PhpAudit Documentation

P.R. Water

Jan 26, 2023

Contents

PhpAudit is a tool for creating and maintaining audit tables and triggers for creating audit trails of data changes in MySQL and MariaDB databases.

PhpAudit has the following features:

- Creates audit tables for tables in your database for which auditing is required.
- Creates triggers on tables for recording inserts, updates, and deletes of rows.
- Helps you to maintain audit tables and triggers when you modify your application's tables.
- Reports differences in table structure between your application's tables and audit tables.
- Disabling triggers under certain conditions.
- Flexible configuration. You can define additional columns to audit tables, for example: logging user and session IDs.

Using the audit trail you track changes made to the data of your application by the users of the application. Even of data that has been deleted or changed back to its original state. Also, you can track how your application manipulates data and find bugs if your application.

CHAPTER 1

Table of Contents

1.1 Getting Started

In this chapter you will learn how to install PhpAudit and start creating audit trails on your application data.

1.1.1 Installing PhpAudit

The preferred way to install PhpAudit is using composer:

```
composer require setbased/php-audit
```

1.1.2 Running PhpAudit

You can run PhpAudit from the command line:

```
./vendor/bin/audit
```

If you have set `bin-dir` in the `config` section in `composer.json` you must use a different path.

1.1.3 Twin Schemata

PhpAudit requires two schemata (databases):

- One schema (database) for your application tables. We call this schema the `data schema`.
- One schema (database) for the audit tables. We call this schema the `audit schema`.

PhpAudit will create an audit table in the `audit schema` for recording the audit trail with the same name as the table in the `data schema`. You can use any (valid) name for these two schemata (database).

1.1.4 The Audit Configuration File

The audit configuration file specification is described in detail in *The Audit Config File*. In this section we provide an example audit configuration file.

```
{
  "database": {
    "host": "localhost",
    "user": "foo_owner",
    "password": "s3cr3t",
    "data_schema": "foo_data",
    "audit_schema": "foo_audit"
  },
  "audit_columns": [
    {
      "column_name": "audit_timestamp",
      "column_type": "timestamp not null default now()",
      "expression": "now()"
    },
    {
      "column_name": "audit_statement",
      "column_type": "enum('INSERT','DELETE','UPDATE') character set ascii collate␣
↪ascii_general_ci not null",
      "value_type": "ACTION"
    },
    {
      "column_name": "audit_type",
      "column_type": "enum('OLD','NEW') character set ascii collate ascii_general_ci␣
↪not null",
      "value_type": "STATE"
    },
    {
      "column_name": "audit_uuid",
      "column_type": "bigint(20) unsigned not null",
      "expression": "@audit_uuid"
    },
    {
      "column_name": "audit_rownum",
      "column_type": "int(10) unsigned not null",
      "expression": "@audit_rownum"
    }
  ],
  "additional_sql": [
    "if (@audit_uuid is null) then",
    "  set @audit_uuid = uuid_short();",
    "end if;",
    "set @audit_rownum = ifnull(@audit_rownum, 0) + 1;"
  ]
}
```

The audit configuration file consists out of 3 sections:

- The `database` section, we will discuss this section below and in detail in *The Database Section*.
- The `audit_columns` section. See *The Audit Columns Section* for a detailed explanation.
- The `additional_sql` section. See *The Additional SQL Section* for a detailed explanation.

The `database` section holds the variables described below:

- `host` The host were the MySQL server is running
- `user` The user that is the *owner* of the tables in the `data schema` and `audit schema`. See *Required Grants* for an exact specification of required grants.
- `password` The password of the *owner*. In *The Database Section* we describe how to store the password outside the audit configuration file.
- `data_schema` The schema (database) with your application tables.
- `audit_schema` The schema (database) for the audit tables. The `data schema` and the `audit schema` must be two different schemata (databases).

Throughout this manual we assume that the audit configuration file is stored in `etc/audit.json`. You are free to choose your preferred path.

Run PhpAudit with the `audit` command:

```
./vendor/bin/audit audit etc/audit.json
```

Output:

```
Found new table FOO_EMPLOYEE
Wrote etc/audit.json
```

The first time you run the audit command PhpAudit will only report the tables found in the `data schema` and add the tables in the `tables` section in the audit configuration file. Suppose you application has a table `FOO_EMPLOYEE`, the `tables` section will look like:

```
{
  "database": {},
  "audit_columns": [],
  "additional_sql": [],
  "tables": {
    "FOO_EMPLOYEE": {
      "audit": null,
      "alias": null,
      "skip": null
    }
  }
}
```

For all tables for which you want an audit trail you must set the audit flag to true. In our example:

```
{
  "database": {},
  "audit_columns": [],
  "additional_sql": [],
  "tables": {
    "FOO_EMPLOYEE": {
      "audit": true,
      "alias": null,
      "skip": null
    }
  }
}
```

and rerun PhpAudit with the `audit` command:

```
./vendor/bin/audit audit etc/audit.json
```

Output:

```
Creating audit table foo_audit.FOO_EMPLOYEE
Wrote etc/audit.json
```

You can now insert, update, and delete rows in/from table `foo_data.FOO_EMPLOYEE` and see the recorded audit trail in table `foo_audit.FOO_EMPLOYEE`.

