MySQL 5.0

Stored Procedures, VIEWs and Triggers

jan@mysql.com

MySQL UC 2005, Santa Clara

Trees in SQL

The problem of how to handle trees in SQL has been talked about alot. The basic 3 ways are:

- store the full path for each entry
- store the parent for each node
- use nested tree

Nested tree is good for read-many-write-less applications where the tree doesn't check over time too much as a write-operation is heavy-weight most of the time.

Referencing the Parent through the full path

Using the variant of the path involves a lot of string handling and is always slow. See below:

```
# use a full path to each node
CREATE TABLE tree_path (
  node_path VARCHAR(1024) PRIMARY KEY,
  name VARCHAR(32) NOT NULL,
  INDEX (name)
) ENGINE = innodb;
INSERT INTO tree_path VALUES
  ( '/0',     'Earth' ),
  ( '/0/0',   'Europe' ),
  ( '/0/0/1', 'Germany' ),
  ( '/1',     'Moon' ),
  ( '/0/1',   'Asia' );

# search for parent of 'Asia'
SELECT t1.name
  FROM tree_path AS t1
 WHERE t1.node_path = (
   SELECT LEFT(t2.node_path,
            LENGTH(t2.node_path)
            - LENGTH(SUBSTRING_INDEX(t2.node_path, '/', -1))
            - 1
          )
     FROM tree_path AS t2
    WHERE t2.name = 'Asia');
+-------+
| name  |
+-------+
| Earth |
+-------+
# get all entries below Earth
SELECT t2.name
  FROM tree_path AS t1
```

```
    JOIN tree_path as t2
 WHERE t1.node_path = LEFT(t2.node_path, LENGTH(t1.node_path))
   AND t1.node_path < t2.node_path
   AND t1.name = 'Earth';
+---------+
| name    |
+---------+
| Europe  |
| Germany |
| Asia    |
+---------+
```

Referencing the Parent by ID

The self referencing variant which is using can't handle all jobs in the DB.

```
# use a full path to each node
CREATE TABLE tree_id (
  node_id INT UNSIGNED NOT NULL PRIMARY KEY,
  name VARCHAR(32) NOT NULL,
  parent_id INT UNSIGNED,
  FOREIGN KEY (parent_id) REFERENCES tree_id (node_id),
  INDEX (name)
) ENGINE = innodb;
INSERT INTO tree_id VALUES
  ( 1, 'Earth',   NULL ),
  ( 2, 'Europe',  1  ),
  ( 3, 'Germany', 2 ),
  ( 4, 'Moon',    NULL ),
  ( 5, 'Asia',    1 );

# search for parent of 'Asia'
SELECT t1.name
  FROM tree_id AS t1
 WHERE t1.node_id = (
   SELECT t2.parent_id
     FROM tree_id AS t2
    WHERE t2.name = 'Asia');
+-------+
| name  |
+-------+
| Earth |
+-------+
1 row in set (0.00 sec)
# get all entries below Earth
## we can't do this in one generic query
```

With the help of Store Procedures we can make it possible as SP provide recursion what is necessary here.

- fetch the first level of childs
- fetch the first level of childs for each childs
- ... and so on until we don't have any new childs

Check this:

```
DELIMTER $$
DROP PROCEDURE get_childs_sub$$
CREATE PROCEDURE get_childs_sub (IN nid INT)
BEGIN
  DECLARE n INT;
  DECLARE done INT DEFAULT 0;
  DECLARE cur CURSOR FOR SELECT node_id FROM tree_id WHERE parent_id = nid;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
  /* if we have not children the FETCH will fail with NOT FOUND */
  OPEN cur;
get_childs_fetch_loop: LOOP
    FETCH cur INTO n;

    IF done = 1 THEN
      LEAVE get_childs_fetch_loop;
    END IF;
    INSERT INTO __childs VALUES ( n );

    CALL get_childs_sub(n);

  END LOOP get_childs_fetch_loop;
  CLOSE cur;
END$$
```

```
### DROP TABLE would result in a problem here.
###
###
DROP PROCEDURE get_childs$$
CREATE PROCEDURE get_childs (IN nid INT)
BEGIN
  DECLARE n INT;

  DROP TEMPORARY TABLE IF EXISTS __childs;
  CREATE TEMPORARY TABLE __childs (
    node_id INT NOT NULL PRIMARY KEY
  );
  /* check if there really are sub-nodes */
  SELECT COUNT(*)
    FROM tree_id
   WHERE parent_id = nid INTO n;

  IF n <> 0 THEN
    /* fetch the node_ids of the childs
      into the __childs table */
    CALL get_childs_sub(nid);
  END IF;

  SELECT t1.*
    FROM tree_id AS t1
      JOIN __childs USING(node_id);

END$$
CALL  get_childs(1)$$
+---------+---------+-----------+
| node_id | name    | parent_id |
+---------+---------+-----------+
|       2 | Europe  |         1 |
|       3 | Germany |         2 |
|       5 | Asia    |         1 |
+---------+---------+-----------+
DELIMITER ;
```

Triggers

The triggers in MySQL 5.0.3 are quite limited, but we can already make some use out of them. The problem we want to handle is optimized search for domains, sub-domains and so on.

