Basismodul Report
Integrating *Now* as Variable with Basic Querying Functionality into PostgreSQL

Mohit Narang
mohit.narang@uzh.ch
Legi: 14-706-618

May 18, 2015

Abstract

This implementation is about the integration of *NOW* with the basic querying funcationality into PostgreSQL. The fundamental approach is to consider *NOW* as a dynamic variable instead of a runtime constant as it is used currently in the PostgreSQL implementation.
Following are the major tasks completed as part of this assignment:

- Literature study about the concepts of using *now*. [2]
- Literature study about implementation details regarding representation of *now* physically in the PostgreSQL kernel.[1]
- Literature study about *now* resolving strategies.
- Implementation of the C code changes to the PostgreSQL kernel based on the literature study.

1. Literature Study

1.1 The concept of *Now*

Now is a time variable used to represent temporal aspects of data being recorded in a database. Its value changes as time passes and should always match current time. It is required in databases, for example in an inventory management, where the current inventory of 3 items start being valid since 1 July and until current time. So the same inventory is also valid from 1 to 2 July or 2 to 3 July as long as there is no change in the inventory. This information is very suitable for using *now* as a value instead of changing the expiry date every day. Currently the databases only support concrete timestamps instead of variables in the time columns.

Problems with previous approaches

- *Pessimisting and optimistic assumptions*
  1. Using *Now* in a database to represent the valid-to gives a too pessimistic view about the future.
  2. Sometimes variables like ∞ are used to overcome this, but they are prone to a view too optimistic.

- *Punctuality assumption*
  Changes have to be made before they happen which is a problem in real life. The database states usually lag some days from the real world due to which this condition is hard to satisfy.

- *Predictive updates*
  If the valid-to is populated with *now* and that happens to be before the valid-from time there is a problem with the notion of time, where valid-from is always before valid-to.

- *Assumptions*
  The accuracy of current time should be very clear because the value of *now* depends on that. This introduces ambiguity when the data is queried on some day but from a different reference point of time. The semantics for the same are not clearly defined currently.

The paper by Clifford [2] tries to define semantics which solve the above problems. In that there is a whole new logical representation framework defined in the paper which is the core of the study for this implementation.

1.2 Logical representations

The semantics proposed in this paper use a fully ground model, which means that there are no variables allowed. There are three different time frames in this representation which are:

- Valid time
- Transaction time
- Reference time

There are also **five important time instants** which are:

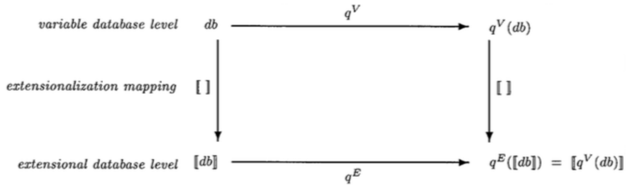
- *Initiation*: Transaction time when a relation is created.
- *Reference time*: Time of database observer's frame of reference. An observer can query the expected database state from any reference time regardless of current time.
- *Query time or Current transaction time*: time at which a query starts being processed.
- *Valid timeslice time*: time for which information is queried.
- *Transaction timeslice time*: time (instants) during which the information must be current in a database.

An example would be: Consider a temporal database relation *inventory* is created on February 2. Current day is March 3. The store manager wants to verify a state of database as reported on February 15. The manager wants to verify that a product after sale on February 7 was deducted in wrong quantity from the inventory on February 10.

- Initiation: February 2, the creation date of the database
- Reference time: February 15, the day of reporting of wrong state
- Current time: March 3
- Valid timeslice: February 7, the real world date when inventory changed
- Transaction timeslice: February 10, when transaction was recorded in database

1.2.1 Extensional and variable database levels

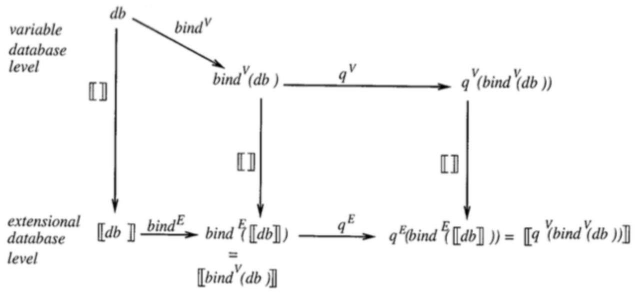
With regard to this framework definition it is assumed that a *time interval* is a set of *time instants* bounded by the starting and terminating instants. In **variable databases** Valid time intervals are made up of *from* and *to* attributes in a tuple and transaction time intervals are made up of *start* and *stop* attributes. This section discusses about how to move to extensional representation of a database starting from variable representation. The basic idea is to get rid any temporal variables in a variable database and all the timestamps are instants instead of intervals. There is an additional temporal attribute called *reference time*. Extensional model presents a very simple treatment of temporal databases with mathematical logic. This conversion of a variable level database to extensional level is called *extensionalization*.



