MasterProject

Christine Graff
Mohit Narang

July 1, 2016

Contents

1 Introduction

Temporal databases has been an important topic of interest in database research since the origin of databases because many real world applications are associated with time intervals. A lot of concepts exist to handle the historical data in huge data warehouses [4]. In this project we implement generally valid concept of *now* in the Postgres kernel. We focus on valid time, which is the real world time during which a fact is true.

In this project we worked on four major areas namely the implementation of *NOW* in the kernel, the query tree of a *normalize* statement, execution of a *normalize* query.

This report introduces the concept of *now* in the section 2. It is a very brief introduction of the problems faced by databases when it comes to time varying data and how can those problems be solved using *now.* We show the implementation from a real world use-case point of view. This gives an understanding of our choices when it comes to implementation. Some technical details are self explanatory while the others which need more reasoning to why we chose a particular way to implement something are described in this report.

Section 4 describes the details about physical implementation of *now* in the kernel along with supporting functions which makes the queries and implementation more efficient. Then follow the generally valid predicates. In next part we go into technical details of how the parse tree of a query using a new keyword *normalize* is built. This is followed by the implementation details of an algorithm to output the tuples for an executor function.

In the next part we take a reference query which shows an overview of the entire workflow of a query which goes from a database user into the postgresql kernel. We show the evolution of a simple text based query and how it goes through the parser finally put into the executor to make actual function calls to the functions created in the implementation of *now.* Next we show the empirical results and performance considerations when it comes to the implementation.

2 Concept of Now

The value of *Now* represents the current real world time at a specific granularity. Suppose an employee, Susan, is hired in March 2014 and her employment extends until November 2016. The valid time of her employment is recorded in the database as Employee("Susan", "$[March 2014, November 2016)$") where Susan is the value of employee name and $[March 2014, November 2016)$ is the value of employment duration. However, suppose Susan's employment contract is open-ended. How can the end time of her employment be represented in the database? In this case the end time could be updated every day (or at every increment of the given granularity) to the current time in order to accurately represent the valid time of Susan's employment thus far. However, updating the database at each time increment is costly. A more efficient way to represent valid time is using the concept of *Now*.

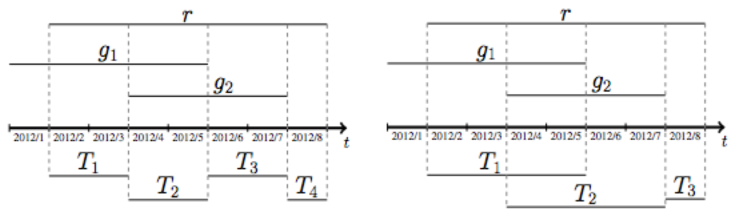
We can specify the valid time range of Susan's employment as "$[March2014, November2016|Now)$". This, represents Susan's employment valid time as beginning on March 2014, continuing at least until November 2016, and possibly beyond. In this way, the open-endedness of her employment can be modeled without the need to constantly update the tuple in the database. Once the current time passes November 2016, the end time of the range becomes *Now*, which represents the current time at each increment.

The advantage of using *Now* in time ranges is that the database does not need to be updated at each time interval, and can perform generally valid calculations, which give correct results at every point in time. This greatly improves the efficiency and reduces the complexity of database operations involving time ranges.

However, there are also a few drawbacks. For example, if the user wants to query whether Susan will be employed at some future time, the database will only consider her valid employment time to continue until the current time. Thus it makes a *pessimistic* assumption about her continued employment, and does not recognize her employee status in the future as valid even though there may be no reason to believe she will be fired. Another problem could occur in cases where the database is updated with some degree of delay, and the *Now* value reflects the real world time of the update rather than of the event it is supposed to describe. In this case, there is an unrealistic assumption about the *punctuality* of an update. For example, if our granularity is days, and Susan is fired on March 3rd 2016, but the database is only updated on March 10th, the *Now* value representing the upper bound of her valid employment time range will be represented as March 10th rather than time she was actually fired, March 3rd. This is because *Now* represents the current time, and will therefore be interpreted as the current time of the update. [4]

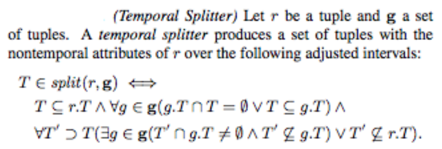
3 Temporal Primitives

In order to get native database support for processing interval timestamped data, Dignos et al [5] implemented the two temporal primitives, temporal splitter and temporal aligner into the kernel of PostgreSQL. These primitives allow operators of temporal algebra to be reduced to their non-temporal counterparts. The temporal splitter splits an argument tuple at each start and end point of all tuples in a specific group, whereas the temporal aligner adjusts an argument tuple according to each tuple of a specific group which it overlaps.



(a) Temporal Splitter. (b) Temporal Aligner.

Temporal alignment and temporal splitter functionality was implemented in version 9.2 of the Postgres kernel. The definition of temporal splitter is as follows:



(Temporal Splitter) Let r be a tuple and $\mathbf{g}$ a set of tuples. A *temporal splitter* produces a set of tuples with the nontemporal attributes of r over the following adjusted intervals:

$$
\begin{aligned}
&T \in split(r, \mathbf{g}) \iff \\
&\quad T \subseteq r.T \wedge \forall g \in \mathbf{g}(g.T \cap T = \emptyset \vee T \subseteq g.T) \wedge \\
&\quad \forall T' \supset T(\exists g \in \mathbf{g}(T' \cap g.T \neq \emptyset \wedge T' \not\subseteq g.T) \vee T' \not\subseteq r.T).
\end{aligned}
$$

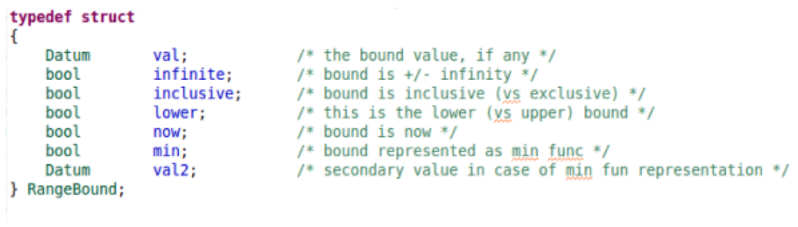
In our project we handle integration of the temporal splitter, normalization, with our implementation of *Now*, however we do not handle temporal alignment.

