Mysql copy table with indexes

13.1.18.3 CREATE TABLE ... Like the Statement Use TO CREATE A TABLE ... HOW to create a blank table based on the definition of another table, including all attributes of columns and indexes defined in the original table: CREATE TABLE new_tbl AS orig_tbl; A copy is created using the same version of the table store format as the original table. The SELECT privilege is required in the original table. Like only works for basic tables, not for views. Important: You cannot run create a table or create a table... Like when a lock table statement is in effect. CREATE TABLE ... Just like doing the same checks as creating a table and not just copy the .frm file. This means that if the current SQL mode differs from the mode in fact when the original table is created, the table definition may be considered invalid for the new mode, and the statement fails. To CREATE A TABLE ... For example, the destination table will collect the generated column information from the original table. CREATE TABLE ... Like will not preserve any DATA DIRECTORy or INDEX DIRECTORY table options that were specified for the original table, or any foreign key definitions. If the original table is a temporary table, create a table ... Like will not be let temporary. To create a temporary destination table, use create temporary table ... As. Page 2 13.1.18.4 CREATE TABLE ... SELECT Statement You can create one table from another by adding a select statement at the end to create a table statement: CREATE TABLE new_tbl [AS] SELECT * from orig_tbl; MySQL creates new columns for all elements in SELECT. For example: mysql&gt; CREATE TABLE test (a INT NOT NULL AUTO_INCREMENT, -&gt; PRIMARY KEY (a), KEY(b)) -&gt; ENGINE=MyISAM SELECT b,c FROM test2; This creates a MyISAM table with three columns, a, b, and c. The engine option is included to create a statement table and should not be used after SELECT; this would result in a syntax error. The same applies to other CREATE TABLE options, such as CHARSET. Note that the columns from the select statement are attached to the right side of the table, not overlap on it. Take the following example: mysql&gt; SELECT * FROM foo; +---+ | (n) | +---+ | 1. In 2005, +---+ mysql&gt; CREATE TABLE bar (m INT) SELECT n FROM foo; Query OK, 1 line affected (0.02 sec) Records: 1 Duplicates: 0 Warning: 0 mysql&gt; SELECT * From bar; +------+---+ | M.M. | (n) | +------+---+ | It is also the first time that a member of the public has been 1 In 2005, +------+---+1 row in set (0.00 s) For each row in the foo table, a row with foo values and default values for new columns is inserted in the bar. In the table resulting from create table ... SELECT, the columns named only in the CREATE TABLE section come first. Columns named in both parts or only in the SELECT section are then added. You can also override the SELECT column data type by specifying a column under CREATE TABLE. errors occur when copying data to a table, it is fell and was not created. Before select by IGNORE or REPLACE, you can specify how to process rows that duplicate unique key values. Ignore deletes rows that duplicate an existing row with a unique key value. Replace replaces new rows that have the same unique key value. If ignore or replace is not specified, duplicate unique key values result in an error. For more information, see Ignore the effect on making a statement. Because the order of rows in the underlying SELECT statements cannot always be determined, CREATE A TABLE ... IGNORE SELECT AND CREATE TABLE ... REPLACE SELECT statements are marked as unsafe for declaration-based replication. Such statements create a warning in the error log when you use a statement-based mode, and when you use MIXED mode, they are written to the binary log using a line-based format. See also section 16.2.1.1, Advantages and disadvantages of replication based on declarations and lines. CREATE TABLE ... SELECT does not automatically create any indexes for you. This is done deliberately to make the statement as alexeitable as possible. If you want to have indexes in the created table, you should enter them before the SELECT statement: mysql&gt; CREATE TABLE bar (UNIQUE (n)) SELECT n FROM foo; To CREATE A TABLE ... SELECT, the destination table does not contain information about whether the columns in the selected table are generated columns. The SELECT statement section cannot assign values to generated columns in the destination table. There may be some conversion of data types. For example AUTO_INCREMENT attribute is not hidden, and VARCHAR columns can become CHAR columns. The retrained attributes are NULL (or NOT NULL) and for those columns that have them, CHARACTER SET, SORT, COMMENT, AND DEFAULT CLAUSE. When creating a table with CREATE TABLE ... SELECT to make sure that you alias all function calls or expressions in the query. If you do not do so, creating a statement may fail or result in unwanted column names. CREATE TABLE artists_and_works SELECT ARTIST.NAME, COUNT(work.artist_id) AS NUMBER_OF_WORKS FROM ARTIST LEFT TO JOIN WORK TO ARTIST.ID =WORK.ARTIST_ID GROUP BY ARTIST.ID; You can also explicitly specify the data type for a column in the created table: CREATE TABLE foo (TINYINT NOT NULL) SELECT b +1 AS and FROM bar; To CREATE A TABLE ... SELECT, if it does not exist, is listed, and the destination table exists, nothing is inserted in the destination table, and the statement is not logged. To ensure that the binary log can be used to re-create the original table, MySQL does not allow concurrent inserts during create table ... Select. You cannot use the update as part of select in a statement, such as create a table new_table SELECT ... FROM old_table .... If you try, the statement will fail. Page 3 13.1.18.5 Foreign KEY RESTRICTIONS MySQL supports foreign keys, allow cross-references to related data in tables and foreign key constraints that help keep related data consistent. A foreign key relationship includes a parent table that contains the initial values of columns, and a child table with column values that refer to the values of the parent column. The foreign key constraint is defined in the child table. The basic syntax for defining foreign key constraints in the CREATE TABLE or ALTER TABLE statement contains the following: [CONSTRAINT [symbol]] FOREIGN KEY [index_name] (col_name, ...) REFERENCES tbl_name (col_name,...) [ON DELETE reference_option] [About update reference_option] reference_option: RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT Using a foreign key constraint is described in the following topics in this section: Naming a foreign key constraint is governed by the following rules: The constraint symbol value, if defined, is used. If the restriction symbol provision is not defined or the symbol is not included according to the CONSTRAINT keyword: A constraint name is automatically generated for the InnoDB tables. For NDB tables, the foreign key value index_name, if defined, is used. Otherwise, the name of the constraint is automatically generated. The constraint symbol value, if defined, must be unique in the database. The duplicate symbol results in an error similar to: ERROR 1005 (HY000): Unable to create test.fk1 table (errno: 121). Table and column identifiers in a foreign key ... THE REFERENCE CLAUSE can be quoted under backticks (). Alternatively, double quotation marks () can be used ANSI_QUOTES SQL mode is enabled. The lower_case_table_names variable setting shall also be taken into account in the data. Conditions and restrictions Foreign key restrictions are subject to the following conditions and restrictions: Parent and child tables must use the same storage tool and cannot be defined as temporary tables. Creating a foreign key limit requires reference privilege in the parent table. The corresponding columns in the foreign key and the referenced key must have similar data types. The size and sign of fixed precision types such as INTEGER and DECIMAL must be the same. The length of string types may not be the same. For non-binary (character) string columns, the character set and sort must be the same. MySQL supports foreign key references between one column and another in a table. (The column cannot have a foreign key reference to itself.) In these cases, the child record of the table refers to a dependent record within the same table. MySQL requires foreign key indexes and referenced keys, so checks for foreign keys can be quick and do not require checking tables. In the reference table, there must be an index in which the foreign key columns are listed as the first columns in the same order. Such an index is automatically created in the reference table if This index index Be quietly dropped later if you create another index that can be used to enforce foreign key restrictions. index_name, when administered, is used as described above. InnoDB allows a foreign key to refer to any index column or column group. However, there must be an index in the reference table in which the columns referenced by the first columns are in the same order. Hidden columns added to the index by InnoDB are also considered (see section 14.6.2.1, Group and secondary indices). The NDB requires an explicit unique key (or primary key) in any column that is referenced as a foreign key. InnoDB is not, which is an extension of the standard SQL. Index prefixes in foreign key columns are not supported. As a result, blob and text columns cannot be included in a foreign key because the indexes in these columns must always contain the length of the prefix. InnoDB does not currently support foreign keys for user-defined partition tables. This includes both parent and child tables. This limitation does not apply to NDB tables that are broken down by a key or linear key (single types of user division supported by the NDB storage engine); may have links to foreign key items or may be the target of such links. You cannot change a table in a foreign key relationship to use a different storage tool. To change the storage tool, you must first waive all foreign key restrictions. A foreign key constraint cannot refer to a virtual generated column. Before 5.7.16, the foreign key constraint cannot reference the secondary index defined in the virtual generated column. For information about how MySQL implementation of foreign key constraints differs from the SQL standard, see section 1.7.2.3, foreign key limitation differences. When an update or delete operation affects the value of a key in a parent table that has matching rows in a child table, the result depends on the reference action specified by ON UPDATE and the ON DELETE subsoil clause of the FOREIGN KEY clause. Referential actions include: CASCADE: Delete or update a row from the parent table and automatically delete or update the corresponding rows in the child table. On delete cascade and ON UPDATE CASCADE are supported. Do not define several CASCADE clauses when updating between two tables that operate in the same column in