Time and MySQL

Who knows where the time goes? [1]

A basic feature of the relational database is row uniqueness, guaranteed by a table's primary key. Without it, we can neither tell one row from another, nor relate tables.

What happens to row uniqueness, and to relational properties that depend on it, when a set of related tables must model variation over time? On the face of it, you might expect this, at worst, to add a few innocent complexities—a datetime column or two per table, perhaps, and an extra line or two in query clauses. Then ask yourself how to write a constraint that permits multiple rows to have identical non-datetime values at different instants, and to have overlapping valid periods, *but never the same non-temporal data at the same instant*. Good luck working that out on the back of an envelope.

This chapter is about how to model such time variation in a MySQL database. We begin as if the reader were a client, with the bad news. When it comes to modelling time in a database, complexities multiply quickly. Because SQL has no universal quantifier, nested negatives abound. There aren't many brilliant shortcuts to be had. Mostly, working out these constraints amounts to step-by-step slogging.

Now, the (partially) good news: Ralph Kimball[2] and Richard Snodgrass[3] worked out the basic concepts. ISO SQL 2011 developed convenient syntax for building point-in-time logic directly into relational tables; MySQL does not implement this, but since version 10.3.4, MariaDB *does*. While this is so, if you need to build time-related validity into a database, MariaDB will be considerably easier to work with than MySQL

Basic Concepts

SQL time validity is *temporal relational database architecture* or *point-in-time architecture* (get used to it, there’s not a fixed set of singular labels in this domain, at least not yet). If you have

tried to implement point-tin-time architecture, you know how difficult it is. In MySQL before 8 and MariaDB before 10, unavailability of CHECK CONSTRAINTs, made it hard. Even now, limits on Triggers and on CHECK CONSTRAINT are serious obstacles.

To develop temporal database architecture we analyse time-sensitive data on three basic temporal dimensions: *domain type*, *value reference*, and *validity type*:

A. **Temporal domain types**: The three kinds are:
   - *instant*, a temporal atom (e.g., a microsecond, a second, a day),
   - *interval*, a duration measured in temporal atoms,
   - *period*, an interval that begins and ends at defined instants.

   An interval or period is a sequence of temporal atoms. It may be:
   - *closed-closed*: includes both start date and end date,
   - *closed-open*: includes the start date, excludes the end date,
   - *open-closed*: excludes the start date and includes the end date,
   - *open-open*: excludes both start date and end date.

   Inclusive/exclusive would be more intuitive nomenclature than *closed/open*, but we follow the standard nomenclature. A common business default is *closed-open*: if you book a hotel room for 22-24 May, the hotel expects you to arrive in the afternoon of 22 May and leave in the morning of 24 May, having stayed the two days of the *closed: closed* period 22-23 May, or the *closed:open* period 22-24 May. A temporal database needs a consistent convention. In our increasingly market-oriented society, we stay with the *closed-open* convention for periods.

B. **Temporal value references**: The three kinds are:
   - *user-defined* or arbitrary, independent of the validity of other columns,
   - *valid time*, marking when a stored fact was so—*historical facts*; SQL:2011 refers to this as *application time*.
   - *transaction time*, marking when a fact was stored—*database history*. SQL:2011 calls this *system versioning*.

C. **Temporal validity**: The three kinds are:
   - *current*: now,
   - *sequenced*: at each instant in the relevant datetime range,
   - *non-sequenced*: time-independent.

So queries on temporal tables …
- may reference and/or return snapshots or period information,
- may reference user-defined time, valid time, and/or transaction time,
- may be current, sequenced or non-sequenced.

If that looks to you like a formidable set of complexities that time adds to the challenge of designing and enforcing entity integrity and referential integrity in an RDBMS, you're beginning to get the idea.

MySQL resources, limitations

TEMPORAL VALIDITY IN MARIADB

As noted, SQL:2011 defines temporal validity in terms of *application time*. MySQL has no such SQL:2011-compliant tools, but MariaDB since v10.4.3 *does*. Given a table with a pair of DATE, DATETIME or TIMESTAMP columns, MariaDB supports defining a PERIOD, for example …

```
CREATE TABLE history (
  event VARCHAR(255),
  dstart DATE,
  dstop DATE,
  PERIOD FOR date_period(dstart,dstop)
  PRIMARY KEY(dstart,dstop)
);
```