1.1.5 Verbosity

In verbose mode (`-v`) the `audit` command will show triggers dropped and created:

```
./vendor/bin/audit -v audit etc/audit.json
```

Output:

```
Creating audit table foo_audit.FOO_EMPLOYEE
Creating trigger foo_data.trg_audit_5d7a1d1e18ada_insert on table foo_data.FOO_
↪EMPLOYEE
Creating trigger foo_data.trg_audit_5d7a1d1e18ada_update on table foo_data.FOO_
↪EMPLOYEE
Creating trigger foo_data.trg_audit_5d7a1d1e18ada_delete on table foo_data.FOO_
↪EMPLOYEE
Wrote etc/audit.json
```

In very verbose mode (`-vv`) PhpAudit will show each executed SQL statement also.

1.2 Installing & Uninstalling PhpAudit

1.2.1 Installing PhpAudit

The preferred way to install PhpAudit is using composer:

```
composer require setbased/php-audit
```

Running PhpAudit

You can run PhpAudit from the command line:

```
./vendor/bin/audit
```

If you have set `bin-dir` in the `config` section in `composer.json` you must use a different path.

For example:

```
{
  "config": {
    "bin-dir": "bin/"
  }
}
```

then you can run PhpAudit from the command line:

```
./bin/audit
```

1.2.2 Uninstalling PhpAudit

Before you uninstall PhpAudit you must delete all audit triggers from the tables in the `data schema`. This can be done with the `drop-triggers` command:

```
./vendor/bin/audit drop-triggers etc/config.json
```

Remove PhpAudit from your project with composer:

```
composer remove setbased/php-audit
```

1.3 An Example

In this section we give a real world example taken from a tournament on the Nahouw. We have reduced the tournament table to two columns and changed some IDs for simplification.

```
select *
from   nahouw_data.NAH_TOURNAMENT
where  trn_id = 4473
```

Output:

trn_id	trn_name
4773	Correct name

The audit trail for this tournament:

```
select *
from   nahouw_audit.NAH_TOURNAMENT
where  trn_id = 4473
```

audit_timestamp	audit_statement	audit_state	audit_uuid	audit_rownum	audit_ses_id	audit_usr_id	trn_id	trn_name
2012-05-05 08:36:06	INSERT	NEW	3106165035085332	789	34532889	65	4773	Wrong name
2013-02-01 10:55:01	UPDATE	OLD	3110371421365215	378	564977477	107	4773	Wrong name
2013-02-01 10:55:01	UPDATE	NEW	3110371421365215	378	564977477	107	4773	Correct name

Notice that the audit table has 7 additional columns. You can configure more or less columns and name them to your needs.

- `audit_timestamp`: The time the statement was executed.
- `audit_statement`: The type of statement. One of INSERT, UPDATE, OR DELETE.
- `audit_sate`: The state of the row. NEW or OLD.

- `audit_uuid`: A UUID per database connection. Using this ID we can track all changes made during a page request.
- `audit_rownum`: The number of the audit row within the UUID. Using this column we can track the order in which changes are made during a page request.
- `audit_ses_id`: The ID the session of the web application.
- `audit_usr_id`: The ID of the user has made the page request.

From the audit trail we can see that user 65 has initially entered the tournament with a wrong name. We see that the tournament insert statement was the second statement executed. Using UUID 310616503508533789 we found the first statement was an insert statement of the tournament's location which is stored in another table. Later user 107 has changed the tournament name to its correct name.

