MariaDB Temporal Tables

Federico Razzoli
$ whoami

Hi, I’m Federico Razzoli from Vettabase Ltd

Database consultant, open source supporter, long time MariaDB and MySQL user

- vettabase.com
- Federico-Razzoli.com
Temporal Tables Implementations
Proprietary DBMSs

- Oracle 11g (2007)
- Db2 (2012)
- SQL Server 2016
- Snowflake

In Db2, a temporal table can use system-period or application-period
Open source databases

- PostgreSQL has a temporal_tables extension
  - not available from the main cloud vendors
- CockroachDB
  - with limitations
- CruxDB (NoSQL)
- HBase stores old row versions and you can retrieve them
MariaDB

MariaDB supports both types of Temporal Tables:

- MariaDB 10.3: system_time
- MariaDB 10.4: application_time

Tables are bitemporal
Application Time
Example

CREATE OR REPLACE TABLE ticket (  
id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,  
state ENUM('OPEN', 'VERIFIED', 'FIXED', 'INVALID') NOT NULL  
  DEFAULT 'OPEN',  
summary VARCHAR(200) NOT NULL,  
description TEXT NOT NULL  
)
  ENGINE InnoDB
;

- We want to start to track changes to bugs over time
Making the table Application-Timed

```
ALTER TABLE ticket
    LOCK = SHARED,
    ALGORITHM = COPY,
ADD COLUMN valid_from DATETIME NOT NULL,
ADD COLUMN valid_to DATETIME NOT NULL,
ADD PERIOD FOR time_period (valid_from, valid_to) ;
```

We can use...

- Any temporal data type that includes a date (DATE, DATETIME, TIMESTAMP)
- Any storage engine
Inserting rows

MariaDB [test]> INSERT INTO ticket (summary, description) VALUES
   ->  ('I cannot login', 'Why is this happening to me?');
ERROR 1364 (HY000): Field 'valid_from' doesn't have a default value

MariaDB [test]> INSERT INTO ticket (summary, description, valid_from, valid_to)
VALUES
   -> ('I cannot login', 'Why is this happening to me?',
   -> '1994-01-01', '2010-01-01');
Query OK, 1 row affected (0.003 sec)
A better Application-Timed table

CREATE TABLE ticket_tmp LIKE ticket;
ALTER TABLE ticket_tmp
    ADD COLUMN valid_from DATETIME NOT NULL
        DEFAULT NOW(),
    ADD COLUMN valid_to DATETIME NOT NULL
        DEFAULT '2038-01-19 03:14:07.999999',
    ADD INDEX idx_valid_from (valid_from),
    ADD INDEX idx_valid_to (valid_to),
    ADD PERIOD FOR time_period(valid_from, valid_to);
A better Application-Timed table

ALTER TABLE ticket_tmp
    DROP PRIMARY KEY,
    ADD PRIMARY KEY (id, valid_to);

-- populate the table

RENAME TABLE ticket TO ticket_old, ticket_tmp TO ticket;

● You will need to do similar operations with UNIQUE indexes
● RENAME TABLE is an atomic operation
MariaDB [test]> SELECT id, summary, valid_from, valid_to FROM ticket;
+----+----------------+---------------------+---------------------+
| id | summary         | valid_from          | valid_to            |
+----+----------------+---------------------+---------------------+
|  1 | I cannot login  | 1994-01-01 00:00:00 | 2010-01-01 00:00:00 |
+----+----------------+---------------------+---------------------+
1 row in set (0.001 sec)

MariaDB [test]> SELECT id, summary, valid_from, valid_to FROM ticket
            -> WHERE NOW() BETWEEN valid_from AND valid_to;
Empty set (0.001 sec)
Deleting rows properly

CREATE OR REPLACE PROCEDURE ticket_delete(p_id INT)
MODIFIES SQL DATA
COMMENT 'Makes a row obsolete by changing its timestamp'
BEGIN
  UPDATE ticket
    SET valid_to = NOW()
    WHERE id = p_id AND valid_to > NOW();
END;
Deleting rows properly

MariaDB [test]> SELECT id, valid_from, valid_to FROM ticket WHERE id = 1;

<table>
<thead>
<tr>
<th>id</th>
<th>valid_from</th>
<th>valid_to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2020-08-23 14:32:22</td>
<td>2038-01-19 03:14:07</td>
</tr>
</tbody>
</table>

MariaDB [test]> CALL ticket_delete(1);

MariaDB [test]> SELECT id, valid_from, valid_to FROM ticket WHERE id = 1;

<table>
<thead>
<tr>
<th>id</th>
<th>valid_from</th>
<th>valid_to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2020-08-23 14:32:22</td>
<td>2020-08-23 14:32:34</td>
</tr>
</tbody>
</table>
Deleting/updating periods

MariaDB [test]> SELECT id, valid_from, valid_to FROM ticket;
+----+---------------------+---------------------+
| id | valid_from          | valid_to            |
+----+---------------------+---------------------+
|  1 | 1994-01-01 00:00:00 | 2010-01-01 00:00:00 |
+----+---------------------+---------------------+

MariaDB [test]> DELETE FROM ticket
 -> FOR PORTION OF time_period FROM '1990-01-01' TO '2000-01-01'
 -> WHERE id = 1;