4 Integration of Now

4.1 Time Domain

The time domain is defined as $\Omega^v = T \cup \{< t|Now >, min(a, < t|Now >)\}$. T represents a linear, discrete time point, such as $01 - 01 - 2015$. The time domain also includes now-relative time points such as $< 01 - 01 - 2015|Now >$ or now-relative minimums such as $min(01 - 01 - 2015, < 01 - 01 - 2017|Now >)$ We represent a time interval as $[T_s, T_e)$, a contiguous set of time points where T_s is the inclusive start point and T_e is the exclusive end point. For example, $[01 - 01 - 2018, < 01 - 01 - 2019|Now >)$.

To implement the concept of *Now* we worked from existing kernel version 9.4, which uses a *daterange* struct to encode time ranges.To represent a now-relative time range, we implemented structs RangeType and RangeBound. RangeType represents a whole time range, and RangeBound represents the upper and lower bounds of the range. Using a struct to represent the start and end times of an interval allows us to represent *Now* and other concepts, through the use of descriptive flags.



```
typedef struct
{
    Datum       val;            /* the bound value, if any */
    bool        infinite;       /* bound is +/- infinity */
    bool        inclusive;      /* bound is inclusive (vs exclusive) */
    bool        lower;          /* this is the lower (vs upper) bound */
    bool        now;            /* bound is now */
    bool        min;            /* bound represented as min func */
    Datum       val2;           /* secondary value in case of min fun representation */
} RangeBound;
```

We use a series of flags to describe the properties of a RangeBound instance. For instance, a bound representing January 1st 2015 has "val" set to the Datum 01-01-2015 and all other flags set to false. *Now* is also represented with a flag. For example, the bound [... , 01-01-2015 — Now) is represented by setting the "val" flag to Datum 01-01-2015 and the "now" flag to true. We can also describe a bound represented as a minimum function, $min(t_a, < t_b|Now >)$, using the flags "min" and "val2". For example, in the case of bound $min(01-01-2015, < 01-01-2017|Now >)$ we set "val" to $01-01-2017$, "val2" to $01-01-2015$, "now" to true, and "min" to true.

Canonicalization Canonicalization is performed on the result in another function, range_serialize, which combines two RangeBounds into a RangeType. In this way we ensure that new ranges are always canonicalized.
Canonicalization is the process of transforming a range T of the form $[T_s, T_e)$ where it undergoes two basic transformations. First transformation transforms the rangebound T_e such that it is max(T_s,T_e). The second step is to apply canonicalization function such that a range is not further reducible from the given possible forms.

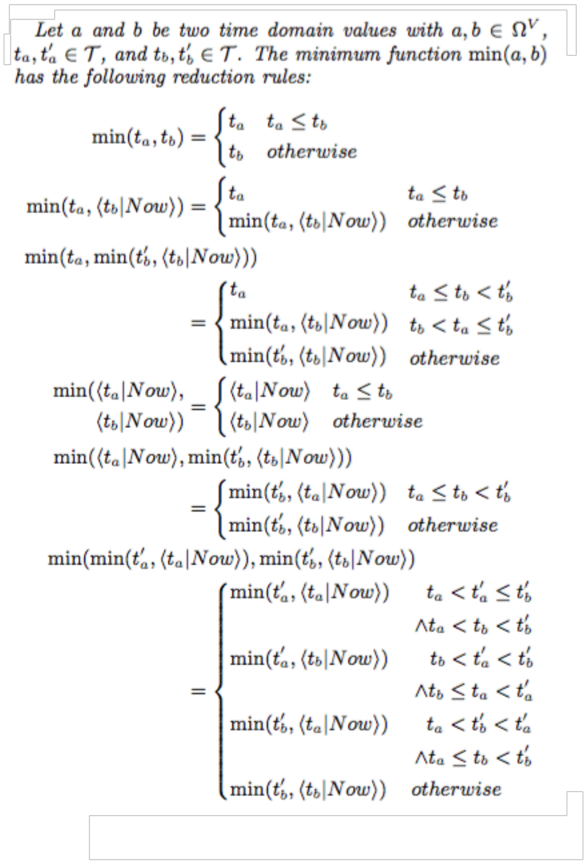
$$\begin{cases} [\min(t_2, \langle t_1|Now\rangle), t_2) & = c([\langle t_1|Now\rangle, \langle t_2|Now\rangle)) \\ [t_1, \langle t_1|Now\rangle) & = c([\min(t_1, \langle t_0|Now\rangle), \\ & \quad \langle t_0|Now\rangle)) \\ [\min(t_2, \langle t_1|Now\rangle), t_2) & = c([\min(t_3, \langle t_1|Now\rangle), \\ & \quad \min(t_3, \langle t_2|Now\rangle))) \\ [t_1, \min(t_2, \langle t_1|Now\rangle)) & = c([\min(t_1, \langle t_0|Now\rangle), \\ & \quad \min(t_2, \langle t_0|Now\rangle))) \\ [t_s, t_e) & = c([t_s, t_e)) \qquad otherwise \end{cases}$$

It has been implemented in the code in `rangeTypes.c` file as a separate function which takes in two rangebounds and canonicalizes it using the min representation reduction rules as shown in the figure above.

4.2 Generally Valid Functions

A function φ on time ranges or time points is generally valid if its result evaluated at every possible current time is equivalent to the result of φ determined over the time ranges or time points evaluated at every possible current time [1]. In order to use intersection and difference queries on now-relative relations, we implemented generally valid intersection and difference functions on time ranges. We implemented generally valid minimum and maximum functions, as part of our intersection and difference function implementations.

Minimum and Maximum Functions The maximum and minimum functions take two RangeBounds as input and return either the maximum or minimum RangeBound respectively. In cases dealing with now-relative time range bounds, determining the minimum or maximum of two bounds often varies with respect to the current time (*Now*). Since our functions must be generally valid (always return the same result regardless of the current time) it is not possible to always return one of the two input arguments as a result. Both the minimum and maximum functions are symmetric. To ensure that the minimum and maximum functions are closed under the time domain, we use reduction rules for all possible combinations of input arguments from the time domain [1].