On table `nahouw_data.NAH_TOURNAMENT` we have three triggers, one for insert statements, one for update statements, and one for delete statements. Below is the code for the update statement (the code for the other triggers look similar).

```
create trigger `nahouw_data`.`trg_trn_update`
after UPDATE on `nahouw_data`.`NAH_TOURNAMENT`
for each row
begin
  if (@audit_uuid is null) then
    set @audit_uuid = uuid_short();
  end if;
  set @audit_rownum = ifnull(@audit_rownum, 0) + 1;
  insert into `nahouw_audit`.`NAH_TOURNAMENT`(audit_timestamp,audit_type,audit_state,
↪audit_uuid,rownum,audit_ses_id,audit_usr_id,trn_id,trn_name)
  values(now(),'UPDATE','OLD',@audit_uuid,@audit_rownum,@audit_ses_id,@audit_usr_id,
↪OLD.`trn_id`,OLD.`trn_name`);
  insert into `nahouw_audit`.`NAH_TOURNAMENT`(audit_timestamp,audit_type,audit_state,
↪audit_uuid,rownum,audit_ses_id,audit_usr_id,trn_id,trn_name)
  values(now(),'UPDATE','NEW',@audit_uuid,@audit_rownum,@audit_ses_id,@audit_usr_id,
↪NEW.`trn_id`,NEW.`trn_name`);
end
```

1.4 The Audit Config File

This chapter is the specification of the audit config file.

For most projects the audit config file must added to the VCS and distributed to the production environment of your project (unless you have some other mechanism for maintaining audit tables and triggers).

The audit config file is a JSON file and consist out of four sections which we discuss in detail in the following sections.

```
{
  "database": {},
  "audit_columns": [],
  "additional_sql": [],
  "tables": {}
}
```

1.4.1 The Database Section

The `database` section holds the variables described below:

- `credentials` (optional) The filename relative to the path of the audit config file of a supplementary configuration file. Any configuration setting in the supplementary configuration file will override the setting in `database` section of the audit config file. You can choose your favorite configuration format for the credentials file: ini, json, xml, or yml. You can only store the password in the supplementary configuration file or all database settings.
- `host` (mandatory) The host were the MySQL server is running
- `user` (mandatory) The user that is the *owner* of the tables in the `data schema` and `audit schema`.
- `password` (mandatory) The password of the *owner*.
- `data_schema` (mandatory) The schema (database) with your application tables.
- `audit_schema` (mandatory) The schema (database) for the audit tables. The `data schema` and the `audit schema` must be two different schemata (databases).
- `port` (optional) The port number for connecting to the MySQL server. Default value is 3306.

Convention

You are encouraged to follow this naming convention for the `data schema` and `audit schema`.

Both schema (databases) names start with the name or abbreviation of your project followed by `_data` for the `data schema` and `_audit` for the `audit schema`. For example `foo_data` and `foo_audit`.

Examples

Example 1

A basic example.

`audit.json`:

```
{
  "database": {
    "host": "localhost",
    "user": "foo_owner",
    "password": "s3cr3t",
    "data_schema": "foo_data",
    "audit_schema": "foo_audit"
  }
}
```

Example 2

In this example the password stored in `credentials.ini` will be used.

`audit.json`:

```
{
  "database": {
    "credentials": "credentials.ini",
    "host": "localhost",
    "user": "foo_owner",
    "password": "foo_owner",
```

(continues on next page)

(continued from previous page)

```
    "data_schema": "foo_data",
    "audit_schema": "foo_audit"
  }
}
```

credentials.ini:

```
[database]
password =  s3cr3t
```

Example 3

In this example the user name and password stored in `credentials.xml` will be used.

audit.json:

```
{
  "database": {
    "credentials": "credentials.xml",
    "host": "localhost",
    "data_schema": "foo_data",
    "audit_schema": "foo_audit"
  }
}
```

credentials.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <database>
        <user>foo_owner</user>
        <password>s3cr3t</password>
    </database>
</config>
```

Example 4

In this example only settings stored in `credentials.json` will be used.

audit.json:

```
{
  "database": {
    "credentials": "credentials.json"
  }
}
```

credentials.json:

```
{
  "database": {
    "host": "127.0.0.1",
    "user": "foo_owner",
```

(continues on next page)

(continued from previous page)

```
    "password": "foo_owner",
    "data_schema": "foo_data",
    "audit_schema": "foo_audit",
    "port": 3307
  }
}
```

1.4.2 The Audit Columns Section

The audit columns section specifies the additional columns that will added to each audit table in the `audit schema`.

The additional column specification become in two flavors:

- value is either the action (i.e. `insert`, `update`, or `delete`) or the state of the row (i.e. `NEW` or `OLD`),
- value is a valid SQL expression that can be used in an insert statement in a trigger.

Example

