Implementation of Temporal Functions Benchmark in PostgreSQL

Facharbeit

Simon Jakob

10-711-000

Dielsdorf, Switzerland

Institut für Informatik der Universität Zürich

Prof. Dr. Michael Böhlen

Supervisor: Amr Noureldin

Deadline: 30.11.2012

Contents

Introduction

Information is the currency of the modern society and thus storing and accessing it effectively becomes increasingly important. To reach this goal, procedural languages were introduced to make more complex computations possible than those possible with declarative SQL.

Now, a great number of the data we produce and want to handle has a time dimension which should be included in the database. To implement procedural functions including the temporal domain is harder than in nontemporal, databases and is therefore a current field of study. An efficient algorithm that converts nontemporal functions, that are easier to implement into temporal is thus of big value.

In this paper, I will first give a short introduction about temporal databases. Then I will show the difficulties of implementing temporal functions, using an example. Second, I will show how the maximally-fragmented slicing approach proposed by Snodgrass et. al. [1] solves this problem, using the same example. Last, I will do performance experiments to evaluate the fitness of this approach for different queries and database sizes.

Temporal Databases

Temporal data is any data provided with a time span when this data is valid , called Valid-Time[2]. It is also possible to save for each information when you knew about it, called Transaction-time, but this is not covered by this paper [2]. Below is an example for a nontemporal and for a temporal Employee relation.

Name	DepID	Salary
A	1	2000
B	2	5000
C	1	4000

Table 1: Nontemporal relation

Name	DepID	Salary	TS	TE
A	1	2000	0	7
B	2	5000	2	5
C	2	1000	0	4
C	1	6000	4	6

Table 2: Temporal relation

The benefit of temporal databases becomes evident in the case of employee C. As C changes departments the TE(valid-**T**ime **E**nd) of that tuple is set to the time when C changes his department. At the same time a new tuple with TS (valid-**T**ime **S**tart) equal to the time when C changes his department is inserted into the database. In nontemporal databases, the only way to handle this change is to update the tuple and so lose the old information.

Invoking a procedural function that gives the employee with the highest salary in the nontemporal database would produce the following output.

Name	DepID	Salary
B	2	5000

Table 3: Nontemporal output

The PostgreSQL-function that does this could look like this:

```
CREATE OR REPLACE FUNCTION maxsalary() RETURNS int AS $$
DECLARE
	maxsalaryint := NULL;
	t RECORD;
BEGIN
	FOR t IN SELECT * FROM employee LOOP
		IF maxsalary IS NULL THEN
			maxsalary := t.salary;
		ELSIF maxsalary<t.salary THEN
			maxsalary := t.salary;
		END IF;
	END LOOP;
RETURN maxsalary;
END;
$$ LANGUAGE PLPGSQL;
SELECT * FROM maxsalary();
```

Codeblock 1: nontemporal query

Asking for the same function in a temporal database the output becomes more complicated as every tuple in the result must be coupled with the time interval during which the tuple is valid:

Name	DepID	Salary	T_Start	T_End
A	1	2000	0	2
B	2	5000	2	4
C	1	6000	4	6
A	1	2000	6	7

Table 4: Temporal output

The PostgreSQL-function that does this could look like this:

```
CREATE TYPE inttemporal AS (maxsalaryint, t int)

CREATE OR REPLACE FUNCTION maxsalary_time() RETURNS SETOF inttemporal as $$
DECLARE
	maxsalaryint := NULL;
	r RECORD;
	t int :=0;
	s inttemporal%ROWTYPE;
BEGIN
	WHILE t<= (SELECT MAX(TE) FROM Employee) LOOP
		Maxsalary := NULL;
		FOR r IN  (SELECT* FROM employee WHERE t >= (SELECT TS FROM Employee) AND
		t <= (SELECT TE FROM Employee)) LOOP
			IF maxsalary IS NULL THEN
				maxsalary := r.salary;
			ELSIF maxsalary<r.salary THEN
				maxsalary := r.salary;
			END IF;
		END LOOP;
		s.maxsalary := maxsalary;
		s.t := t;
		RETURN NEXT s;
		T := t+1;
	END LOOP;
END;

$$ LANGUAGE  PLPGSQL;
```