Let a and b be two time domain values with $a, b \in \Omega^V$, $t_a, t'_a \in \mathcal{T}$, *and* $t_b, t'_b \in \mathcal{T}$. *The minimum function* $\min(a, b)$ *has the following reduction rules:*

$$\min(t_a, t_b) = \begin{cases} t_a & t_a \leq t_b \\ t_b & \textit{otherwise} \end{cases}$$

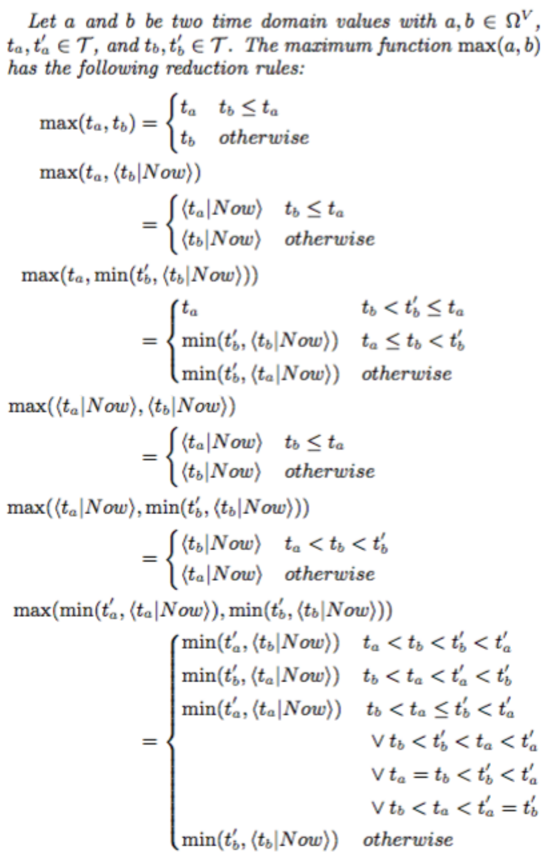
$$\min(t_a, \langle t_b|Now\rangle) = \begin{cases} t_a & t_a \leq t_b \\ \min(t_a, \langle t_b|Now\rangle) & \textit{otherwise} \end{cases}$$

$$\min(t_a, \min(t'_b, \langle t_b|Now\rangle)) = \begin{cases} t_a & t_a \leq t_b < t'_b \\ \min(t_a, \langle t_b|Now\rangle) & t_b < t_a \leq t'_b \\ \min(t'_b, \langle t_b|Now\rangle) & \textit{otherwise} \end{cases}$$

$$\min(\langle t_a|Now\rangle, \langle t_b|Now\rangle) = \begin{cases} \langle t_a|Now\rangle & t_a \leq t_b \\ \langle t_b|Now\rangle & \textit{otherwise} \end{cases}$$

$$\min(\langle t_a|Now\rangle, \min(t'_b, \langle t_b|Now\rangle)) = \begin{cases} \min(t'_b, \langle t_a|Now\rangle) & t_a \leq t_b < t'_b \\ \min(t'_b, \langle t_b|Now\rangle) & \textit{otherwise} \end{cases}$$

$$\min(\min(t'_a, \langle t_a|Now\rangle), \min(t'_b, \langle t_b|Now\rangle)) = \begin{cases} \min(t'_a, \langle t_a|Now\rangle) & t_a < t'_a \leq t'_b \\ & \wedge t_a < t_b < t'_b \\ \min(t'_a, \langle t_b|Now\rangle) & t_b < t'_a < t'_b \\ & \wedge t_b \leq t_a < t'_a \\ \min(t'_b, \langle t_a|Now\rangle) & t_a < t'_b < t'_a \\ & \wedge t_a \leq t_b < t'_b \\ \min(t'_b, \langle t_b|Now\rangle) & \textit{otherwise} \end{cases}$$



Let a and b be two time domain values with $a, b \in \Omega^V$, $t_a, t'_a \in \mathcal{T}$, *and* $t_b, t'_b \in \mathcal{T}$. *The maximum function* $\max(a, b)$ *has the following reduction rules:*

$$\max(t_a, t_b) = \begin{cases} t_a & t_b \leq t_a \\ t_b & \textit{otherwise} \end{cases}$$

$$\max(t_a, \langle t_b|Now\rangle) = \begin{cases} \langle t_a|Now\rangle & t_b \leq t_a \\ \langle t_b|Now\rangle & \textit{otherwise} \end{cases}$$

$$\max(t_a, \min(t'_b, \langle t_b|Now\rangle)) = \begin{cases} t_a & t_b < t'_b \leq t_a \\ \min(t'_b, \langle t_b|Now\rangle) & t_a \leq t_b < t'_b \\ \min(t'_b, \langle t_a|Now\rangle) & \textit{otherwise} \end{cases}$$

$$\max(\langle t_a|Now\rangle, \langle t_b|Now\rangle) = \begin{cases} \langle t_a|Now\rangle & t_b \leq t_a \\ \langle t_b|Now\rangle & \textit{otherwise} \end{cases}$$

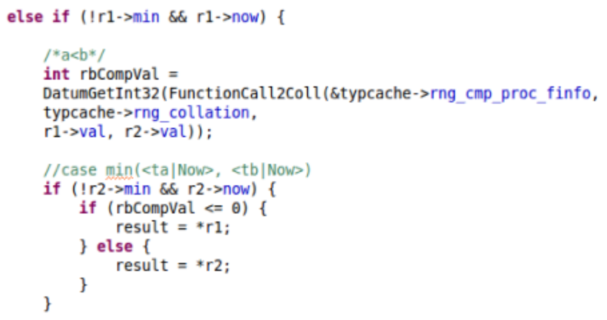
$$\max(\langle t_a|Now\rangle, \min(t'_b, \langle t_b|Now\rangle)) = \begin{cases} \langle t_b|Now\rangle & t_a < t_b < t'_b \\ \langle t_a|Now\rangle & \textit{otherwise} \end{cases}$$

$$\max(\min(t'_a, \langle t_a|Now\rangle), \min(t'_b, \langle t_b|Now\rangle)) = \begin{cases} \min(t'_a, \langle t_b|Now\rangle) & t_a < t_b < t'_b < t'_a \\ \min(t'_b, \langle t_a|Now\rangle) & t_b < t_a < t'_a < t'_b \\ \min(t'_a, \langle t_a|Now\rangle) & t_b < t_a \leq t'_b < t'_a \\ & \vee t_b < t'_b < t_a < t'_a \\ & \vee t_a = t_b < t'_b < t'_a \\ & \vee t_b < t_a < t'_a = t'_b \\ \min(t'_b, \langle t_b|Now\rangle) & \textit{otherwise} \end{cases}$$

In our implementation, the maximum and minimum functions take a TypeCacheEntry and two RangeBounds as arguments.

```
RangeBound range_minimum(TypeCacheEntry *typcache, RangeBound *r1, RangeBound *r2);
RangeBound range_maximum(TypeCacheEntry *typcache, RangeBound *r1, RangeBound *r2);
```