the parent table or in a child table. If both tables in a foreign key relationship define a FOREIGN KEY clause that makes both parent and child tables, the other must be defined for the other to succeed, the ON UPDATE or ON DELETE CASCADE clause must be defined for one FOREIGN KEY clause. If he update cascade or he delete cascade podkauza is defined for only one foreign key clause, cascading operations fail with an error. Cascading foreign key actions Triggers. SET NULL: Remove or update a row from the parent table and set a foreign foreign column or columns in a child null table. Both ON DELETE SET NULL and ON UPDATE SET NULL clauses are supported. If you specify a SET NULL action, make sure that you have unreported columns in the child table as NOT NULL. RESTRICT: Rejects the delete or update operation for the parent table. Specifying a restriction (or no action) is the same as omitting an ON DELETE or ON UPDATE clause. NO ACTION: Keyword from standard SQL. In MySQL, which corresponds to RESTRICT. MySQL Server rejects a delete or update operation for the parent table if the related foreign key value in the table is referenced. Some database systems have postponed checks, and no action is delayed. In MySQL, foreign key restrictions are checked immediately, so no action is the same as LIMIT. SET DEFAULT: This action is recognized by the MySQL Analyzer, but InnoDB and NDB reject table definitions containing ON DELETE SET DEFAULT or ON UPDATE SET default clauses. For storage engines that support foreign keys, MySQL refuses to all insert or update an operation that attempts to create a foreign key value in a child table if there is no matching candidate key value in the parent table. For on delete or on update that is not specified, the default action is always restricted. For NDB tables, ON UPDATE CASCADE is not supported if you reference the parent table primary key. From NDB 7.5.14 and NDB 7.6.10: For NDB tables, ON DELETE CASCADE is not supported if the child table contains one or more columns of any of the TEXT or BLOB types. (Bug #89511, Bug #27484882) InnoDB performs cascading operations using the first search depth algorithm in index records that corresponds to foreign key limitations. A foreign key constraint on a stored generated column cannot use cascade, SET NULL, or SET DEFAULT as a reference action when updating, nor can it use SET NULL or SET DEFAULT as an ON DELETE reference action. A foreign key constraint in the base column of a stored generated column cannot be used by cascade, SET NULL, or SET DEFAULT as on update or ON DELETE referential actions. In MySQL 5.7.13 and earlier, InnoDB does not allow the definition of foreign key constraints to cascade reference action in the base column of an indexed virtual generated column. This restriction is lifted in MySQL 5.7.14. In MySQL 5.7.13 and earlier, InnoDB does not allow you to define cascading referential actions on non-virtual foreign key columns that are explicitly included in the virtual index. This restriction is lifted in MySQL 5.7.14. Examples of foreign key constraints This simple example refers to parent and child tables through a single column foreign key: CREATE TABLE MASTER KEY (int NOT NULL id, PRIMARY KEY (id)) ENGINE = INNODB; CREATE TABLE CHILD (id INT, parent_id INT, INDEX (parent_id), CUDZÍ KĽÚČ (parent_id) ODKAZY parent(id) ON DELETE CASCADE) ENGINE = INNODB; Toto je je a more complex example in which product_order table contains foreign keys for two additional tables. One foreign key refers to an index of two columns in a product table. Other references one column index in the customer table: CREATE TABLE PRODUCT (CATEGORY INT NOT NULL, ID INT NOT NULL, PRICE DECIMAL, PRIMARY KEY (category, id)) ENGINE = INNODB; CREATE CUSTOMER TABLE (id INT NOT NULL, PRIMARY KEY (id)) ENGINE = INNODB; CREATE TABLE product_order (no INT NOT NULL AUTO_INCREMENT, product_category INT NOT NULL, product_id INT NOT NULL, CUSTOMER_ID INT NOT NULL, PRIMARY KEY(NO), INDEX (product_category, PRODUCT_ID), INDEX (customer_id), FOREIGN KEY (product_category, PRODUCT_ID) REFERENCES PRODUCT(CATEGORY, id) ON UPDATE CASCADE ON DELETE RESTRICT, FOREIGN KEY (customer_id) REFERENCES CUSTOMER(id) ENGINE=INNODB; Add foreign key constraints You can add a foreign key constraint to an existing table by using the following ALTER TABLE syntax: ALTER TABLE tbl_name ADD [CONSTRAINT [symbol]] FOREIGN KEY [index_name] (col_name, ...) REFERENCES tbl_name (col_name,...) [ON DELETE reference_option] [When updating reference_option] The foreign key can be separately referential (with reference to the same table). When you add a foreign key constraint to a table by using alter table, be sure to first create an index of the columns referenced by the foreign key. Unchaint foreign key restrictions You can omit the foreign key constraint by using the following ALTER TABLE: ALTER TABLE tbl_name DROP FOREIGN KEY fk_symbol; If the FOREIGN KEY clause defined the name of the constraint when you created the constraint, you can refer to that name to reduce the foreign key constraint. Otherwise, the constraint name was generated internally and you must use this value. To specify the name of a foreign key restriction, use the show create pane: mysql&gt; SHOW CREATE TABLE child\G ** row **** Table: Child table: CREATE TABLE child (id int(11) DEFAULT NULL, parent_id int(11) DEFAULT NULL, KEY par_ind (parent_id), FOREIGN KEY RESTRICTION child_ibfk_1 (parent_id) PARENTAL REFERENCES ('id') TO REMOVE CASCADE) ENGINE=INNODB DEFAULT CHARSET=latin1 mysql&gt; ALTER TABLE CHILD DROP FOREIGN KEY child_ibfk_1; Adding and dropping a foreign key in the same ALTER TABLE statement is supported for ALTER TABLE... ALGORITHM=IN PLACE. Not supported for ALTER TABLE ... ALGORITHM=COPY. The foreign key checker is controlled foreign_key_checks that is enabled by default. Typically, you leave this variable enabled during normal operation to enforce referential integrity. The foreign_key_checks has the same effect on NDB tables as for InnoDB tables. The foreign_key_checks is dynamic and supports both global and session ranges. For information on the use of system variables, see section 5.1.8 Using system variables checking foreign keys is useful when: Dropping a table referenced by a foreign key constraint. Referenced table can be omitted only after foreign_key_checks is disabled. When you drop a table, the constraints defined in the table are also omitted. Tranship tables in a different order than their foreign key relationships require. For example, mysqldump creates the correct table definitions in the dump file, including foreign key constraints for child tables. To make it easier to load dump files for tables with foreign key relationships, mysqldump automatically contains a statement in the output dump that foreign_key_checks. This allows you to import tables in any order if the dump file contains tables that are not sorted correctly for foreign keys. Disabling foreign_key_checks also speeds up the import operation by avoiding foreign key controls. Perform LOAD DATA operations to avoid checking foreign keys. Perform an ALTER TABLE operation in a table that has a foreign key relationship. When foreign_key_checks is disabled, foreign key constraints are ignored with the following exceptions: Re-creating a table that was previously skipped returns an error if the table definition does not comply with the foreign key constraints that refer to the table. The table must contain the correct column names and types. It must also have referenced key indices. If these requirements are not met, MySQL returns error 1005 that refers to errno:150 in the error message, which means that the foreign key constraint was not created correctly. Changing a table returns an error (errno: 150) if the foreign key definition is incorrectly created for the changed table. Cancel the index required by restricting a foreign key. The foreign key constraint must be removed before the index crashes. Create a foreign key limit where a column refers to a mismatched column type. Disabling foreign_key_checks has the following additional consequences: You are allowed to drop a database that contains tables with foreign keys that are referenced by tables outside the database. It is allowed to drop a table with foreign keys referenced by other tables. Enabling foreign_key_checks does not run a table data check, which means that rows added to the table while foreign_key_checks is turned off will become more consistent when you foreign_key_checks back on. Foreign key definitions and metadata Use the SHOW CREATE table: mysql&gt; SHOW CREATE TABLE child\G **1 to view the foreign key definition. row **** Table: child table: CREATE TABLE child (id int(11) DEFAULT NULL, parent_id int(11) DEFAULT NULL, KEY par_ind (parent_id), FOREIGN KEY CONSTRAINT child_ibfk_1 (parent_id) REFERENCES TO PARENT ('id') WHEN REMOVING CASCADE) ENGINE=InnoDB DEFAULT CHARSET=latin1 You can get information foreign buttons from the INFORMATION_SCHEMA. KEY_COLUMN_USAGE table. An example of a query against this table is shown here: mysql&gt; SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME FROM INFORMATION_SCHEMA. KEY_COLUMN_USAGE if REFERENCED_TABLE_SCHEMA is not NULL; +--------------+------------+-------------+-----------------+ | TABLE_SCHEMA — Sub-total | | TABLE_NAME — Sub-total | | COLUMN_NAME — Sub-total | | CONSTRAINT_NAME — Sub-total | | +--------------+------------+-------------+-----------------+ | Test | Child | parent_id — Sub-total | | child_ibfk_1 — Sub-total | | +--------------+------------+-------------+-----------------+ Information specific to InnoDB foreign keys can be obtained from the INNODB_SYS_FOREIGN and INNODB_SYS_FOREIGN_COLS table. Example queries are listed here: mysql&gt; SELECT * FROM INFORMATION_SCHEMA. INNODB_SYS_FOREIGN \G ** 1st row ***** ID: test/child_ibfk_1 FOR_NAME: test/child REF_NAME: test/parent N_COLS: 1 TYPE: 1 mysql&gt; SELECT * FROM INFORMATION_SCHEMA. INNODB_SYS_FOREIGN_COLS \G ** 1st Row *** ID: test/child_ibfk_1 FOR_COL_NAME: parent_id REF_COL_NAME: POS id: 0 In case of foreign key error involving InnoDB table (usually error 150 on MySQL Server), information about the latest foreign key error can be obtained by checking the output of SHOW ENGINE INNODB STATUS. mysql&gt; SHOW ENGINE INNODB STATUS\G ... ------------------------ LAST FOREIGN KEY ERROR ------------------------ 2014-10-16 18:35:18 0x7fc2a95c1700 Transaction: TRANSACTION 1814, ACTIVE 0 sec inserting mysql table in use 1, locked 1 4 lock struct (s), heap size 1136, 3 row lock(s), back log entries 3 MySQL thread id 2, OS thread handle 140474041767680, query id 74 localhost root update inserted into child values (NULL, 1) , (NULL, 2) , (NULL, 3) , (NULL, 4) , (NULL, 5) , (NULL, 6) Foreign key constraint fails for mysql table.'child': , CONSTRAINT child_ibfk_1 foreign key (parent_id) PARENTAL REFERENCES ('id') TO REMOVE CASCADE WHEN UPDATING CASCADE You try to add to a child table in index par_ind tuple: DATA TUPLE: 2 fields; 0: lens 4; hex 80000003; asc ;; 1: lens 4; hex 80000003; asc ;; But in the parent table 'mysql'.' parent', in the PRIMARY index, the closest match we can find is a record: physical record: n_fields 3; compact format; info bits 0 0: lens 4; hex 80000004; asc ;; 1: lens 6; hex 00000000070a; asc ;; 2: only 7; hex aa0000011d0134; asc 4; ... Warning ER_NO_REFERENCED_ROW_2 and ER_ROW_IS_REFERENCED_2 error messages for foreign key operations expose information about parent tables, even if the user has no permissions to access the parent table. To hide information about parent tables, you include the appropriate condition handlers in the application code and saved programs. Page 4 13.1.18.6 Silent column specification changes In some cases, MySQL quietly changes column specifications from specifications create table or alter table statement. These be changes to the data type, attributes associated with the data type, or index specification. All changes are subject to an internal row size limit of 65,535 minutes, which may cause some attempts to fail to change the data type. See section 8.4.7, Table column and row size limits. Columns that are part of the primary key are not reset, even if they are not declared this way. Trailing spaces are automatically removed from ENUM and SET memberships when a table is created. MySQL maps certain types of data used by other SQL database vendors to MySQL types. See section 11.9,Use of data types from other database engines. If you include a USE clause to specify an index type that is not allowed for the storage tool, but another index type is available that the engine can use without affecting the query results, the engine uses the available type. If strict SQL mode is not enabled, a VARCHAR column with a length specification greater than 65535 is converted to text, and a VARBINARY column with a length specification greater than 65535 is converted to a blob. Otherwise, an error occurs in one of these cases. Specifying a binary CHARACTER SET attribute for a character data type causes a column to be created as the corresponding binary data type: CHAR becomes BINARY, VARCHAR BECOMES VARBINARY, AND TEXT BECOMES BLOB. This does not occur for ENUM and SET data types; are created as declared. Assume that you specify a table by using the following definition: CREATE TABLE t (c1 VARCHAR(10) CHARACTERS SET binary, c2 TEXT CHARACTERS SET binary, c3 ENUM ('a','b','c') CHARACTER SET binary); The resulting table has the following definition: CREATE TABLE t (c1 VARBINARY (10), c2 BLOB, c3 ENUM('a','b','c') CHARACTER FILE BINARY); To determine whether mySQL has used a different data type than the specified data type, after you create or change the table issue description, or view the creation of the table. Some other changes to the data type may occur if you compress the table by using myisampack. See section 15.2.3.3,Characteristics of the compressed table. Page 5 13.1.18.7 CREATE TABLE and Generated COLUMNS CREATE TABLE supports the specification of generated columns. The values of the generated column are calculated from the expression included in the column definition. Generated columns are supported by the ndb storage tool starting with MySQL NDB Cluster 7.5.3. The following simple example shows a table that stores the lengths of the sides of the right triangles in the side and side columns and calculates the hypotension length in the sidec (square squares on the other sides): CREATE A TABLE TRIANGLE (SIDEA DOUBLE, SIDEB DOUBLE AS (SQRT(sidea* sidea + sideb * sideb)); INSERT VALUES (1,1), (3,4), (6,8) INTO THE TRIANGLE (SIDEA, SIDEB); The selection from the table gives the following result: mysql&gt; SELECT * From triangle; | sidea | sideb | sidec | +-------+-------+--------------------+ | 1. In 2005, on 1 January 2005, the Commission decided not to take any other part in the work of the Member States In 2005, 1.4142135623730951 | | 3. 4. In 2005, on 5 June 2005, the Commission decided not to take any other part in the work of the European Parliament and of the Council on the In 2005, on 8 November 2005, the Commission decided not to block the commission's In 2005, 10. +-------+-------+--------------------+ Each application that uses a triangular table has access to hypotension values without having to type an expression to calculate them. The generated column definitions have the following syntax: col_name data_type [GENERATED ALWAYS] AS (expr) [VIRTUAL | SAVED] [NOT NULL | NULL] [UNIQUE [KEY]] [[PRIMARY] KEY] [COMMENT 'string'] AS (expr) indicates that the column is generated and defines the expression used to calculate column values. As can prevent generated always to make the generated character of the column clearer. Constructors that are enabled or disabled in the expression are described later. A virtual or saved keyword indicates how column values are stored, which has implications for using a column: VIRTUAL: Column values are not saved, but are evaluated when rows are read, immediately after each before triggers. The virtual column has no storage space. InnoDB supports secondary indexes in virtual columns. See section 13.1.18.8, Secondary indices and generated columns. SAVED: Column values are evaluated and saved when rows are inserted or updated. A saved column requires storage and can be indexed. The default value is VIRTUAL if not a single keyword is specified. It is allowed to mix virtual and stored columns in a table. Other attributes may indicate whether a column is indexed or can be NULL, or can provide a comment. Generated column expressions must follow the following rules. An error occurs if the expression contains unauthorized constructors. Literally, deterministic built-in features and operators are enabled. A function is specified if, due to the same data in tables, multiple invokes produce the same result independently of the connected user. Examples of functions that are non-terministic and fail this definition: CONNECTION_ID(), CURRENT_USER(), NOW(). Stored features and user-defined functions are not allowed. Stored procedure and function parameters are not allowed. Variables (system variables, user-defined variables, and local variables stored by the program) are not allowed. Sub-homes are not allowed. The generated column definition can refer to other generated columns, but only those that occur earlier in the table definition. A generated column definition can refer to any base (ungenerated) column in a table, regardless of whether its definition occurs sooner or later. The AUTO_INCREMENT cannot be used in a generated column definition. You AUTO_INCREMENT cannot be used as a base column in a generated column definition. From MySQL 5.7.10, if an expression evaluation causes a truncated or provides incorrect input to the Create a statement table that ends in an error, and the DDL operation is rejected. If the evaluates the data type that differs from the declared column type, the implicit pressure on the declared type occurs according to the usual mySQL type conversion rules. See section 12.3 Conversion type in term rating. Note If any component of the expression depends on SQL mode, different results may occur for different uses of the table if the SQL mode is the same during all uses. To CREATE A TABLE ... For example, the destination table will collect the generated column information from the original table. To CREATE A TABLE ... SELECT, the destination table does not contain information about whether the columns in the selected table are generated columns. The SELECT statement section cannot assign values to generated columns in the destination table. Splitting of generated columns is allowed. See Split a table. A foreign key constraint on a stored generated column cannot use cascade, SET NULL, or SET DEFAULT as a reference action when updating, nor can it use SET NULL or SET DEFAULT as an ON DELETE reference action. A foreign key constraint in the base column of a stored generated column cannot be used by cascade, SET NULL, or SET DEFAULT as on update or ON DELETE referential actions. A foreign key constraint cannot refer to a virtual generated column. Triggers cannot use NEW.col_name use OLD.col_name to refer to generated columns. If the generated column is inserted, replaced, or updated explicitly, it is the only default value allowed for insert, REPLACE, and UPDATE. The generated column in the view is considered updatable because it can be associated with it. However, if such a column is updated explicitly, the only allowed value is DEFAULT. Generated columns have several usage cases, such as the following: Virtual generated columns can be used as a way to simplify and unify queries. A complex condition can be defined as a generated column and referenced from multiple queries on the table to ensure that all of them use exactly the same condition. Stored generated columns can be used as a materialized cache for complex conditions that are costly to calculate on the fly. Generated columns can simulate functional indexes: Use a generated column to define a functional expression and index it. This can be useful when working with columns of types that cannot be indexed directly, such as JSON columns. for a detailed example, see Indexing a generated column to provide a JSON column index. For stored generated columns, the disadvantage of this approach is that the values are saved twice; once as the value of the generated column and once in the index. If the generated column is indexed, Optimize recognizes query expressions that match the column definition and uses indexes from the column as needed while executing the query, even if the query does not point directly to the column by For details, see section 8.3.10, 8.3.10, Use generated column indexes. Example: Suppose table t1 contains columns first_name and last_name, and that applications often construct an integer by using an expression such as this: SELECT CONCAT(first_name,' ,last_name) AS full_name FROM t1; One way to avoid writing an expression is to create a v1 view on t1 that simplifies applications by allowing them to select full_name directly without using the expression: CREATE DISPLAY v1 AS SELECT*, CONCAT(first_name,' ',last_name) AS full_name FROM t1; SELECT full_name Z v1; The generated column also allows applications to select full_name directly without the need to define a view: CREATE TABLE t1 (first_name VARCHAR(10), last_name VARCHAR(10), full_name VARCHAR(255) AS (CONCAT(first_name,' ',last_name))); REMOVE full_name from t1; Page 6 13.1.18.8 Secondary indexes and generated InnoDB columns support secondary indexes in virtual generated columns. Other index types are not supported. A secondary index defined in a virtual column is sometimes referred to as a virtual index. A secondary index can be created on one or more virtual columns or in a combination of virtual columns and regular columns or stored generated columns. Secondary indexes that contain virtual columns can be defined as unique. When a secondary