Stealthy migrating MySQL tables and MySQL data access interfaces using enlarged updateable VIEW functionality

by Kris Köhntopp and Oli Sennhauser, MySQL AB.

Applications occasionally require redesign. However, redesigning an application cannot be done in one step because the application is distributed or several versions of applications must be supported. MySQL 5.0 provides the necessary means to stealthy migrate your data.

In a short overview let's look at what we plan to do:

**Original state:** We have a typical application accessing the data via SELECT and DML (INSERT, REPLACE, UPDATE, DELETE) commands. Because the table structure has to change in the near future we have to hide this against the application.

**Step 1, create interfaces:** To hide the table structure we cover it with a layer of VIEW's and Stored Procedures (SP). The Application can still access the data via the original paths. New applications can access the new interfaces.

**Step 2, clean-up of application:** When the interfaces are properly defined and implemented, the application can be migrated step by step to the new interfaces.

**Step 3, Change table structure:** When all the tables are covered by the new interfaces, the table structure can be changed and interface versions can be upgraded to the new table structure in one step.

**Step 4, Add new interface versions:** From now on, new interface versions can be added to provide new features, table structures can be changed, and support for older application versions can be better guaranteed.
**Step by step**

Let's first simulate the original state:

```sql
CREATE TABLE company_employee
(
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    company VARCHAR(128) NOT NULL,
    employee VARCHAR(128) NOT NULL
);

INSERT INTO company_employee
VALUES (1, 'MySQL', 'Hans Meier'),
       (2, 'MySQL', 'Hugo Huber'),
       (3, 'Tante Emma Laden', 'Hanne Hitz'),
       (4, 'Gegenueber Shop', 'Fritz Froehlich');
```

Our application consists of the following operations:

```sql
SELECT * FROM company_employee;

INSERT INTO company_employee
VALUES (NULL, 'Linux', 'Anton Albern');

UPDATE company_employee
SET employee = 'Berta Bach'
WHERE id = 5;

DELETE FROM company_employee
WHERE id = 5;
```

Everything is fine now. We can access our data and we can modify it. But, ... hmm, ok the data model behind this example is not that perfect. We sometimes read about normalization and 3rd normal form. So the goal is to stealthy migrate this table in 3rd normal form.

**Create the interfaces**

To hide this change, we have to first create some interfaces:
CREATE VIEW ce_select_if_v1 AS
SELECT id, company, employee
FROM company_employee;

DELIMITER //

CREATE PROCEDURE ce_insert_if_v1
(IN  company_name  VARCHAR(128)
  , IN  employee_name VARCHAR(128)
)
BEGIN
  INSERT INTO company_employee
  VALUES (NULL, company_name, employee_name);
END;
//

CREATE PROCEDURE ce_update_if_v1
(IN  company_id    INT
  , IN  employee_name VARCHAR(128)
)
BEGIN
  UPDATE company_employee
  SET employee = employee_name
  WHERE id = company_id;
END;
//

CREATE PROCEDURE ce_delete_if_v1
(IN  company_id    INT
)
BEGIN
  DELETE
  FROM company_employee
  WHERE id = company_id;
END;
//

DELIMITER ;

Now the old operations, as well as the following new ones should still work:

SELECT *
FROM ce_select_if_v1;

CALL ce_insert_if_v1('Linux', 'Anton Albern');
CALL ce_update_if_v1(6, 'Berta Bach');
CALL ce_delete_if_v1(6);

**Clean-up application**

Clean-up of the application code cannot be shown here. There are several ways to ensure that the data is being accessed solely using the provided interfaces:

- You can revoke the privileges for a certain user to directly access the data and grant these privileges only to the stored procedures or
- You can create some triggers on these tables which are logging some informations about the user, time, etc. or
- You can parse the general query log for suspicious queries or
- You can rename the underlying tables and the queries will fail and the client will get an error message.