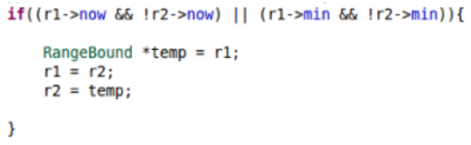
Inside the functions we check to see which reduction rule case applies by comparing the values of the first and second RangeBound flags with one another in a series of if statements. We define a new RangeBound instance, result, and based on the applicable case, we modify the flags of the result instance and then return it at the end of the function. For example, the case $min(< 01 - 01 - 2017|Now >, < 01 - 01 - 2020|Now >)$ is captured in the range_minimum function as follows:



```
else if (!r1->min && r1->now) {

    /*a<b*/
    int rbCompVal =
    DatumGetInt32(FunctionCall2Coll(&typcache->rng_cmp_proc_finfo,
    typcache->rng_collation,
    r1->val, r2->val));

    //case min(<ta|Now>, <tb|Now>)
    if (!r2->min && r2->now) {
        if (rbCompVal <= 0) {
            result = *r1;
        } else {
            result = *r2;
        }
    }
```

We check whether both RangeBounds r1 and r2 have their ”min” flag set to false and their ”now” flag set to true to determine whether the input bounds fit the description $min(< t_a|Now > | < t_b|Now >)$. The applicable case in the reduction rules state that if $t_a \leq t_b$, the result should be $< t_a|Now >$, which is the same as RangeBound input r1, and otherwise the result should be $< t_b|Now >$, which is the same as RangeBound input r2. We compare the ”val” flags of r1 and r2, and if $t_a \leq t_b$ we assign *r1 to *result*, otherwise we assign *r2 to *result*. In our example, $min(< 01 - 01 - 2017|Now >, < 01 - 01 - 2020|Now >)$, the result should therefore be $< 01 - 01 - 2017|Now >$.

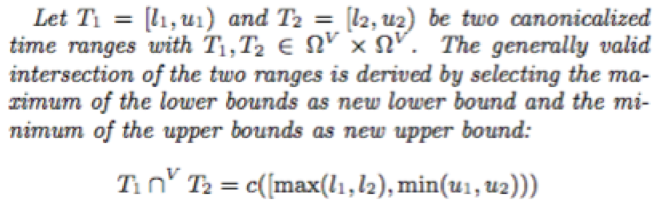
To ensure symmetry, we add a check to the start of the range_maximum and range_minimum functions:



```
if((r1->now && !r2->now) || (r1->min && !r2->min)){

    RangeBound *temp = r1;
    r1 = r2;
    r2 = temp;

}
```

In our implementation of the reduction rules, we assume that in cases such as $min(t_a, < t_b|Now >)$ and $max(t_a, min(t_b^{'}, < t_b|Now >))$ we will always have RangeBound inputs with their ”now” or ”min” flags set to true come second after RangeBound inputs with ”now” or ”min” set to false respectively. If this is not the case, then the order of the inputs has to be switched. To do this, we simply assign r1 to the variable r2 and r2 to the variable r1. In a case where the first input has ”now” and ”min” set to true, but the second input only has ”now” set to true and not ”min”, we also switch the order of the inputs. For example, if the inputs are $[< 01 - 01 - 2015|Now >, 01 - 01 - 2016)$, we would assign the second input to r1 and the first input to r2. If the inputs are $[min(01 - 01 - 2016, < 01 - 01 - 2014|Now >))$, we would also assign the first input to r2 and the second input to r1.

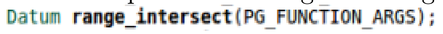
Intersection Function The generally valid intersection function makes use of the generally valid maximum and minimum functions. The intersection function is defined as follows [1]:



Let $T_1 = [l_1, u_1)$ and $T_2 = [l_2, u_2)$ be two canonicalized time ranges with $T_1, T_2 \in \Omega^V \times \Omega^V$. The generally valid intersection of the two ranges is derived by selecting the maximum of the lower bounds as new lower bound and the minimum of the upper bounds as new upper bound:

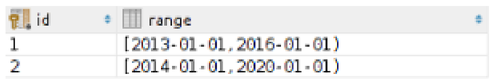
$$T_1 \cap^V T_2 = c([\max(l_1, l_2), \min(u_1, u_2)))$$

Version 9.4 of the project contains a function range_intersect to handle intersection queries involving time ranges.



```
Datum range_intersect(PG_FUNCTION_ARGS);
```

Consider the following table containing two tuples with time ranges and ids.



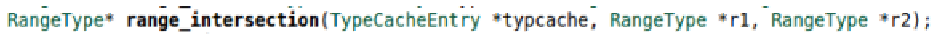
id	range
1	[2013-01-01,2016-01-01)
2	[2014-01-01,2020-01-01)

The existing range_intersect function is called when the user issues the query:

SELECT (a.range * b.range) FROM table as a, table as b where a.id = 1 AND b.id = 2

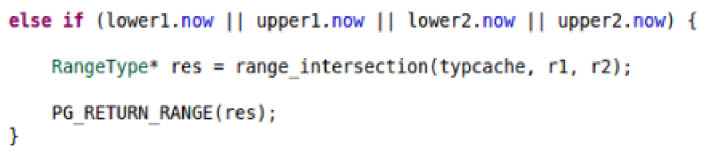
The result returned from this query is the intersection of the two time ranges: $[01-01-2014, 01-01-2016)$.

However, the existing range_intersect function cannot handle queries on time ranges including *Now*. We implement a second intersection function, range_intersection to handle cases using the definition of the generally valid intersection function. Our implementation takes a TypeCacheEntry and two RangeTypes as input.



```
RangeType* range_intersection(TypeCacheEntry *typcache, RangeType *r1, RangeType *r2);
```

We modify the existing range_intersect function to check whether any of the RangeBounds from the input include *Now*. If so, we call our implementation, range_intersection, and return the result of that function.



```
else if (lower1.now || upper1.now || lower2.now || upper2.now) {
    RangeType* res = range_intersection(typcache, r1, r2);
    PG_RETURN_RANGE(res);
}
```

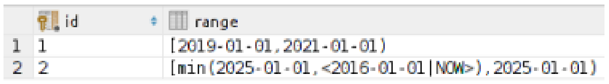
Inside the function range_intersection, as in the existing range_intersect, we call the function range_deserialize(TypeCacheEntry *typecache, RangeType *range, RangeBound *lower, RangeBound *upper, bool *empty), which takes a TypeCacheEntry and a RangeType, and deconstructs it into its upper and lower RangeBounds, which are stored in the variables upper and lower.

```
range_deserialize(typcache, r1, &lower1, &upper1, &empty1);
range_deserialize(typcache, r2, &lower2, &upper2, &empty2);
```