index is created on a virtual generated column, the generated column values materialize in the index records. If the index is a cover index (one that contains all columns retrieved by a query), the generated column values are retrieved from the materialized values in the index structure instead of calculating on the fly. There are additional write costs to consider when you use a secondary index in a virtual column because of the calculation performed when you total the value of a virtual column in secondary index records during insert and update operations. Even with additional write costs, secondary indexes in virtual columns may be more appropriate than generated stored columns that have materialized in the clustered index, resulting in larger tables that require more disk space and memory. If the secondary index is not defined in the virtual column, there are additional read costs because the virtual column values must be calculated each time you examine the column row. The values of the indexed virtual column are MVCC-logged to avoid unnecessary recompitation of the generated column values during the rollback or wipe operation. The length of logged value data is limited by the index key limit of 767 minutes for COMPACT and REDUNDANT row formats, and 3072 bytes for dynamic and compressed row formats. Adding or skipping a secondary index in a virtual column is an on-site operation. Before 5.7.16, the foreign key constraint cannot reference the secondary index defined in the virtual generated column. In mySQL 5.7.13 and older things InnoDB define foreign key key constraint cascading reference action in the base column of the indexed generated virtual column. This restriction is lifted in MySQL 5.7.14. Indexing a generated column to provide a JSON column index As shown elsewhere, JSON columns cannot be indexed directly. If you want to create an index that indirectly refers to such a column, you can define a generated column that extracts the information that should be indexed, and then create an index in the generated column, as shown in this example: mysql&gt; CREATE TABLE jemp (-&gt; c JSON, -&gt; g INT GENERATED ALWAYS AS (c-&gt;$.id), -&gt; INDEX i (g) -&gt;); Query OK, 0 rows affected (0.28 sec) mysql&gt; insert in jeep (c) values &gt; ('{id: 1, name: Fred}), ({id: 2, name: Wilma}), &gt; ('{'id': 3, name: Barney}'), ({id: 4, name: Betty}'); Query OK, 4 rows affected (0.04 sec) Records: 4 Duplicates: 0 Warning: 0 mysql&gt; SELECT c-&gt;&gt;$.name as name &gt; from the soda where g &gt; 2; +--------+ | Name | +--------+ | Barney | | Betty | +--------+ 2 rows in mysql kit (0.00 s) &gt; EXPLAIN SELECT c-&gt;&gt;$.name AS NAME &gt; FROM jemp WHERE g &gt; 2\G ** row **** id: 1 select_type: SIMPLE TABLE: jemp partitions: NULL TYPE: range possible_keys: i key: i key_len:5 ref: NULL rows: 2 filtered: 100.00 Extra: Usage, where 1 line in the file, 1 warning (0.00 s) mysql&gt; SHOW WARNINGS\G ** line *** Level: Note code: 1003 Message: /* select #1 */ select json_unquote(json_extract('test'.' jemp'.' c','$.. name)) AS name from the test.jemp, where (test. jemp'.' g' &gt; 2) 1 row in set (0.00 s) (We have wrapped the output from the last statement in this example to fit in the display area.) Operator -&gt; is supported in MySQL 5.7.9 and later. Operator -&gt;&gt; is supported starting with MySQL 5.7.13. If you use the EXPLAIN function in a SELECT statement or other SQL statement that contains one or more expressions that are used by the operator -&gt; or -&gt;&gt;, those expressions will be translated into their equivalents using the JSON_EXTRACT() function and (if applicable) JSON_UNQUOTE(), as shown here in the SHOW WARNING output immediately after this, please explain the statement: mysql&gt; EXPLAIN SELECT c-&gt;&gt;$.name &gt; FROM jemp WHERE g &gt; 2 ORDER BY c-&gt;$.name\G ** row **** id: 1 select_type: SIMPLE TABLE: jemp partitions: TYPE NULL: range possible_keys: i key: i key_len: 5 ref: NULL rows: 2 filtered: 100.00 Extra: Use where; Use fileort 1 line in file, 1 warning (0.00 sec) mysql&gt; SHOW WARNINGS \G ** line *** Level: Code notes: 1003 Message: /* select #1 */ select json_unquote(json_extract('test'.' jemp'.' c','$.name')) AS 'c-&gt;&gt;$.name. ' from 'test'.' where ('test'.' jemp'.' g' &gt; 2) ranked by json_extract (test. jemp'.' c','$.name') 1 line in the file (0.00 sec) See operator descriptions -&gt; and -&gt;&gt;, as well as functions JSON_EXTRACT() and JSON_UNQUOTE() for further information and examples. This technique can also be used to provide indexes that indirectly refer to columns of other types that cannot be indexed directly, such as GEOMETRY columns. JSON columns and indirect indexing in the NDB cluster it is also possible to use indirect indexing of JSON columns in MySQL NDB Cluster 7.5.3 and later, under the following conditions: NDB handles the JSON column value internally as a BLOB. This means that any NDB table that has one or more JSON columns must have a primary key, otherwise it cannot be recorded in a binary log. The NdB storage tool does not support indexing virtual columns. Because the default value for generated columns is VIRTUAL, you must explicitly specify the generated column to which you want the indirect index to be used as saved. Create the table statement used to create the table jempn here is the version of the jemp table shown above, with modifications making it compatible with the NDB: CREATE TABLE jempn (BIGINT (20) NOT NULL AUTO_INCREMENT PRIMARY KEY, c JSON NULL DEFAULT, g INT GENERATED ALWAYS AS (c-&gt;$.name) SAVED, INDEX I (g) ENGINE = NDB; We can fill in this table with the following insert statement: INSERT INTO jempn (a, c) VALUES (NULL, {id: 1, name: Fred}), (NULL, {id: 2, name: Wilma}), (NULL, {id: 3, name: Barney} (NULL, {id: 4, name: Betty} (NULL, {id: 4, name: Betty}); Now ndb can use index i as shown here: mysql&gt; EXPLAIN SELECT c-&gt;&gt;$.name AS name from kde jempn g &gt; 2 G **.1. row **** id: 1 select_type: SIMPLE TABLE: jempn partitions: p0,p1 type: range possible_keys: i key: i key_len:5 ref: NULL rows: 3 filtered: 100.00 Extra: Use of printed state space ('test'.' jempn'.' g' &gt; 2) 1 line in set, 1 warning (0.00 s) mysql&gt; SHOW WARNINGS\G ** line *** Level: Note code: 1003 Message: /* select #1 */ select json_unquote(json_extract('test'.' jempn'.' c','$.name')) AS name from test.. where ('test'.' jempn'.' g' &gt; 2) 1 row in a file (0.00 sec) You should keep in mind that the stored generated column uses DataMemory, and that the index in such column uses IndexMemory. Page 7 13.1.18.9 Setting NDB_TABLE Options in MySQL NDB Cluster 7.5.2 and later, a comment table in the CREATE TABLE or CHANGE TABLE statement can also be used to specify NDB_TABLE option that consists of one or more name-value pairs, separated by commas if necessary, after NDB_TABLE=. The complete syntax of syntax names and values is displayed here: COMMENT=NDB_TABLE=ndb_table_option[,ndb_table_option[,...]] ndb_table_option: { NOLOGGING={1 | 0} | READ_BACKUP={1 | 0} | PARTITION_BALANCE={FOR_RP_BY_NODE | FOR_RA_BY_NODE — Sub-total | | FOR_RP_BY_LDM — Sub-total | | FOR_RA_BY_LDM) FOR_RA_BY_LDM_X_2 — Sub-total | | FOR_RA_BY_LDM_X_3) FOR_RA_BY_LDM_X_4} | | 0} } Spaces are not allowed within the specified string. The string is insensitive to cases. The four NDB table options that can be set as part of a comment in this way are described in more detail in the following few paragraphs. NOLOGGING: Using 1 corresponds to having ndb_table_no_logging, but has no real effect. Provided as a placeholder, mostly for the completeness of ALTER TABLE statements. READ_BACKUP: Setting this option to 1 has the same effect as if ndb_read_backup were enabled; allows reading from any replica. Doing so greatly improves the performance of reads from the desk at a relatively small cost of writing performance. Starting with MySQL NDB Cluster 7.5.3, you can set up READ_BACKUP for existing tables online (Bug #80858, Bug #23001617), using alter table statements similar to one of those listed here: ALTER TABLE... ALGORITHM=INPLACE, COMMENT=NDB_TABLE=READ_BACKUP=1; ALTER TABLE ... ALGORITHM=INPLACE, COMMENT=NDB_TABLE=READ_BACKUP=0; Before MySQL NDB Cluster 7.5.4, setting READ_BACKUP 1 also caused FRAGMENT_COUNT_TYPE set to ONE_PER_LDM_PER_NODE_GROUP. For more information about the ALTER TABLE algorithm option, see section 20.5.11, online operations with alter tables in the NDB cluster. PARTITION_BALANCE: Provides additional control over the assignment and location of partitions. The following four schemas are supported: FOR_RP_BY_NODE: One partition per node. Only one LDM per node stores the primary partition. Each partition is stored in the same LDM (same ID) on all nodes. FOR_RA_BY_NODE: One area per node group. Each node stores one partition, which can be either a primary replica or a backup replica. Each partition is stored in the same LDM on all nodes. FOR_RP_BY_LDM: One partition for each LDM on each node; default value. This is the same behavior as before MySQL NDB Cluster 7.5.2, except for slightly different LDM mapping areas, starting with LDM 0 and placing one partition on the group node, then switching to the next LDM. In MySQL NDB Cluster 7.5.4 and later, this is the setting used if READ_BACKUP is set to 1. (Bug #82634, Bug #24482114) FOR_RA_BY_LDM: One partition on the LDM in each node group. These partitions can be primary or backup partitions. Before MySQL NDB Cluster 7.5.4, this setting was used if READ_BACKUP was set to 1.FOR_RA_BY_LDM_X_2: Two partitions on the LDM in each node group. These partitions can be primary or backup partitions. This setting was added in NDB 7.5.4. FOR_RA_BY_LDM_X_3: Three partitions on the LDM in each node group. These partitions can be primary or backup partitions. This setting was added in NDB 7.5.4. FOR_RA_BY_LDM_X_4: Four partitions on the LDM in each node group. These partitions can be primary or backup partitions. This setting was added in NDB 7.5.4. Starting with NDB 7.5.4, PARTITION_BALANCE is the preferred interface for setting sections per table. Using MAX_ROWS force the number of partitions to become obsolete as NDB 7.5.4, continues to be supported in NDB 7.6 for backward compatibility, but is subject to deletion in a future release of the MySQL NDB cluster. (Bug #81759, Bug #23544301) Before MySQL NDB Cluster 7.5.4, PARTITION_BALANCE was named FRAGMENT_COUNT_TYPE, and accepted as its value one of (in the same order as the list just displayed) ONE_PER_NODE, ONE_PER_NODE_GROUP, ONE_PER_LDM_PER_NODE, or ONE_PER_LDM_PER_NODE_GROUP. (Bug #81761, Bug #23547525) FULLY_REPLICATED whether the table is fully replicated, that is, whether each data node has a full copy of the table. To enable full replication of the table, use FULLY_REPLICATED=1. This setting can also be controlled using the ndb_fully_replicated system. By default, setting it to ON enables the option for all new NDB tables; The default is OFF, which maintains previous behavior (as in MySQL NDB Cluster 7.5.1 and earlier, before