Temporal databases

Storing the history of data

Learning goals

1. Understand the need for modelling temporal changes in a database
2. Recognise the three axes of time
3. Learn to create a temporal database using the tools provided by a RDMS.

Changes in time

- Earlier, you have learnt logical database design, i.e. the design of the database schema.
- In real world, things happen in time, and as a consequence, data changes.
- As data changes, historical data accumulates.
  - By historical data, we mean old (updated or deleted) values of the data items.
  - In some contexts, it may be vital to store the historical data.
- Possible queries to historical data:
  - What earlier values has the data item had?
  - At what time has each change of the data item taken place?
  - What has been the period of validity for each data point in history?
- Most real-life databases contain a temporal dimension at least to some extent.
- Its modelling by tables and fields in the schema may be challenging.
- The database management systems (such as MariaDB) provide tools to temporal modelling.

A database without a temporal dimension

id	firstname	lastname	salary
1	Viivi	Virta	2 000,00
2	Ahmed	Bakir	2 000,00

- In the example, there is a simple Employee table.
- Let's assume that Viivi gets a pay rise to 2200 euros.
- Simple solution: update the value of the salary field.
- The history of Viivi's salary is not saved. For example, the following challenges arise:
  1. What to do if the history of her salary needs to be investigated later?
     - For example, how to retrieve a salary that was valid on 14 March 2023?
  2. What to do when Viivi's salary is to be raised to 2500 euros after two months, as her probation ends?
     - Can the new salary somehow be updated in advance?

Temporal database

- Problems such as those described before are avoided by creating a temporal database
  - temporal = time-related.
- Temporal databases may include three axes of time:
  1. transaction time
  2. valid time
  3. decision time
- Of these, one or more axis can be applied simultaneously. Often, the transaction time and the valid time, or either of them, is used.

Axis 1/3: transaction time

id	firstname	lastname	salary	transaction_time
1	Viivi	Virta	2 000,00	2022-08-12 10:14:41.649413
1	Viivi	Virta	2 300,00	2022-08-12 10:14:44.708788
2	Ahmed	Bakir	2 000,00	2022-08-12 10:14:41.654144

- The transaction time tells
  - The point in time when the record has been stored in the database, or
  - The period of time when the record has been current (i.e. the last record in the table)
- In the example, Viivi's salary has been updated three seconds after the initial insert.
- The table now includes the transaction time axis (of a temporal database)
- Historical values as well as update times will be stored.
- The transaction time is not necessarily the time when the updated data has been valid from an application's point of view.
  - Viivi may have started her job in the beginning of August. Her salary may well be 2300 €/mo. from the beginning of August.
- Is recording the transaction time a valid solution to store the salary history?
  - It may be, if all updates are made between consequtive paydays, and the new value is always valid for the entire month.
  - However, future payrises cannot be stored e.g. two months in advance.

Axis 2/3: valid time

id	firstname	lastname	salary	startdate	enddate
1	Viivi	Virta	2 500,00	2022-11-01	2024-12-31
2	Ahmed	Bakir	2 000,00	2022-08-01	2024-12-31
3	Viivi	Virta	2 000,00	2022-08-01	2022-11-01

- Valid time, or application time, tells when the data is valid from the point of view of data usage.
- From this point of view, it is irrelevant when the data has been stored or updated in the database.

Axis 3/3: decision time

id	firstname	lastname	salary	decision_time
1	Viivi	Virta	2 000,00	2022-07-24
2	Ahmed	Bakir	2 000,00	2022-07-31

- The decision time tells when the decision about the validity of the record has been made.
  - It can differ from both the transaction time and the decision time.
- An example of the three axes of time:
- In the board meeting on 24.7.2022 it was decided that Viivi's salary will be 2000 €/mo. in the time period of 1.8.2022 to 31.12.2022. This fact was entered into the system on 27.7.2022.
  - Decision time: 24.7.2022
  - Valid time: 1.8.2022 to 31.12.2022
  - Transaction time: 27.7.2022
- From the database's point of view, the decision time is sotred in a regular date field. Thus, it is not particularly interesting when technical aspects are considered.

SQL and axes of time

- SQL:2011 standard defines operations for a bitemporal database.
  - A bitemporal database contains both the valid time and the transaction time.
- In MariaDB, support for bitemporal databases has been present since version 10.3.4.
  - system versioning
  - application-time period versioning
- More functionality is included as versions evolve: for example, since version 10.5.3, there has been support for WITHOUT OVERLAPS clause that prevents overlapping intervals in the valid time.
  - A potential use case: reservations to a hotel room. Two reservations for the same room must not overlap.
- Next, let's get acquainted with the use of mobth axes of time in MariaDB.

