Transaction time validity in MySQL

Peter Brawley

Relational databases are terrific, but they implement amnesia. Unless we take steps to preserve old values, a simple update to a column value in a row consigns the previously recorded value to oblivion. Sometimes that's OK. But what if users need access to those old values?

Questions about overwritten and deleted values come in two basic varieties:

1. What is the history of the *correct facts*, so far as we know them, as they affect a particular row in a particular column?
2. What is the history of *recorded values* of a particular column in a particular row?

Together, the two questions define what Ralph Kimball calls *point-in-time architecture*. Obviously many complex queries can be developed from them. The state of affairs recorded at time T did so over what period of time? Over what periods were particular database representations incorrect? Complexities multiply quickly.

Question 1 concerns the problem of *history*—all the information about a system, current and historical, that we now believe to be true. For some applications—especially accounting apps—history is integral to current data. An interest-rate view must show not only current rates but previous rates. Richard Snodgrass calls this *temporal validity*. It requires that we track the starting effective date and the ending effective date for every table value in the database. It occupies the biggest part of a big, advanced chapter in ***Get It Done With MySQL 5&Up*** by myself and Arthur Fuller.

Question 2, the history of recorded values, is our subject here. What did we think this customer's name was last June 30th? What did we think her account balance was a month earlier? Snodgrass calls this *transaction time validity*—the guarantee that the state of a table can be reported exactly as it existed at any time in its history.

It's the problem of the *audit trail*: how to track change datetimes and authorship for all data in all rows. There are two basic solutions:

1. Maintain a *log* or *logging table* that tracks every database modification,
2. Turn every table that needs an audit trail into a *transaction state table* by adding logging columns to it.

If you have a correct tracking log, you can derive a transaction state table, and vice-versa, so you can guarantee transaction time validity with either approach. Logging preserves the current interface of the database, and that may be crucial if you are retrofitting transaction time validity to a running app. A transaction state table, though, may reduce disk space usage and will likely simplify querying of past table states.

Tracking logs

In MySQL, the most basic logging method for any table, or set of tables, is to enable binary logging. Given a valid binary log, the state of any table (or combination of them) at any previous instant can be rebuilt correctly with the *mysqlbinlog* utility.

But if that's the only audit trail we have, then every time we run an audit report, we have to regenerate at least part of the database. Not practical. An audit trail has to be directly available via straightforward queries. That requires a tracking log table.

The simplest possible tracking log table has

1. Every column that exists in the tracked table,

2. An action column to record the data-changing action (update, insertion or deletion),
3. A time-of-change column with the desired datetime granularity, and if desired
4. A who-changed-this column.

What is the desired datetime tracking granularity? Whatever is guaranteed to distinguish all possible updates. In most systems including MySQL, two identical modification commands to the same table cannot be executed in the same second, so `change_time TIMESTAMP` is reasonable.

After creating the tracking table, we write triggers to save dated tracking copies of all row modifications, and we prohibit deletions in the tracking table.

This approach delivers two large benefits—it's simple, and all tracking logic remains under the hood, invisible to an application using the tracked table. The downside is redundancy: every change is written twice, once to the table and once to its tracking twin. Three times if you count the binary log.

Consider this simplified table for tracking employees at a movie studio:

Script 1: Create an employees table

```
CREATE TABLE emp (
  id INT PRIMARY KEY AUTO_INCREMENT,
  lastname CHAR(32) NOT NULL,
  firstname CHAR(32) NOT NULL,
  gender CHAR(1),
  dob datetime,
  marital CHAR(1),
  SSN CHAR(9)
);
```

Script 2: Populate emp

```
INSERT INTO emp VALUES
(1, 'Black', 'Mary', 'F', '1972-10-31', 'M', '135792468'),
(2, 'Higgins', 'Henry', 'M', '1955-2-28','W','246813579'),
(3, 'Turunen', 'Raija', 'F', '1949-5-15','M','357902468'),
(4, 'Garner', 'Sam', 'M', '1964-8-15','M','468013579');
```

To create the tracking log table:

1. Create it as a clone of the tracked table,
2. Add an `action CHAR(1)` column,
3. Add a `change_time TIMESTAMP` column,
4. Add a `changed_by` column to track the change author,
5. Make the primary key the current one plus the `change_time` column,
6. Write three required triggers.