Codeblock 2: Brute force temporal query

In this example the query is evaluated for every time step from zero to the latest time found in the relation. As this example confirms the statement of Snodgrass that ”every temporal function is three times longer in terms of lines than its nontemporal equivalent” [3] and the premonition that performance of this approach may not be the best, the need for a powerful convertion-algorithm is evident.

Maximally-Fragmented Slicing

Basic Idea

The basic idea of maximally-fragmented slicing (MFS) is to first collect all the tables referenced by the function to then compute constant periods (CP). During a constant period, none of the collected tables undergoes any change and the evaluation of a query thus gives the same result during any subinterval of the constant period. Knowing this, the DBMS will then be requested to evaluate the query for each constant period and then give back the temporal result. The MFS was proposed by Snodgrass et. al. [2].

Implementation

First the constant periods have to be computed. This can be done by first collecting all the timestamps (start time and end time) from the input-tables and put them into a list. In PostgreSQL:

```
CREATE TEMPORARY TABLE timestamps  AS (
SELECT TS AS time_point FROM employee
UNION
SELECT TE AS time_point FROM employee);
```

Codeblock 3: Timestamps

Using this, the constant periods are computed by making `begintime-endtime-pairs` of the timestamps, where `begintime` is always earlier than `endtime` and there is no timestamp in between `begintime`. It is also allowed to just have a `begintime`, if no timestamp exists that is later than it. This conditions are given in the relational calculus expression shown in Figure 1.

$$cp(r_1, r_2, ..., r_n) \quad = \quad \{\langle bt, et\rangle | bt \in ts \wedge et \in ts \wedge bt < et \\ \wedge \neg \exists t \in ts(bt < t < et)\}$$

Figure 1: Expression to extract constant periods [2]

The implementation as a View in PostgreSQL:

```
CREATE VIEW CP AS (
SELECT timestamps1.time_point AS begin_time, timestamps2.time_point AS end_time
FROM timestamps AS timestamps1, timestamps AS timestamps2
WHERE timestamps1.time_point < timestamps2.time_point AND
NOT EXISTS (SELECT time_point FROM timestamps WHERE timestamps1.time_point <time_point AND
time_point< timestamps2.time_point)
ORDER BY timestamps1.time_point) ;
```

Codeblock 4: Constant Periods

Second, changes have to be made to the query that invokes a function. As it should give back a temporal table we have to add the `begintime` and `endtime` of a CP to the `SELECT`-clause and therefore the table CP to the `FROM`-clause. Additionally the WHERE-clause has to be equipped with conditions granting that every tuple that is given to the function overlaps with the `cp.begintime` it is given with.This ensures the validity of the input at a given time. Per definition the evaluation of a query can not change during a CP and because of that it is more efficient to just compare with the `begintime` rather than all of the constant period, which would be much slower.

```
SELECT employee.Name, employee.DepID, employee.Salary, cp.begin_time, cp.end_time
FROM employee,cp
WHERE employee.name = (SELECT name FROM employee WHERE salary = maxsalary(cp.begin_time)
AND  employee.T_Start<= cp.begin_time
AND cp.begin_time<employee.T_End;
```