```
{
  "audit_columns": [
    {
      "column_name": "flavor 1",
      "column_type": "...",
      "value_type": "..."
    },
    {
      "column_name": "flavor 2",
      "column_type": "...",
      "expression": "..."
    }
  ]
}
```

Both flavors have the fields `column_name` and `column_type` in common.

- `column_name` The name of the additional column in the audit table. You must choose a name that is not been used in any of your tables in the `data schema` (for which auditing is enabled).
- `column_type` The column type specification as used in a `CREATE TABLE` statement.
- `value_type` Either `ACTION` or `STATE`.
  - `ACTION` The action of the SQL statement that has fired the audit trigger. Possible values are `INSERT`, `UPDATE`, or `DELETE`.
  - `STATE` The state of the row.
    - An insert statement will insert one row in the audit table with value `NEW`.
    - A delete statement will insert one row in the audit table with value `OLD`.
    - An update statement will insert two rows in the audit table: `OLD` with the values of the row (in the data table) before the update statement and `NEW` with the values of the row (in the data table) after the update statement.
- `expression` Any valid SQL expression that can be used in an insert statement in a trigger.

Convention

You free to choose any column name for an additional table column as long the column name does not collide with a column name in a data table.

You are encouraged to follow this naming convention for the additional table column: the name of an additional table column has prefix `audit_`.

Examples

In this section we provide several useful examples for additional columns.

Additional columns are optional, however, in practice additional columns given in examples 1, 2, and 3 are at least required to record a useful audit trail.

Examples 4 and 5 for recording all data changes made in a database session and the order in which they are made.

Example 1: Timestamp

Recording the time of the data change.

```
{
  "audit_columns": [
    {
      "column_name": "audit_timestamp",
      "column_type": "timestamp not null default now()",
      "expression": "now()"
    }
  ]
}
```

Example 2: Statement Type

Recording the statement type of the query.

```
{
  "audit_columns": [
    {
      "column_name": "audit_statement",
      "column_type": "enum('INSERT','DELETE','UPDATE') character set ascii collate␣
↪ascii_general_ci not null",
      "value_type": "ACTION"
    }
  ]
}
```

Example 3: Row State

Recording the state of the row.

```
{
  "audit_columns": [
    {
      "column_name": "audit_type",
      "column_type": "enum('OLD','NEW') character set ascii collate ascii_general_ci␣
↪not null",
      "value_type": "STATE"
    }
  ]
}
```

Example 4: Database Session

Recording the database session (a single connection from your PHP application to the MySQL instance). See *The Additional SQL Section* for setting the user defined variable `@audit_uuid` in MySQL.

```
{
  "audit_columns": [
    {
      "column_name": "audit_uuid",
      "column_type": "bigint(20) unsigned not null",
      "expression": "@audit_uuid"
    }
  ],
  "additional_sql": [
    "if (@audit_uuid is null) then",
    "  set @audit_uuid = uuid_short();",
    "end if;",
  ]
}
```

Example 5: Order

Recording the order of the data changes. See *The Additional SQL Section* for setting the user defined variable `@audit_rownum` in MySQL.

```
{
  "audit_columns": [
    {
      "column_name": "audit_rownum",
      "column_type": "int(10) unsigned not null",
      "expression": "@audit_rownum"
    }
  ],
  "additional_sql": [
    "set @audit_rownum = ifnull(@audit_rownum, 0) + 1;"
  ]
}
```

Example 6: Database User

Recording the database user connection to the server. This example is useful when different database user can connect to your database. For example you have an application with a HTML frontend connecting to the database with user

web_user, a REST API connecting to the database with user api_user, and some background process connecting to the database with user mail_user.

```
{
  "audit_columns": [
    {
      "column_name": "audit_user",
      "column_type": "varchar(80) character set utf8 collate utf8_bin not null",
      "expression": "user()"
    }
  ]
}
```

On MariaDB the maximum length of a user name is 80 characters, on mysql the maximum length of a user name is 32 characters.

Example 7: Application Session

Recording the session ID. This example is useful tracking data changes made in multiple page request in a single session of a web application.

```
{
  "audit_columns": [
    {
      "column_name": "audit_ses_id",
      "column_type": "int(10) unsigned",
      "expression": "@audit_ses_id"
    }
  ]
}
```

When retrieving the session you must set the user defined variable @audit_ses_id in MySQL from your PHP application. See *Setting User Defined Variables in MySQL* for examples of setting user defined variables in MySQL.

Example 8: End User

Recording the user ID. This example is useful for recording the end user who has modified the data using your PHP application.

```
{
  "audit_columns": [
    {
      "column_name": "audit_usr_id",
      "column_type": "int(10) unsigned",
      "expression": "@audit_usr_id"
    }
  ]
}
```

When retrieving the session and when signing in or off you must set the user defined variable @audit_usr_id in MySQL from your PHP application. See *Setting User Defined Variables in MySQL* for examples of setting user defined variables in MySQL.

1.4.3 The Additional SQL Section

The additional SQL section specifies additional SQL statements that are placed at the beginning of the body of each created audit trigger.

Example

This example show how to set the variables `@audit_uuid` and `@audit_rownum` mentioned in sections *Example 4: Database Session* and *Example 5: Order*.