In our example want to concentrate on 'a user might have multiple email-addresses'.

```
CREATE TABLE user (
  user_id INT NOT NULL auto_increment PRIMARY KEY,
  name VARCHAR(32) NOT NULL
) engine = innodb;
CREATE TABLE email (
  email_id INT NOT NULL auto_increment PRIMARY KEY,
  user_id INT NOT NULL,
  address VARCHAR(128) NOT NULL,
  UNIQUE (address),
  FOREIGN KEY (user_id) REFERENCES user (user_id)
) engine = innodb;
INSERT INTO user VALUES
  ( NULL, 'Joe Dohn' ),
  ( NULL, 'Karl Heinze' ),
  ( NULL, 'Jet Wong' );

INSERT INTO email (user_id, address) VALUES
  ( 1, 'joe@yahoo.com' ),
  ( 1, 'joe@hotmail.com' ),
  ( 2, 'karl@gmx.de' ),
  ( 2, 'karl@bt.co.uk' ),
  ( 3, 'jet@hotmail.cn' );
```

As a basic operation we want to check which user has a account at in .uk domain.

```
SELECT u.name
  FROM user AS u
 WHERE u.user_id IN (
   SELECT user_id
     FROM email
    WHERE address LIKE '%.uk');
```

This is slow if you have a long list of email addresses. LIKE can only be optimized if the percent-sign is at the end.

REVERSE() can handle this for us at INSERT and UPDATE time. But we want to keep this transparent for the user.

```
ALTER TABLE email
  ADD reverse_address VARCHAR(128) NOT NULL;
UPDATE email SET reverse_address = REVERSE(address);
ALTER TABLE email
  ADD UNIQUE(reverse_address);
```

Whenever the user INSERTs something into the table the reverse_address field should be set directly without user-intervention. We need pre-INSERT trigger which replaces the default value by a REVERSE() of the address field.

```
DELIMITER $$
CREATE TRIGGER tbi_email_reverse_address
 BEFORE INSERT ON email FOR EACH ROW
BEGIN
  IF NEW.address IS NOT NULL THEN
    SET NEW.reverse_address = REVERSE(NEW.address);
  END IF;
END$$
DELIMITER ;
```

Let's insert a new email address for Karl and see if it works.

```
SELECT *
  FROM email;
+----------+---------+-----------------+-----------------+
| email_id | user_id | address         | reverse_address |
+----------+---------+-----------------+-----------------+
|        1 |       1 | joe@yahoo.com   | moc.oohay@eoj   |
|        2 |       1 | joe@hotmail.com | moc.liamtoh@eoj |
|        3 |       2 | karl@gmx.de     | ed.xmg@lrak     |
|        4 |       2 | karl@bt.co.uk   | ku.oc.tb@lrak   |
|        5 |       3 | jet@hotmail.cn  | nc.liamtoh@tej  |
+----------+---------+-----------------+-----------------+
INSERT INTO email (user_id, address) VALUES
  (2, 'karl@web.de' );

SELECT *
  FROM email;
+----------+---------+-----------------+-----------------+
| email_id | user_id | address         | reverse_address |
+----------+---------+-----------------+-----------------+
|        1 |       1 | joe@yahoo.com   | moc.oohay@eoj   |
|        2 |       1 | joe@hotmail.com | moc.liamtoh@eoj |
|        3 |       2 | karl@gmx.de     | ed.xmg@lrak     |
|        4 |       2 | karl@bt.co.uk   | ku.oc.tb@lrak   |
|        5 |       3 | jet@hotmail.cn  | nc.liamtoh@tej  |
|        6 |       2 | karl@web.de     | ed.bew@lrak     |
+----------+---------+-----------------+-----------------+
```

It works. The same is neccessary for the update of the address field. The trigger does exactly the same.

```
DELIMITER $$
CREATE TRIGGER tbu_email_reverse_address
 BEFORE UPDATE ON email FOR EACH ROW
BEGIN
  IF NEW.address IS NOT NULL THEN
    SET NEW.reverse_address = REVERSE(NEW.address);
  END IF;
END$$
DELIMITER ;
```

