MySQL 5.1

Past, Present and Future

jan@mysql.com

MySQL UC 2006

Santa Clara, CA

Abstract

Last year at the same place MySQL presented the 5.0 release introducing Stored Procedures, Views and Triggers to the MySQL world. A lot has happened since then. The community came up with a standard set of stored procedures, I have finished the promised SQL-Tree functions and MySQL 5.1 has become a beta release featuring temporal events, partitioning and nifty, small features like log-to-table or the processlist in the information schema.

The Past

At the MySQL Users Conference in 2005 I gave a tutorial about the new features of MySQL 5.0 covering the 3 big new features: Stored Procedures, Views and Triggers. I wanted to show how those features can simplify your life as a DBA or Developer and how they can improve the performance of the applications you are developing.

To demonstrate the power of the stored procedures I implemented a parent-id based tree in SQL. Each node in the tree is made up of a child-parent relation and we use stored procedures to walk the tree. Instead of using a recursive function as used in classic application programming we used the more SQL like (and several times faster) JOIN implementation which is iterative, one iteration for each level.

After the User Conference I finished the implementation and ported it to dynamic SQL which moved the dependency on the table names out of the stored procedures. Instead of having static SPs which are written for a specific table, dynamic SQL is forming the SQL statements at run-time. This is making the tree-class more generic as you can easily use if for all your tree-tables as long as they have a `id`-field to mark the current node a `parent_id` field to "point" to the parent-node.

Dynamic SQL

Without Dynamic SQL you have to hard-code the SQL-statements including the table-names into the Stored Procedures.

```
CREATE PROCEDURE sp_tree_get_parent(IN node_id INT)
BEGIN
  SELECT id FROM tree WHERE parent_id = node_id;
END$$
```

If you have a second tree table, you would have to create the same set of tables, just to replace the table name.

In MySQL 5.0.13 the bugs which were blocking the use of Prepared statements in Stored Procedures have been removed and now you can create your statements at runtime.

```
CREATE PROCEDURE sp_tree_get_parent(IN tbl_name VARCHAR(64),
  IN node_id INT)
BEGIN
  SET @stmt := CONCAT('SELECT id FROM ', tbl_name,
      ' WHERE parent_id = ?');
  PREPARE stmt_sp_tree_get_parent FROM @stmt;
  EXECUTE stmt_sp_tree_get_parent USING node_id;
  DEALLOCATE stmt_sp_tree_get_parent;
END$$
```

The table name is passed into the SP as string and the `SELECT` statement is built when the procedure is called.

As long as the tables are having the same fields to connect the node-id with node-id of the parent you can now create a set of SPs to work with trees which are independent of the underlying tables.

On http://jan.kneschke.de/projects/mysql/sp/ you can find the set of SPs to walk, search and modify a tree using dynamic SQL and stored procedures.

A standard Stored Procedure library

Some weeks after MySQL 5.0 became stable a set of generic stored procedures as published in the forums of mysql.com.

http://savannah.nongnu.org/projects/mysql-sr-lib/

The present

With MySQL 5.1 two major, new features have been introduced:

- Partitioning
- Temporal Triggers aka Events

Both are a nice addition to the toolbox of a advanced DBA. While Partitioning will help you to handle large datasets efficiently, temporal triggers add a simple way to run cron-jobs in the database.

Events

Part if the normal job of DBA performing regular checks if the database is database is still performing as expected and to check if the database is not running out of space and if, to cleanup.

Before MySQL 5.1.6 you had to use a cronjob Unix or task-scheduler on Windows. With the help of perl you could get the list of tables, check their size, do calculations and trigger an action to report to the admin, move data to a seperate storage, ...

Sometimes it is better to do such jobs directly in the database, especially when you don't have to leave the database to perform the job. In MySQL 5.1.6 temporal triggers have been introduced.

Temporal triggers behave like triggers, they are stored procedures which are executed when a time-condition is triggered. In the Stored Procedures you have access to the INFORMATION_SCHEMA to find out what is the current state of the database and you have access to the CHECK, ALTER, CREATE statements to change the database.

Events are created with the CREATE EVENT statement:

```
CREATE EVENT ev_cleanup_logs
  ON SCHEDULE /**/
  DO /* SP body */
```

The statement allows you to create re-occuring events or one-shot events.