Once we have the upper and lower RangeBounds of the RangeType we can feed them to the range_maximum and range_minimum functions to get the upper and lower bounds of the result.

```
RangeBound rbmin = range_minimum(typcache, &lower2, &upper1);
RangeBound rbmax = range_maximum(typcache, &upper2, &lower1);
```

Our implementations of the generally valid maximum and minimum functions, range_maximum and range_minimum, cannot be called directly by the user in a query. However, since our implementation of the now-inclusive intersection function, range_intersection, uses range_maximum and range_minimum, we can test range_maximum and range_minimum with intersection queries to make sure the reduction rules of the various generally valid minimum and maximum cases are working correctly. For example suppose our database contains a relation "table" containing the following tuples:



	id	range
1	1	[2019-01-01,2021-01-01)
2	2	[min(2025-01-01,<2016-01-01|NOW>),2025-01-01)

We can use a query to find the intersection of the ranges with "id" values 1 and 2:
SELECT (a.range * b.range) FROM table as a, table as b where a.id = 1 AND b.id = 2
As stated in the definition of the generally valid intersection function, the lower bound of the result is $max(l_1, l_2)$ and the upper bound of the result is $min(u_1, u_2)$. This coincides with the generally valid maximum and minimum reduction rule cases $min(t_a, < t_b|Now >)$ and $max(t_a, < t_b|Now >)$. According to these rules, the result of the query should be $[min(01-01-2016, < 01-01-2019|Now >, < 01-01-2-15|Now >)$. In this way, we can issue queries coinciding with all of the minimum and maximum reduction rules to check whether our implementation of range_minimum, range_maximum and range_intersection are working correctly.

Difference Function The generally valid minimum and maximum functions are also used by our implementation of the difference function, which allows the user to query the difference between two now-relative time ranges. The difference function is defined as follows [1]:

Let $T_1 = [l_1, u_1)$ and $T_2 = [l_2, u_2)$ be two canonicalized time ranges with $T_1, T_2 \in \Omega^V \times \Omega^V$. The difference of the two ranges is either range T_1 or two ranges, the first one derived from the lower bounds of T_1 and T_2, and the second one from the upper bounds of these ranges.

$$T_1 -^V T_2 = \begin{cases} T_1 & T_1 \cap^V T_2 =^V \emptyset \\ \{c(T_a), c(T_b)\} & \textit{otherwise} \end{cases}$$

with

$$T_a = [l_1, \min(l_2, u_1))$$
$$T_b = [\max(u_2, l_1), u_1)$$

The difference function can return either one, two, or zero results. In cases where there are two results of a the difference function, both T_a and T_b return a non-empty range. In cases where there is only one result, either T_a or T_b returns a non-empty range, and in cases where there are zero results, both T_a and T_b return empty ranges. T_a covers the range between the lower bound of the first input range and the lower bound of the second input range. The minimum function in its upper bound makes sure that this value never exceeds the upper bound of the first input range. T_b covers the range between the upper bound of the second input range and the upper bound of the first input range. The maximum function in its lower bound ensures that the range never gets below the lower bound of the first input range.

Postgres only supports contiguous difference functions. It is not possible to return two results from a single function at once. Therefore we chose to implement the generally valid difference function by creating two separate functions for the possible lower and upper results of difference, T_a and T_b, and defining them in pg_proc as functions minus_ta and minus_tb, so that they can be directly called in queries. We also implement a new function range_difference which, similar to range_intersection, takes a TypeCacheEntry and two RangeTypes as input. However, because there can be up to two results, it returns a pointer to a RangeList struct instead of a RangeType.

```
RangeType* minus_ta(PG_FUNCTION_ARGS);
RangeType* minus_tb(PG_FUNCTION_ARGS);
RangeList* range_difference(TypeCacheEntry *typcache, RangeType *r1, RangeType *r2);
```

The RangeList struct can store up to two pointers to RangeTypes, which are the results of the difference query. The range_difference function calculates

T_a and T_b and adds the results to the RangeList before returning it.

```
typedef struct
{
    RangeType* ta;
    RangeType* tb;
    bool empty;

} RangeList;
```

As in the case of range_intersection, canonicalization occurs in the serialize function responsible for making the result RangeType out of two RangeBounds. To query now-relative difference, the user calls both minus_ta and minus_tb. Both functions call the range_difference function. If there is one result, it is returned by minus_ta while minus_tb returns an empty range. If there are two results, minus_ta returns the lower result and minus_tb returns the upper result. In the case of zero results, both functions return empty ranges.

4.3 Predicates

We implemented the three predicates, equals, overlaps, and isEmpty as they are required for a correct implementation of the generally valid interseciton and difference function and for filtering tuples within queries.

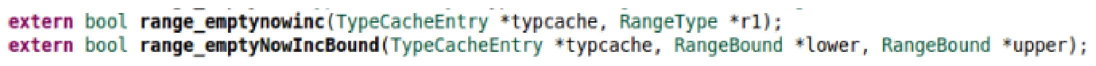
Equals The equals predicate is true if two time ranges describe the same range at every possible point in time. For example, the time ranges $[2015, < 2015|Now >)$ and $[2015, < 2014|Now >)$ are equal. Until the current time reaches 2015, both time ranges are empty, and after 2015 both ranges represent a valid time range which starts at 2015 and ends at the current time or *Now* [1]. Our implementation of the function which calculates the equals predicate is called range_equals. It returns a boolean representing the equals predicate, and takes a TypeCacheEntry and two RangeBounds as arguments.

```
bool range_equals(TypeCacheEntry *typcache, RangeBound *r1, RangeBound *r2)
```

Inside the function, we compare the values of all of the flags of the two RangeBound inputs r1 and r2, except the lower and inclusive flags. If every relevant flag of r1 has the same value as the corresponding flag of r2, then the ranges are equal and the function returns true. Otherwise they are not equal and the function returns false.

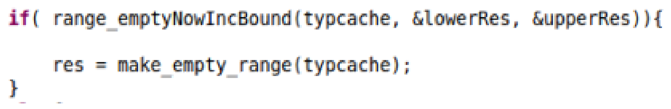
Empty In cases where the result of the range_difference or range_intersection function is an empty time range, we have to specifically make and return an empty RangeType. To detect whether now-relative time ranges are empty, we implement two functions: range_emptynowinc, which takes a TypeCacheEntry and a RangeType as arguments, and range_emptyNowIncBound, which takes a

TypeCacheEntry and two RangeBounds as arguments.