$$\begin{tikzcd} db \arrow[r, "q^V"] \arrow[d, "\llbracket\,\rrbracket"] & q^V(db) \arrow[d, "\llbracket\,\rrbracket"] \\ \llbracket db \rrbracket \arrow[r, "q^E"'] & q^E(\llbracket db \rrbracket) = \llbracket q^V(db) \rrbracket \end{tikzcd}$$

Figure 1: Variable to Extensional

The top level of Figure 1 represents **varaible level** of a database *db*, which after going through q^V queries goes to a state q^V(db). The part of the database supports variables like *now*.
An example would be a tuple in the inventory database:
<Xbox,8 remaining,Electronics,[February 7,now]>
A query is applied on February 25 to this database: "Number of remaining XBox in inventory on February 10"
The result will be: <Xbox, 8 remaining, Electronics>
The bottom level of Figure 1 represents extensional temporal data model.
The extensional mapping of a tuple in the inventory database:
<Xbox,8 remaining,Electronics,[February 7,now]> tuple in the variable db will be represented in extensional model as:
{*<Xbox,8 remaining,Electronics,[February 7,February 25]>*,
<Xbox,8 remaining,Electronics,[February 8,February 25]>,
... *<Xbox,8 remaining,Electronics,[February 25,February 25]>* }



$$\begin{tikzcd} db \arrow[dr, "bind^V"] \arrow[dd, "\llbracket\,\rrbracket"] & & \\ & bind^V(db) \arrow[r, "q^V"] \arrow[d, "\llbracket\,\rrbracket"] & q^V(bind^V(db)) \arrow[d, "\llbracket\,\rrbracket"] \\ \llbracket db \rrbracket \arrow[r, "bind^E"] & bind^E(\llbracket db \rrbracket) = \llbracket bind^V(db) \rrbracket \arrow[r, "q^E"] & q^E(bind^E(\llbracket db \rrbracket)) = \llbracket q^V(bind^V(db)) \rrbracket \end{tikzcd}$$

Figure 2: Preprocessing

In figure 2, a variable database gets mapped into an intermediate where a $bind^V$ operator maps all the tuples' variables with timestamps. This allows to use the same TSQL or SQL based query engines without much modification.

The paper [2] presents the semantics for extensionalization of ValidTime databases with 4 different kinds of variables:

- Now
- Now relative
- Indeterminate
- Now relative indeterminate

In this implementation now based valid time is used, with following resolving strategy:

***NOW* as a lower bound**

- Current time : t_{curr}
- Future time : t_{future}

$NOW = \max\,(t_{curr}, t_{future})$

***NOW* as an upper bound**

- Lower bound (start time) : t_s
- Current tiem : t_{curr}

$$\begin{cases} [t_s \begin{cases} NOW] \begin{cases} t_{curr} \text{ when } t_s \leq t_{curr} \\ t_s \text{ otherwise} \end{cases} \\ NOW) \begin{cases} t_{curr} \text{ when } t_s \leq t_{curr} \\ t_s + 1 \text{ otherwise} \end{cases} \end{cases} \\ (t_s \begin{cases} NOW] \begin{cases} t_{curr} \text{ when } t_s + 1 \leq t_{curr} \\ t_s + 1 \text{ otherwise} \end{cases} \\ NOW) \begin{cases} t_{curr} \text{ when } t_s + 1 \leq t_{curr} \\ t_s + 2 \text{ otherwise} \end{cases} \end{cases} \end{cases}$$

1.3 Physical representation

Physical representation of *NOW* is a very important part of the whole equation because it affects the core performance of the database during the data access [K. Torp, C. S. Jensen, and M. Bohlen. Layered im- plementation of temporal DBMS concepts and tech- niques. A TimeCenter Technical Report TR-2, 1999.]. Following approaches have been proposed:

- *NULL*
  Advantage: It takes less space.
  Disadvantage: Indexing will be a problem due to null values.
- *MIN* (smallest timestamp)
  Disadvantage: Range indexing problem
- *MAX* (largest timestamp)

The paper [1] by Stantic proposes to use time instants instead of ranges to solve these problems. It only needs to make sure about the granularity to be meaningful enough for the application requirements. The query experiments done using this approach give following results as in Figure 3 and 4 which are quite impressive.

