The behavior of the auto-increment mechanism is not defined if a user assigns a negative value to the column or if the value becomes bigger than the maximum integer that can be stored in the specified integer type.

When accessing the auto-increment counter, InnoDB uses a special table-level AUTO-INC lock that it keeps to the end of the current SQL statement, not to the end of the transaction. The special lock release strategy was introduced to improve concurrency for inserts into a table containing an AUTO_INCREMENT column. Nevertheless, two transactions cannot have the AUTO-INC lock on the same table simultaneously, which can have a performance impact if the AUTO-INC lock is held for a long time. That might be the case for a statement such as INSERT INTO t1 ... SELECT ... FROM t2 that inserts all rows from one table into another.

InnoDB uses the in-memory auto-increment counter as long as the server runs. When the server is stopped and restarted, InnoDB reinitializes the counter for each table for the first INSERT to the table, as described earlier.

Beginning with MySQL 5.0.3, InnoDB supports the AUTO_INCREMENT = *N* table option in CREATE TABLE and ALTER TABLE statements, to set the initial counter value or alter the current counter value. The effect of this option is canceled by a server restart, for reasons discussed earlier in this section.

14.2.6.4. FOREIGN KEY Constraints

InnoDB also supports foreign key constraints. The syntax for a foreign key constraint definition in InnoDB looks like this:

```
[CONSTRAINT symbol] FOREIGN KEY [id] (index_col_name, ...)
    REFERENCES tbl_name (index_col_name, ...)
    [ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
    [ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
```

Foreign keys definitions are subject to the following conditions:

- Both tables must be InnoDB tables and they must not be TEMPORARY tables.
- Corresponding columns in the foreign key and the referenced key must have similar internal data types inside InnoDB so that they can be compared without a type conversion. *The size and sign of integer types must be the same*. The length of string types need not be the same. For non-binary (character) string columns, the character set and collation must be the same.
- In the referencing table, there must be an index where the foreign key columns are listed as the *first* columns in the same order. Such an index is created on the referencing table automatically if it does not exist.
- In the referenced table, there must be an index where the referenced columns are listed as the *first* columns in the same order.
- Index prefixes on foreign key columns are not supported. One consequence of this is that BLOB and TEXT columns cannot be included in a foreign key, because indexes on those columns must always include a prefix length.
- If the CONSTRAINT *symbol* clause is given, the *symbol* value must be unique in the database. If the clause is not given, InnoDB creates the name automatically.

InnoDB rejects any INSERT or UPDATE operation that attempts to create a foreign key value in a child table if there is no a matching candidate key value in the parent table. The action InnoDB takes for any UPDATE or DELETE operation that attempts to update or delete a candidate key value in the parent table that has some matching rows in the child table is dependent on the *referential action* specified using ON UPDATE and ON DELETE subclauses of the FOREIGN KEY clause. When the user attempts to delete or update a row from a parent table, and there are one or more matching rows in the child table, InnoDB supports five options regarding the action to be taken:

- CASCADE: Delete or update the row from the parent table and automatically delete or update the matching rows in the child table. Both ON DELETE CASCADE and ON UPDATE CASCADE are supported. Between two tables, you should not define several ON UPDATE CASCADE clauses that act on the same column in the parent table or in the child table.
- SET NULL: Delete or update the row from the parent table and set the foreign key column or columns in the child table to NULL. This is valid only if the foreign key columns do not have the NOT NULL qualifier specified. 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 not declared the columns in the child table as NOT NULL*.
- NO ACTION: In standard SQL, NO ACTION means *no action* in the sense that an attempt to delete or update a primary key value is not allowed to proceed if there is a related foreign key value in the referenced table. InnoDB rejects the delete or update operation for the parent table.
- RESTRICT: Rejects the delete or update operation for the parent table. NO ACTION and RESTRICT are the same as omitting the ON DELETE or ON UPDATE clause. (Some database systems have deferred checks, and NO ACTION is a deferred check.

In MySQL, foreign key constraints are checked immediately, so NO ACTION and RESTRICT are the same.)

- SET DEFAULT: This action is recognized by the parser, but InnoDB rejects table definitions containing ON DELETE SET DEFAULT or ON UPDATE SET DEFAULT clauses.

Note that InnoDB supports foreign key references within a table. In these cases, “child table records” really refers to dependent records within the same table.

InnoDB requires indexes on foreign keys and referenced keys so that foreign key checks can be fast and not require a table scan. The index on the foreign key is created automatically. This is in contrast to some older versions, in which indexes had to be created explicitly or the creation of foreign key constraints would fail.

If MySQL reports an error number 1005 from a CREATE TABLE statement, and the error message refers to errno 150, table creation failed because a foreign key constraint was not correctly formed. Similarly, if an ALTER TABLE fails and it refers to errno 150, that means a foreign key definition would be incorrectly formed for the altered table. You can use SHOW ENGINE INNODB STATUS to display a detailed explanation of the most recent InnoDB foreign key error in the server.

Note: InnoDB does not check foreign key constraints on those foreign key or referenced key values that contain a NULL column.