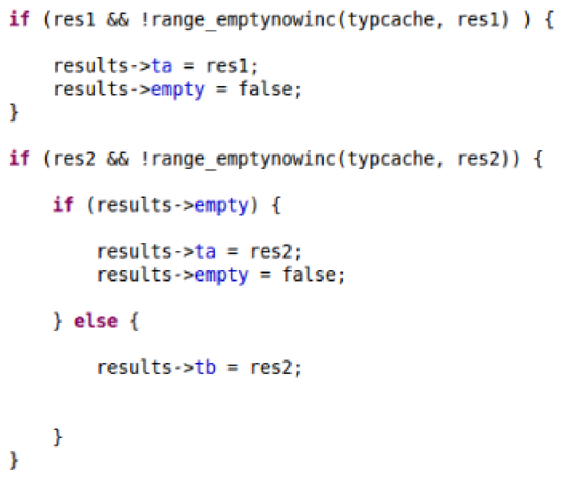
```
extern bool range_emptynowinc(TypeCacheEntry *typcache, RangeType *r1);
extern bool range_emptyNowIncBound(TypeCacheEntry *typcache, RangeBound *lower, RangeBound *upper);
```

The range_intersection function checks whether the upper and lower bound of the resulting time range constitute an empty time range, and if so makes and returns and empty RangeType.



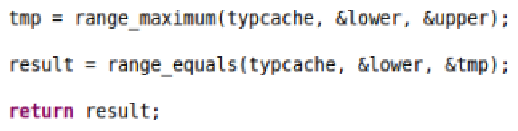
```
if( range_emptyNowIncBound(typcache, &lowerRes, &upperRes)){

    res = make_empty_range(typcache);
}
```

The range_difference function checks the results of T_a and T_b and only adds them to the RangeList result if they are non-empty.



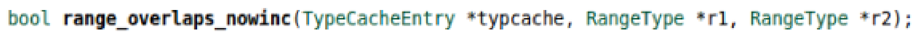
```
if (res1 && !range_emptynowinc(typcache, res1) ) {

    results->ta = res1;
    results->empty = false;
}

if (res2 && !range_emptynowinc(typcache, res2)) {

    if (results->empty) {

        results->ta = res2;
        results->empty = false;

    } else {

        results->tb = res2;


    }
}
```

To determine whether a now-relative time range is empty in both the range_emptynowinc and range_emptyNowIncBound functions, we first determine the maximum of the upper and lower RangeBounds of the time range, and then compare the lower RangeBound with the result using range_equals. If range_equals returns true, then the lower RangeBound must be greater than the upper RangeBound, and thus the time range is empty.



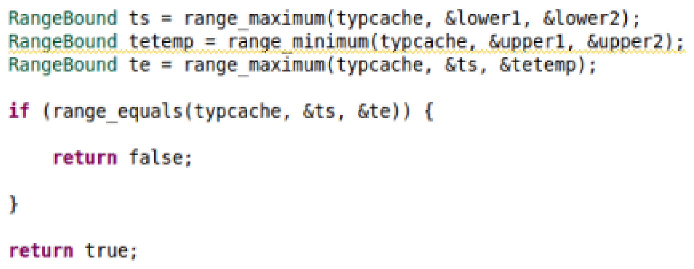
```
tmp = range_maximum(typcache, &lower, &upper);

result = range_equals(typcache, &lower, &tmp);

return result;
```

Overlaps The overlaps predicate is true if two now-relative time ranges overlap one another. We use the intersection and the equals predicate to determine whether two time ranges overlap. For example, if we have time ranges $[2013, 2017)$ and $[2013, < 2013|Now >)$, and the current time is 2015, then their intersection is $[2013, 2015)$. Since this non-empty intersection exists for at least one point in time, we know that the time ranges must overlap. [1] We implement this calculation in the function range_overlaps_nowinc, which takes a TypeCacheEntry and two RangeTypes, and returns a boolean which is true if the RangeTypes overlap one another and false if they do not overlap.



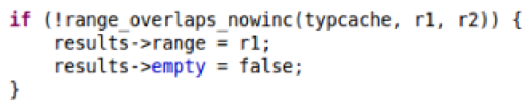
```
bool range_overlaps_nowinc(TypeCacheEntry *typcache, RangeType *r1, RangeType *r2);
```

To determine whether two RangeTypes overlap, we find the upper and lower RangeBounds of the intersection, and then check whether the maximum of these two RangeBounds is equal to the lower bound of the intersection range. If so, then the intersection must be empty, and thus the input time ranges do not overlap.



```
RangeBound ts = range_maximum(typcache, &lower1, &lower2);
RangeBound tetemp = range_minimum(typcache, &upper1, &upper2);
RangeBound te = range_maximum(typcache, &ts, &tetemp);

if (range_equals(typcache, &ts, &te)) {

    return false;

}

return true;
```

We use the range_overlaps_nowinc function in our range_difference function. If the ranges to not overlap, we know that the difference of the two RangeTypes is equal to r1.



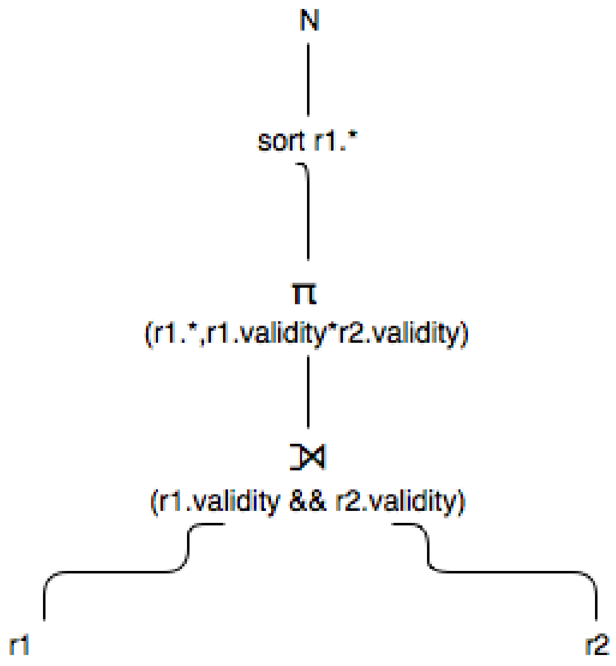
```
if (!range_overlaps_nowinc(typcache, r1, r2)) {
    results->range = r1;
    results->empty = false;
}
```

5 Normalize Primitive

Normalization was already implemented in postgresql version 9.2 for the temporal alignment project [2] using datetime types and two attributes ts and te. In this project we first ported that implementation to postgresql version 9.4. After that we changed the `analyze.c and nodeadjustment.c` files to start using the newly updated `Daterange` to work with time domain Ω^v and NOW. Normalize is a type of select predicate which is implemented in `execAdjustment` function for its execution. The simple text query from a terminal client like **psql** is first parsed in the `transformAdjustmentStmt` function in **analyze.c**. This function takes in the parse state and the statement as parameter and transforms the normalize part of the query into standard parse tree which removes the normalize keyword and instead replaces it with a custom select statement. This new tree is then passed on to the executor function `execAdjustment` in `nodeAdjustment.c` file in the form of a data structure called AdjustmentState which includes extra information like result tuples as well.