We assume `changed_by` will contain a `host@user` value from the `mysql.user` table:

Script 3: Create a tracking log table for emp

```
CREATE TABLE emplog LIKE emp;
ALTER TABLE emplog ADD COLUMN action CHAR(1) DEFAULT '';
ALTER TABLE emplog ADD COLUMN change_time TIMESTAMP DEFAULT NOW();
ALTER TABLE emplog ADD COLUMN changed_by VARCHAR(77) NOT NULL;;
ALTER TABLE emplog MODIFY COLUMN id INT DEFAULT 0;
ALTER TABLE emplog DROP PRIMARY KEY;
ALTER TABLE emplog ADD PRIMARY KEY (id, change_time );

-- Insertion tracking trigger for emp table:
CREATE TRIGGER emplog_insert AFTER INSERT ON emp
FOR EACH ROW
```

```
INSERT INTO emplog VALUES (
  NEW.id,NEW.lastname,NEW.firstname,NEW.gender,
  NEW.dob,NEW.marital,NEW.SSN,'I',NULL,USER());

-- Update tracking trigger for the emp table:
CREATE TRIGGER emplog_update AFTER UPDATE ON emp
FOR EACH ROW
INSERT INTO emplog VALUES (
 NEW.id,NEW.lastname,NEW.firstname,NEW.gender,
  NEW.dob,NEW.marital,NEW.SSN,'U',NULL,USER());

-- Deletion tracking trigger for the emp table:
CREATE TRIGGER emplog_delete AFTER DELETE ON emp
FOR EACH ROW
INSERT INTO emplog VALUES (
  OLD.id,OLD.lastname,OLD.firstname,OLD.gender,
  OLD.dob,OLD.marital,old.SSN,'D',NULL,USER());
```

Given the tracking log's primary key, the update and deletion triggers will fail if two identical modification commands are attempted on the same row in the same second. That side-effect is probably desirable. If you need a finer granularity than one second, then if your MySQL server is version 5.6.4 or later you can add up to 6 digits of microsecond precision by defining the `change_time` column as TIMESTAMP(n) with 1 <= n <= 6; if you are using an earlier MySQL release, you can either use a Milliseconds() function to your server, or define transaction times as integers and populate them from a frontend application which can report time in fractions of seconds.

For this to be complete, the tracking table and tracking triggers must be created before any rows are inserted into the tracked table. To set up tracking for an existing, populated table:

1. Save the table definition and data with *mysqldump*.
2. Write a tracking table and trigger creation script modelled on Script 3.
3. Drop the table.
4. Run the DDL portion of the dump script.
5. Run the logging table and trigger creation script.
6. Run the data portion of the dump script.

Of course such retrofitting makes no audit trails for any moment before the retrofit.

For the `emp` table, just write ...

```
TRUNCATE emp;
```

... then run Script 2. To test this here, we cheat by backdating the log a month and add a subsequent, trackable change—we learned today that actress Mary Black got divorced:

```
UPDATE emplog SET change_time = change_time – interval 30 day;
UPDATE emp SET marital='D' WHERE id=1;
```

How to reconstruct a dated table state from a tracking log table

Given this setup, a query to reconstruct any previous state of a tracked table from its tracking log is dead simple:

1. Find rows having no deletion action up to the target time, and for each of these,
2. Find the row with the latest non-deletion action.

This query varies only on one parameter, *as-of date*, so it is a natural for a MySQL stored procedure:

Script 4: Snapshot of emp as of a specified historical instant

```
CREATE PROCEDURE emp_asof( IN pdt TIMESTAMP )
BEGIN
  SELECT id, lastname, firstname, gender, dob, marital, ssn
  FROM emplog As log1
  WHERE log1.action <> 'D'
  AND log1.change_time = (
    SELECT MAX( log2.change_time )
    FROM emplog AS log2
    WHERE log1.id = log2.id AND log2.change_time < pdt
  );
END;
```