Turning transaction time on

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT,
    firstname VARCHAR(40),
    lastname VARCHAR(40),
    salary DECIMAL(8,2),
    PRIMARY KEY (id)
) WITH SYSTEM VERSIONING;
```

- The WITH SYSTEM VERSIONING clause in the end turns on the system versioning based on the transaction time.
- There will be hidden fields `row_start` and `row_end` in the table. They are timestamp fields for the transaction time (i.e. the start and end of the period when the record has been current)
  - These fields are invisible by default (SELECT * ... won't display them).

Queries in presence of the transaction time

```
INSERT INTO person VALUES (NULL, "Viivi", "Virta", 2000);
INSERT INTO person VALUES (NULL, "Ahmed", "Bakir", 2000);
```

1. `SELECT * FROM person;`

```
SELECT SLEEP(3);
UPDATE person SET salary=2300 WHERE id=1;
```

2. `SELECT * FROM person;`
3. `SELECT *, row_start AS transaction_time FROM person FOR SYSTEM_TIME ALL;`
4. `SELECT * FROM person FOR SYSTEM_TIME AS OF TIMESTAMP SUBTIME(NOW(), "1.0");`

1.

id	firstname	lastname	salary
1	Viivi	Virta	2 000,00
2	Ahmed	Bakir	2 000,00

2.

id	firstname	lastname	salary
1	Viivi	Virta	2 300,00
2	Ahmed	Bakir	2 000,00

3.

id	firstname	lastname	salary	transaction_time
1	Viivi	Virta	2 000,00	2022-08-12 12:51:32.725143
1	Viivi	Virta	2 300,00	2022-08-12 12:51:35.780547
2	Ahmed	Bakir	2 000,00	2022-08-12 12:51:32.729443

4.

id	firstname	lastname	salary
1	Viivi	Virta	2 000,00
2	Ahmed	Bakir	2 000,00

Queries in presence of the transaction time

- With system versioning (that is, recording the transaction time), the operations for the current record (SELECT, INSERT, UPDATE, DELETE) can be writted as if the transaction time didn't exist.
- A regular SELECT retrieves the current record.
- A regular DELETE marks the record as invalid,. It doesn't delete the record physically from the disk.
  - If needed, all versions can be explicitly deleted with a DELETE HISTORY statement.

Queries based on the transaction time

- It is possible to refer to a historical version be ending a FOR SYSTEM_TIME clause in the end of the query.
- Examples:
  - SELECT ... FOR SYSTEM_TIME ALL retrieves all historical versions separately.
  - SELECT ... FOR SYSTEM TIME SUBTIME(NOW(), ”365 0:0:0”) retrieves a version that is 365 days old.
  - SELECT ... FOR SYSTEM TIME TIMESTAMP("2022-08-24 18:55") retrieves a version that has been current on 24.8.2022 at 18:55.

Turning valid time on

```
CREATE TABLE person (
   id INT NOT NULL AUTO_INCREMENT,
   firstname VARCHAR(40),
   lastname VARCHAR(40),
   salary DECIMAL(8,2),
   startdate DATE,
   enddate DATE,
   PERIOD FOR valid_time(startdate, enddate),
   PRIMARY KEY (id)
);
```

- In the table, specific fields will be introduced for both the start and end dates of the validity period.
  - The permitted data types are DATE, DATETIME, and TIMESTAMP. The types must be same for the two fields.
- PERIOD FOR clause turns on the versioing based on the valid time.
  - In the example, `valid_time` is an identifier with which you can refer to the valid time period later. The naming of the identifier is free.

Queries based on the valid time

```
INSERT INTO person VALUES (NULL, "Viivi", "Virta", 2000, "2022-08-01", "2024-12-31");
INSERT INTO person VALUES (NULL, "Ahmed", "Bakir", 2000, "2022-08-01", "2024-12-31");
```

1. `SELECT * FROM person;`

```
UPDATE person
FOR PORTION OF valid_time FROM "2022-11-01" TO "2024-12-31"
SET salary = 2500
WHERE id=1;
```

2. `SELECT * FROM person;`
3. `SELECT * FROM person WHERE NOW() BETWEEN startdate AND enddate;`
4. `SELECT * FROM person WHERE TIMESTAMP("2022-11-07") BETWEEN startdate AND enddate;`

Note: in example 3., **NOW()** returns a timestamp in Sep 2022.

1.

id	firstname	lastname	salary	startdate	enddate
1	Viivi	Virta	2 000,00	2022-08-01	2024-12-31
2	Ahmed	Bakir	2 000,00	2022-08-01	2024-12-31

2.

id	firstname	lastname	salary	startdate	enddate
1	Viivi	Virta	2 500,00	2022-11-01	2024-12-31
2	Ahmed	Bakir	2 000,00	2022-08-01	2024-12-31
3	Viivi	Virta	2 000,00	2022-08-01	2022-11-01

3.

id	firstname	lastname	salary	startdate	enddate
2	Ahmed	Bakir	2 000,00	2022-08-01	2024-12-31
3	Viivi	Virta	2 000,00	2022-08-01	2022-11-01

4.

id	firstname	lastname	salary	startdate	enddate
1	Viivi	Virta	2 500,00	2022-11-01	2024-12-31
2	Ahmed	Bakir	2 000,00	2022-08-01	2024-12-31

Queries based on the valid time

- SELECT queries are written in a regular fashion. If the records need to be selected on the basis of valid time, the condition is included in the WHERE clause.
- UPDATE and DELETE queries may contain a FOR PORTION clause that changes the data for the given valid given interval only. As a consequence, new versions of the record may emerge:
  - In the previous example, the UPDATE statement created a new period with a higher salary. The old period (with a lower salary) shrank automatically.
  - Note that the UPDATE statement created a new record into the person table that carries a new primary key value. If id needs to be a fixed person number (instead of just a row key without further meaning), it must be made a regular field, not a primary key field.

A bitemporal table