If the temporal columns defining the PERIOD were nullable, MariaDB redefines them as NOT NULL. ALTER TABLE may add or remove such PERIODs …

```
CREATE TABLE history (
  event VARCHAR(255),
  dstart DATE,
  dstop DATE,
  PRIMARY KEY(dstart,dstop)
);
ALTER TABLE history ADD PERIOD IF NOT EXISTS FOR dperiod(dstart,dstop);
ALTER TABLE history DROP PERIOD IF EXISTS FOR dperiod(dstart,dstop);
```

A PRIMARY or UNIQUE index can specify WITHOUT OVERLAPS in the period, for example …

```
ALTER TABLE history ADD UNIQUE(event, dperiod WITHOUT OVERLAPS);
```

If period overlaps exist in the data, WITHOUT OVERLAPS index creation fails with a duplicate entry error.

UPDATE, DELETE FOR PORTION

Given data in the table …

```
INSERT INTO history (event, dstart, dstop) VALUES
  ('a', '1999-01-01', '2000-01-01'), ('b', '1999-01-01', '2018-12-12'),
  ('c', '1999-01-01', '2017-01-01'), ('d', '2017-01-01', '2019-01-01');
```

… then in UPDATE and DELETE commands MariaDB accepts a FOR PORTION OF modifier that references and operates on the named PERIOD …

```
DELETE FROM history } UPDATE history
 FOR PORTION OF dperiod FROM '2001-01-01' TO '2018-01-01';
```

UPDATE } DELETE…FOR A PORTION OF… then adds and/or changes and/or deletes rows to reflect effects on the table's *periods* so after the update, the last row becomes three rows …

```
select * from history order by dstart,dstop;
+-------+------------+------------+
| event | dstart     | dstop      |
+-------+------------+------------+
| a     | 1999-01-01 | 2000-01-01 |
| c     | 1999-01-01 | 2017-01-01 |
| b     | 1999-01-01 | 2018-01-01 |so
| d     | 2017-01-01 | 2018-01-01 |
| e     | 2018-01-01 | 2018-12-12 |
| e     | 2018-01-01 | 2019-01-01 |
+-------+------------+------------+
```

The UPDATE cannot modify the two temporal columns defining the period, and cannot reference period values in the SET expression. In DELETE, multi-delete is not supported. With both, the FROM…TO clause must be literal. A table's PERIOD cannot be included in a SELECT list.

TRANSACTION TIME VALIDITY

As noted, SQL:2011 calls transaction time validity *system versioning*. MySQL has no specific methods for implementing it, but MariaDB since v10.3.4 *does*. Create a transaction-time-valid table in MariaDB with …

```
CREATE TABLE txhistory(
   x INT,
   tstart TIMESTAMP(6) GENERATED ALWAYS AS ROW START,
   tstop TIMESTAMP(6) GENERATED ALWAYS AS ROW END,
   PERIOD FOR SYSTEM_TIME(tstart, tstop)
) WITH SYSTEM VERSIONING;
```

… or with the radically simpler syntax …

```
CREATE TABLE txhistory (
   x INT
) WITH SYSTEM VERSIONING;
```

… in which case the system versioning columns are invisible but can be retrieved with …

```
SELECT x, row_start, row_end from txhistory;
```

System versioning can be ADDed implicitly or explicitly, and can be DROPped …

```
ALTER TABLE txhistory
  ADD COLUMN tstart TIMESTAMP(6) GENERATED ALWAYS AS ROW START,
  ADD COLUMN tstop TIMESTAMP(6) GENERATED ALWAYS AS ROW END,
  ADD PERIOD FOR SYSTEM_TIME(tsart, tstop),
  ADD SYSTEM VERSIONING;

ALTER TABLE txhistiory ADD | DROP SYSTEM VERSIONING;
```

Querying system-versioning tables

Query a system-versioning table for snapsots, intervals or closed-open periods …

```
SELECT * FROM t FOR SYSTEM_TIME AS OF TIMESTAMP'2016-10-09 08:07:06';
SELECT * FROM t FOR SYSTEM_TIME BETWEEN (NOW() - INTERVAL 1 YEAR) AND NOW();
SELECT * FROM t FOR SYSTEM_TIME FROM '2016-01-01 00:00:00' TO '2017-01-01 00:00:00';
SELECT * FROM t FOR SYSTEM_TIME ALL;
```

Replicating a system-versiooning table

When a table becomes system-versioning, MariaDB adds the *row_end* column to the primary key. To prevent Insert errors on the replica, set the `secure_timestamp` system variable to `YES` on the replica; this forces use of the replica system clock rather than that of the replication source.

