in the process makes it possible to simplify joins for queries with outer joins having trivial conditions, such as this one:

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 WHERE condition_2 OR 0 = 1
```

The optimizer now sees during preparation that 0 = 1 is always false, making `OR 0 = 1` redundant, and removes it, leaving this:

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 where condition_2
```

Now the optimizer can rewrite the query as an inner join, like this:

```
SELECT * FROM t1 LEFT JOIN t2 WHERE condition_1 AND condition_2
```

For more information, see Section 8.2.1.9, “Outer Join Optimization”.

- In MySQL 8.0.16 and later, MySQL can use constant folding at optimization time to handle comparisons between a column and a constant value where the constant is out of range or on a range boundary with respect to the type of the column, rather than doing so for each row at execution time. For example, given a table `t` with a `TINYINT UNSIGNED` column `c`, the optimizer can rewrite a condition such as `WHERE c < 256` to `WHERE 1` (and optimize the condition away altogether), or `WHERE c >= 255` to `WHERE c = 255`.

  See Section 8.2.1.14, “Constant-Folding Optimization”, for more information.

- Beginning with MySQL 8.0.16, the semijoin optimizations used with `IN` subqueries can now be applied to `EXISTS` subqueries as well. In addition, the optimizer now decorrelates trivially-correlated equality predicates in the `WHERE` condition attached to the subquery, so that they can be treated similarly to expressions in `IN` subqueries; this applies to both `EXISTS` and `IN` subqueries.

  For more information, see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”.

- As of MySQL 8.0.17, the server rewrites any incomplete SQL predicates (that is, predicates having the form `WHERE value`, in which `value` is a column name or constant expression and no comparison operator is used) internally as `WHERE value <> 0` during the contextualization phase, so that the query resolver, query optimizer, and query executor need work only with complete predicates.

  One visible effect of this change is that, for Boolean values, `EXPLAIN` output now shows `true` and `false`, rather than `1` and `0`.

  Another effect of this change is that evaluation of a JSON value in an SQL boolean context performs an implicit comparison against JSON integer 0. Consider the table created and populated as shown here:

```
mysql> CREATE TABLE test (id INT, col JSON);

mysql> INSERT INTO test VALUES (1, '{"val":true}'), (2, '{"val":false}');
```

  Previously, the server attempted to convert an extracted `true` or `false` value to an SQL boolean when comparing it in an SQL boolean context, as shown by the following query using `IS TRUE`:

```
mysql> SELECT id, col, col->"$.val" FROM test WHERE col->"$.val" IS TRUE;
+------+---------------+--------------+
| id   | col           | col->"$.val" |
+------+---------------+--------------+
|    1 | {"val": true} | true         |
```