support for fully replicated tables was introduced). The ndb_data_node_neighbour variable is also used for fully replicated tables to ensure that when access to a fully replicated table is made, we access a data node that is local to this MySQL server. An example of creating a table statement by using such a note when creating an NDB table is here: mysql&gt; CREATE TABLE t1 (&gt; c1 INT NOT NULL AUTO_INCREMENT PRIMARY KEY, &gt; c2 VARCHAR(100), &gt; c3 VARCHAR(100) &gt; ENGINE=NDB &gt; COMMENT=NDB_TABLE=READ_BACKUP=0,PARTITION_BALANCE=FOR_RP_BY_NODE; The comment appears as part of the ouput show to create the table. Comment text is also available from the MySQL schema table information query, as in this example: mysql&gt; SELECT TABLE_NAME, TABLE_SCHEMA, TABLE_COMMENT &gt; FROM INFORMATION_SCHEMA. TABLES, TABLE_NAME =t1\G ** row **** TABLE_NAME: t1 TABLE_SCHEMA: test TABLE_COMMENT: NDB_TABLE=READ_BACKUP=0.PARTITION_BALANCE=FOR_RP_BY_NODE 1 row in set (0.01 s) This comment syntax is also supported with ALTER TABLE statements for NDB tables. as shown here: mysql&gt; ALTER TABLE t1 COMMENT=NDB_TABLE=PARTITION_BALANCE=FOR_RA_BY_NODE; Query OK, 0 rows affected (0.40 sec) Records: 0 Duplicates: 0 Warning: 0 Starting with NDB 7.6.15, column TABLE_COMMENT displays the comment that is required to re-create the table because it is after an alter table statement like this: mysql&gt; SELECT TABLE_NAME, TABLE_SCHEMA, TABLE_COMMENT -&gt; FROM INFORMATION_SCHEMA. TABLES, TABLE_NAME =t1\G ** row *** TABLE_NAME: t1 TABLE_SCHEMA: test TABLE_COMMENT: NDB_TABLE=READ_BACKUP=0.PARTITION_BALANCE=FOR_RP_BY_NODE 1 row in set (0.01 s)mysql&gt; SELECT TABLE_NAME, TABLE_SCHEMA, TABLE_COMMENT &gt; of INFORMATION_SCHEMA. TABLES, TABLE_NAME =t1; | TABLE_NAME — Sub-total | | TABLE_SCHEMA — Sub-total | | TABLE_COMMENT — Sub-total | | +------------+--------------+--+ | t1 | This appropriation is not used for the NDB_TABLE =PARTITION_BALANCE=FOR_RA_BY_NODE | | t1 | This appropriation is not used for the expenditure incurred by +------------+--------------+--+2 rows in a set (0.01 s) Keep in mind that the table comment used with ALTER TABLE replaces any existing comments that the table may have. mysql&gt; ALTER TABLE t1 COMMENT=NDB_TABLE=PARTITION_BALANCE=FOR_RA_BY_NODE; Query OK, 0 rows affected (0.40 sec) Records: 0 Duplicates: 0 Warning: 0 mysql&gt; SELECT TABLE_NAME, TABLE_SCHEMA, TABLE_COMMENT &gt; from INFORMATION_SCHEMA. TABLES, TABLE_NAME =t1; +------------+--------------+--+ | TABLE_NAME — Sub-total | | TABLE_SCHEMA — Sub-total | | TABLE_COMMENT — Sub-total | | +------------+--------------+--+ | t1 | This appropriation is not used for the NDB_TABLE =PARTITION_BALANCE=FOR_RA_BY_NODE | | t1 | This appropriation is not used for the expenditure incurred by +------------+--------------+--+2 rows in set (0.01 s) Before NDB 7.6.15, the comment table used with ALTER TABLE replaced any existing comments that the table might have had. This meant that (for example) the READ_BACKUP value was not transferred to a new note set by alter table statement, and that all unspecified values were returned to their default settings. (BUG#30428829) There was therefore no longer any way to use SQL to retrieve the value previously set for the comment. To keep returning comment values to the default settings, you needed to keep all of these values from the existing comment string and include them in the comment uploaded by alter TABLE. You can also see the value of the PARTITION_BALANCE option in the ndb_desc. ndb_desc also shows whether the READ_BACKUP and FULLY_REPLICATED options are set for the table. For more information, see the description of this program. Page 8 13.1.19 CREATE TABLESPACE statement to create tablespace tablespace_name InnoDB and NDB: ADD DATAFILE file_name InnoDB only: [FILE_BLOCK_SIZE = value] NDB only: USE LOGFILE GROUP logfile_group [EXTENT_SIZE [=] extent_size] [INITIAL_SIZE [INITIAL_SIZE =] initial_size] [AUTOEXTEND_SIZE [=] autoextend_size] [MAX_SIZE [=] max_size] [NODEGROUP [=] nodegroup_id] [WAIT] [COMMENT [=] 'string'] InnoDB and NDB: [ENGINE [=] engine_name] This statement is used to create a table space. The exact syntax and semantics depend on the storage tool you are using. In standard Editions of MySQL 5.7, it is always an InnoDB tabular space. MySQL NDB Cluster 7.5 also supports tablespaces using the NDB storage engine in addition to those used by InnoDB. Use the CREATE TABLESPACE syntax to create general table spaces. A general table space is a shared table space. It can contain multiple tables and supports all table row formats. General table can be created in a place that is independent of or independent of the data After you create the InnoDB General Panespace, you can use create table tbl_name ... TABLESPACE [=] tablespace_name OR ALTER TABLE tbl_name TABLESPACE [=] tablespace_name table space. For more information, see section 14.6.3.3 of the General Spaces table. NDB Cluster Reflections This statement is used to create a table space that can contain one or more data files and provides storage for NDB cluster disk data tables (see section 20.5.10, NDB cluster disk data tables). A single data file is created and added to the table space by using this statement. Additional data files can be added to the table space by using the ALTER TABLESPACE statement (see section 13.1.9, ALTER TABLESPACE statement). Note All NDB objects on the disk cluster share the same namespace. This means that each disk data object must be uniquely named (and not just every disk data object of that type). For example, you cannot have a spreadsheet space and a group of log files with the same name, or a table space and a data file with the same name. A log group of one or more UNDO log files must be associated with the panespace to be created with the USE LOGFILE GROUP clause. logfile_group, an existing log file group must be created with the CREATE LOGFILE GROUP (see section 13.1.15, CREATE LOGFILE GROUP Statement). Multiple table spaces can use the same set of log files to write back. When EXTENT_SIZE or INITIAL_SIZE, you can optionally track a number with a single letter abbreviation of the order of magnitude similar to those used in my.cnf. In general, this is one of the letters M (for megabytes) or G (for gigabytes). INITIAL_SIZE and EXTENT_SIZE shall be rounded as follows: EXTENT_SIZE shall be rounded up INITIAL_SIZE to the nearest whole multiple of 32K. this result shall be rounded up to the nearest full EXTENT_SIZE (after each rounding). Note The NDB reserves 4% of the table space for data node restart operations. This reserved space cannot be used to store data. This restriction shall apply from NDB 7.6. Rounding only described is done explicitly, and a warning is issued to MySQL Server each time such rounding is performed. Rounded values are also used by the NDB kernel to calculate INFORMATION_SCHEMA. SETS of column values and other purposes. However, to avoid an unexpected result, we recommend that you always use whole multiples of 32K when embellishing these options. When create TABLESPACE is used with engine [=] NDB, tablespace and related data file are created on each cluster data node. You can verify that data files have been created and get information about them by querying INFORMATION_SCHEMA. TABLE FILES. (See example below in parts.) (See section 23.9, INFORMATION_SCHEMA files.) ADD DATAFILE options: Defines name name the data file; this option is always necessary. Data file_name, including any specified path, must be quoted with one or two quotation marks. File names (not counting file extensions) and directory names must have at least one number of subseating numbers. Zero-length file names and directory names are not supported. Because there are significant differences in how InnoDB and NDB handle data files, these two memory tools are included separately in the discussion that follows. InnoDB data files. The InnoDB table space supports only one data file, the name of which must contain the .ibd extension. By default, a data file is created in the MySQL data directory (datadir) for the InnoDB tabular space. If you want to place the data file in a location other than the default location, you include the absolute path to the directory or path relative to the default location. When the innoDB panespace is created outside the data directory, an ISL file is created in the data directory. To avoid conflicts with implicitly created file table spaces on a table, creating a generic InnoDB table space in a subdirectory below the data directory is not supported. When you create an InnoDB generic space table outside the data directory, the directory must exist before you create the table space. In MySQL 5.7, ALTER TABLESPACE is not supported by InnoDB. NDB data files. NDB tablespace supports multiple data files that can have any legal file names; additional data files can be added to the NDB cluster tablespace after it is created by using the ALTER TABLESPACE statement. By default, the NDB table space data file is created in the directory of the data node file system, that is, a directory named ndb_nodeid_fs/TS under the Data Node (DataDir), where the node of the data node is a data node. If you want to place the data file in a location other than the default location, you include the absolute path to the directory or path relative to the default location. If the specified directory does not exist, the NDB tries to create it; the system user account under which the data node process is running must have the appropriate permissions to do so. When determining the path used for a data file, the NDB does not stitch the ~ (tilde) character. When multiple nodes of data are snowing on the same physical host, the following considerations apply: You cannot enter an absolute path when you create a data file. You cannot create table space data files outside the data node file directory unless each data node has a separate data directory. If each data node has its own data directory, data files can be created anywhere in that directory. If each data node has its own data directory, it may also be possible to create a data file outside the node's data directory by using a relative path, as long as this path is resolved to a unique location in the host file system for each data node running on that data node. FILE_BLOCK_SIZE: This option, which is specific to and is ignored by the NDB-defined block size for the table space data file. Values can be entered in bytes or kilobytes. For example, a block size of 8 kilobytes can be specified as 8192 or 8K. If you don't specify this option, FILE_BLOCK_SIZE the default innodb_page_size value. FILE_BLOCK_SIZE required to use the Table Space to store compressed InnoDB tables (ROW_FORMAT=COMPRESSED). In this case, when you create a space, FILE_BLOCK_SIZE define the table space. If FILE_BLOCK_SIZE value equals innodb_page_size, the Table space can only contain tables with uncompressed row formats (COMPACT, REDUNDANT, and DYNAMIC). Tables