Note: Currently, triggers are not activated by cascaded foreign key actions.

Deviation from SQL standards: If there are several rows in the parent table that have the same referenced key value, InnoDB acts in foreign key checks as if the other parent rows with the same key value do not exist. For example, if you have defined a RESTRICT type constraint, and there is a child row with several parent rows, InnoDB does not allow the deletion of any of those parent rows.

InnoDB performs cascading operations through a depth-first algorithm, based on records in the indexes corresponding to the foreign key constraints.

Deviation from SQL standards: A FOREIGN KEY constraint that references a non-UNIQUE key is not standard SQL. It is an InnoDB extension to standard SQL.

Deviation from SQL standards: If ON UPDATE CASCADE or ON UPDATE SET NULL recurses to update the *same table* it has previously updated during the cascade, it acts like RESTRICT. This means that you cannot use self-referential ON UPDATE CASCADE or ON UPDATE SET NULL operations. This is to prevent infinite loops resulting from cascaded updates. A self-referential ON DELETE SET NULL, on the other hand, is possible, as is a self-referential ON DELETE CASCADE. Cascading operations may not be nested more than 15 levels deep.

Deviation from SQL standards: Like MySQL in general, in an SQL statement that inserts, deletes, or updates many rows, InnoDB checks UNIQUE and FOREIGN KEY constraints row-by-row. According to the SQL standard, the default behavior should be deferred checking. That is, constraints are only checked after the *entire SQL statement* has been processed. Until InnoDB implements deferred constraint checking, some things will be impossible, such as deleting a record that refers to itself via a foreign key.

Here is a simple example that relates parent and child tables through a single-column foreign key:

```
CREATE TABLE parent (id INT NOT NULL,
                     PRIMARY KEY (id)
) ENGINE=INNODB;
CREATE TABLE child (id INT, parent_id INT,
                    INDEX par_ind (parent_id),
                    FOREIGN KEY (parent_id) REFERENCES parent(id)
                      ON DELETE CASCADE
) ENGINE=INNODB;
```

A more complex example in which a product_order table has foreign keys for two other tables. One foreign key references a two-column index in the product table. The other references a single-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 TABLE customer (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),
                            FOREIGN KEY (product_category, product_id)
                              REFERENCES product(category, id)
                              ON UPDATE CASCADE ON DELETE RESTRICT,
                            INDEX (customer_id),
                            FOREIGN KEY (customer_id)
                              REFERENCES customer(id)) ENGINE=INNODB;
```

InnoDB allows you to add a new foreign key constraint to a table by using ALTER TABLE:

```
ALTER TABLE tbl_name
    ADD [CONSTRAINT symbol] FOREIGN KEY [id] (index_col_name, ...)
    REFERENCES tbl_name (index_col_name, ...)
    [ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
    [ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
```

Remember to create the required indexes first. You can also add a self-referential foreign key constraint to a table using ALTER TABLE.

InnoDB also supports the use of ALTER TABLE to drop foreign keys:

```
ALTER TABLE tbl_name DROP FOREIGN KEY fk_symbol;
```

If the FOREIGN KEY clause included a CONSTRAINT name when you created the foreign key, you can refer to that name to drop the foreign key. Otherwise, the *fk_symbol* value is internally generated by InnoDB when the foreign key is created. To find out the symbol value when you want to drop a foreign key, use the SHOW CREATE TABLE statement. For example:

```
mysql> SHOW CREATE TABLE ibtest11c\G
*************************** 1. row ***************************
       Table: ibtest11c
Create Table: CREATE TABLE `ibtest11c` (
  `A` int(11) NOT NULL auto_increment,
  `D` int(11) NOT NULL default '0',
  `B` varchar(200) NOT NULL default '',
  `C` varchar(175) default NULL,
  PRIMARY KEY  (`A`,`D`,`B`),
  KEY `B` (`B`,`C`),
  KEY `C` (`C`),
  CONSTRAINT `0_38775` FOREIGN KEY (`A`, `D`)
REFERENCES `ibtest11a` (`A`, `D`)
ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `0_38776` FOREIGN KEY (`B`, `C`)
REFERENCES `ibtest11a` (`B`, `C`)
ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=INNODB CHARSET=latin1
1 row in set (0.01 sec)

mysql> ALTER TABLE ibtest11c DROP FOREIGN KEY `0_38775`;
```

You cannot add a foreign key and drop a foreign key in separate clauses of a single ALTER TABLE statement. Separate statements are required.

The InnoDB parser allows table and column identifiers in a FOREIGN KEY ... REFERENCES ... clause to be quoted within backticks. (Alternatively, double quotes can be used if the ANSI_QUOTES SQL mode is enabled.) The InnoDB parser also takes into account the setting of the lower_case_table_names system variable.

InnoDB returns a table's foreign key definitions as part of the output of the SHOW CREATE TABLE statement:

```
SHOW CREATE TABLE tbl_name;
```

mysqldump also produces correct definitions of tables to the dump file, and does not forget about the foreign keys.

You can also display the foreign key constraints for a table like this:

```
SHOW TABLE STATUS FROM db_name LIKE 'tbl_name';
```

The foreign key constraints are listed in the Comment column of the output.

When performing foreign key checks, InnoDB sets shared row-level locks on child or parent records it has to look at. InnoDB checks foreign key constraints immediately; the check is not deferred to transaction commit.

To make it easier to reload dump files for tables that have foreign key relationships, mysqldump automatically includes a statement in the dump output to set FOREIGN_KEY_CHECKS to 0. This avoids problems with tables having to be reloaded in a particular order when the dump is reloaded. It is also possible to set this variable manually:

```
mysql> SET FOREIGN_KEY_CHECKS = 0;
mysql> SOURCE dump_file_name;
mysql> SET FOREIGN_KEY_CHECKS = 1;
```

This allows you to import the tables in any order if the dump file contains tables that are not correctly ordered for foreign keys. It also speeds up the import operation. Setting FOREIGN_KEY_CHECKS to 0 can also be useful for ignoring foreign key constraints during LOAD DATA and ALTER TABLE operations. However, even if FOREIGN_KEY_CHECKS=0, InnoDB does not allow the creation of a foreign key constraint where a column references a non-matching column type. Also, if an InnoDB table has foreign

key constraints, ALTER TABLE cannot be used to change the table to use another storage engine. To alter the storage engine, you must drop any foreign key constraints first.

InnoDB does not allow you to drop a table that is referenced by a FOREIGN KEY constraint, unless you do SET FOREIGN_KEY_CHECKS=0. When you drop a table, the constraints that were defined in its create statement are also dropped.

If you re-create a table that was dropped, it must have a definition that conforms to the foreign key constraints referencing it. It must have the right column names and types, and it must have indexes on the referenced keys, as stated earlier. If these are not satisfied, MySQL returns error number 1005 and refers to errno 150 in the error message.

14.2.6.5. InnoDB and MySQL Replication

MySQL replication works for InnoDB tables as it does for MyISAM tables. It is also possible to use replication in a way where the storage engine on the slave is not the same as the original storage engine on the master. For example, you can replicate modifications to an InnoDB table on the master to a MyISAM table on the slave.

To set up a new slave for a master, you have to make a copy of the InnoDB tablespace and the log files, as well as the .frm files of the InnoDB tables, and move the copies to the slave. If the innodb_file_per_table variable is enabled, you must also copy the .ibd files as well. For the proper procedure to do this, see Section 14.2.8, "Backing Up and Recovering an InnoDB Database".

If you can shut down the master or an existing slave, you can take a cold backup of the InnoDB tablespace and log files and use that to set up a slave. To make a new slave without taking down any server you can also use the non-free (commercial) InnoDB Hot Backup tool.

You cannot set up replication for InnoDB using the LOAD TABLE FROM MASTER statement, which works only for MyISAM tables. There are two possible workarounds:

- Dump the table on the master and import the dump file into the slave.
- Use ALTER TABLE *tbl_name* ENGINE=MyISAM on the master before setting up replication with LOAD TABLE *tbl_name* FROM MASTER, and then use ALTER TABLE to convert the master table back to InnoDB afterward. However, this should not be done for tables that have foreign key definitions because the definitions will be lost.

Transactions that fail on the master do not affect replication at all. MySQL replication is based on the binary log where MySQL writes SQL statements that modify data. A transaction that fails (for example, because of a foreign key violation, or because it is rolled back) is not written to the binary log, so it is not sent to slaves. See Section 13.4.1, "START TRANSACTION, COMMIT, and ROLLBACK Syntax".

14.2.7. Adding and Removing InnoDB Data and Log Files

This section describes what you can do when your InnoDB tablespace runs out of room or when you want to change the size of the log files.

The easiest way to increase the size of the InnoDB tablespace is to configure it from the beginning to be auto-extending. Specify the autoextend attribute for the last data file in the tablespace definition. Then InnoDB increases the size of that file automatically in 8MB increments when it runs out of space. The increment size can be changed by setting the value of the innodb_autoextend_increment system variable, which is measured in MB.

Alternatively, you can increase the size of your tablespace by adding another data file. To do this, you have to shut down the MySQL server, change the tablespace configuration to add a new data file to the end of innodb_data_file_path, and start the server again.

If your last data file was defined with the keyword autoextend, the procedure for reconfiguring the tablespace must take into account the size to which the last data file has grown. Obtain the size of the data file, round it down to the closest multiple of 1024 × 1024 bytes (= 1MB), and specify the rounded size explicitly in innodb_data_file_path. Then you can add another data file. Remember that only the last data file in the innodb_data_file_path can be specified as auto-extending.

As an example, assume that the tablespace has just one auto-extending data file ibdata1:

```
innodb_data_home_dir =
innodb_data_file_path = /ibdata/ibdata1:10M:autoextend
```

Suppose that this data file, over time, has grown to 988MB. Here is the configuration line after modifying the original data file to not be auto-extending and adding another auto-extending data file:

```
innodb_data_home_dir =
innodb_data_file_path = /ibdata/ibdata1:988M;/disk2/ibdata2:50M:autoextend
```