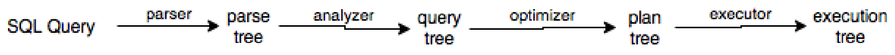
5.1 Parsing of the query

The parser takes in the simple text query of the form:
`SELECT ALIASNAME.* FROM (R1 NORMALIZE R2 USING()) ALIASNAME;`
The normalize part is then simplified by using `transformAdjustmentStmt` function in **analyze.c**. The parsing is done according to the below query tree:

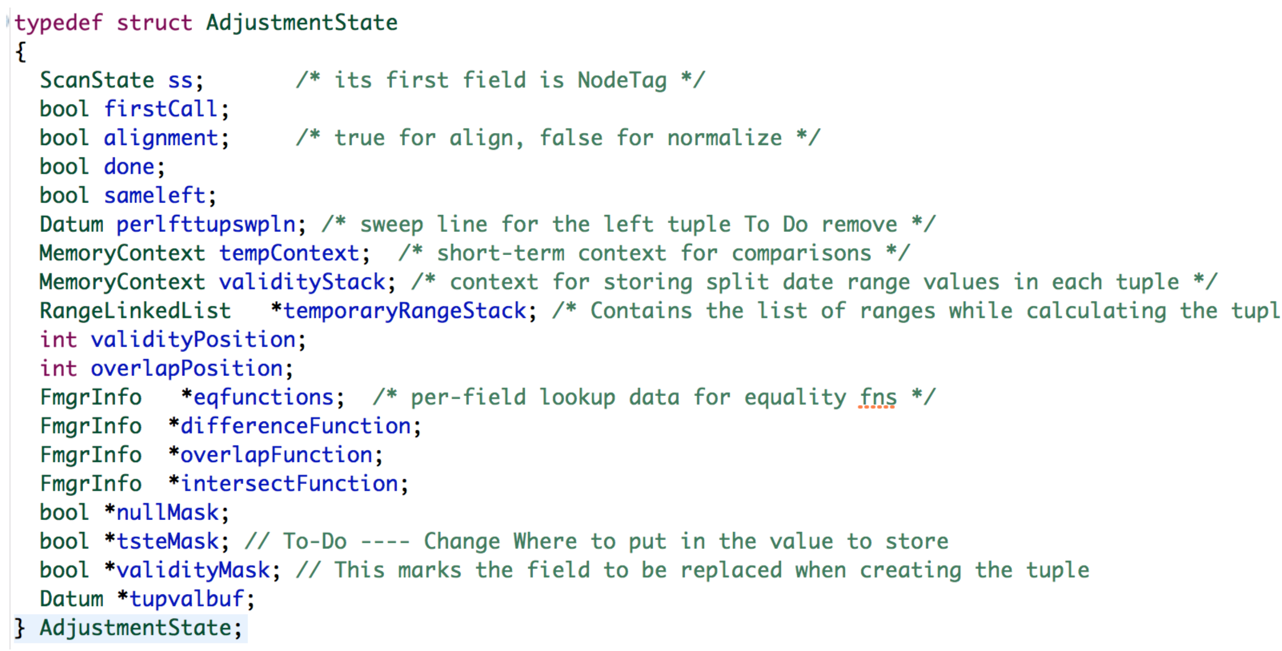


5.2 Execution of the query

The high level of processing of a query in postgres kernel is as following:

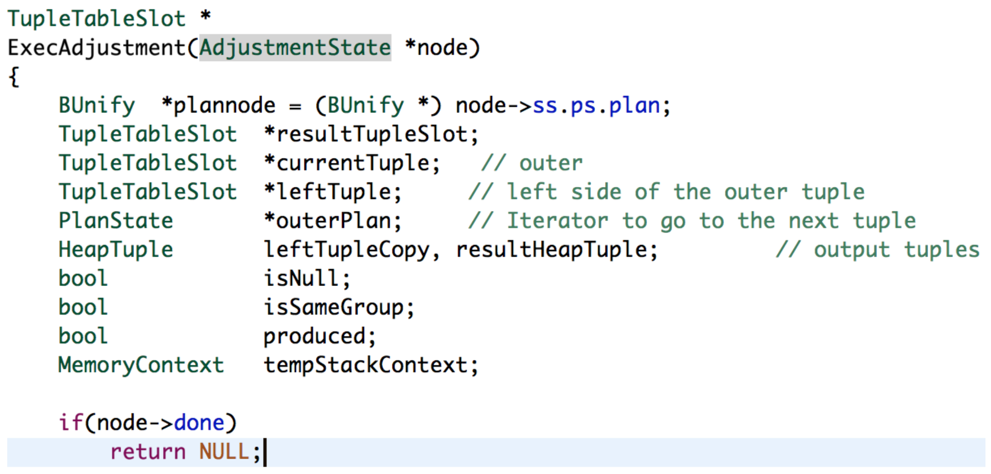


The executor function takes in the adjustmentstate node as an argument. The adjustment state is a data structure defined in the `execnodes.h` file.



```
typedef struct AdjustmentState
{
  ScanState ss;       /* its first field is NodeTag */
  bool firstCall;
  bool alignment;     /* true for align, false for normalize */
  bool done;
  bool sameleft;
  Datum perlfttupswpln; /* sweep line for the left tuple To Do remove */
  MemoryContext tempContext;  /* short-term context for comparisons */
  MemoryContext validityStack; /* context for storing split date range values in each tuple */
  RangeLinkedList   *temporaryRangeStack; /* Contains the list of ranges while calculating the tupl
  int validityPosition;
  int overlapPosition;
  FmgrInfo   *eqfunctions;  /* per-field lookup data for equality fns */
  FmgrInfo  *differenceFunction;
  FmgrInfo  *overlapFunction;
  FmgrInfo  *intersectFunction;
  bool *nullMask;
  bool *tsteMask; // To-Do ---- Change Where to put in the value to store
  bool *validityMask; // This marks the field to be replaced when creating the tuple
  Datum *tupvalbuf;
} AdjustmentState;
```

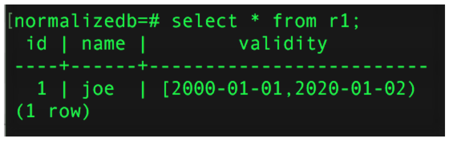
The adjustment state stores the status of execution in variable `ss`. The basic idea of the algorithm is to go through each tuple group from the query tree. The algorithm expects *validity* attribute as default attribute in a relation to perform normalize upon and it is defined in the structure as `validityPosition`.