```
{
   "additional_sql": [
      "if (@audit_uuid is null) then",
      "  set @audit_uuid = uuid_short();",
      "end if;",
      "set @audit_rownum = ifnull(@audit_rownum, 0) + 1;"
    ]
}
```

1.4.4 The Tables Section

The tables sections holds an entry for each table in the `data schema`. New tables are automatically added to the tables section and obsolete tables are automatically removed from the tables section when your run PhpAudit with the `audit` command.

Foreach table in the table section there are three fields:

- `audit` The audit flag. A boolean indication auditing is enabled or disabled.
  - `true` Recording of an audit trail for this table is enabled.
  - `false` Recording of an audit trail for this table is disabled.
  - `null` Recording of an audit trail for this table is not specified. Each time your run PhpAudit with the `audit` command PhpAudit will report that a new table has been found.
- `alias` An alias for the table. This alias must be unique and will be used in the names of the audit trigger for this table. If you don't specify a value PhpAudit will generate automatically an alias when auditing is enabled.
- `skip` An optional variable name. When the value of this variable is not null the audit trigger will skip recording data changes.

When you disable recording of an audit trail of a table the audit triggers will be removed, however, the audit table will remain in the `audit schema`.

Examples

Example 1: No audit trail

No audit trail will be recorded for table `TMP_IMPORT`.

```
{
  "tables": {
      "TMP_IMPORT": {
        "audit": false,
```

(continues on next page)

(continued from previous page)

```
        "alias": null,
        "skip": null
      }
    }
}
```

Example 2: Audit trail

An audit trail will be recorded for table `FOO_USER`.

```
{
  "tables": {
      "FOO_USER": {
        "audit": true,
        "alias": "usr",
        "skip": "@audit_skip_foo_user"
      }
    }
}
```

When user defined variable `@audit_skip_foo_user` in MySQL is set no audit triggers will record data changes. In the SQL code below updating column `usr_last_login` will not be recorded.

```
set @audit_skip_foo_user = 1;

update FOO_USER
set    usr_last_login = now()
where  usr_id = p_usr_id
;

set @audit_skip_foo_user = null;
```

1.5 The PhpAudit Program

The PhpAudit program is a Symphony console application with four additional commands:

- `alter-audit-table`
- `audit`
- `diff`
- `drop-triggers`

We discuss the additional commands in the sections below.

Some commands provide additional output when in verbose mode (`-v`). All commands show the queries been executed in very verbose mode (`-vv`).

```
$ ./vendor/bin/audit
audit 1.0.0

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output,
2 for more verbose output and 3 for debug

Available commands:
  alter-audit-table  Creates alter SQL statements for audit tables
  audit              Maintains audit tables and audit triggers
  diff               Compares data tables and audit tables
  drop-triggers      Drops all triggers
  help               Displays help for a command
  list               Lists commands
$
```

1.5.1 The `alter-audit-table` command

The `alter-audit-table` command generates SQL statements to align the tables in the `audit schema` with the tables in the `data schema` and the additional columns section in the audit config file, see *The Audit Columns Section*.

```
./vendor/bin/audit alter-audit-table etc/audit.json
```

Output:

```
alter table `test_audit`.`FOO_EMPLOYEE`
change column `emp_name` `emp_name` varchar(80) character set utf8 collate utf8_
↪general_ci null
;
```

You must inspect each SQL statement manually before executing. See *Changed Column Type* for an explanation about incompatible column types.

1.5.2 The `audit` command

The `audit` command creates and alters audit tables (in the `audit schema` and audit triggers on tables in the `data schema`.

Usage of the `audit` command is discussed ind detail in *Getting Started* and *Schema Changes and Deployment*.

1.5.3 The `diff` command

The diff command show in a graphical manner the differences between the tables in the `audit schema` with the tables in the `data schema` and the additional columns section in the audit config file, see *The Audit Columns Section*.

Without the `--full` option only tables with differences and only different columns are shown. With the `--full` option all tables and all columns are shown.