**Change table**

Now the table can be changed:

```sql
CREATE TABLE company
(
  id            INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  company_name  VARCHAR(128) NOT NULL
);

CREATE TABLE employee
(
  id            INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  first_name    VARCHAR(128) NOT NULL,
  last_name     VARCHAR(128) NOT NULL,
  company_id    INT NOT NULL
);

INSERT INTO company
SELECT id, company
FROM company_employee;

# Not perfect but for demo it is enough.
INSERT INTO employee
SELECT NULL, SUBSTRING(employee, 1, locate(' ', employee)-1),
      SUBSTRING(employee, locate(' ', employee)+1), id
FROM company_employee;
```

And the interfaces have to be adapted too:
RENAME TABLE ce_select_if_v1 TO ce_selct_if_v1_old;

CREATE VIEW ce_select_if_v1 AS
SELECT company.id, company_name AS 'company',
     CONCAT(first_name, ' ', last_name) AS 'employee'
FROM company INNER JOIN employee ON company.id = employee.company_id;

DELIMITER //

DROP PROCEDURE ce_insert_if_v1 //

CREATE PROCEDURE ce_insert_if_v1
(IN  company_name  VARCHAR(128), IN  employee_name VARCHAR(128))
BEGIN
    DECLARE id INT;
    INSERT INTO company
    VALUES (NULL, company_name);
    SET id = LAST_INSERT_ID();
    INSERT INTO employee
    VALUES (NULL, SUBSTRING(employee_name, 1, locate(' ', employee_name)-1),
            SUBSTRING(employee_name, locate(' ', employee_name)+1), id);
END;
//

DROP PROCEDURE ce_update_if_v1 //

CREATE PROCEDURE ce_update_if_v1
(IN  id    INT, IN  employee_name VARCHAR(128))
BEGIN
    UPDATE employee
    SET first_name = SUBSTRING(employee_name, 1, locate(' ', employee_name)-1),
        last_name = SUBSTRING(employee_name, locate(' ', employee_name)+1)
    WHERE company_id = id;
END;
//

DROP PROCEDURE ce_delete_if_v1 //

CREATE PROCEDURE ce_delete_if_v1
(IN  id    INT)
BEGIN
    DELETE
    FROM employee
    WHERE company_id = id
Now the operations should still work using the new interfaces and table structures:

```sql
SELECT *
FROM ce_select_if_v1;
CALL ce_insert_if_v1('Linux', 'Anton Albern');
CALL ce_update_if_v1(5, 'Berta Bach');
CALL ce_delete_if_v1(5);
```

**New interfaces**

Now, new interfaces can be built for future development.

**Discussion**

During the migration there was no cleanup in company table. So the table still contains redundant information which needs to be cleaned-up:

```
mysql> select * from company;
+----+------------------+
<table>
<thead>
<tr>
<th>id</th>
<th>company_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>MySQL</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
+----+------------------+
```

The interfaces do not yet provide full functionality to properly maintain the data. It needs to be defined if record #5 should have been deleted or not:

```
mysql> select * from company;
+----+------------------+
<table>
<thead>
<tr>
<th>id</th>
<th>company_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MySQL</td>
</tr>
<tr>
<td>2</td>
<td>MySQL</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Linux</td>
</tr>
</tbody>
</table>
+----+------------------+
```
**Updateable views**

MySQL 5.0 provides updateable views but with some restrictions. With the shown stored procedure wrappers we can reduce the number of restrictions.

On this VIEW we can update. It is an updateable view:

```
CREATE VIEW ce_select_if_v1 AS
    SELECT id, company, employee
    FROM company_employee
;
```

But on a VIEW with an underlying JOIN you will receive the following errors:

```
INSERT INTO ce_select_if_v1 (id, company, employee)
VALUES (1, 'MySQL', 'Hans Meier')
;
ERROR 1393 (HY000): Can not modify more than one base table through a join view 'test.ce_select_if_v1'

UPDATE ce_select_if_v1
    SET employee = 'Hilde Fischer'
    WHERE id = 3
;
ERROR 1348 (HY000): Column 'employee' is not updatable

DELETE
    FROM ce_select_if_v1
    WHERE id = 1
;
ERROR 1395 (HY000): Can not delete from join view 'test.ce_select_if_v1'
```

This is one of the reasons why stored procedures were used for this example. The solution using Stored Procedures provides much more flexibility for implementing logic about how the interface should behave.