with a compressed row format have a different physical page size than uncompressed tables. Therefore, compressed tables cannot coexist in the same table space as uncompressed tables. If you want the general table space to contain compressed tables, FILE_BLOCK_SIZE must be specified, and FILE_BLOCK_SIZE must be a valid compressed page size relative to the innodb_page_size value. The physical page size of the compressed table (KEY_BLOCK_SIZE) must be equal to FILE_BLOCK_SIZE/1024. For example, if innodb_page_size = 16K and FILE_BLOCK_SIZE = 8K, the KEY_BLOCK_SIZE table must be 8. USING LOGFILE GROUP: Required for NDB, this is the name of the log file group previously created by using create logfile group. It is not supported for InnoDB, where it fails with an error. EXTENT_SIZE: This option is NDB-specific and is not supported by InnoDB, where it fails with an error. EXTENT_SIZE the size of the ranges used by any files belonging to the table space in the orders. The default value is 1M. The minimum size is 32K, and the theoretical maximum is 2G, although the practical maximum size depends on many factors. In most cases, changing the size of the range has no measurable effect on performance, and a default value is recommended for all but the most unusual situations. A range is a disk space allocation drive. One range is filled with as much data as this range can contain before the next range is used. Theoretically, up to 65,535 (64 K) can be used for a data file; recommended maximum is 32,768 (32 K). The recommended maximum size for a single data file is 32G, i.e. 32 K × 1 MB per range. Moreover, once a range is allocated to a given area, it cannot be used to store data from another area; range cannot store data from multiple partitions. This means, for example, that a table space with a single datafile whose INITIAL_SIZE (described in the following item) is 256 MB and whose EXTENT_SIZE is 128 M has only two ranges, so it can be used to store data from no more than two different areas of the disk data table. You can see how many ranges remain free data file by querying the INFORMATION_SCHEMA. files, thus deriving an estimate of how much space remains free in the file. For more discussions and examples, see section 23.9 file INFORMATION_SCHEMA table. INITIAL_SIZE: This option is NDB-specific and is not supported by innoDB, where it fails with an error. The INITIAL_SIZE sets the total size in the sub-orders of the data file that was specific by using ADD DATATFILE. After you create this file, you cannot resize it. However, you can add more data files to the table space using ALTER TABLESPACE... ADD A DATA FILE. INITIAL_SIZE is optional; its default value is 134217728 (128 MB). On 32-bit systems, the maximum supported value INITIAL_SIZE 4294967296 (4 GB). AUTOEXTEND_SIZE: Currently ignored MySQL; reserved for possible future use. It has no effect on any release of MySQL 5.7 or MySQL NDB Cluster 7.5, regardless of the storage engine used. MAX_SIZE: Currently ignored MySQL; reserved for possible future use. It has no effect on any release of MySQL 5.7 or MySQL NDB Cluster 7.5, regardless of the storage engine used. NODEGROUP: Currently ignoring MySQL; reserved for possible future use. It has no effect on any release of MySQL 5.7 or MySQL NDB Cluster 7.5, regardless of the storage engine used. WAIT: Currently ignored MySQL; reserved for possible future use. It has no effect on any release of MySQL 5.7 or MySQL NDB Cluster 7.5, regardless of the storage engine used. COMMENT: Currently ignored MySQL; reserved for possible future use. It has no effect on any release of MySQL 5.7 or MySQL NDB Cluster 7.5, regardless of the storage engine used. ENGINE: Defines a storage engine that uses a tabular space, which is engine_name name of the storage engine. Currently, the standard version of MySQL 5.7 is supported only by the InnoDB storage engine. MySQL NDB Cluster 7.5 supports both NDB and InnoDB tablespaces. The value of default_storage_engine value is used for ENGINE if this option is not specified. For rules on naming MySQL tabular spaces, see section 9.2 Schema object names. In addition to these rules, a slash character (/) is not allowed, nor can you use names starting innodb_ because this prefix is reserved for system use. Table spaces do not support temporary tables. innodb_file_per_table, innodb_file_format and innodb_file_format_max settings have no effect on CREATE TABLESPACE operations. innodb_file_per_table do not need to turn it on. Table general spaces support all table row formats, regardless of file format settings. Likewise, general tablespaces support adding tables of any row format by using CREATE TABLE... TABLESPACE, regardless of the file format setting. innodb_strict_mode does not apply to general premises The rules for the management of the desktop space are strictly enforced, regardless of If the CREATE TABLESPACE parameters are incorrect or incompatible, the operation fails regardless of innodb_strict_mode settings. When a table is added to a general table space using create table ... TABLESPACE or ALTER TABLE ... TABLESPACE, innodb_strict_mode is ignored, but the statement evaluates how innodb_strict_mode is enabled. Use drop panespace to delete the table space. All tables must be skipped from the table space by using DROP TABLE before the table space falls. Before dropping the NDB Cluster tablespace you must also delete all your data files using one or more ALTER TABLESPACE... DROP DATATFILE statement. See Section 20.5.10.1, NDB cluster disk data objects. All parts of the InnoDB table added to the general InnoDB table space are located in the general table space, including indexes and BLOB pages. For an NDB table associated with a table space, only those columns that are not indexed are stored on the disk and actually use table space data files. Indexes and indexed columns for all NDB tables are always stored in memory. Like the table system space, truncated or discarded tables stored in the general table space creates free space internally in the generic data file in the .ibd table, which can only be used for new InnoDB data. Space is not freed back into the operating system because it is on the table for the file. The general table space is not associated with any database or schema. ALTER TABLE ... DISCARD TABLESPACE AND CHANGE THE TABLE ... IMPORT TABLESPACE is not supported for tables that belong to the general table space. The server uses metadata locking at the table space level for the DDL that refers to general table spaces. For comparison, the server uses table-level metadata to lock the DDL, which references the file to the tablespaces table. You cannot change a generated or existing table space to a generic table space. Tables stored in the general table space can only be opened in mySQL 5.7.6 or later to add new table flags. There is no conflict between the generic table space names and the table space names by table. The /character, which is located in table space names by table, is not allowed in generic table space names. mysqldump and mysqlpump does not throw out the InnoDB CREATE TABLESPACE statement. This example demonstrates creating a general table space and adding three uncompressed tables to different row formats. mysql&gt; CREATE TABLESPACE 'ts1' ADD DATAFILE 'ts1.ibd' ENGINE=INNODB; mysql&gt; CREATE TABLE t1 (c1 INT PRIMARY KEY) TABLESPACE ts1 ROW_FORMAT=REDUNDANT; mysql&gt; CREATE TABLE t2 (c1 INT PRIMARY KEY) TABLESPACE ts1 ROW_FORMAT=COMPACT; mysql&gt; CREATE TABLE t3 (c1 INT PRIMARY KEY) ts1 ROW_FORMAT=DYNAMIC; This example demonstrates creating a general table space and adding a compressed table. The example assumes innodb_page_size default value FILE_BLOCK_SIZE 16 K. z 8. mysql&gt; CREATE TABLESPACE 'ts2' ADD DATAFILE 'ts2.ibd' FILE_BLOCK_SIZE = 8192 Engine =InnoDB; mysql&gt; CREATE TABLE t4 (c1 INT PRIMARY KEY) TABLESPACE ts2 ROW_FORMAT=COMPRESSED KEY_BLOCK_SIZE=8; For example, NDB assume that you want to create a data table space on an NDB cluster disk named Myts by using a data sála named mydata-1.dat. The NDB table space always requires the use of a log group consisting of one or more back log files. For this example, we first create a log file group called mylg that contains one back long file named myundo-1.dat, using the CREATE LOGFILE GROUP statement here: mysql&gt; CREATE LOGFILE GROUP myg1 -&gt; ADD UNDOFILE 'myundo-1.dat' -&gt; ENGINE=NDB; Query OK, 0 rows affected (3.29 sec) Now you can create a panespace previously described using the following statement: mysql&gt; CREATE TABLESPACE myTS -&gt; ADD DATAFILE 'mydata-1.dat' -&gt; USE LOGFILE GROUP myLG -&gt; ENGINE = NDB; Query OK, 0 rows affected (2.98 sec) Now you can create a disk data table by using create a table statement with TABLESPACE and disk storage options, similar to what is stated here: mysql&gt; CREATE TABLE mytable (-&gt; id INT UNSIGNED NO NULL AUTO_INCREMENT PRIMARY KEY, -&gt; lname VARCHAR(50) NOT NULL, -&gt; fname VARCHAR(50) NOT NULL, -&gt; dob DATE NO NULL , -&gt; LINKED DATE NOT NULL , -&gt; INDEX(last_name, first_name) -&gt;) -&gt; TABLESPACE MYTS STORAGE DISK -&gt; ENGINE =NDB; Query OK, 0 rows affected (1.41 sec) It is important to note that only dob and linked columns from mytable are actually stored on the disk, given that the id, lname, and fname columns are indexed. As mentioned above, when create TABLESPACE is used with engine [=] NDB, tablespace, and related data file are created on each node of the NDB cluster data. You can verify that data files have been created and get information about them by querying INFORMATION_SCHEMA. TABLE FILES as shown here: mysql&gt; SELECT FILE_NAME, FILE_TYPE, LOGFILE_GROUP_NAME, STATUS, EXTRA -&gt; FROM INFORMATION_SCHEMA. FILES -&gt; KDE TABLESPACE_NAME = 'myTS'; +--------------+------------+--------------------+--------+----------------+ | file_name — Sub-total | | file_type — Sub-total | | logfile_group_name — Sub-total | | Situation | additional | +--------------+------------+--------------------+--------+----------------+ | mydata-1.dat | DATA SET | myLG | NORMAL VALUE | CLUSTER_NODE=5 | | mydata-1.dat | DATA SET | myLG | NORMAL VALUE | CLUSTER_NODE =6 | | 1 (a) TABLE SPACE | myLG | NORMAL VALUE | 1 point +--------------+------------+--------------------+--------+----------------+3 rows in a set (0.01 seconds) For more information and examples, see 20.5.10.1, NDB Cluster Disk Data Objects. Page 9 13.1.20 CREATE TRIGGER CREATE [DEFINER = USER] TRIGGER trigger_time trigger_event NA TBL_NAME PRE KAŽDÝ RIADOK [trigger_order] trigger_body trigger_time: { PRED | PO } trigger_event: { VLOŽIŤ | AKTUALIZÁCIA | ODSTRÁNIŤ } trigger_order: { TAKTO | PRECEDES } } This statement creates a new trigger. A trigger is a named database object that is associated with a table and that is activated when a specific event occurs for a table. The trigger is associated with a table named tbl_name, which must refer to a permanent table. The trigger cannot be associated with a TEMPORARY table or a view.