For one-shot events you can specify when the event should be executed:

```
  ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 2 HOUR
```

while you will use

```
  ON SCHEDULE EVERY 1 DAY STARTS '01:00:00'
```

to say that you want to execute a stored procedure every day at 1AM in the morning.

To demonstrate events we use a history table rotator as example. It assumes that you have a history table this log changes in the database:

- who changed the row
- when was it changed
- which schema and table

Such table are useful for auditing or for tracking the changes on database objects. In boths cases the table is only used for SELECT, INSERT and sometimes DELETE. These tables grow very fast and saving space is a goog thing. We want to move all the data from the last day into a new table, a ARCHIVE table. It is compressed, allows SELECT and INSERT and can be moved easily onto a tape or another backup medium if necessary.

What has to be done:

- find the oversized tables (larger than 128 Mb)
- create a new ARCHIVE table taking the current-date is part of the name
- moving the data to the new table
- delete it from the old table

To test the implementation we use a stored procedure which can be tracked easier than a event that you have to wait to be triggered. The event will just call our procedure when we are done with the implementation and testing.

Implementation

We use Dynamic SQL alot in our code as you have to create a table-name on the fly based on the original table and the current date. A small procedure that is doing the PREPARE, EXECUTE, DEALLOCATE PREPARE is appropiate:

```
CREATE PROCEDURE sp_dynsql_execute(IN stmt TEXT)
BEGIN
  SET @stmt = stmt;
  PREPARE stmt_sp_dynsql_exec FROM @stmt;
  EXECUTE stmt_sp_dynsql_exec;
  DEALLOCATE PREPARE stmt_sp_dynsql_exec;
END$$
```

As we described before we want to move the content of a table from a history table into a archived table. For each day we create a new table, appending the date to the table name.

In case the table exists, reuse it. And to make sure that we only delete from the original table what copied to the archive already we open a transaction and use SELECT ... FOR UPDATE to lock the rows we copied.

```
CREATE PROCEDURE sp_cleanup_table (IN tn VARCHAR(64))
BEGIN
  DECLARE table_exists INT DEFAULT 0;
  DECLARE CONTINUE HANDLER FOR 1050 SET table_exists = 1;

  /* build the table name for the archive table */
  SET @date = DATE_FORMAT(CURRENT_DATE, "%Y%m%d");
  SET @atn = CONCAT(tn, "_", @date);

  CALL sp_dynsql_execute(CONCAT("CREATE TABLE ", @atn, " LIKE ", tn));
  IF table_exists = 0 THEN
    CALL sp_dynsql_execute(CONCAT(
      "ALTER TABLE ", @atn, " ENGINE = ARCHIVE"));
  END IF;

  /* move the data */
  START TRANSACTION;
  CALL sp_dynsql_execute(CONCAT(
    "INSERT INTO ", @atn, " SELECT * FROM ", tn, " FOR UPDATE"));
  CALL sp_dynsql_execute(CONCAT("DELETE FROM ", tn));
  COMMIT;
END$$
```

Now we have everything to move all tables which are larger than 128MByte into a archive. The for_each_table() function from the Standard Routine lib can interate over all tables in a database and execute SQL on it.

We use a filter on data_length and table_name to limit the cleaned up tables.

```
CREATE PROCEDURE sp_cleanup_logs (IN max_size INT)
BEGIN
  SET @cond = CONCAT("data_length > ", max_size,
      " AND NOT (table_name RLIKE  '_')");
  CALL for_each_table(database(), @cond, "CALL sp_cleanup_table('$T')");
END$$
```

Time for testing:

```
CREATE TABLE logs ( log TEXT ) ENGINE = innodb;
INSERT INTO logs VALUES ( "create logs" );
SHOW TABLES;
+------------------+
| Tables_in_world |
+------------------+
| logs             |
+------------------+
SELECT * FROM logs;
+-------------+
| log         |
+-------------+
| create logs |
+-------------+
CALL sp_cleanup_logs(1 * 1024);
SELECT * FROM logs;
Empty set (0.00 sec)
SHOW TABLES;
+------------------+
| Tables_in_world |
+------------------+
| logs             |
| logs_20060421    |
+------------------+
SELECT * FROM logs_20060421;
+-------------+
| log         |
+-------------+
| create logs |
+-------------+
```

Creating the Event

We have everything in place. We just have to create the event and watch how it works:

```
CREATE EVENT ev_cleanup_logs
  ON SCHEDULE EVERY 1 MINUTE
  ON COMPLETION PRESERVE
  DO CALL sp_cleanup_logs(1024);
```

