimizer cannot predict the number of iterations because it cannot predict at what point the WHERE clause becomes false. CTE actual cost may also be affected by result set size. A CTE that produces many rows may require an internal temporary table large enough to be converted from in-memory to on-disk format and may suffer a performance penalty. If so, increasing the permitted in-memory temporary table size may improve performance; see Section 8.4.4, “Internal Temporary Table Use in MySQL”. Limiting Common Table Expression Recursion It is important for recursive CTEs that the recursive SELECT part include a condition to terminate recursion. As a development technique to guard against a runaway recursive CTE, you can force termination by placing a limit on execution time: The cte_max_recursion_depth system variable enforces a limit on the number of recursion levels for CTEs. The server terminates execution of any CTE that recurses more levels than the value of this variable. The max_execution_time system variable enforces an execution timeout for SELECT statements executed within the current session. The MAX_EXECUTION_TIME optimizer hint enforces a per-query execution timeout for the SELECT statement in which it appears. Suppose that a recursive CTE is mistakenly written with no recursion execution termination condition: WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte) SELECT * FROM cte; By default, cte_max_recursion_depth has a value of 1000, causing the CTE to terminate when it recurses past 1000 levels. Applications can change the session value to adjust for their requirements: SET SESSION cte_max_recursion_depth = 10; -- permit only shallow recursion SET SESSION cte_max_recursion_depth = 1000000; -- permit deeper recursion You can also set the global cte_max_recursion_depth value to affect all sessions that begin subsequently. For queries that execute and thus recurse slowly or in contexts for which there is reason to set the cte_max_recursion_depth value very high, another way to guard against deep recursion is to set a per-session timeout. To do so, execute a statement like this prior to executing the CTE statement: SET max_execution_time = 1000; -- impose one second timeout Alternatively, include an optimizer hint within the CTE statement itself: WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte) SELECT /*+ SET_VAR(cte_max_recursion_depth = 1M) */ * FROM cte; WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte) SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM cte; Beginning with MySQL 8.0.19, you can also use LIMIT within the recursive query to impose a maximum nuber of rows to be returned to the outermost SELECT, for example: WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte LIMIT 10000) SELECT * FROM cte; You can do this in addition to or instead of setting a time limit. Thus, the following CTE terminates after returning ten thousand rows or running for one thousand seconds, whichever occurs first: WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte LIMIT 10000) SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM cte; If a recursive query without an execution time limit enters an infinite loop, you can terminate it from another session using KILL QUERY. Within the session itself, the client program used to run the query might provide a way to kill the query. For example, in mysql, typing Control+C interrupts the current statement. Recursive Common Table Expression Examples As mentioned previously, recursive common table expressions (CTEs) are frequently used for series generation and traversing hierarchical or tree-structured data. This section shows some simple examples of these techniques. A Fibonacci series begins with the two numbers 0 and 1 (or 1 and 1) and each number after that is the sum of the previous two numbers. A recursive common table expression can generate a Fibonacci series if each row produced by the recursive SELECT has access to the two previous numbers from the series. The following CTE generates a 10-number series using 0 and 1 as the first two numbers: WITH RECURSIVE fibonacci (n, fib_n, next_fib_n) AS (SELECT 1, 0, 1 UNION ALL SELECT n + 1, next_fib_n, fib_n + next_fib_n FROM fibonacci WHERE n < 10) SELECT * FROM fibonacci; The CTE produces this result: +------+-------+------------+ | n | fib_n | next_fib_n | +------+-------+------------+ | 1 | 0 | 1 | | 2 | 1 | 1 | | 3 | 1 | 2 | | 4 | 2 | 3 | | 5 | 3 | 5 | | 6 | 5 | 8 | | 7 | 8 | 13 | | 8 | 13 | 21 | | 9 | 21 | 34 | | 10 | 34 | 55 | +------+-------+------------+ How the CTE works: n is a display column to indicate that the row contains the n-th Fibonacci number. For example, the 8th Fibonacci number is 13. The fib_n column displays Fibonacci number n. The next_fib_n column displays the next Fibonacci number after number n. This column provides the next series value to the next row, so that row can produce the sum of the two previous series values in its fib_n column. Recursion ends when n reaches 10. This is an arbitrary choice, to limit the output to a small set of rows. The preceding output shows the entire CTE result. To select just part of it, add an appropriate WHERE clause to the top-level SELECT. For example, to select the 8th Fibonacci number, do this: mysql> WITH RECURSIVE fibonacci ... ... SELECT fib_n FROM fibonacci WHERE n = 8; +-------+ | fib_n | +-------+ | 13 | +-------+ Date Series Generation A common table expression can generate a series of successive dates, which is useful for generating summaries that include a row for all dates in the series, including dates not represented in the summarized data. Suppose that a table of sales numbers contains these rows: mysql> SELECT * FROM sales ORDER BY date, price; +------------+--------+ | date | price | +------------+--------+ | 2017-01-03 | 100.00 | | 2017-01-03 | 200.00 | | 2017-01-06 | 50.00 | | 2017-01-08 | 10.00 | | 2017-01-08 | 20.00 | | 2017-01-08 | 150.00 | | 2017-01-10 | 5.00 | +------------+--------+ This query summarizes the sales per day: mysql> SELECT date, SUM(price) AS sum_price FROM sales GROUP BY date ORDER BY date; +------------+-----------+ | date | sum_price | +------------+-----------+ | 2017-01-03 | 300.00 | | 2017-01-06 | 50.00 | | 2017-01-08 | 180.00 | | 2017-01-10 | 5.00 | +------------+-----------+ However, that result contains “holes” for dates not represented in the range of dates spanned by the table. A result that represents all dates in the range can be produced using a recursive CTE to generate that set of dates, joined with a LEFT JOIN to the sales data. Here is the CTE to generate the date range series: WITH RECURSIVE dates (date) AS (SELECT MIN(date) FROM sales UNION ALL SELECT date + INTERVAL 1 DAY FROM dates WHERE date + INTERVAL 1 DAY WITH RECURSIVE ... ... SELECT * FROM employees_extended WHERE id IN (692, 4610) ORDER BY path; +------+-------+-----------------+ | id | name | path | +------+-------+-----------------+ | 4610 | Sarah | 333,198,29,4610 | | 692 | Tarek | 333,692 | +------+-------+-----------------+ Common Table Expressions Compared to Similar Constructs Common table expressions (CTEs) are similar to derived tables in some ways: Both constructs are named. Both constructs exist for the scope of a single statement. Because of these similarities, CTEs and derived tables often can be used interchangeably. As a trivial example, these statements are equivalent: WITH cte AS (SELECT 1) SELECT * FROM cte; SELECT * FROM (SELECT 1) AS dt; However, CTEs have some advantages over derived tables: A derived table can be referenced only a single time within a query. A CTE can be referenced multiple times. To use multiple instances of a derived table result, you must derive the result multiple times. A CTE can be self-referencing (recursive). One CTE can refer to another. A CTE may be easier to read when its definition appears at the beginning of the statement rather than embedded within it. CTEs are similar to tables created with CREATE [TEMPORARY] TABLE but need not be defined or dropped explicitly. For a CTE, you need no privileges to create tables. Page 21 13.3 Transactional and Locking Statements MySQL supports local transactions (within a given client session) through statements such as SET autocommit, START TRANSACTION, COMMIT, and ROLLBACK. See Section 13.3.1, “START TRANSACTION, COMMIT, and ROLLBACK Statements”. XA transaction support enables MySQL to participate in distributed transactions as well. See Section 13.3.8, “XA Transactions”.