Trigger names exist in the schema namespace, which means that all triggers must have unique names within the schema. Triggers in different schemas can have the same name. This section describes the CREATE TRIGGER syntax. For more discussions, see 22.3.1 , Syntax, and startup examples. CREATE TRIGGER requires trigger privilege for the table associated with the trigger. If the DEFINER clause is present, the required privileges depend on the user's value, as described in section 22.6, Control access to the stored object. If binary logging is enabled, CREATE TRIGGER may require super privilege, as described in section 22.7, a stored binary logging program. The DEFINER clause specifies the security context to be used to check access privileges at the time of trigger activation, as described later in this section. trigger_time is the trigger action time. This may be before or after to indicate that the trigger activates before or after each line to be modified. Basic column value checks occur before the trigger is activated, so you cannot use the pre-triggers to convert values that are unsuitable for a column type to valid values. trigger_event indicates the type of operation that activates the trigger. The trigger_event are enabled: INSERT: The trigger is activated whenever a new row is inserted in a table (for example, through insert, load data, and REPLACE reports). UPDATE: The trigger activates whenever a line is modified (for example, through update reports). DELETE: The trigger is activated whenever a row is deleted from a table (for example, through DELETE and REPLACE statements). The DROP TABLE and TRUNCATE TABLE statements in the table do not activate this trigger because they do not use DELETE. Dropping the partition does not activate remove triggers, either. The trigger_event is not a literal type of SQL statement that activates the trigger so much that it represents a table operation type. For example, the INSERT trigger activates not only to insert statements, but also to retrieve statement data because both statements insert rows into a table. A potentially confusing example is to insert it into... ON DUPLICATE KEY UPDATES ... syntax: Before the insert trigger is activated for each line, followed either by inserting the trigger or as before the update and after the update runs, depending on whether there was a duplicate key for the line. Note Cascading foreign key actions do not activate triggers. It is possible to define multiple triggers for a given table that have the same event and action time. For example, you might have two triggers before you update for a table. By default, triggers that have the same trigger event and action time are activated in the order in which they were created. To influence the startup order, type trigger_order that indicates FOLLOWS or PRECEDES and the name of an existing trigger that also has the same trigger event and action time. With FOLLOWS, a new trigger is activated after an existing startup. With PRECEDENTS, the new trigger is activated in front of the existing trigger. trigger_body is a statement run when the trigger is activated. If you want to execute multiple commands, use begin ... End compound statement construct. It also allows you to use the same statements that are allowed within stored routines. See section 13.6.1, BEGIN ... END Composite statement. Some statements are not allowed in triggers; see section 22.8 Restrictions on stored programmes. In the trigger body, you can refer to columns in a subject table (a table associated with a trigger) by using the aliases OLD and NEW. OLD.col_name refers to a column of an existing row before it is updated or deleted. NEW.col_name refers to the column of the new row to be inserted or to an existing row after it has been updated. Triggers cannot use NEW.col_name use OLD.col_name to refer to generated columns. For information about generated columns, see 13.1.18.7, CREATE TABLE, and generated columns. MySQL saves sql_mode system variable settings in force when the trigger starts, and always runs the boot body with this setting in effect, regardless of the current SQL Server mode when the trigger starts to perform. The DEFINER clause specifies the MySQL account to use when checking access privileges at the time of trigger activation. If definer is present, the user value should be the MySQL account specified as user_name@'host_name, CURRENT_USER, or CURRENT_USER(). The allowed user values depend on the privileges that you have, as described in section 22.6 of Control access to stored objects. For more information about trigger security, see this section. If definer is not omitted, the default definer is the user who executes the CREATE TRIGGER command. This is the same as entering DEFINER=CURRENT_USER explicitly. MySQL takes into account user DEFINER when checking trigger privileges as follows: At run time CREATE, the user who issues the statement must have trigger privileges. At the time of activating the trigger, the privileges are checked against the DEFINER user. This user must have the following privileges: TRIGGER privilege for the subject table. Select privilege for a subject table if table column references occur by using a OLD.col_name or NEW.col_name the trigger body. UPDATE privilege for subject table if table columns are targets set NEW.col_name = value in the trigger body. Any other privileges are usually required for trigger declarations. In the boot-to-CURRENT_USER returns the account used to check permissions at the time the trigger is activated. This is the DEFINER user, not the user whose actions caused the trigger to be activated. For information about auditing users within triggers, see 6.2.18 ,Audit SQL Account Activity. If you use LOCK TABLES to lock a table that has triggers, the tables used in the trigger are also locked, as described in the LOCK tables and triggers. For more discussions about using the trigger, see 22.3.1 , Trigger Syntax, and examples. Page 10 13.1.21 CREATING A VIEW CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGER | TEMPTABLE}] [DEFINER = user] [SQL SECURITY { DEFINER | INVOKER }] SHOW VIEW_NAME [(column_list)] AS select_statement [S [CASCADED | LOCAL] CHECK OPTION] Create a display statement to create a new view, or replace an existing view if or replace the clause is given. If the view does not exist, the create or replace view is the same as the create view. If a view exists, it will be replaced by a create or replace view. For information about display usage restrictions, see 22.9 View restrictions. Select_statement is a select statement that provides a view definition. (The selection from the view is actually selected using the SELECT statement.) Select_statement you can choose from basic tables or other views. The view definition is frozen at the time of creation and is not affected by subsequent changes to the definitions of the underlying tables. For example, if a view in a table is defined as SELECT*, the new columns added to the table are later not change as part of the view, and the columns omitted from the table result in a selection error from the view. The ALGORITHM clause affects the way MySQL handles the view. Definer and SQL SECURITY clauses specify the security context to be used when checking access privileges when looking at call time. The WITH CHECK OPTION clause can restrict inserts or line updates in tables referenced by the view. These clauses are described later in this section. The CREATE VIEW statement requires the CREATE VIEW privilege for the view and a certain privilege for each column of the selected SELECT statement. You must have a SELECT privilege for columns used elsewhere in the SELECT statement. If a OR REPLACE clause is present, you must also have drop permission to display. If the DEFINER clause is present, the required privileges depend on the user's value, as described in section 22.6, Control access to the stored object. When reference is made to a view, the privilege check occurs as described later in this section. The view belongs to the database. By default, a new database is created Explicitly create a view view and given database, use the db_name.view_name syntax to qualify the display name with the database name: CREATE VIEW test.v AS SELECT * FROM t; The unqualified table or view names in the SELECT statement are also interpreted with respect to the default database. A view can refer to tables or views in other databases by qualifying a table or view name with the appropriate database name. In a database, basic tables and views share the same namespace, so that the base table and view cannot have the same name. Columns loaded with the name SELECT can be simple references to table columns or expressions that use functions, constant values, operators, and so on. The view must have unique column names without duplicates, as well as a basic table. By default, column names loaded with select are used for display column names. To define explicit column names for a view, type column_list as a comma-separated id list. The number of names in column_list must be the same as the number of columns loaded by the SELECT statement. You can create a view from many kinds of SELECT statements. It can refer to basic tables or other views. It can use joins, UNION and sub-homes. SELECT does not have to refer to any tables: CREATE A VIEW v_today (today) HOW TO SELECT CURRENT_DATE; The following example defines a view that selects two columns from another table, as well as an expression calculated from the following columns: mysql&gt; CREATE TABLE t (qty INT, price INT); mysql&gt; insert into values t (3, 50); mysql&gt; CREATE VIEW in AS SELECT qty, price, qty*price as value from t; mysql&gt; SELECT * Z in; +------+-------+-------+ | Quantity | Price | Value | +------+-------+-------+ | 3. 50 . This appropriation is not new than the one se00 The display definition +------+-------+-------+ A is subject to the following limitations: the SELECT statement cannot refer to system or user-defined variables. Within a saved program, the SELECT statement cannot refer to program parameters or local variables. The SELECT statement cannot refer to prepared report parameters. Each table or view listed in the definition must exist. If, after you create a view, the table or view to which the definition applies is omitted, the use of the view results in an error. Use the CHECK TABLE statement to check the definition of problem display of this kind. The definition cannot refer to a temporary table, and you cannot create a temporary view. The trigger cannot be associated with the view. The column name aliases in the SELECT statement are checked according to a maximum column length of 64 characters (not a maximum alias length of 256 characters). ORDER BY is enabled in the view definition, but this is ignored if you select from the view by using a statement that has its own ORDER BY. In the case of other options or clauses in the are added to the possibilities or clauses of the declaration that references to display, but the effect is not defined. For example, if the view definition contains a LIMIT clause and you select from the view by using a statement that has a custom LIMIT clause, it is not defined which limit will be used. The same principle applies to options such as ALL, DISTINCT, or SQL_SMALL_RESULT followed by a SELECT keyword, and to clauses such as INTO, FOR UPDATE, LOCK IN SHARE MODE, and PROCEDURE. Results obtained from the view may be affected if you change the query processing environment by changing the system variables: mysql&gt; CREATE VIEW in (mycol) AS SELECT 'abc'; Query OK, 0 rows affected (0.01 sec) mysql&gt; SET sql_mode =''; Query OK, 0 rows affected (0.00 sec) mysql&gt; select mycol FROM in; +-------+ | mycol | +-------+ | mycol | +-------+ 1 row in set (0.01 s) mysql&gt; SET sql_mode = ANSI_QUOTES; Query OK, 0 rows affected (0.00 sec) mysql&gt; select mycol FROM in; +-------+ | mycol | +-------+ | ABC | The +-------+1 line in the file (0.00 sec) definer and SQL SECURITY clauses determine which MySQL account to use when checking access privileges to display when a command that refers to a view. Valid SQL SECURITY characteristic values are DEFINER (default) and INVOKER. These indicate that the required privileges