You'd prefer a parameterised View? Me too. Are you desperate enough to accept a kluge? Here's one. Assuming you have a `sys` database where you keep generic data tables and stored routines, execute this:

```
USE sys;
CREATE TABLE viewparams (
  id int(11) PRIMARY KEY AUTO_INCREMENT,
  db varchar(64) DEFAULT NULL,
  viewname varchar(64) DEFAULT NULL,
  paramname varchar(64) DEFAULT NULL,
  paramvalue varchar(128) DEFAULT NULL
);
```

Then if your `emp` table is in `test` ...

```
USE test;
DELETE FROM sys.viewparams WHERE db='test' AND viewname='vEmpHist';
INSERT INTO sys.viewparams (db, viewname, paramname, paramvalue)
VALUES ('test', 'vEmpHist', 'date', '2008-9-1' );

DROP VIEW IF EXISTS vEmpHist;
CREATE VIEW vEmpHist AS
SELECT id, lastname,firstname,gender,dob,marital,ssn
FROM emplog As log1
WHERE log1.action <> 'D'
AND log1.change_time = (
  SELECT MAX( log2.change_time )
  FROM emplog AS log2
  WHERE log1.id = log2.id
    AND log2.change_time < (
      SELECT paramvalue FROM sys.viewparams
      WHERE db='test' AND viewname='vEmpHist' AND paramname='date'
    )
);
```

Or, till MySQL has parameterised Views, we can use sprocs like `emp_asof()` to retrieve any previous table state, for example what the employees table looked like on 1 September 2008:

```
CALL emp_asof('2008-9-1');
+----+----------+-----------+--------+---------------------+---------+-----------+
| id | lastname | firstname | gender | dob                 | marital | ssn       |
+----+----------+-----------+--------+---------------------+---------+-----------+
|  1 | Black    | Mary      | F      | 1972-10-31 00:00:00 | M  <--- | 135792468 |
|  2 | Higgins  | Henry     | M      | 1955-02-28 00:00:00 | W       | 246813579 |
|  3 | Turunen  | Raija     | F      | 1949-05-15 00:00:00 | M       | 357902468 |
|  4 | Garner   | Sam       | M      | 1964-08-15 00:00:00 | M       | 468013579 |
+----+----------+-----------+--------+---------------------+---------+-----------+
```

Transaction state tables

A transaction state table supports accurate *snapshots* of past states of a table, and accurate queries of any past *period*, for example "Over what period of time did our records show Joe Splunk's account more than $5,000 in arrears?" It has columns to track change datetime and who-changed-this. All rows come into existence with an `INSERT` or `UPDATE`, so either they have not been modified subsequently, or their transaction states end with an `UPDATE` or `DELETE`.

If we have a transaction tracking log as set out above, we can build a *transaction state View* from it to emulate a transaction state table:

```
CREATE VIEW emp_transaction_history AS
SELECT
  log1.id, log1.lastname, log1.firstname, log1.gender, log1.dob, log1.marital,
  log1.ssn, log1.change_time AS start_time, log2.change_time AS end_time,
  log1.changed_by AS changed_by1, log2.changed_by as changed_by2
FROM emplog AS log1
JOIN emplog AS log2
WHERE log1.change_time < log2.change_time
  AND log1.action <> 'D'
  AND NOT EXISTS (
    SELECT * FROM emplog AS log3
    WHERE log3.id = log1.id
      AND log1.change_time < log3.change_time
      AND log3.change_time < log2.change_time
  )
UNION
SELECT
  log1.id, log1.lastname, log1.firstname, log1.gender, log1.dob, log1.marital,
  log1.ssn, log1.change_time AS start_time, NOW() AS end_time,
  log1.changed_by AS changed_by1, NULL
FROM emplog AS log1
WHERE log1.action <> 'D'
AND NOT EXISTS (
  SELECT * FROM emplog AS log3
  WHERE log1.id = log3.id
    AND log1.change_time < log3.change_time  );
```

