Binlog scope

ClustrixDB supports multiple binlogs per system. Each binlog can have a different scope. This gives administrators control over each binlog and overall logging coverage, but introduces restrictions on the queries that can be logged.

Configuring binlog scope

Each ClustrixDB binlog has LOG and IGNORE lists, which are similar to MySQL's --binlog-do-db and --binlog-ignore-db options. These lists control the scope of each binlog, and can be manipulated using DDL statements.

CREATE BINLOG

Binlog scope can be set during binlog creation by adding LOG and IGNORE clauses to the CREATE BINLOG statement.

```
/* log all changes on the entire system */
master> CREATE BINLOG 'binny';

/* only log changes to database `test` */
master> CREATE BINLOG 'binny' LOG(`test`);

/* only log changes to databases `test` and `baz` */
master> CREATE BINLOG 'binny' LOG(`test`, `baz`);

/* log all changes except changes to database `test` */
master> CREATE BINLOG 'binny' IGNORE(`test`);

/* log all changes except changes to databases `test` and `baz` */
master> CREATE BINLOG 'binny' IGNORE(`test`, `baz`);
```

Binlog scope is independent of binlog format.

```
/* these both work */
master> CREATE BINLOG 'binny' LOG(`test`), FORMAT='ROW';
master> CREATE BINLOG 'binny' LOG(`test`), FORMAT='STATEMENT';
```

Create a table scope binlog following this sample:

```
/* only log changes to table `test`.`foo` */
master> CREATE BINLOG 'binny' LOG(`test`.`foo`);

/* log all changes except changes to tables `test`.`foo` and `baz`.`bar` */
master> CREATE BINLOG 'binny' IGNORE(`test`.`foo`, `baz`.`bar`);
```

Table scope and database scope can be combined.

```
/* only log changes to database `test` excluding changes to table `test`.`foo` */
master> CREATE BINLOG 'binny' LOG(`test`), IGNORE(`test`.`foo`);

/* only log changes to database `test` and table `baz`.`bar` */
master> CREATE BINLOG 'binny' LOG(`test`, `baz`.`bar`);
```

ALTER BINLOG

Binlog scope can be completely reset by executing ALTER BINLOG statements with LOG and IGNORE clauses.

```
/* get rid of the existing configuration and log everything */
master> ALTER BINLOG 'binny' LOG ALL;

/* get rid of the existing configuration and log nothing */
master> ALTER BINLOG 'binny' IGNORE ALL;

/* get rid of the existing configuration and only log changes to database `test` */
master> ALTER BINLOG 'binny' LOG(`test`);

/* get rid of the existing configuration and log everything except changes to database `test` */
master> ALTER BINLOG 'binny' IGNORE(`test`);
```

Or, the existing LOG and IGNORE lists can be adjusted with ADD and DROP.

```
/* add database `test` to the current LOG list */
master> ALTER BINLOG 'binny' ADD LOG(`test`);

/* add database `test` to the current IGNORE list */
master> ALTER BINLOG 'binny' ADD IGNORE(`test`);

/* remove database `test` from the current LOG list  */
master> ALTER BINLOG 'binny' DROP LOG(`test`);

/* remove database `test` from the current IGNORE list */
master> ALTER BINLOG 'binny' DROP IGNORE(`test`);
```

Multiple directives can be included in one ALTER BINLOG; they are applied left-to-right.

```
/* remove database `baz` and add database `test` to the current LOG list */
master> ALTER BINLOG 'binny' DROP LOG(`baz`), ADD LOG(`test`);

/* get rid of the existing configuration and only log changes to databases `baz` and `test`, excluding changes
to table `test`.`foo` */
master> ALTER BINLOG 'binny' IGNORE ALL, ADD LOG(`baz`, `test`), ADD IGNORE(`test`.`foo`);
```

Checking Binlog Scope

There are two ways to check the LOG and IGNORE lists for a binlog. The first is to use SHOW MASTER STATUS.

```
master> CREATE BINLOG 'binny' LOG(`test`, `baz`.`bar`), IGNORE(`test`.`foo`),
FORMAT='ROW';
Query OK, 0 rows affected (0.10 sec)

master> SHOW MASTER STATUS 'binny';
+--------------+----------+---------------+------------------+
| File         | Position | Binlog_Do_DB  | Binlog_Ignore_DB |
+--------------+----------+---------------+------------------+
| binny.000001 |        4 | baz.bar, test | test.foo         |
+--------------+----------+---------------+------------------+
1 row in set (0.05 sec)
```