```
$ ./vendor/bin/audit diff etc/audit.json
FOO_EMPLOYEE
+----------+-------------------------+-------------------------+
| column   | audit table             | config / data table     |
+----------+-------------------------+-------------------------+
| emp_name | varchar(60)             | varchar(80)             |
|          | [utf8] [utf8_general_ci] | [utf8] [utf8_general_ci] |
+----------+-------------------------+-------------------------+
+----------+-------------------------+-------------------------+

$ ./vendor/bin/audit diff --full etc/audit.json
FOO_EMPLOYEE
+--------------------+-----------------------------------------+-----------------------------------------+
| column             | audit table                             | config / data table                     |
+--------------------+-----------------------------------------+-----------------------------------------+
| audit_timestamp    | timestamp not null                      | timestamp not null                      |
| audit_statement    | enum('INSERT','DELETE','UPDATE') not null | enum('INSERT','DELETE','UPDATE') not null |
|                    | [ascii] [ascii_general_ci]              | [ascii] [ascii_general_ci]              |
| audit_type         | enum('OLD','NEW') not null              | enum('OLD','NEW') not null              |
|                    | [ascii] [ascii_general_ci]              | [ascii] [ascii_general_ci]              |
| audit_uuid         | bigint(20) unsigned not null            | bigint(20) unsigned not null            |
| audit_rownum       | int(10) unsigned not null               | int(10) unsigned not null               |
| audit_user         | varchar(80) not null                    | varchar(80) not null                    |
|                    | [utf8] [utf8_bin]                       | [utf8] [utf8_bin]                       |
| emp_id             | int(10) unsigned                        | int(10) unsigned                        |
| emp_name           | varchar(60)                             | varchar(80)                             |
|                    | [utf8] [utf8_general_ci]                | [utf8] [utf8_general_ci]                |
| emp_salary         | decimal(10,2)                           | decimal(10,2)                           |
| emp_role           | varchar(20)                             | varchar(20)                             |
|                    | [utf8] [utf8_general_ci]                | [utf8] [utf8_general_ci]                |
+--------------------+-----------------------------------------+-----------------------------------------+
| engine             | InnoDB                                  | InnoDB                                  |
| character_set_name | utf8                                    | utf8                                    |
| table_collation    | utf8_general_ci                         | utf8_general_ci                         |
+--------------------+-----------------------------------------+-----------------------------------------+

$
```

1.5.4 The `drop-triggers` command

The `drop-triggers` command will drop all triggers in the `data schema`.