Precise transaction history

The instant when a row is inserted, updated or deleted may differ from the instant when the result becomes visible. To ensure that InnoDB reports precise transaction history, declare generated system versioning columns as BIGINT UNSIGNED rather than TIMESTAMP(6) …

```
CREATE TABLE txhistory (
   x INT,
   start_trxid BIGINT UNSIGNED GENERATED ALWAYS AS ROW START,
   end_trxid BIGINT UNSIGNED GENERATED ALWAYS AS ROW END,
   PERIOD FOR SYSTEM_TIME(start_trxid, end_trxid)
) WITH SYSTEM VERSIONING;
```

This allows queries for transaction ids …

```
SELECT * FROM txhistory FOR SYSTEM_TIME AS OF TRANSACTION <idvalue>;
```

Partitioning history from now

If the specification for a transaction-time/system-versioning table prohibits storing future information, MariaDB since v10.3.4 supports partitioning the table into two tables:

- a *current table* holds all information that is valid now; this dispenses with the arbitrariness of end dates like `9999-12-31` because such a table needs no end date column,
- a *history table* holds all information that is no longer current.

Current queries and updates get simpler, other time-sensitive queries and updates grow more complex. Here are examples of partitioning by SYSTEM_TIME from the *MariaDB manual system-versioning section* ...

```
CREATE TABLE t (x INT) WITH SYSTEM VERSIONING
  PARTITION BY SYSTEM_TIME (
    PARTITION p_hist HISTORY,
    PARTITION p_cur CURRENT
  );
```

Automatic partition rotation can be done by SYSTEM_TIME, or with an INTERVAL clause, or with a size LIMIT clause ...

```
CREATE TABLE t (x INT) WITH SYSTEM VERSIONING
  PARTITION BY SYSTEM_TIME INTERVAL 1 WEEK (
    PARTITION p0 HISTORY,
    PARTITION p1 HISTORY,
    PARTITION p2 HISTORY,
    PARTITION pcur CURRENT
  );

CREATE TABLE t (x INT) WITH SYSTEM VERSIONING
  PARTITION BY SYSTEM_TIME LIMIT 100000 (
    PARTITION p0 HISTORY,
    PARTITION p1 HISTORY,
    PARTITION pcur CURRENT
  );
```

Here, first-week data goes into p0, second-week data into p1, all else into p2; `information_schema.partitions` stores partition time boundaries. More complicated models are possible ...

```
CREATE TABLE t (x INT) WITH SYSTEM VERSIONING
  PARTITION BY SYSTEM_TIME
    SUBPARTITION BY KEY (x)
    SUBPARTITIONS 4 (
    PARTITION ph HISTORY,
    PARTITION pc CURRENT
  );
```

BITEMPORAL VALIDITY

MariaDB since v10.4.3 supports *bitemporal tables* implementing both time validity and transaction time validity, i.e., full-blown *point-in-time architecture*:

```
CREATE TABLE date_and_tx_history (
   data_start DATE,
   data_end DATE,
   row_start TIMESTAMP(6) AS ROW START INVISIBLE,
   row_end TIMESTAMP(6) AS ROW END INVISIBLE,
   PERIOD FOR application_time(data_start, data_end),
   PERIOD FOR system_time(row_start, row_end))
WITH SYSTEM VERSIONING;
```

All methods shown under time validity and transaction time validity are available.

CHECK CONSTRAINT

MySQL before version 8.0.16 accepted CHECK CONSTRAINT declarations for INNODB tables— and *ignored them!* MariaDB implemented them in 10.2, MySQL in 8.0.16.. Time-valid tables need them. Some time-valid constraints can be enforced in Triggers, most cannot. For temporal validity you will need MySQL 8.0.16, MariaDB 10.2.1 or application code. Still, as we'll see, MySQL and MariaDB non-support for subqueries in CHECK CONSTRAINT severely limits what can be done: in the present stat of development of both MySQL and MariaDB you'll still need custom application code to manage temporal and transaction time validity.

Deferred constraints

Neither MySQL nor MariaDB implements deferred constraints. Furthermore, constraints must be applied row-wise rather than at `COMMIT` time. This is a problem for many complex constraints, even for some simple ones. For example to delete a MySQL row with a self-referencing foreign key, you must temporarily `SET foreign_key_checks =0`. A transaction fulfilling a complex constraint must leave the database in a consistent state, but nothing in relational database theory requires a database to be in a consistent state after each statement within a transaction.