The second is to query the system tables that hold binlog configuration.

```
master> SELECT name, log_id, log_all_dbs
        FROM system.binlogs
        WHERE name = 'binny';
+-------+--------------------+-------------+
| name  | log_id             | log_all_dbs |
+-------+--------------------+-------------+
| binny | 6057996416246436865 |           0 |
+-------+--------------------+-------------+
1 row in set (0.01 sec)

master> SELECT `database` AS LOG
        FROM system.binlog_log_databases
        WHERE log_id = 6057996416246436865
        UNION
            SELECT CONCAT(`database`, '.',
`table`)
            FROM system.binlog_log_tables
            WHERE log_id =
6057996416246436865;
+---------+
| LOG     |
+---------+
| test    |
| baz.bar |
+---------+
2 rows in set (0.01 sec)

master> SELECT `database` AS IGNORE
        FROM system.binlog_ignore_databases
        WHERE log_id = 6057996416246436865
        UNION
            SELECT CONCAT(`database`, '.',
`table`)
            FROM system.binlog_ignore_tables
            WHERE log_id =
6057996416246436865;
+----------+
| IGNORE   |
+----------+
| test.foo |
+----------+
1 row in set (0.01 sec)
```

How Scoping Works

A binlog records all statements that modify something within its scope. More specifically, a statement that modifies something is recorded to all binlogs that log the modified object, and all binlogs that log everything, except for binlogs that IGNORE the modified object. This is not determined by the default database, but by the statement itself.

The following statement will be recorded to all binlogs that LOG test.foo. The statement is the same for both statement-based and row-based logging and is irrespective of the database currently in use.

```
master> INSERT INTO test.foo VALUES (1);
```

The above statement will be logged to each of these binlogs.

```
master> CREATE BINLOG 'b1' LOG(`test`.`foo`);
master> CREATE BINLOG 'b2' LOG(`test`);
master> CREATE BINLOG 'b3';
master> CREATE BINLOG 'b4' IGNORE(`baz`);
master> CREATE BINLOG 'b5' IGNORE(`baz`.`bar`);
... etc ...
```

If a statement modifies multiple objects, (i.e. a multi-table update), it will be recorded to all binlogs that log both test.foo and test.bar.

```
master> UPDATE test.foo, test.bar
        SET test.foo.f = 2, test.bar.b = 2
        WHERE f = b AND f = 1;
```

The above sample will be logged to each of these binlogs.

```
master> CREATE BINLOG 'b1' LOG(`test`.`foo`, `test`.`bar`);
master> CREATE BINLOG 'b2' LOG(`test`);
master> CREATE BINLOG 'b3';
... etc ...
```

If at least one binlog does not include logging for all the tables of a multi-table update, the statement would be considered unsafe. See Unsafe Queries.

If the statement is part of a multi-statement transaction, only write statements will be recorded. For example, in this transaction, the highlighted statements are logged.

```
master> BEGIN;
master> SELECT * FROM test.bar;
master> INSERT INTO test.foo VALUES (1);
master> SELECT COUNT(*) FROM test.foo;
master> UPDATE test.foo SET f = 3 WHERE f = 2;
master> COMMIT;
```

Differences from MySQL

MySQL's --binlog-do-db and --binlog-ignore-db options have slightly different semantics than our binlog scopes. The MySQL documentation explains it well. Notable differences are:

- For statement-based logging, MySQL decides whether to log based on current default database, not based on statement. ClustrixDB decides based on the database modified by the statement.
- For row-based logging, statements that modify both something inside and something outside of the MySQL binlog are partially recorded. (These are *unsafe queries* in ClustrixDB.)

Restrictions

Some queries are not safe to log to database-scope or table-scope binlogs.

Unsafe Queries

Note that this section on unsafe queries applies to both statement-based (SBR) and row-based (RBR) logging.

Queries that modify both something inside and something outside of a binlog's scope cannot be safely logged. In other words, all writes must fall completely inside or completely outside of each binlog's scope.