Does it work ?

```
UPDATE email
   SET address = 'karl@hotmail.com'
 WHERE email_id = 6;
SELECT *
  FROM email;
+----------+---------+-----------------+-----------------+
| email_id | user_id | address         | reverse_address |
+----------+---------+-----------------+-----------------+
```

```
|        1 |       1 | joe@yahoo.com    | moc.oohay@eoj    |
|        2 |       1 | joe@hotmail.com  | moc.liamtoh@eoj  |
|        3 |       2 | karl@gmx.de      | ed.xmg@lrak      |
|        4 |       2 | karl@bt.co.uk    | ku.oc.tb@lrak    |
|        5 |       3 | jet@hotmail.cn   | nc.liamtoh@tej   |
|        6 |       2 | karl@hotmail.com | moc.liamtoh@lrak |
+----------+---------+------------------+------------------+
```

It does. Now we have this automagically generated field. The user hasn't been told about it.

```
DELIMITER $$
CREATE PROCEDURE sp_get_email_by_domain ( IN adr VARCHAR(128) )
BEGIN
  SELECT *
    FROM email
   WHERE reverse_address LIKE REVERSE(CONCAT('%',adr));
END$$
DELIMITER ;
CALL sp_get_email_by_domain('.uk');
+----------+---------+---------------+-----------------+
| email_id | user_id | address       | reverse_address |
+----------+---------+---------------+-----------------+
|        4 |       2 | karl@bt.co.uk | ku.oc.tb@lrak   |
+----------+---------+---------------+-----------------+
DELIMITER $$
CREATE PROCEDURE sp_get_user_by_domain ( IN adr VARCHAR(128) )
BEGIN
  SELECT u.*
    FROM user AS u JOIN email USING (user_id)
   WHERE reverse_address LIKE REVERSE(CONCAT('%',adr));
END$$
DELIMITER ;
CALL sp_get_user_by_domain('.uk');
+---------+-------------+
| user_id | name        |
+---------+-------------+
|       2 | Karl Heinze |
+---------+-------------+
```

As the REVERSE() puts the percent-sign in front the query results in a range-request which is compared to the table-scan very fast.

Function and Triggers with DML

The manual still says that this is not possible, but look yourself.

```
DELIMITER $$
DROP FUNCTION get_child_name$$
CREATE FUNCTION get_child_name (id INT)
  RETURNS VARCHAR(128)
BEGIN
  DECLARE a VARCHAR(128);
  SELECT name FROM tree_id WHERE node_id = id INTO a;
  RETURN a;
END$$
SELECT get_child_name(1) AS name$$
+-------+
| name  |
+-------+
| Earth |
+-------+
DELIMITER ;
```

Is the same true for DML statements in triggers ?

```
DELIMITER $$
DROP TABLE IF EXISTS log_updates$$
CREATE TABLE log_updates (
  tbl_name VARCHAR(64) NOT NULL,
  row_id BIGINT NOT NULL,
  user VARCHAR(64) NOT NULL,
  mtime DATETIME NOT NULL
) engine = innodb$$
DROP TRIGGER user.tau_log_updates$$
CREATE TRIGGER tau_log_updates
  AFTER UPDATE ON user FOR EACH ROW
```

```
BEGIN
  DECLARE a VARCHAR(128);

  INSERT INTO log_updates
    ( tbl_name, row_id, user, mtime) VALUES
    ( "user", OLD.user_id, CURRENT_USER, CURRENT_TIMESTAMP);
END$$
DELIMITER ;
SET @@autocommit = 0;
BEGIN;
## explicit locking for the trigger
## this is currently necessary
LOCK TABLE log_updates WRITE, user WRITE, mysql.proc READ;
UPDATE user SET name = 'Karl Heise' WHERE user_id = 2;
SHOW WARNINGS;
SELECT * FROM log_updates;
# +----------+--------+----------------+---------------------+
# | tbl_name | row_id | user           | mtime               |
# +----------+--------+----------------+---------------------+
# | user     |      2 | root@localhost | 2005-04-02 14:51:32 |
# +----------+--------+----------------+---------------------+
ROLLBACK;
UNLOCK TABLES;
SELECT * FROM log_updates;
# empty set
```

All the tables are transactional here and as you see the trigger works and is rollback as expected. Fine.

LOCKing in SP in MySQL 5.0.3

While writing the examples above I ran into some locking issues which got into 5.0.3 which were not present in the previous releases.