Codeblock 5: Query invoking the function

Third, changes in the function-body are required. The `WHERE`-clause has to be modified similarly as in the query invoking the function, that is to say by adding conditions to compare the tuples with the `begintime` of the CP. This ensures that the time-valid input is only compared with valid tuples at the `CP-begintime` of the input. Doing so for all the valid tuples for all constant periods gives the final result.

```
CREATE OR REPLACE FUNCTION maxsalary_temp(begin_time_inInt) RETURNS int AS $$
DECLARE
	maxsalaryint := NULL;
	t RECORD;
BEGIN
	FOR t IN SELECT * FROM employee
	WHERE  employee.T_Start<= begin_time_in
	AND begin_time_in<employee.T_End
	LOOP
		IF maxsalary IS NULL THEN
			maxsalary := t.salary;
		ELSIF maxsalary<t.salary THEN
			maxsalary := t.salary;
		END IF;
	END LOOP;
RETURN maxsalary;
END;

$$ LANGUAGE PLPGSQL;
```

Codeblock 6: Function body

Special Case

If the temporal function is invoked in the `FROM`-Clause of the query invoking the function the before explained method does not work entirely. This, because it is not possible to equip every result tupple with its constant period. This special case can be handled by implementing an additional function which then is called by the query. This function could look like this:

```
CREATE TABLE res ( taupsm_item_id CHARACTER (10), sometime DATE, TS DATE, TE DATE);

CREATE OR REPLACE FUNCTION getresult (first_n int)
RETURNS setof rowtype_t AS $$
DECLARE
	cpen record;
BEGIN
	FOR cpen IN (SELECT * FROM CP) LOOP
		INSERT INTO res (SELECT *,cpen.begin_time,cpen.end_time
				FROM get_first_n_items_about_hockey_temp
```

```
                                (first_n,cpen.begin_time));
        END LOOP;
        RETURN QUERY SELECT * FROM res;
END;

$$LANGUAGE PLPGSQL;

SELECT * FROM getresult(10);
```

Codeblock 7:example of a special case result forming function

First a table is created including the variables the original function returns and the `begintime` and `endtime` of the constant period. The function is implemented taking the same input as the original function and returning a set of the before created table. After that a record variable iterates through all cp entries and in this loop all the results of the original query are equipped with their proper constant period and are inserted into the result table. In the end this table is returned and is what you get when you call this function in the `FROM`−clause.

Experiments

In the course of my facharbeit I implemented 17 temporal functions with different functionality and components disregarding the temporal domain. Then I converted them into temporal functions using the maximally-fragmented slicing approach. I then measured evaluation-time for all of these functions applying them on six different databases with the following settings.

dataset	ds1_m2	ds1_m3	ds2_m2	ds2_m3	ds3_m2	ds3_m3
Number of tuples valid in the beginning	25 882	103 768	25 882	103 768	25 882	103 768
Number of slices in the database	104	104	104	104	693	693
Number of inserts/updates/deletes that take place per time step	240	240	240	240	30	30
Selection type	uniform	uniform	gaussian	gaussian	uniform	uniform

Table 5: database settings

In a nutshell the difference between ds1 and ds2 is a different selection type and the difference to ds3 is a different number of slices and changes. The difference in between m2 and m3 is the different size of the databases.

Full results are in the appendix, and shown in figure 2.

Evaluation

Analyzing the results two phenomena come up. First, the overall pattern how the queries perform on the different databases and second, the way some queries have a way better performance in a temporal setting than others.

Performance differences in between datasets

The difference in between ds1 and ds2 are really small and one time ds1 has better performance sometimes ds2. . Table 6 shows the factor that the evaluation of a query on ds1 takes longer than the same query on ds2.

Factor ds1_m2/ds2_m2	Factor ds1_m3/ds2_m3	Average
0.89	1.09	0.99

Table 6: performance difference factor between ds1 and ds2

As the difference really is in the range of the measuring inaccuracy it can be stated that the selection type has just a really small influence on performance of a temporal query produced using maximally fragmented slicing. The only possible finding is that the bigger the dataset gets (m3) the better the Gaussian selectiontype works.

The difference in performance between ds1 (and ds2) and ds3 are considerable.