```
DROP TABLE IF EXISTS person;

CREATE TABLE person (
   id INT NOT NULL AUTO_INCREMENT,
   firstname VARCHAR(40),
   lastname VARCHAR(40),
   salary DECIMAL(8,2),
   startdate DATE,
   enddate DATE,
   PERIOD FOR valid_time(startdate, enddate),
   PRIMARY KEY (id)
) WITH SYSTEM VERSIONING;

INSERT INTO person VALUES (NULL, "Viivi", "Virta", 2000, "2022-08-01", "2024-12-31");
INSERT INTO person VALUES (NULL, "Ahmed", "Bakir", 2000, "2022-08-01", "2024-12-31");

SELECT *, row_start, row_end FROM person;
```

- Both axis of time (transaction time and valid time) can be taken into use simultaneously.

id	firstname	lastname	salary	startdate	enddate	row_start	row_end
1	Viivi	Virta	2 000,00	2022-08-01	2024-12-31	2022-08-12 14:05:37.071090	2038-01-19 05:14:07.999999
2	Ahmed	Bakir	2 000,00	2022-08-01	2024-12-31	2022-08-12 14:05:37.077544	2038-01-19 05:14:07.999999

Another option: modelling in the schema

- Before, we discussed the temporal database functionality of a relational database management system (MariaDB).
- Is it possible to achieve the same functionality without the provided functionality?
- Alternatively, the history tables and fields can be modelled in the schema.
  - For each INSERT or UPDATE, the transaction time can be generated with the SQL NOW() function.
- With devotion, the correct schema can be designed. However, the UPDATE and SELECT operations may become cumbersome.

Another option: modelling in the schema

```
CREATE TABLE person (
   id INT NOT NULL AUTO_INCREMENT,
   firstname VARCHAR(40),
   lastname VARCHAR(40),
   PRIMARY KEY (id)
);

CREATE TABLE salary (
   salaryid INT NOT NULL AUTO_INCREMENT,
   personid INT,
   transactiontime TIMESTAMP DEFAULT NOW(),
   salary DECIMAL(8,2),
   startdate DATE,
   enddate DATE,
   PRIMARY KEY (salaryid),
   FOREIGN KEY (personid) REFERENCES person(id)
);
```

Another option: modelling in the schema

- The queries and the update operations become more difficult.
- Let's raise Viivi's salary from a date in the future.
  - This operation becomes quite complicated...
  - ... and the things like this become even more difficult as size of the schema grows.

```
UPDATE salary SET enddate="2022-11-01"
WHERE personid=1
AND enddate>"2022-11-01"
AND startdate<="2024-12-31";
INSERT INTO salary(personid, salary, startdate, enddate)
VALUES(1, 2500, "2022-11-01", "2024-12-31");

SELECT * FROM salary;
```

salaryid	personid	transactiontime	salary	startdate	enddate
1	1	2022-08-12 14:49:02	2 000,00	2022-08-01	2022-11-01
2	1	2022-08-12 14:49:05	2 500,00	2022-11-01	2024-12-31

Modelling in the scema: benefits and drawbacks

- Benefits:
  - The temporal nature of data is made explicitly visible in the design phase.
  - The solution is guaranteed to work in any RDBMS.
- Drawbacks:
  - The schema becomes complicated.
  - The queries and the update operations become extremely complex.
    - This complexity is likely to transfer to the application layer.
    - Consequently, the performance may be hampered.

A lightweight solution: logging

- If the queries to the historical data are rare, and not needed in the daily operation, logging may be a sufficient solution.
- In MariaDB, general query logging can be turned on. As a consequence, each SQL statement is written into a text file.
- Historical data is still retrievable, but it is an exceptional procedure that cannot be expressed with normal SQL queries.

Issues to consider

- The need for temporal data should be analyzed in the modelling phase.
- With temporal databases, there is the risk of over-modelling: is history really needed for all potentially changing data.
    - Example: can a name of a Finnish municipality change? Is it necessary to retrieve the old name in its original shape?
- It usually pays to take advantage of the temporal database functionality in a RDBMS.
  – Specifically, MariaDB supports bitemporal tables.