```
$ ./vendor/bin/audit drop-triggers etc/audit.json
Dropping trigger trg_audit_emp_delete from table FOO_EMPLOYEE
Dropping trigger trg_audit_emp_insert from table FOO_EMPLOYEE
Dropping trigger trg_audit_emp_update from table FOO_EMPLOYEE
 File etc/audit.json is up to date
$
```

1.6 Schema Changes and Deployment

During the life time of your application there will schema changes:

- new tables will be created,
- obsolete tables will be dropped,
- tables will be renamed,
- table options will change,
- new columns will added to a table,
- obsolete columns will be drop from a table,
- columns will be renamed,
- column types will change.

In this chapter we discuss how to handle all these types of changes. Also, we discuss how to deploy schema changes on the production environment.

PhpAudit becomes with two commands that helps you to compare the `data schema` with the `audit schema`:

- The `diff` command, see *The diff command*.
- The `alter-audit-table` command, see *The alter-audit-table command*.

1.6.1 Schema Changes

In this section we discuss all possible schema changes one by one. You can combine many schema changes on one go.

New Table

When adding a new table to the database of your application you must decide whether auditing is required for this table.

- Run the DDL statements for creating the new table.
- Run the `audit` command of PhpAudit. PhpAudit will report that it has found a new table.
  - Auditing is not required for the new table:
    - Set the audit flag for the new table to `false`.
  - Auditing is required for the new table:
    - Set the audit flag for the new table to `true`.
    - Run the `audit` command of PhpAudit again. This time an audit table and audit triggers will be created for the new table.
- Commit the changes in the audit config file to your VCS.

Obsolete Table

- Run the DDL statements for dropping the obsolete table.
- Run the `audit` command of PhpAudit. PhpAudit will report that it has found an obsolete table.
  - PhpAudit will remove the obsolete table from the tables section.
  - PhpAudit will not drop the table from the `audit schema`. The corresponding table in the `audit schema` is still a part of your application's audit trail.
- Commit the changes in the audit config file to your VCS.

If you decide now or later that the corresponding table in the `audit schema` is not longer required you must drop the corresponding table in the `audit schema` your self. This does not affect PhpAudit at all nor requires any action by PhpAudit.

Renamed Table

When you rename a table in the `data schema` there is no reliable way for PhpAudit to detect a table has been renamed. PhpAudit will see an obsolete and a new table.

- Run the DDL statements for renaming the table in the `data schema`.
- Run similar DDL statements for renaming the corresponding table in the `audit schema`.
- Rename the table in the tables section of the audit config file.
  - At this moment the audit triggers on the table in `data schema` are still using the old table name in the `audit schema`.
- Run the `audit` command of PhpAudit.
  - The audit triggers on the table in `data schema` are using new table name in the `audit schema` now.
- Commit the changes in the audit config file to your VCS.

If you omit renaming the table in the audit config file, PhpAudit will report an obsolete table and a new table. I this case you must restore the table configuration (i.e. the audit file, alias and skip variable) in the aut config file.

If you omit renaming the corresponding table in the `audit schema`, PhpAudit will create a new table in the `audit schema`.

Table Options

- Run the DDL statements for altering the table options of the table in the `data schema`.
- Run similar DDL statements for altering the table options of the corresponding table in the `audit schema`.

PhpAudit is unaware of most table options. It only considers the following table options:

- CHARACTER SET
- COLLATE
- ENGINE

Running the `audit` command of PhpAudit will not affect the table options of any table in the `audit schema`.

See XXX for a discussing about transactional and non transaction storage engines.

New Column

- Run the DDL statements for adding the new column to the table `data schema`.
- Run the `audit` command of PhpAudit.
  - The new column will be added to the corresponding table in the `audit schema` and added to the queries in the audit triggers.

Obsolete Column

- Run the DDL statements for dropping the obsolete column from the table `data schema`.
- Run the `audit` command of PhpAudit.
  - The obsolete column will be removed from the queries in the audit triggers.
  - The obsolete column in the corresponding table in the `audit schema` is still a part of your application's audit trail and will not be dropped.

If you decide now or later that the obsolete column in the corresponding table in the `audit schema` is not longer required you must drop the obsolete column in the corresponding table in the `audit schema` your self. This does not affect PhpAudit at all nor requires any action by PhpAudit.

Renamed Column

When you rename a column of a table in the `data schema` there is no reliable way for PhpAudit to detect a column has been renamed. PhpAudit will see an obsolete and a new column.

- Run the DDL statements for renaming the column of the table in the `data schema`.
- Run similar DDL statements for renaming the column of the corresponding table in the `audit schema`.
  - At this moment the audit triggers on the table in `data schema` are still using the old column name.
- Run the `audit` command of PhpAudit.
  - The audit triggers on the table in the `data schema` are using the new column name now.

Changed Column Type

We consider two types of column type changes:

- Changing the column type to a more comprehensive column type. For example:
  - `varchar(10) charset utf8 collation utf8_general_ci => varchar(20) charset utf8 collation utf8_general_ci`
  - `varchar(80) charset ascii collation ascii_general_ci => varchar(80) charset utf8 collation utf8_general_ci`
  - `smallint(4) => int(6)`
- Changing the column type to a less comprehensive or incompatible column type: For example:
  - `varchar(10) charset utf8 collation utf8_general_ci => int(10)`
  - `varchar(80) charset utf8 collation utf8_general_ci => varchar(80) charset latin1 collation latin1_general_ci`
  - `longblob => medium text`

Currently, automatically modification of columns of tables in the `audit schema` is not implemented and planned for a future release.

We consider three kinds of less comprehensive or incompatible column types:

- The audit trail does not contain any data that cannot be converted to the new column type. For example:
  - A `varchar(10)` that holds only integers (as strings) in both the data and audit table can be modified to an `int(10)` without any issues.

  - A `varchar(80) charset utf8 collation utf8_general_ci` that holds only latin1 characters in both the data and audit table can be modified to an `varchar(80) charset latin1 collation latin1_general_ci` without any issues.

- The audit trail does contain data that cannot be converted to the new column type however a more comprehensive column type (for the actual data in both columns in the `data schema` and `audit schema`) is available. For example:
  - A `varchar(10) charset utf8 collation utf8_general_ci` (that must be modified to `varchar(30) charset latin1 collation latin1_general_ci`) that holds only latin1 characters in the data table, but the audit table holds data outside the latin1 character set. In this case the column in the `data schema` can be converted to `varchar(30) charset latin1 collation latin1_general_ci` and the column in the `audit schema` can be converted to `varchar(30) charset utf8 collation utf8_general_ci`.
- The audit trail does contain data that cannot be converted to the new column type and a more comprehensive column type is not available. For example:
  - A `varbinary(10)` (that must be modified to `int(10)`) column holding binary in the audit trail but not any more in the data table.

  In this case to only solution is to rename the column in the audit table. The `audit` command of PhpAudit will create a new column in the audit table with the new column type.

1.6.2 Deployment

In the above section we discuss all possible schema changes one by one. of course you can combine all schema changes in one go. The basic rules are simple:

- Renaming tables:
  - Rename the tables in the `data schema`.
  - Rename the corresponding tables in the `audit schema`.
  - Rename the tables in the audit config file.
- Renaming columns:
  - Rename the columns in the `data schema`.
  - Rename the columns in the corresponding tables in the `audit schema`.
- Changing column types:
  - Change the column types in the `data schema`.
  - Change the column types in the corresponding tables in the `audit schema`.
  - See *Changed Column Type* for incompatible column type changes.
- Run the `audit` command of PhpStratum.

Simple Deployment

If your deployment script has only DDL statements (affecting tables that require auditing), followed by only (or none) DML statements (affecting tables that require auditing), it is called a simple deployment. You must your deployment as scripts as follows:

- Run the DDL statements.
- Run the `audit` command of PhpAudit.

  - Use the latest version of your audit config file.
  - All audit tables and triggers are in a proper state to capture the data changes caused by the following DML statements.
- Run the DML statements.

Complex Deployment

If your deployment script has only DDL statements (affecting tables that require auditing), followed by only (or none) DML statements (affecting tables that require auditing), followed by only DDL statements (affecting tables that require auditing), followed by only (or none) DML statements (affecting tables that require auditing) and so on, it is called a complex deployment. You must your deployment as scripts as follows:

- Run DDL statements.
- Run the `audit` command of PhpAudit (with the latest version of you audit config file).
  - Use the latest version of your audit config file.
  - Make sure that the audit flags for are still correct.
  - All audit tables and triggers are in a proper state to capture the data changes caused by the following DML statements.
- Run DML statements.
- Run DDL statements.
- Run the `audit` command of PhpAudit (with the latest version of you audit config file).
  - Use the latest version of your audit config file.
  - Make sure that the audit flags for are still correct.
  - All audit tables and triggers are in a proper state to capture the data changes caused by the following DML statements.
- Run DML statements.
- and so on

1.7 Miscellaneous

In this chapter we discuss miscellaneous aspects of PhpAudit.

1.7.1 Required Grants

The (MySQL) user under which PhpAudit is connecting to the database instance requires the following grants:

- `data schema`:
  - `lock tables`
  - `select`
  - `trigger`
- `audit schema`:
  - `create`

- drop
- insert
- select

For example:

```
create user `foo_audit`@`localhost`;
grant lock tables, select, trigger on `foo_data`.* to `foo_audit`@`localhost`;
grant create, drop, insert, select on `foo_audit`.* to `foo_audit`@`localhost`;
```

Remember a trigger is running under the definer, i.e. the user which the trigger is created.

1.7.2 Indexes

PhpAudit does not create any indexes on tables in the `audit schema`. Creating an audit trail is about inserting rows in audit tables only. Hence, PhpAudit does not requires any indexes.

If your application is querying on tables in the `audit schema` you are free to add indexes on the tables in the `audit schema`. PhpAudit will not drop or alter any indexes in the `audit schema`.

Be careful with unique indexes. A key of a table in the `data schema` will (very likely) not be a key of the corresponding table in the `audit schema`.

1.7.3 Setting User Defined Variables in MySQL

There are several ways for setting user defined variables in MySQL from your PHP application. In this section we discuss two methods. More information about user defined variables in MySQL can be found at https://mariadb.com/kb/en/user-defined-variables/ and https://dev.mysql.com/doc/refman/8.0/en/user-variables.html

Explicit Query From PHP

The PHP snippet below is an example of setting a user defined variable in MySQL from a PHP application.

```
// User has signed in and variable $usrId holds the ID of the user and
// $mysql is the connection to MySQL.
$mysql->real_query(sprintf('set @audit_usr_id = %s', $usrId ?? 'null'));
```

Implicit in SQL Query

The SQL statement below is an example of setting user defined variables in MySQL in a SQL statement (in this example session data is stored in table *FOO_SESSION*).

```
select @audit_ses_id := ses_id
,      @audit_usr_id := usr_id
,      ses_data
from   FOO_SESSION
where  ses_token = 'the-long-token-stored-in-the-session-cookie-of-the-user-agent'
;
```

1.7.4 Limitations

PhpAudit has the following limitations:

- A `TRUNCATE TABLE` statement will remove all rows from a table and does not activate any triggers. Hence, the removing of those rows will not be logged in the audit table.
- A delete or update of a child row caused by a cascaded foreign key action of a parent row will not activate triggers on the child table. Hence, the update or deletion of those rows will not be logged in the audit table.

Both limitations arise from the behavior of MySQL. In practice these limitations aren't of any concern. In applications where tables are "cleaned" with a `TRUNCATE TABLE` we never had the need to audit these tables. We found the same for child tables with a `ON UPDATE CASCADE` or `ON UPDATE SET NULL` reference option.

1.8 License

The project is licensed under the MIT license.