Then we can find our record of Mary's divorce without mentioning her in the query:

```
SELECT DISTINCT
  h1.id, h1.lastname, h1.firstname,
  h1.marital AS Mbefore, h1.start_time, h2.marital AS MAfter, h2.start_time
FROM emp_transaction_history AS h1
JOIN emp_transaction_history AS h2 USING (id)
WHERE h1.marital <> h2.marital;
+----+----------+-----------+---------+---------------------+--------+---------------------+
| id | lastname | firstname | MBefore | start_time          | MAfter | start_time          |
+----+----------+-----------+---------+---------------------+--------+---------------------+
|  1 | Black    | Mary      | M       | 2008-08-11 14:55:47 | D      | 2008-09-10 14:56:32 |
+----+----------+-----------+---------+---------------------+--------+---------------------+
```

In this way, we seem to still have our cake even though we ate it—we've added a form of transaction time validity without radical database and application re-engineering. The cost of this trick is the redundancy of the View—the byzantine `UNION` query in the `emp_transaction_history` View will bog down transaction query performance as the table grows.

A less redundant solution comes at the cost of the re-engineering we've so far avoided: *replace the original table with a transaction state version of it, and implement the original table as a View on the transaction state table.*

For this we need a convention for representing that a particular column value is now current, i.e., that it has not been changed. There are two such conventions: set the end time column to NULL, or set it to the largest possible value for the datetime type we are using (for datetimes 9999-12-31, for timestamps 2037-12-31). I find these queries easier to write and understand using the latest-possible-date convention.

We begin by reverting to the original version of the emp table, before Mary told us about her divorce:

```
TRUNCATE emp;
INSERT INTO emp VALUES
(1, 'Black', 'Mary', 'F', '1972-10-31', 'M', '135792468'),
(2, 'Higgins', 'Henry', 'M', '1955-2-28','W','246813579'),
(3, 'Turunen', 'Raija', 'F', '1949-5-15','M','357902468'),
(4, 'Garner', 'Sam', 'M', '1964-8-15','M','468013579');
```

In a transaction state table, each row defines the period when its values are valid, so it must have all columns of the table whose transactions it tracks, plus columns to mark the beginning and end of the row's valid transaction period, plus columns to identify who executed the start time and end time transactions. A row's transaction period begins when the row is inserted in the tracked table, and continues 'forever' or until updated or deleted. Our convention is that 'forever' is until 9999-12-31 if the column is DATETIME, or until 2037-12-31 if it is a TIMESTAMP.

Future transactions are logically impossible, and rewriting history would destroy it, so a transaction state table admits current modifications only. The *sequenced primary key* of a table which accepts only current modifications consists of the tracked table's primary key plus the period end date. We use the identifier suffix _ts to indicate that a table or column tracks transaction state. We need two columns to track change authors—one for the author of a start, and one for the author of an end when there is one. To prevent MySQL from auto-updating our transaction timestamps, we declare them NULL:

Script 5: Create and populate a transaction state table for the* emp *table

```
CREATE TABLE emp_ts LIKE emp;
ALTER TABLE emp_ts ADD COLUMN start_ts TIMESTAMP NULL;
ALTER TABLE emp_ts ADD COLUMN end_ts TIMESTAMP NULL;
ALTER TABLE emp_ts ADD COLUMN started_by VARCHAR(77);
ALTER TABLE emp_ts ADD COLUMN ended_by VARCHAR(77);
ALTER TABLE emp_ts MODIFY COLUMN id INT DEFAULT 0;
ALTER TABLE emp_ts DROP PRIMARY KEY;
ALTER TABLE emp_ts ADD PRIMARY KEY (id, end_ts );
INSERT INTO emp_ts
  SELECT
    id,lastname,firstname,gender,dob,marital,ssn,
    '2008-8-15','2037-12-31',USER(),NULL
  FROM emp;
```

As with transaction logs, the simplest way to automate maintenance of a transaction state table without affecting existing programs is with Triggers. For the emp_ts table: The INSERT trigger adds a emp_ts copy of the new emp row with start_ts=NOW(), end_ts='2037-12-31', started_by=USER() and ended_by=NULL. The DELETE trigger sets emp_ts.end_ts=NOW() and ended_by=USER() for rows matching the deleted emp row. The UPDATE trigger requires two commands, one to set emp_ts.end_ts=NOW() and ended_by=USER() in emp_ts rows matching the updated emp rows, and a second to add a emp_ts copy of the updated state row with start_ts=NOW(), end_ts='2037-12-31'. started_by=USER() and ended_by=NULL.