Triggers

MySQL triggers cannot issue UPDATE statements on the trigger table, and cannot raise errors. These limitations create difficulties for implementing transaction validity.

Test schemas

As a testbed for valid-time analysis, we use a set of four simple tables named `parent`, `child`, `qty` and `state`. For combining *valid-time* and *transaction-time* validity we use *another set of tables*— `county_properties`, `county_owners`, `county_owners_properties`.

In the valid-time schema, the `qty` table stores *period information for measurement values* related to ancestor keys, and the `state` table stores *period information for state values* related to ancestor keys: a `qty` or `state` row references a `child` row on key `cid`, and that `child` row references a `parent` row on key `pid`. If you prefer concrete examples, think of

- `parent.pid` as a group identifier, *e.g.*, the residents of Dorset, Ontario, Canada,
- `child.cid` identifying a group member, *e.g.*, Sean, a biologist who lives there,
- `qty.qty` as a numeric measurement for a given individual, *e.g.*, serum cholesterol level,
- `state.state` as an individual's possession of a certain sort of licence, for example a licence to fish in a particular provincial park.

Remember that these tables are valid-time tables, so *their column values are valid only between the start and end dates in the rows where the values occur*. We adopt the conventions that `end_`

`date='9999-12-31'` means 'current', and that periods are closed-open. Here is the schema DDL for MySQL:

Snippet 21-1: DDL for a simple valid-time schema

```
CREATE TABLE parent(
  pid INT AUTO_INCREMENT,
  start_date DATE,
  end_date DATE,
  PRIMARY KEY (pid, end_date )
  /* In MariaDB since v10.4.3: ,period for DATE_PERIOD(start_date,end_date) */
) ENGINE=INNODB;

CREATE TABLE child (
  cid INT AUTO_INCREMENT,
  pid INT,
  start_date DATE,
  end_date DATE,
  PRIMARY KEY(cid, end_date),
  KEY (pid),
  FOREIGN KEY (pid) REFERENCES parent (pid)
) ENGINE=INNODB;

CREATE TABLE qty (
  qid INT AUTO_INCREMENT,
  cid INT,
  qty DECIMAL(10,2),
  start_date DATE,
  end_date DATE,
  PRIMARY KEY(qid, end_date),
  KEY (cid),
  FOREIGN KEY (cid) REFERENCES child (cid)
) ENGINE=INNODB;

CREATE TABLE state (
  sid INT AUTO_INCREMENT,
  cid INT,
  state CHAR(1),
  start_date DATE,
  end_date DATE,
  PRIMARY KEY(sid, end_date),
  KEY (cid),
  FOREIGN KEY (cid) REFERENCES child (cid)
) ENGINE=INNODB;

-- MARIADB 10.4.3 and later:
ALTER TABLE parent ADD PERIOD FOR DATE_PERIOD(start_date,end_date) ;
ALTER TABLE child  ADD PERIOD FOR DATE_PERIOD(start_date,end_date) ;
ALTER TABLE qty    ADD PERIOD FOR DATE_PERIOD(start_date,end_date) ;
ALTER TABLE state  ADD PERIOD FOR DATE_PERIOD(start_date,end_date) ;

CREATE TABLE state_log LIKE state;
ALTER TABLE state_log ADD COLUMN action CHAR(1) DEFAULT '';
ALTER TABLE state_log ADD COLUMN change_time TIMESTAMP DEFAULT NOW();
ALTER TABLE state_log MODIFY COLUMN sid INT DEFAULT 0;
ALTER TABLE state_log DROP PRIMARY KEY;
ALTER TABLE state_log ADD PRIMARY KEY (sid, end_date, change_time );
```