Temporary tables don't work in SPs

```
DELIMITER $$
DROP PROCEDURE IF EXISTS test1$$
CREATE PROCEDURE test1()
BEGIN
  DROP TABLE IF EXISTS __test1;
  CREATE TEMPORARY TABLE __test1 (
    a INT
  );

  SELECT * FROM __test1;
END$$
CALL test1()$$
ERROR 1146 (42S02): Table 'pasta.__test1' doesn't exist
DELIMITER ;
```

Bug 9563

READ lock aquired even if write is needed by sub-sequent tables

The necessary locks are check before the procedure is called. This doesn't take in account the a sub-sequent call might need a higher lock.

```
DELIMITER $$
DROP TABLE IF EXISTS __test2$$
CREATE TABLE __test2 (
  a INT
)$$
DROP PROCEDURE IF EXISTS test2_sub$$
CREATE PROCEDURE test2_sub()
BEGIN
  INSERT INTO __test2 VALUES ( 1 );
END$$
DROP PROCEDURE IF EXISTS test2$$
CREATE PROCEDURE test2()
```

```
BEGIN
  CALL test2_sub();
  SELECT * FROM __test2;
END$$
CALL test2()$$
ERROR 1099 (HY000): Table '__test2' was locked with a READ lock and can't be updated
DELIMITER ;
```

Bug 9565

Illegal state if external LOCKing is used

```
DELIMITER $$
DROP TABLE IF EXISTS __test3$$
CREATE TABLE __test3 (
  a INT
)$$
DROP PROCEDURE IF EXISTS test3$$
CREATE PROCEDURE test3()
BEGIN
  SELECT * FROM __test3;
END$$
LOCK TABLE __test3 WRITE$$
CALL test3()$$
ERROR 1100 (HY000): Table 'proc' was not locked with LOCK TABLES
UNLOCK TABLES$$
DROP PROCEDURE test3$$
ERROR 1305 (42000): PROCEDURE pasta.test3 does not exist
DELIMITER ;
```

The first ERROR is ok as you have to lock all necessary tables if you want to lock by hand. Including 'proc'.

Bug 9566

Triggers and Functions can handle tables without problems as you see above. But currently you have to handle the locking itself in some cases.

```
DELIMITER $$
DROP TABLE IF EXISTS test4$$
CREATE TABLE test4 (
  id INT NOT NULL,
  mtime DATETIME NOT NULL
) engine = innodb$$
DROP TABLE IF EXISTS test5$$
CREATE TABLE test5 (
  id INT NOT NULL
) engine = innodb$$
DROP TRIGGER test5.tau_log_update$$
CREATE TRIGGER tau_log_update
  AFTER UPDATE ON test5 FOR EACH ROW
BEGIN
  INSERT INTO test4
    ( id, mtime ) VALUES
    ( OLD.id, CURRENT_TIMESTAMP);
END$$
DELIMITER ;
SET @@autocommit = 0;
BEGIN;
## explicit locking for the trigger
## this is currently necessary
LOCK TABLE test4 WRITE, test5 WRITE, mysql.proc READ;
INSERT INTO test5 VALUES ( 2 ), ( 3 );
UPDATE test5 SET id = 1 WHERE id = 2;
SHOW WARNINGS;
SELECT * FROM test4;
# +----+---------------------+
# | id | mtime               |
# +----+---------------------+
# |  2 | 2005-04-02 15:03:03 |
# +----+---------------------+
COMMIT;
UNLOCK TABLES;
SET @@autocommit = 0;
BEGIN;
## explicit locking for the trigger
## this is currently necessary
LOCK TABLE test4 WRITE, test5 WRITE, mysql.proc READ;
INSERT INTO test5 VALUES ( 2 ), ( 3 );
```

```
UPDATE test5 SET id = 1 WHERE id = 2;
SHOW WARNINGS;
(root@localhost) [pasta]> SHOW WARNINGS;
# +-------+------+-----------------------------------------------+
# | Level | Code | Message                                       |
# +-------+------+-----------------------------------------------+
# | Error | 1100 | Table 'test4' was not locked with LOCK TABLES |
# +-------+------+-----------------------------------------------+
SELECT * FROM test4;
# +----+---------------------+
# | id | mtime               |
# +----+---------------------+
# |  2 | 2005-04-02 15:03:03 |
# +----+---------------------+
COMMIT;
UNLOCK TABLES;
```

And somehow the LOCKing is still not working as expected. In the second round the locking of the tables seems to be ignored.

Bug 9581

Another query which is taking the wrong LOCKs to execute the query. But this time it was not possible to circumvent it by using external locking.

```
DROP TABLE IF EXISTS test6a;
DROP TABLE IF EXISTS test6b;
CREATE TABLE test6a ( id int );
CREATE TABLE test6b ( id int );
DROP PROCEDURE IF EXISTS test6;
DELIMITER $$
CREATE PROCEDURE test6 ( IN i INT )
BEGIN
  UPDATE test6a JOIN test6b USING (id)
     SET test6a.id = 2
   WHERE test6b.id = i;
END$$
DELIMITER ;
CALL test6();
# ERROR 1099 (HY000): Table 'test6a' was locked with a READ lock and can't be updated
```