For example, we first create two tables in the test database:

```
master> CREATE TABLE test.foo (f INT PRIMARY KEY);
master> CREATE TABLE test.bar (b INT PRIMARY KEY);
```

Next, we create a table-scope binlog for one of them:

```
master> CREATE BINLOG 'binny' LOG(`test`.`foo`);
```

Queries that modify just one table are fine:

```
/* This will be logged to binny */
master> INSERT INTO test.foo VALUES (1), (3);

/* This will NOT be logged to binny */
master> INSERT INTO test.bar VALUES (1), (3);
```

Queries that modify both tables cannot be logged to the binlog:

```
master> UPDATE test.foo JOIN test.bar ON (f = b)
        SET    f = 2, b = 2
        WHERE  b = 1;
ERROR 1 (HY000): [11314] This statement cannot be replicated safely: binlogs {binny} do not log both `test`.
`foo` and `test`.`bar`

master> DELETE test.foo.*, test.bar.*
        FROM   test.foo JOIN test.bar ON (f = b)
        WHERE  b = 3;
ERROR 1 (HY000): [11314] This statement cannot be replicated safely: binlogs {binny} do not log both `test`.
`foo` and `test`.`bar`

master> DROP TABLE test.foo, test.bar;
ERROR 1 (HY000): [11314] This statement cannot be replicated safely: binlogs {binny} do not log both `test`.
`foo` and `test`.`bar`
```

Resolve this issue by changing the binlog configuration or modifying the query.

Possibly unsafe statements

The following section applies to statement-based logging (SBR) and DDL with row-based logging (RBR).

Statements that read something outside of a binlog's scope before modifying something inside that binlog's scope might not be safe to log. It depends on whether the object being read (called a dependency) exists where and when the binlog is replayed.

Given this setup:

```
master> CREATE TABLE test.foo (f INT PRIMARY KEY);
master> CREATE TABLE test.bar (b INT PRIMARY KEY);

master> CREATE BINLOG 'binny' LOG(`test`.`foo`);
```

The following statements have test.bar as a dependency. Since it is unclear whether test.bar will exist where and when binny is replayed, ClustrixDB issues an error.

```
master> INSERT INTO test.foo
               SELECT * FROM test.bar;
ERROR 1 (HY000): [11314] This statement cannot be replicated safely: binlog binny does not log `test`.`bar`,
consequently the statement might not replay correctly

master> UPDATE test.foo JOIN test.bar ON (f = b)
        SET f = 2
        WHERE b = 1;
ERROR 1 (HY000): [11314] This statement cannot be replicated safely: binlog binny does not log `test`.`bar`,
consequently the statement might not replay correctly

master> DELETE test.foo.*
        FROM test.foo JOIN test.bar ON (f = b)
        WHERE b = 3;
ERROR 1 (HY000): [11314] This statement cannot be replicated safely: binlog binny does not log `test`.`bar`,
consequently the statement might not replay correctly
```

Unsafe Transactions

The following section on unsafe transactions applies to both statement-based (SBR) and row-based (RBR) logging.

Like unsafe queries, transactions that modify both something inside of and something outside of a binlog's scope are unsafe.

Given this setup:

```
master> CREATE TABLE test.foo (f INT PRIMARY KEY);
master> CREATE TABLE test.bar (b INT PRIMARY KEY);

master> CREATE BINLOG 'binny' LOG(`test`.`foo`);
```

Transactions that modify both tables cannot be logged to binny:

```
master> BEGIN;
Query OK, 0 rows affected (0.01 sec)

master> INSERT INTO test.foo VALUES (1), (3);
Query OK, 2 rows affected (0.03 sec)

master> INSERT INTO test.bar VALUES (1), (3);
ERROR 1 (HY000): [11314] This statement cannot be replicated safely: binlogs {binny} only record some of the
changes made by the transaction
```

Rules for specific statements and objects

Account Management Statements

Account management statements (CREATE USER, DROP USER, RENAME USER, SET PASSWORD, GRANT, REVOKE) count as modifying the system database. For example, these binlogs log system activity.

```
master> CREATE BINLOG 'b1';
master> CREATE BINLOG 'b2' IGNORE(`test`);
master> CREATE BINLOG 'b3' LOG(`system`);
```

All of the following statments will be logged to those binlogs:

```
master> CREATE USER 'example'@'%';
master> RENAME USER 'example'@'%' TO 'example1'@'localhost';
master> SET PASSWORD FOR 'example1'@'localhost' = PASSWORD('12345');
master> GRANT ALL ON *.* TO 'example1'@'localhost' WITH GRANT OPTION;
master> REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'example1'@'localhost';
master> DROP USER 'example1'@'localhost';
```

GRANT and REVOKE statements that refer to specific databases or tables only count as modifying the system database, and not the objects referenced.

See also Replicating User Account Management Statements.

Temporary tables

In addition to the normal binlog scope rules, there are special rules for temporary tables.

