Lukas Yu 14-720-866

Vertiefung: Integration of Ongoing Time Points into PostgreSQL

Application Scenario

A lot of work has gone into making a *now* timestamp that is dependent on the current time possible, but what are the use cases for such a timestamp? What are the advantages to use such a special variable compared to just sticking to traditional static timestamps?

Let's think of an example: We want to track the time interval of an employee named Jane, who starts working on January 1st. The interval is denoted by two timestamps, the start timestamp and the end timestamp. It describes the interval for which the fact that Jane is an employee is true. The tuple in the database would look like this on the February 2nd:

Employment

Employee	**Start**	**End**
Jane	01.01.2017	02.01.2017

However, keeping the tuples updated every day is not optimal, especially with growing numbers. It also does not tell the user if the employee is still employed or stopped working on the same day, making it more difficult to decide which tuples to update. It would be nice to have the ability to save a variable *now*, which then gets replaced with the current time when the tuple gets queried. The table would look like this:

Employee	**Start**	**End**
Jane	01.01.2017	*now*

The now variable as the end timestamp means that Jane is still employed. If a user now wants to know the employment time of Jane on January 3rd, the *now* variable gets replaced with the current timestamp (01.01.2017 – 03.01.2017), which is the correct interval. If Jane stops working one day, the end timestamp gets replaced with a fixed timestamp which then will not be replaced with the current timestamp when queried.

With the *now* timestamp it is now possible to store timestamps that are dependent on the *current time*, the time it is queried.

Several extensions to this idea can be made:

- Now-relative timestamps. E.g. *now* + 3 days
- Additional to valid time interval (valid in real world), transaction time (valid in database) is added to represent internal database changes
- Reference time. For many applications, it is insufficient just to replace *now* with the current time when the query is executed. With reference time, any time can be queried as *now*.

Implementation approaches

The concept of the *now* timestamp is simple, but the actual implementation into a DBMS is no trivial. Several approaches have been proposed, but all of them have some trade-offs.

MIN, MAX or NULL value

One approach to represent the *now* value is to take the minimal or maximal possible value of the date type. These values usually are not used in a real world application, so its semantics can be replaced with *now*.

The problem of the MIN and MAX approach is that it can't be efficiently indexed. In a B-Tree, all the *now* values are either on the extreme left (MIN value) or on the extreme right (MAX value), lowering the performance (**index range problem)**. Another problem is that all these values are indexed to the same special value, causing the index to sequentially search through each value (**index redundancy problem**).

Another approach is to use null value to represent *now*. It has the advantage that it does not require as much space than a regular timestamp, but a null value cannot be indexed by conventional DBMS at all, leading to unacceptable access times.

A problem for all three approaches is that it gives special values a new meaning, potentially leading to **overloading**. Despite its advantage that it only uses integrated data types and no modification on the DBMS is required, it can be concluded that these three approaches are far from optimal and are should not be used in a real-world application.

Empty range

Stantic *et al.* described a new approach. If the start and end timestamps of an interval is the same (e.g. [01.01.2017 – 01.01.2017]), it means that this tuple is valid from the date until *now*. This type of interval does not have a meaningful use in the real world: It starts on one date and ends before the date has begun. The meaning of the useless tuple gets replaced with a *now* semantic.

An example:

The first row shows how it is physically stored in the database. It has the same semantics as the second row, but does not require a special value for *now*.

Employee	**[Start**	**End)**
Jane	01.01.2017	01.01.2017
Jane	01.01.2017	*now*

The advantage of this is that is does not require a special data type, as it only uses conventional timestamps, thus no modification to the DBMS is needed. The disadvantage an additional layer between the user and database needed to "translate" the notation from the database into a human readable form, which is a problem with all approaches that only use integrated functionalities of the DBMS.

Approach in modification semantics

In this paper from K. Torp *et al.* a possible approach to implement a database with *now-relative* timestamps is described.

With this approach the valid time gets represented with 4 date variables:

- V-Begin

- V-Begin-Offset
- V-End
- V-End-Offset

Offset values different from NULL indicates a max and min function, respectively. An example:

Name	**V-Begin**	**V-Begin-Offset**	**V-End**	**V-End-Offset**
Jane	10.01.2003	NULL	20.01.2003	*Now*

This tuple pretty much means [10.01.2003 – min(20.01.2003, *now*)). It starts on January 10th and ends on *now* if the reference time is until January 20th.

While this approach doesn't extend any data types, this cannot be implemented in Postgres without losing the functionality that comes with the daterange data type.

Generally Valid Queries on Databases with Ongoing Time Points

In previous approaches, a reference time is specified when querying a temporal database, which then is bound to every involved *now* variable.

A query that is frequently executed and is requiring heavy computing, it can be significantly improved with the optimization described in the paper. The concept is that a result is computed that is independent of reference time to get a generally valid result. The result of the query is then saved into another table consisting of the fact and a reference time interval for which the fact is true. Any future requests of the same query just have to check if the reference time falls into the interval, significantly reducing the work load.

In the same paper another approach to represent *now* is used. The date data type of Postgres gets extended:

- Date a
- Date c
- boolean bottomed
- boolean topped

Name	**Date a**	**Date c**	**bottomed**	**topped**
Jane	10.01.2017	20.01.2017	true	true

The tuple above describes a date that evaluates to *now* between the two timestamps and is limited to these dates beyond them. The limitations can be disabled by setting the corresponding Booleans to false.

A valid time interval is described by two of them. Because the new date data type is an extension of the existing integrated Postgres date data type, the daterange data type provided by Postgres can be used to contain two extended date types.

Advantages and challenges of timestamps with and without extensions of data types

The **Stratum approach** does not modify the underlying DBMS. It adds an layer between the user and database to translate the variable queries and modifications into conventional SQL. It does not require any modifications to the actual database. However, optimization might be difficult and the data cannot be understood directly without the translator as the layer is essentially a workaround.

The **integrated approach** modifies and adds the temporal features directly into the database. It is generally more difficult to implement than the stratum approach, but it should yield better performance and does not require an additional layer. The only disadvantages comes from the difficulties when extending integrated data types. Functionalities may break and must be repaired again to ensure a safely working system.

Date data type extension in a Postgres environment

As mentioned before, if the date data type gets extended, functions and predicates of the extended date and daterange data type may break.

The approach from the paper “Generally Valid Queries” is used. When the date data type of Postgres is extended, some operators and functions of date and daterange must be redefined. Especially the “<” operator is essential, as it is used for **ORDER BY**, **indexing**, **merge joins** etc.

The problem with the extension of date is that it’s reference time dependent. Operators as easy as “<” which previously only returned *true* or *false* is now a generally valid boolean. When comparing two extended date types to find out which one is smaller, the system has to compute an answer which can evaluate to true or false depending on the reference time, which is not known yet when generating a generally valid answer from a query. The same procedure must be done for daterange, which is dependent on the date data type.

The logical connectors for those generally valid Booleans also must be defined as they are not simple Booleans anymore.

Fortunately, this work has already been done. Details on the implementation plan are all described in the paper.