Is it really there ?

```
root@localhost [world]> show events\G
*************************** 1. row ***************************
            Db: world
          Name: ev_cleanup_logs
       Definer: root@localhost
          Type: RECURRING
    Execute at: NULL
Interval value: 1
Interval field: MINUTE
        Starts: 0000-00-00 00:00:00
          Ends: 0000-00-00 00:00:00
        Status: ENABLED
```

We see two things:

1. each event is bound to a Database
2. each event has definer

Another definer can create the same event in this database. The event-name has to be unique for the combination of definer and database. That means you can create the same even on multiple databases and another user can create a event with the same name on this database.

By default the event-scheduler is disabled, none of the events will be executed as long as you don't enable the scheduler.

```
SET GLOBAL event_scheduler = ON;
```

If you try this with 5.1.7 you will run into a bug which is blocking you from using EVENTs with Dynamic SQL. See below for more.

Maintaining Events

ALTER EVENT and DROP EVENT are used to change or remove a event. ALTER TABLE allows you for example to disable a event for a while without removing it:

```
ALTER EVENT ev_cleanup_logs DISABLE;
```

Bugs

Every year I try out new things for UC and stumble over bugs which show that most features are working, just the last little piece to the full joy is missing.

This year it is related to triggers and dynamic SQL.

```
DROP PROCEDURE IF EXISTS sp_bug$$
CREATE PROCEDURE sp_bug ()
BEGIN
  SET @q = 'SET @a = 1';
  PREPARE s FROM @q;
  EXECUTE s;
  DEALLOCATE PREPARE s;
END$$

CREATE EVENT bug
  ON SCHEDULE AT CURRENT_TIMESTAMP
  DO CALL sp_bug()$$
```

If you are interested in the progress of this bug, check http://bugs.mysql.com/19264

Partitioning

Partitioning is the other major feature in 5.1.x. It is grown out of the MySQL Cluster Storage Engine into a general server feature. The partitioning feature is giving you, the DBA, a control over the physical layout of the data in a table. The physical layout has not influence to the usage of the table.

What we did above was partitioning, completly handmade and in no way transparent for the end user as we ended up in multiple tables. The Partitioning features is providing the same result but without a change for the way the user works with the table.

Why to split a table ?

The most prominant features you get by splitting up your table into sub-tables are

- Speed
- Easier Management

The speed comes from the idea that you can reduce the data to search as you can decide from the start if a partition can be part of the scan or not.

Easier management means that you can delete a large part of the data (a partition) by just droping a partition instead of deleting the data.

How to split

To split the table into multiple sub-tables (partitions) you need a rule. MySQL supports 4 different ways to split table which all have there usecases:

- HASH and KEY for automatic distribution of the data over partitions
- LIST and RANGE for strict control of the distribution

Examples

The most simple way to create partitions is using the KEY method:

```
CREATE TABLE part_key (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(32),
  added DATETIME NOT NULL
) ENGINE = innodb
  PARTITION BY KEY()
  PARTITIONS 2;
```

The data is seperated on the primary key. It is hashed with either a internal hash-function or MD5 [cluster]. The result of the hash is taken modulo the number of partitions and stored in the partition.

As the data is hash with a function which is turning any a byte-stream into a integer any datatype can be used for a KEY method.

With the HASH method you get more control over the why data is distributed:

```
CREATE TABLE part_hash (
  id INT NOT NULL,
  name VARCHAR(32),
  added DATETIME NOT NULL,
  INDEX (id)
) ENGINE = innodb
  PARTITION BY HASH( WEEKDAY(added) )
  PARTITIONS 7;
```

Is a useful partitioning rule if you often have to search for the data from yesterday, or if you have can remove data after a week.

With this example we have hit the first limitation:

- If the table has a primary key, the PK has to be part of the partitioning expression
- This includes UNIQUE keys too.
- As soon as you don't hash on the PK, partitioned tables can't use UNIQUE keys

```
CREATE TABLE part_list (
  id INT NOT NULL,
  name VARCHAR(32),
  added DATETIME
) ENGINE = innodb
  PARTITION BY LIST( WEEKDAY(added) ) (
    PARTITION workdays VALUES IN (1, 2, 3, 4, 5),
    PARTITION weekend VALUES IN (0, 6)
);
```

Instead of distributing the data automaticly through MODULO over all partitions, you define the partitions yourself and which value is stored in which partition. If you try to insert into the table and the partitioning expression results in a value that is not defined in the list of excepted values, the INSERT fails.