1. If binlog format is ROW, we do not log temporary tables.
2. If the binlog format is switched to ROW or the binlog fell out of scope, we stop logging that temporary table. Resetting the binlog's format to STAT EMENT does not restart the logging.
3. But there's an exception to these two rules: DROP TEMPORARY TABLE IF EXISTS statements must sometimes be logged to RBR binlogs or even binlogs with a different scope.

If temporary tables were never involved in permanent modifications, we wouldn't need to log them at all. For row format this is true, and thus we have the first rule. But in statement format it is possible to mix permanent modifications and temporary tables. For example:

```
master> CREATE TABLE perm (id INT PRIMARY KEY);
master> CREATE TEMPORARY TABLE temp (id INT PRIMARY KEY);
master> INSERT INTO temp VALUES (1), (2), (3);
master> INSERT INTO perm SELECT * FROM temp;
```

In statement format we need to log temporary tables. It is the same in MySQL.

The second rule is to prevent users from accidentally losing some information if a temporary table is logged incompletely. For example, if we create a temporary table and then switch binlog format to ROW, we must not continue logging that temporary table even if binlog format is switched back to STATE MENT.

```
master> CREATE BINLOG bin FORMAT='STATEMENT';
master> CREATE TABLE perm (id INT PRIMARY KEY);
master> CREATE TEMPORARY TABLE temp (id INT PRIMARY KEY);
master> INSERT INTO temp VALUES (1), (2), (3);
master> SET SESSION binlog_format = 'ROW';                /* At this point we stop logging modifications to
temp because of the first rule. */
master> INSERT INTO temp VALUES (4), (5), (6);
master> SET SESSION binlog_format = 'STATEMENT';
master> INSERT INTO perm SELECT * FROM temp;              /* We must not allow this, otherwise perm will be
missing data when the binlog is replayed. */
```

The only exception to this rule is the third rule: DROP TABLE statements sometimes need to be logged, even when we've stopped logging a temporary table, and even in row format. For example, if we create a temporary table and then switch binlog format, we must still log the DROP TABLE statement or we'll leak the temporary table.

```
master> SET SESSION binlog_format = 'STATEMENT';
master> CREATE TEMPORARY TABLE temp (id INT PRIMARY KEY);
master> INSERT INTO temp VALUES (1), (2), (3);
master> SET SESSION binlog_format = 'ROW';
master> INSERT INTO temp VALUES (4), (5), (6);    /* We do not log this insert, because of the first rule. */
master> DROP TEMPORARY TABLE temp;                /* But we do log this DROP TABLE, to prevent leaking the
temporary table on the slave when replaying the binlog. */
```

A similar situation occurs when a temporary table falls out of scope for some reason:

```
master> CREATE BINLOG bin LOG(test.foo), FORMAT='STATEMENT';
master> CREATE TEMPORARY TABLE test.foo (f INT PRIMARY KEY);
master> INSERT INTO foo VALUES (1), (2), (3);
master> ALTER BINLOG bin LOG(test.bar);
master> INSERT INTO foo VALUES (4), (5), (6);     /* We do not log this insert, because of binlog scope. */
master> DROP TEMPORARY TABLE foo;                 /* But we do log this DROP TABLE, even though it's no longer
in the binlog's scope. */
```

MySQL also logs artificial DROP TEMPORARY TABLE IF EXISTS statements when necessary.

And of course, all of the restrictions from above still apply. So if we have a row-format binlog, statements writing to both a temporary table and a normal table are not allowed.

Caveats for Binlog Scope

The following caveats apply for binlogs the are scoped by database or table.

- The WebUI does not show information about the scope of binlogs.
- When using SBR, LOAD DATA INFILE is treated like a system operation and may not be replicated correctly in the binlog. To work around this, you can use SET SESSION binlog_format = "ROW" before executing LOAD DATA INFILE.
- Removing all scope from a binlog does not automatically convert it to a non-scoped binlog. To do this you must ALTER BINLOG ... LOG ALL.

Caveats for Table Scope Binlogs

- Table-scope binlogs that use SBR (statement-based replication) and Triggers are not supported.
- Table-scope binlogs must be created after their parent databases are created.
- Views:
    - Statements affecting Views are logged to the binlog of their parent database.
    - It is not possible to create binlogs scoped by views.
- Stored Procedures:
    - For SBR, Stored Procedures are associated with a database and not a table, so related events will be logged to binlogs logging the database and not binlogs only logging tables.
    - If you use Stored Procedures and table-scope binlogs, RBR is recommended.