MySQL does not yet support multiple triggers with the same action time and event for one table, so we have to delete emp triggers written earlier in this article for the tracking log.

Script 6: Emp triggers for maintaining emp_ts

```
DROP TRIGGER IF EXISTS emplog_insert;
CREATE TRIGGER emp_ts_insert AFTER INSERT ON emp  FOR EACH ROW
 INSERT INTO emp_ts VALUES (
    NEW.id,NEW.lastname,NEW.firstname,NEW.gender,NEW.dob,
     NEW.marital,NEW.ssn,NOW(),'2037-12-31',USER(),NULL  );
```

```
DROP TRIGGER IF EXISTS emplog_update;
CREATE TRIGGER emp_ts_update AFTER UPDATE ON emp  FOR EACH ROW
BEGIN
  UPDATE emp_ts
  SET end_ts=NOW(), ended_by=USER()
  WHERE emp_ts.id=OLD.id AND emp_ts.end_ts='2037-12-31';
  INSERT INTO emp_ts VALUES (
    NEW.id,NEW.lastname,NEW.firstname,NEW.gender,NEW.dob,
    NEW.marital,NEW.ssn,NOW(),'2037-12-31',USER(),NULL   );
END;
DROP TRIGGER IF EXISTS emplog_delete;
CREATE TRIGGER emp_ts_delete AFTER DELETE ON emp FOR EACH ROW
  UPDATE emp_ts
  SET end_ts=NOW(), ended_by=USER()
  WHERE emp_ts.id = OLD.id AND emp_ts.end_ts = '2037-12-31';
```

Test this by recording Mary's divorce:

```
UPDATE emp SET marital='D' WHERE id=1;
SELECT
  lastname as last,firstname as first,
  marital as m,start_ts,started_by,end_ts,ended_by
FROM emp_ts;
```

last	first	m	start_ts	started_by	end_ts	ended_by
Black	Mary	M	2008-08-15 00:00:00	admin01@localhost	2008-09-11 01:39:20	admin01@localhost
Black	Mary	D	2008-09-11 01:39:20	admin01@localhost	2037-12-31 00:00:00	NULL
Higgins	Henry	W	2008-08-15 00:00:00	admin01@localhost	2037-12-31 00:00:00	NULL
Turunen	Raija	M	2008-08-15 00:00:00	admin01@localhost	2037-12-31 00:00:00	NULL
Garner	Sam	M	2008-08-15 00:00:00	admin01@localhost	2037-12-31 00:00:00	NULL

It works, but it's redundant. We are again writing every data point to two tables. How to lose the redundancy? Lose the emp table, emulate it with a View, and implement transaction state logic in modification code for emp_ts:

Script 7: Replace the emp table with a view on emp_ts

```
DROP TABLE emp;
CREATE VIEW emp AS
  SELECT id,lastname,firstname,gender,dob,marital,ssn
  FROM emp_ts
WHERE end_ts = '2037-12-31';
SELECT * FROM emp;
```

id	lastname	firstname	gender	dob	marital	ssn
1	Black	Mary	F	1972-10-31 00:00:00	D	135792468
2	Higgins	Henry	M	1955-02-28 00:00:00	W	246813579
3	Turunen	Raija	F	1949-05-15 00:00:00	M	357902468
4	Garner	Sam	M	1964-08-15 00:00:00	M	468013579

Now the emp table is implemented as an updatable View. If we were to define the following trigger for emp_ts INSERTs, application code for insertions into emp could be left unchanged:

```
CREATE TRIGGER emp_ts_insert BEFORE INSERT ON emp_ts FOR EACH ROW
SET NEW.start_ts=NOW(),NEW.end_ts='2037-12-31',NEW.started_by=USER(),NEW.ended_by=NULL;
```