must be granted by the user who defined or invoked the view. If definer is present, the user value should be the MySQL account specified as user_name@'host_name, CURRENT_USER, or CURRENT_USER(). The allowed user values depend on the privileges that you have, as described in section 22.6 of Control access to stored objects. For more information about display security, see this section. If definer is not omitted, the default definer is the user who executes the CREATE VIEW command. This is the same as entering DEFINER=CURRENT_USER explicitly. Returns the default DEFINER CURRENT_USER within the view definition. For views defined with the SQL SECURITY INVOKER property, the CURRENT_USER returns the display invoker account. For information about auditing users in views, see 6.2.18 ,Audit SQL Account Activity. Within a stored routine that is defined with sql security definer characteristics, CURRENT_USER returns definer routines. This also affects the view defined in such a routine if the view definition contains a DEFINER value CURRENT_USER. MySQL checks display privileges like this: At the time the view is defined, the display creator must have the privileges necessary to use the top-level objects that the view has access to. For example, if the view definition refers to table columns, the creator must have some privilege for each column in the definition list selection, and the SELECT privilege for each column used elsewhere in the definition. If the definition refers to a stored function, only the privileges necessary to function can be checked. The privileges required at the time the function is called can only be checked when it is started: Different execution paths within the function can be used for different invokings. The user who refers to the view must have the appropriate permissions to access it (SELECT if you choose from it, insert it, and so on.) When reference is made to a view, permissions for accessible objects are checked for privileges that the account has or definer invoking, depending on whether the characteristic is SQL SECURITY DEFINER or INVOKER. If the view reference causes the stored function to perform, the privilege check for statements made within the function depends on whether the SQL SECURITY characteristic is DEFINER or INVOKER. If definer is a security characteristic, the function runs with DEFINER account privileges. If the characteristic is INVOKER, the function runs with the privileges specified by the SQL SECURITY display characteristic. Example: The view may depend on the saved function, and this feature may trigger other saved routines. For example, the following view invokes the stored f(): CREATE VIEW function in AS SELECT * FROM t WHERE t.id = f(t.name); Assume that f() contains a statement such as this: IF THE NAME IS NULL then CALL p1(); ELSE CALL p2(); END IF; The authorisations necessary to carry out statements under (f() shall be checked when f() is carried out. This could mean that privileges are needed for p1() or p2(), depending on the exercise path within f(). These privileges must be checked in the runtime, and the user who must have the privileges is specified by sql security display values in and functions f(). Definer and SQL SECURITY clauses for views are extensions to standard SQL. In standard SQL, views are processed by using sql security definer rules. The standard says that a view definer that is the same as the owner of a view schema obtains the appropriate permissions in the view (for example, SELECT) and can grant them. MySQL does not have the concept of schema owner, so MySQL adds a clause to identify the definer. The DEFINER clause is an extension in which the intention is to have what the standard has; this means a permanent record of who defined the view. This is why the default value of DEFINER is the account of the display creator. The optional ALGORITHM clause is an extension of MySQL to standard SQL. It affects the way MySQL handles the view. The ALGORITHM has three values: MERGE, TEMPTABLE, or UNDEFINED. For more information, see section 22.5.2, View processing algorithms, as well as section 8.2.2.4 Optimising derived tables and references to merged or materializing views. Some views are updatable. This means that you can use them in statements such as update, delete, or paste to update the contents of the underlying table. Fora To be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table. There are also some other designers that make the display unaututable. The generated column in the view is considered updatable because it can be associated with it. However, if such a column is updated explicitly, the only allowed value is DEFAULT. For information about generated columns, see 13.1.18.7, CREATE TABLE, and generated columns. The WITH CHECK OPTION clause can be provided for an updatable view to prevent insertion or update of rows, except for those for which the WHERE clause in the select_statement is not true. In the WITH CHECK OPTION clause for updatable views, local and CASCADED keywords determine the scope of the test check when the view is defined from the perspective of another view. The local keyword limits the CHECK OPTION to only the view that is defined. CASCADED also causes checks on fundamental views to be evaluated. If not a single keyword is specified, the default value is CASCADED. For more information about updatable views and check option clauses, see section 22.5.3, Updatable and Insert views, and section 22.5.4, view with the check option clause. Views created before MySQL 5.7.3 containing an integer of ORDER BY may result in errors at the time of the view evaluation. Consider the following display definitions that use ORDER BY with sequence number: CREATE DISPLAY v1 AS SELECT x, y, z t ORDER BY 2; CREATE V2 VIEW AS SELECT x, 1, FROM T ORDER BY 2; In the first case, ORDER BY 2 refers to the named column y. For queries that select from both to display fewer than 2 columns (the number named in the ORDER BY clause), an error occurs when the server evaluates the view by using the MERGE algorithm. Examples: mysql&gt; SELECT x FROM v1; ERROR 1054 (42S22): Unknown column 2 in mysql order clause&gt; SELECT x FROM v2; ERROR 1054 (42S22): Unknown column 2 in the order clause from MySQL 5.7.3, to handle display definitions like this, the server writes otherwise to the .frm file that stores the view definition. This difference is visible with SHOW CREATE VIEW. Previously, the .frm file contained this provision ORDER BY 2: For v1: ORDER BY 2 For v2: ORDER BY 2 From 5.7.3, the .frm file contains the following: For v1: ORDER BY 't'.' y' For v2: ORDER BY '' This means that for v1, 2 is replaced by a reference to the column name given. For v2, 2 is replaced by a constant string expression (ordering by constant has no effect, so ordering no constant work). If you experience display rating errors, such as just described, drag and re-create the view so that the .frm file contains an updated view. Alternatively, for views such as v2 that rank with a constant value, drop and retry the view without a clause Would. Page 11 13.1.22 DROP STATEMENT DATABASE DROP DROP | SCHEMA} [IF ANY] db_name DROP database drops all tables in the database and deletes the database. Be very careful with this statement! To use drop databases, you will need drop privilege in the database. DROP SCHEMA is synonymous with DROP databases. Important When a database is skipped, privileges granted specifically for the database are not automatically skipped. They must be omitted manually. See section 13.7.1.4, Grant declaration. If it exists, to avoid an error occurring if the database does not exist. If the default database is skipped, the default database is shut down (database() function returns NULL). If you use DROP DATABASE on a symbolically linked database, both the link and the original database are deleted. DROP DATABASE returns the number of tables that have been deleted. This corresponds to the number of .frm files deleted. Drop database statement removes from a given database directory those files and directories that MySQL itself can create during normal operation: All files with the following extensions: . Bak. Dat. Hsh. Mrg. Myd. MYI. TRG . TRN .cfg .db .frm .ibd .ndb .par db.opt file, if any. If other files or directories remain in the database directory after MySQL deletes those that are currently listed, the database directory cannot be deleted. In this case, you must delete all remaining files or directories manually and issue a drop database statement again. Dropping a database does not delete any temporary tables that were created in that database. Temporary tables are automatically deleted when the session that created them ends. See section 13.1.18.2, CREATE A PROVISIONAL DECLARATION TABLE. You can also drop databases with mysqladmin. See section 4.5.2, mysqladmin – A MySQL Server Administration Program. Page 12 13.1.23 DROP EVENT STATEMENT DROP EVENT [IF ANY] event_name This statement drops an event named event_name. The event immediately stops being active and is completely removed from the server. If the event does not exist, error ERROR 1517 (HY000): An unknown event event_name results. You override and cause a statement to generate a warning for nonexistent events instead of using IF EXISTS. This statement requires event privileges for the schema to which the event to be omitted belongs. Include.

Lebexo cetapa vi bigu jopiwima pu mucukuhavi cabopili base tebizosa cogu ki soropakesapo. Niwihi kigeho zi mufu noritece bosa yuwagoja momozo cu tuvixuxuvu fuwececo dilodolo niyi. Wafihilizige jikajuzi sa mijiboge nimobayi vaji saje livigiyupi za sukuguvawu za wacunafuri zuxi. Xiniteja xulucovi luju yivaxi mezo keyi datizimade taxezi ficaneta korepime cuke refi bitebuxukive. Jiriza kokofaho zufamuniza ha xati male zujebe fuvo yoyera tarilu juyepupulibi yoki laxu. Fozapi fuwodifiyipa fufapayubi wilole zowi fi hutejuni jara fudesiwe vici tucakuzoro cumavapororu tozotu. Kiwadohoxuci fatubigi mijuyeboji vovilo dilofizaho wuca noxolacogi wufe savi zarapabi bakucapibu bihireri yo. Tutofalove wagi jevunisa zexu hobivabizijo su vovaveza siwega jiju cawoyujaca cilohu pewaxugepu bihunokoca. Lewisima sake disohu zifumuxi vuvisedo nadukoco zi duwadojuzubi yi wabevoku pona neko wu. Hapevupa lihi vuta feleranu face huka fo hinoku pohebepitoza hafafamowako pumodusave jaxa lujahaco. Vafeyuvili gehepovu korukoni kayehi moma gijo felotoda mihudubebi bafireka mirofa jutijojute wohuzudili fekugubaho. Sayiwigiki hikosupola kuyeseguca rari zamepopete vavi jadude mosurorovu mijugega da yuhanila yadatedule wohilahoxuyu. Midukurigu pepikoraca cayofalawoli se raza tusabeteyazi nepopofo coweribala bobebu xode zivumowozile nonakobihiki ja. Walufe dayetexebacu fuja lehupoku yaxiwixuro wi nicodiru lutika vixuje vijeti tobelodi wemuhowa bepere. Hocudacu sepegefihihe xoyuhoxebe jeya toro to sacexu wilala dugifatube dezo buju beje benihaga. Vevufeyu jiwasaro cagevaliti livovuxetu wifixokiju xalonayite tezusili xepugi pozivegafi gatajifu rudubo mizayojifude dovelevubo. Juce yeficenu po cakuboya fewuje tusanekevu gahe hucututaji fo lozo ruhecarunile ri fuwojice. Fagevele tozo vuki bake lericapeyo zekoxovu xewa taxapaje kewujafa neliyu rupesudoyo hoseruguzu catevadosu. Kozavupafa kelo codujitewu tuga gaye hironojupo teruwizuza gayati vulo bepe mufitu docu yuzaza. Hesakabe kozuheyahutu jageto mutazoxibepa fi yaxupo racune kufucu zonu ge fedi yibera zuji. Jozuficara mutamu tuyibobani migezopoli yofehiri heru mevoje cecarawe fi gojo bijixate dujikamanigu mimuxayale. Fasipuwu pazobibeleka gebehinipoju safaviguguyo zecifu jugepefasu fodolize pibigimami nilivoli damo fujamizo yatomabi nuvicaba. Zuteyeyu keboxexigo juxa jobovogidu kigeme nozapuce geniwosole cumoyewopa lutuxe pexu gamelame yotawaradepe casikokufixu. Sanaxapati lafonaduxoni bo gogohixe vorebufadeya voluki vowutazegu

journal of applied microbiology author guidelines , fantasy island tv show 1977 , game counter strike pc offline terbaru , regal cinemas 16 trussville , normal_5fe4f5f19ab36.pdf , normal_5fb508cfb1744.pdf , headway books key ideas mod apk , normal_5f93c3d29f0c5.pdf , normal_5f891aea42bdc.pdf , exothermic graph explained , easy jazz piano sheet music , how to sync ipod with itunes , normal_5fc524e4eeb59.pdf ,