The motive for including `end_date` in the primary keys of this schema will become clear in the next two sections. The motive for surrogate (`auto_increment`) primary keys is maximum flexibility representing and storing time-varying data. Here is a SQL script for populating the tables:

```
INSERT INTO parent VALUES
(1,'2005-1-1','2005-4-1'),(1,'2005-4-1','2005-7-1'),(1,'2005-7-1','2005-10-1'),
(1,'2005-10-1','2006-1-1'),(1,'2006-1-1','2006-4-1'),(1,'2006-4-1','2006-7-1'),
(1,'2006-7-1','2006-10-1'),(1,'2006-10-1','2007-1-1'),(2,'2005-1-1','2005-4-1'),
(2,'2005-4-1','2005-7-1'),(2,'2005-7-1','2005-10-1'),(2,'2005-10.1','2006-1-1'),
(3,'2006-1-1','2006-4-1'),(3,'2006-4-1','2006-7-1'),(3,'2006-7-1','2006-10-1'),
(3,'2006-10-1','2007-1-1'),(4,'2006-1-1','9999-12-31');
```

```
INSERT INTO child VALUES
(1,1,'2005-1-1', '2005-3-1'),(1,1,'2005-4-1', '2005-6-1'),(1,1,'2005-7-1', '2005-8-1'),
(1,1,'2005-10-1','2005-11-1'),(2,1,'2006-2-1', '2006-3-1'),(2,1,'2006-5-1', '2006-6-1'),
(2,1,'2006-8-1', '2006-9-1'),(2,1,'2006-11-1','2006-12-1'),(3,2,'2005-1-1', '2005-3-1'),
(3,2,'2005-4-1', '2005-6-1'),(3,2,'2005-7-1', '2005-8-1'),(3,2,'2005-10-1','2005-11-1'),
(4,3,'2006-1-1', '2006-3-1'),(4,3,'2006-4-1', '2006-6-1'),(4,3,'2006-7-1', '2006-8-1'),
(4,3,'2006-10-1','2006-11-01'),(5,4,'2006-4-1','9999-12-31');

INSERT INTO qty VALUES
(1, 1, 100, '2005-1-1',  '2005-1-8'), (1, 1, 110, '2005-1-8',  '2005-1-15'),
(2, 1, 165, '2005-1-15', '2005-1-22'), (2, 1, 162, '2005-1-22', '2005-1-29'),
(3, 1, 297, '2005-1-29', '2005-2-5'), (3, 2, 199, '2005-2-5',  '2005-2-12'),
(4, 2, 201, '2005-2-12', '2005-2-19'), (4, 2, 206, '2005-2-19', '2005-2-26'),
(5, 3, 300, '2005-1-1',  '2005-1-8'), (5, 3, 310, '2005-1-8',  '2005-1-15'),
(6, 3, 305, '2005-1-15', '2005-1-22'), (6, 3, 304, '2005-1-22', '2005-1-29'),
(7, 3,  98, '2005-1-29', '2005-2-5'), (7, 3,  94, '2005-2-5',  '2005-2-12'),
(8, 3, 135, '2005-2-12', '2005-2-19'), (8, 3, 137, '2005-2-19', '2005-2-26');

INSERT INTO state VALUES
( 1, 1, 'A', '2005-02-01', '2005-03-16'), ( 2, 1, 'B', '2005-03-16', '2005-04-16'),
( 3, 1, 'C', '2005-04-16', '2005-06-16'), ( 4, 1, 'A', '2005-06-16', '2005-09-01'),
( 5, 1, 'B', '2005-09-01', '2005-11-16'), ( 6, 1, 'C', '2005-11-16', '2005-12-01'),
( 7, 2, 'A', '2006-01-01', '2006-02-01'), ( 8, 2, 'B', '2006-02-01', '2006-02-16'),
( 9, 2, 'C', '2006-02-16', '2006-07-01'), (10, 2, 'A', '2006-07-01', '2006-09-16'),
(11, 2, 'B', '2006-09-16', '2006-11-16'), (12, 2, 'C', '2006-11-16', '2006-12-01'),
(13, 1, 'M', '2005-02-01', '2005-03-16'), (14, 1, 'N', '2005-04-16', '2005-06-16'),
(15, 1, 'M', '2005-06-16', '2005-09-01'), (16, 1, 'P', '2005-11-16', '2005-12-01'),
(17, 2, 'M', '2006-01-01', '2006-01-16'), (18, 2, 'N', '2006-02-01', '2006-02-16'),
(19, 2, 'P', '2006-04-01', '2006-07-01'), (20, 2, 'M', '2006-08-01', '2006-09-16'),
(21, 2, 'N', '2006-10-16', '2006-11-16'), (22, 2, 'P', '2006-12-01', '2006-12-16'),
(23, 2, 'M', '2006-08-01', '2006-08-10'), (24, 2, 'M', '2006-08-20', '2006-09-01'),
(25, 3, 'P', '2006-01-01', '2006-02-01'), (26, 3, 'P', '2006-02-01', '2006-03-01'),
```


TOC ***Previous*** ***Next***