Here we hit a roadblock. A transaction-valid update resolves to an UPDATE plus an INSERT, and a transaction-valid deletion resolves to an UPDATE. To implement this logic on emp_ts without changing the previous emp app interface would require INSTEAD OF Triggers, which MySQL does not support. The application interface will have to change to allow use of stored procedures instead of CRUD commands. The insertion and update stored procedures need IN parameters for every emp column. They do not need parameters for the transaction state columns:

Script 8: Stored procedures for transaction-valid inserts, updates, deletes on emp_ts

```
CREATE PROCEDURE emp_insert(
  pid INT, plastname CHAR(32), pfirstname CHAR(32), pgender CHAR(1),
  pdob DATE, pmarital CHAR(1), pssn CHAR(9) )
BEGIN
  INSERT INTO emp_ts
  VALUES (pid, plastname, pfirstname, pgender, pdob, pmarital, pssn, NOW(), USER(), '2037-12-
31', NULL); END;

CREATE PROCEDURE emp_update(
  pid INT, plastname CHAR(32), pfirstname CHAR(32), pgender CHAR(1),
  pdob DATE, pmarital CHAR(1), pssn CHAR(9)
)
BEGIN
  UPDATE emp_ts
  SET end_ts = NOW(), ended_by=USER()
  WHERE emp_ts.id=pid AND emp_ts.end_ts='2037-12-31';
  INSERT INTO emp_ts VALUES
    (pid, plastname, pfirstname, pgender, pdob, pmarital, pssn, NOW(), USER(), '2037-12-31',
NULL);
END;

CREATE PROCEDURE emp_delete( pid INT )
BEGIN
  UPDATE emp_ts
  SET end_ts=NOW(), ended_by=USER()
  WHERE emp_ts.id=pid AND end_ts='2037-12-31';
END;
```

Transaction state queries

A snapshot query on a transaction state table is trivially easy:

```
CREATE PROCEDURE emp_snapshot( pwhen DATETIME )
SELECT id,lastname,firstname,gender,dob,marital,ssn
FROM emp_ts
WHERE start_ts <= pwhen AND end_ts > pwhen;
CALL emp_snapshot( '2008-9-1' );
+----+----------+-----------+--------+---------------------+---------+-----------+
| id | lastname | firstname | gender | dob                 | marital | ssn       |
+----+----------+-----------+--------+---------------------+---------+-----------+
|  1 | Black    | Mary      | F      | 1972-10-31 00:00:00 | M       | 135792468 |
|  2 | Higgins  | Henry     | M      | 1955-02-28 00:00:00 | W       | 246813579 |
|  3 | Turunen  | Raija     | F      | 1949-05-15 00:00:00 | M       | 357902468 |
|  4 | Garner   | Sam       | M      | 1964-08-15 00:00:00 | M       | 468013579 |
+----+----------+-----------+--------+---------------------+---------+-----------+
```

Finding periods when some specific state of affairs was recorded can also be straightforward, for example find the periods when any person is recorded in `emp_ts` as married:

```
SELECT
  a.lastname, a.firstname, a.marital,
  IF( a.start_ts > b.start_ts, a.start_ts, b.start_ts) AS FirstRecorded,
  IF( a.end_ts < b.end_ts, a.end_ts, b.end_ts ) AS LastRecorded
FROM emp_ts AS a
JOIN emp_ts AS b ON a.id=b.id AND a.marital='M' AND b.marital='M'
HAVING FirstRecorded < LastRecorded;
```

```
+----------+-----------+---------+---------------------+---------------------+
| lastname | firstname | marital | FirstRecorded       | LastRecorded        |
+----------+-----------+---------+---------------------+---------------------+
| Black    | Mary      | M       | 2008-08-15 00:00:00 | 2008-09-11 01:39:20 |
| Turunen  | Raija     | M       | 2008-08-15 00:00:00 | 2037-12-31 00:00:00 |
| Garner   | Sam       | M       | 2008-08-15 00:00:00 | 2037-12-31 00:00:00 |
+----------+-----------+---------+---------------------+---------------------+
```

Summary

MySQL throws up small roadblocks to ideal audit trails—notably by not supporting parameterised Views and INSTEAD OF Triggers—but robust implementation of audit trails is doable, whether you want to go the tracking log route, or whether you opt for full-blown transaction state tables.

[Last updated 26 June 2019]