MariaDB [test]> SELECT id, valid_from, valid_to FROM ticket;
+----+---------------------+---------------------+
| id | valid_from          | valid_to            |
+----+---------------------+---------------------+
|  2 | 2000-01-01 00:00:00 | 2010-01-01 00:00:00 |
+----+---------------------+---------------------+
System Versioning
CREATE OR REPLACE TABLE ticket (  
id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,  
state ENUM('OPEN', 'VERIFIED', 'FIXED', 'INVALID') NOT NULL DEFAULT 'OPEN',  
summary VARCHAR(200) NOT NULL,  
description TEXT NOT NULL  
)  
ENGINE InnoDB  
;
Making the table System-Versioned

ALTER TABLE ticket
    LOCK = SHARED,
    ALGORITHM = COPY,
    ADD SYSTEM VERSIONING;
Making the table System-Versioned

ALTER TABLE ticket
  LOCK = SHARED,
  ALGORITHM = COPY,
  ADD COLUMN inserted_at TIMESTAMP(6) GENERATED ALWAYS AS ROW START INVISIBLE,
  ADD COLUMN deleted_at TIMESTAMP(6) GENERATED ALWAYS AS ROW END INVISIBLE,
  ADD PERIOD FOR SYSTEM_TIME(inserted_at, deleted_at)
  ADD SYSTEM VERSIONING;

Limitations:

- Temporal columns don’t have to be INVISIBLE, if they’re often needed
- MDEV-15968: System versioning and CONNECT engine don't work well together: current data is not returned
- MDEV-17448: Support DATETIME(6) for ROW START, ROW END
Querying a Sysver Table

-- get current version of the rows
-- without the temporal columns (they’re INVISIBLE)
SELECT * FROM ticket;

-- get current version of the rows
-- with the temporal columns
SELECT *, inserted_at, deleted_at FROM ticket;

-- all current and old data
SELECT *, inserted_at, deleted_at
    FROM ticket FOR SYSTEM_TIME ALL;
Get old versions of the rows

-- get deleted rows
SELECT *, inserted_at, deleted_at
  FROM ticket FOR SYSTEM_TIME
       FROM '1970-00-00' TO NOW() - 1 MICROSECOND;

SELECT *, inserted_at, deleted_at
  FROM ticket FOR SYSTEM_TIME
       BETWEEN '1970-00-00' AND NOW() - 1 MICROSECOND;

SELECT *, inserted_at, deleted_at
  FROM ticket FOR SYSTEM_TIME ALL
 WHERE deleted_at < NOW();
History of a row

SELECT id, state, inserted_at, deleted_at
    FROM ticket FOR SYSTEM_TIME ALL
    WHERE id = 3
    ORDER BY deleted_at;
Read a row from a specific point in time

```sql
SELECT id, state
FROM ticket
FOR SYSTEM_TIME AS OF TIMESTAMP '2020-08-22 08:52:36'
WHERE id = 3;
```

```sql
SELECT id, state
FROM ticket
FOR SYSTEM_TIME ALL
WHERE id = 3 AND
'2020-08-22 08:52:36' BETWEEN inserted_at AND deleted_at;
```
Temporal JOINs

-- rows that were present on 07/01
-- whose state did not change after one month

SELECT t1.id, t1.inserted_at, t1.deleted_at
FROM ticket       FOR SYSTEM_TIME ALL AS t1
LEFT JOIN ticket  FOR SYSTEM_TIME ALL AS t2
  ON
      t1.id = t2.id
      AND t1.state = t2.state
WHERE
      '2020-07-01 00:00:00' BETWEEN t1.inserted_at AND t1.deleted_at
      AND '2020-08-01 00:00:00' BETWEEN t2.inserted_at AND t2.deleted_at
      AND t2.id IS NULL
ORDER BY t1.id;
Indexes

The ROW END column is automatically appended to:

- The Primary Key;
- All UNIQUE indexes.

Queries can use a whole index of its leftmost part, so once a regular table becomes System Versioned queries performance will not degrade.

For Application Timed tables, indexes remain unchanged.
The power of [bi]Temporal Tables
Hints about things you can do

- A table can be both system-versioned and application-timed (bitemporal)
- Stats on added/deleted rows by year, month, weekday, day, daytime...
- Stats on rows lifetime
- Get rows that never changed
- Get rows that change too often, or change at “strange” times
- Examine history of a row to find problems
- ...

(bitemporal)
Hints about things that you should do

● PK should never change, or tracking rows history will be impossible
  ○ If necessary, use a trigger that throws an error if OLD.id != NEW.id
● Application Time tables: no hard deletions/updates
● If you have to drop a column, move it to a new table to avoid losing the history
● If you have to add a column that is not often read/written, consider putting it into a new table
● If you run stats or complex queries involving temporal columns, add PERSISTENT columns and indexes on them to make queries faster
What we left out

This was a short introductory session, so we left out some features:

- **ALTER TABLEs**
  - They may erase or change parts of the history, so they’re disabled by default

- **Partitioning**
  - You can record the history in a separate partition, or multiple partitions

- **Backups**
  - Check the docs for problems with Sysver Tables and mysqldump

- **Replication / binlog**
  - Check the documentation for possible problems with Sysver Tables
  - MariaDB can be a replica of a MySQL server, and make use of Temporal Tables to let analysts run certain analyses that they couldn’t run on MySQL
Thanks for attending!
Question time :-)