Factor ds3_m2/ds1_m2	Factor ds3_m3/ds1_m3	Average
7.68	7.89	7.79

Table 7: performance difference factor between ds1 and ds3

As the table shows, the evaluation of a query in ds3 takes in average seven to eight times as long as in ds3. Moreover, the factor is quite identical no matter how big the dataset is. In table 5 it was shown, that ds3 has roughly 7 times as many slices as ds1 but has 8 times less changes per slice. Knowing about the about 8 times longer evaluationtime on ds3 it is obvious that the number of slices affects the performance much more than the number of changes per slice. It is not possible to say for sure with my experiments, but the guess that the number of slices affect the performance in a quadratic way and changes per slice affect it just linearly lies close at hand.

The difference between m2 and m3 is considerable as well. Table 8 shows the factor that the evaluation of a query on m3 takes longer than the same query on m2.

Factor ds1_m2/ds1_m3	Factor ds2_m2/ds2_m3	Factor ds3_m2/ds3_m3	Average
3.31	2.90	3.15	3.12

Table 8: performace difference factor between m2 and m3

The factor is similar in all 3 datasets and shows it roughly takes 3 times as long to evaluate a query in a dataset roughly 4 times as big. Furthermore the factor for ds2 confirms the earlier finding that the gaussian selectiontype works better the bigger the dataset gets, as it is smaller than the other factors.

Performance differences in between queries

There were temporal queries with a really poor performance in my experiment but at the same time there were others which took some thousand times less long to evaluate.

On the long side queries 11 and 14 have to be mentioned. Query 14 took too long to actually measure and 11 even caused memory problems on my computer. Query 14 uses a cursor in the function body which is thus opened, fetched and closed for every constant period. As the experiment shows this causes substantial performance loss. Query 11 creates a temporal table every time the function is invoked. This produces so many temporal data that either my computer or my software cannot handle it. Even if it could handle it, the performance would almost certainly be poor.

On the sort evaluation time side we have for example query 5 which does not return any tuple, so every time the function is invoked it only has to do one comparison to know that no tuple with such specification exist and it is not needed to compare it with any validtime. This obviously results in a short evaluation time. The same applies to query 17b.

The other queries have an evaluation time between 20 and 90 seconds (for ds1). These smaller differences are due to differences in complexity of the functions. I should think that these difference factors are comparable to the factors in a nontemporal setting.

Conclusion

In my work I showed how temporal functions are implemented using the maximally fragmented slicing approach and how to handle special cases. I then showed how temporal functions produced with MFS perform on different datasets and how selectiontype, size of the dataset, number of slices, and number of changes per slice have an influence on performance. Last I showed how different queries perform in a temporal function.

Further studies could extend the experiments to get a quantitative understanding of the influence of the different factors of a dataset on the performance of the temporal function. In addition the performance of MFS temporal functions could be compared with the performance of functions using different methods.

Appendix

All benchmarks are running on a Acer Aspire 5735Z, Intel Pentium dual core processor (2.16GHZ, 667 MHz FSB, 1 MB L2 cache) and are evaluated using pgAdmin. Results in milliseconds.

QUERY	DS1	DS2	DS3
2	78421	85192	553535
2b	83257	87687	542365
3	41808	41216	238103
5	515	2449	2434
6	6146	5850	48750
7	50310	50356	343917
7b	48438	47455	327460
8	31730	29312	225951
9	22354	23322	156281
10	1451	3557	25865
11			
14	907514	864412	>1500000
17	88858	72774	646063
17b			
19	61137	59717	406317
20	249	1466	8736

Literature

[1] Richard T. Snodgrass ,DengfengGao , Rui Zhang , and Stephen W. Thomas. Temporal Support for Persistent Stored Modules. In ICDE, pages 114-125, 2012.

[2] Richard T. Snodgras. A Case Study of Temporal Data

[3] R. T. Snodgrass. Developing Time-Oriented Database Applications in SQL.Morgan Kaufmann Publishers, 2002.