The error-msg you get in MySQL 5.1.7 is mis-leading, as we have a partition for the value 0 and we can insert into the table if the partitioning expression results in 0. But we don't have a VALUE for NULL.

```
insert into part_list values (1, 'first', NULL);
ERROR 1504 (HY000): Table has no partition for value 0
select weekday(now() + INTERVAL 3 DAY);
0
insert into part_list values (1, 'first', NOW() + INTERVAL 3 DAY);
```

The last function is RANGE which extends LIST for RANGEs:

```
CREATE TABLE part_range (
  id INT NOT NULL,
  name VARCHAR(32),
  added DATETIME
) ENGINE = innodb
  PARTITION BY RANGE( YEAR(added) ) (
    PARTITION y2004 VALUES LESS THAN (2005),
    PARTITION y2005 VALUES LESS THAN (2006),
    PARTITION y2006 VALUES LESS THAN MAXVALUE
);
```

Range expressions are very useful when you plan to drop old data in one chunk.

```
ALTER TABLE part_range DROP PARTITION y2004;
```

instead of deleting everything below year 2005 is several times faster for larger datasets.

Partition Pruning

The second important point for partitions was a better optimization for non-indexed data. If you know by the partitioning expression that a value can only be found in one or more partitions you save to read the other partitions. This reduces the scan time alot.

To see how pruning works the EXPLAIN command was extended:

```
EXPLAIN PARTITIONS SELECT * FROM part_range WHERE added = now()\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: part_range
   partitions: y2006
         type: ALL
possible_keys: NULL
...
         rows: 2
        Extra: Using where
```

Horizontal Partitioning

Horizontal Partitioning describes the way how to split a table by column instead of by row. While MySQL 5.1 doesn't provide any special support for it knowing its benifits it useful.

Let's take a example:

```
CREATE TABLE article (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(32) NOT NULL,
  comment TEXT
  UNIQUE (name)
) ENGINE = InnoDB;
```

Whenever you select from user-table the whole row is fetched from the disk. If the average comment is quite long alot of data is read from disk and kept in the caches.

The idea is to split the comments into a seperate table and JOIN it when neccesary.

```
CREATE TABLE article (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(32) NOT NULL,
  UNIQUE (name)
) ENGINE = InnoDB;

CREATE TABLE article_comments (
  id INT NOT NULL PRIMARY KEY,
  comment TEXT NOT NULL,
  FOREIGN KEY (id) REFERENCES article (id)
) ENGINE = InnoDB;

CREATE VIEW article_all AS
  SELECT a.id, a.name, c.comment
    FROM article AS a JOIN
         article_comments AS c
      ON a.id = c.id;

SELECT * FROM article_all;
```

Using Updatable Views

While you can select from the VIEW with ease, you can't always use modifying statements on VIEWs. While a most simple JOIN VIEWs are updatable, they are sadly never insertable or deletable.

Let's try the 3 statements:

```
INSERT INTO article_all (name, comment) VALUES (
  'car', 'a huge car' );
ERROR 1393 (HY000): Can not modify more than one base table through a join view
'test.article_all'
INSERT INTO article (name) VALUES ('car');
INSERT INTO article_comments (id, comment) VALUES (
  LAST_INSERT_ID(), 'big car');
UPDATE article_all SET comment='small car' WHERE id = 1;
DELETE FROM article_all WHERE id = 1;
ERROR 1395 (HY000): Can not delete from join view 'test.article_all'
```

The DELETE case is partly solvable by using ON DELETE CASCADE.

```
ALTER TABLE article_comments
  DROP FOREIGN KEY article_comments_ibfk_1,
  ADD FOREIGN KEY (id) REFERENCES article (id) ON DELETE CASCADE;
DELETE FROM article WHERE id = 1;
```

To summarize:

Updatable VIEWs allows you to simulate horizontal partitioning to a wide extend. Most of the handwork is still left. A full implementation would:

- SELECT would read only the required the partitions required for the result
- UPDATE would update both tables (not possible with VIEWs)
- INSERT would generate a one ID for both tables and INSERT into both
- DELETE would delete from both partitions itself

The more I think about it, perhaps we can solve this with Dynamic SQL and Stored Procedures ... But we leave this for next year.