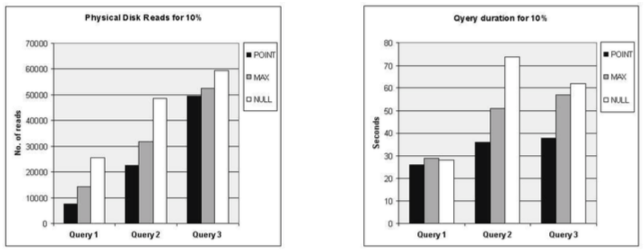


Figure 3: Results

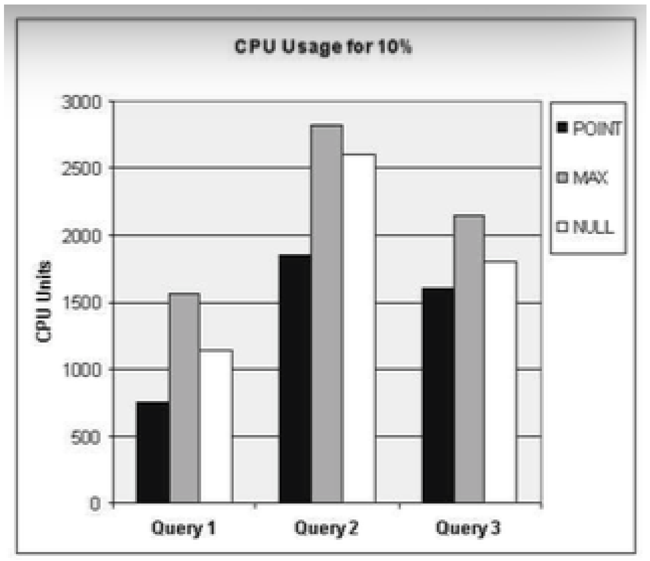


Figure 4: Results

2 PostgreSQL kernel

2.1 Implementation of Now and limited query functionality

A new boolean variable is added to RangeBound struct in rangetypes.h header file. This is the base of representing *NOW* using the literature studies. It is stored physically for the daterange datatypes in each of the relations whenever a new insert is done.
The basic idea implemented in the kernel is to intercept the daterange based SELECT and INSERT queries. Whenever a *NOW* value is intercepted the kernel uses new logic to manipulate the range values based on the now resolving strategy [Optional citing of Yvonne Mülle's research]. The methods changed are range_in() and range_out() for INSERT and SELECT statements respectively to handle the input and output of the *NOW* based data.

2.2 Algorithmic Data flow

Whenever a query is entered in the *psql* utility it is checked for the rangebound. If any of the attributes in the resulting tuples contain rangebound it enters into function range_in() or range_out() depending upon whether its an *INSERT* or a *SELECT* query.

Case 1: INSERT
In this case the landing function is range_in(). The range is first converted into Datum and then checked if the range is valid by comparing the upper and lower bounds. Whenever a *NOW* is detected in this range the resolving strategy kicks

in in this function and the compare function is also updated to accomodate the new way of handling the range. At the time of storage the value of *now* is replaced with a fixed date depending upon the resolving strategy. The fact that the range contains a variable *now* is stored in a new boolean variable added to the rangebound struct called ***bool now***.

Case 2: SELECT

In this case the landing function is range_out(). The data is loaded from the RangeBound Datum and converted to a string. Then based on the resolving strategy and presence of ***bool now*** flags the Datum is converted to a string and passed to the output.

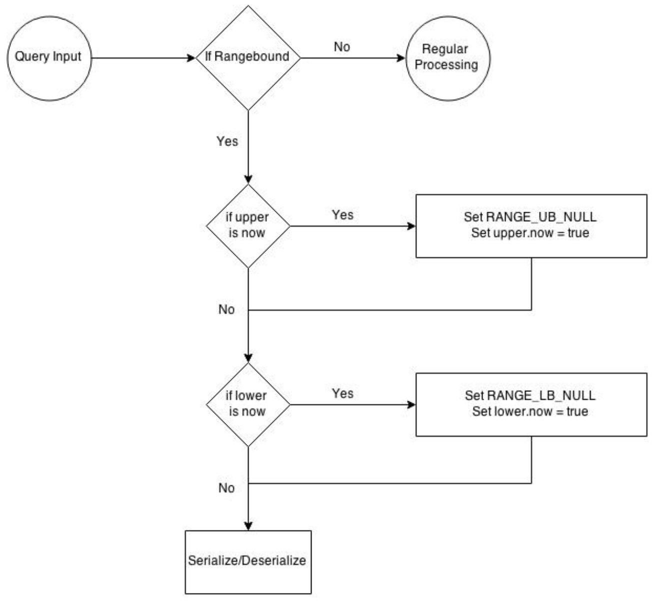


Figure 5: Algorithm

References and Conclusion

I would like to thank Yvonne Mülle and Dr. Michael Böhlen for guiding through this Basismodul study. It had been a pleasure to learn from their expertise and experience. It was a very steep learning curve to start working with the postgresql kernel, but was indeed a very satisfying work to do. I hope to work more with the database group at the ifi for masterproject as well in future.

Bibliography

[1] Bela Stantic, John Thornton, Abdul Sattar, *A Novel Approach to Model NOW in Temporal Databases*, Fourth International Conference on Temporal Logic (IEEE 2003).

[2] James Clifford, Curtis Dyreson, Tomas Isakowitz, Christian S. Jensen, *On the Semantics of "NOW" in Databases*, ACM Transactions on Database Systems, Vol. 22, No. 2, June 1997, Pages 171-214.