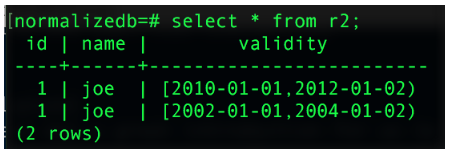
```
TupleTableSlot *
ExecAdjustment(AdjustmentState *node)
{
    BUnify  *plannode = (BUnify *) node->ss.ps.plan;
    TupleTableSlot  *resultTupleSlot;
    TupleTableSlot  *currentTuple;   // outer
    TupleTableSlot  *leftTuple;     // left side of the outer tuple
    PlanState       *outerPlan;     // Iterator to go to the next tuple
    HeapTuple       leftTupleCopy, resultHeapTuple;         // output tuples
    bool            isNull;
    bool            isSameGroup;
    bool            produced;
    MemoryContext   tempStackContext;

    if(node->done)
        return NULL;
```

The algorithm for executor is implemented in function `ExecAdjustment` in the file `nodeAdjustment.c`. As soon as the function gets called the first time it initializes the variables `plannode, currentTuple` which contain the execution plan state and the current tuple in the plan respectively. The group is maintained in the variable `leftTuple`. The `temporaryRangeStack` variable in the adjustment state maintains list of range splits for the processing of each group

in the input node which persists multiple calls to the executor during an execution plan. As soon as that group is processed the `temporaryRangeStack` is emptied and the algorithm proceeds to process the next group in the input adjustmentState node. To perform the normalize splits the the algorithm gets the difference of a group with all the tuples, and existing stack. Then stores it into the stack until there are no more possible splits and then starts to output the tuples one by one until the temporaryRangeStack gets empty.
A more detailed pseudo code of the algorithm is shown below. We show an example run of the algorithm with a relation r1 and r2 containing following data where r1 is normalized by r2:



```
[normalizedb=# select * from r1;
 id | name |        validity
----+------+-------------------------
  1 | joe  | [2000-01-01,2020-01-02)
(1 row)
```



```
[normalizedb=# select * from r2;
 id | name |        validity
----+------+-------------------------
  1 | joe  | [2010-01-01,2012-01-02)
  1 | joe  | [2002-01-01,2004-01-02)
(2 rows)
```

During the first call to the algorithm, the variables `leftTuple and currentTuple` are both initialized by a tuple which is the result of left join from the query tree result. In this case it would contain the data $[2010-01-01, 2012-01-02)$ in the overlap position of the join result tuple. Since this is the first call of the execution plan the variable `firstcall` is true and hence `leftTuple` which represents a group is set to current tuple and `firstcall` is updated to false. Next it moves to check the temporaryRangeStack to see if a previous group is already in execution in which case the function outputs those tuples. But since we are in the first call to the function we move on to the next part and check if the tuples belong to the same group to determine if the algorithm should proceed to start outputting the tuple of still keep updating the temporaryRangeStack. Then we go into a while loop, iterating over all the existing values in temporaryRangeStack for each new tuple in the same group. Whenever a new overlap is found after difference and intersection between the validity of a tuple and the stack the stack is updated with newly split ranges. This keeps on iterating until a group is successfullly processed till the end. Then the algorithm goes to next group and hence enters a base case of outputting the exisiting tuples.

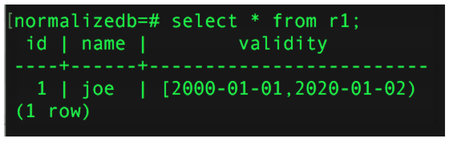
5.2.1 Pseudo code of the algorithm for execution of the query

```
Function: ExecAdjustment(n)
Input: Node n in execution tree
Output: A single output tuple or $\omega$
Copy variables of n to local variables
samegroup < − false
if firstcall then
    currentTuple < − n.outerplan; leftTuple < − currentTuple;
    if currentTuple = null then
        n.done < − true; return null;
    end
    firstcall < − false;
end
if temporaryRangeStack $\neq$ null then
    resultTuple < − heapTuple(temporaryRangeStack.range)
    produced < − true
    temporaryRangeStack < − temporaryRangeStack.next
    return resultTuple
else
    if currentTuple = null then
        n.done < − true; return null;
    end
    leftTuple < − currentTuple;
end
if leftTuple = currentTuple then
    samegroup < − true;
end
while samegroup = true do
    if temporaryRangeStack = null then
        differenceResult < − difference(leftTuple.validity, currentTuple.overlap)
        if differenceResult $\neq$ null then
            currentRangeStack < − diffrenceResult.range
            if differenceResult.next $\neq$ null then
                currentRangeStack.next < − diffrerenceResult.next
            end
        end
        temporaryRangeStack < − currentRangeStack
        intersectResult < − intersect(leftTuple.validity, currentTuple.overlap)
        if intersectResult $\neq$ null then
            intersectResult.next < − temporaryRangeStack
            temporaryRangeStack < − intersectResult
        end
    else
    else
        copyOfTempStack < − temporaryRangeStack
        resultStack < − null
        while copyOfTempStack ≠ null do
            differenceResult < − difference(copyOfTempStack.range, currentTuple.overlap)
            tempRangeStackDiff < − null; tempRangeStackInter < − null;
            if diffrerenceResult = null then
                tempRangeStackDiff < − resultStack
            else
                tempRangeStackDiff < − diffrerenceResult.range
                if differenceResult.next ≠ null then
                    tempRangeStackDiff.next < − differenceResult.next
                    tempRangeStackDiff.next.next < − resultStack
                else
                    tempRangeStackDiff.next < − resultStack
                end
            end
            intersectResult < − intersect(copyOfTempStack.range, currentTuple.overlap)
            if intersectionResult = null then
                tempRangeStackInter < − tempRangeStackDiff
            else
                tempRangeStackInter < − intersectionResult
                tempRangeStackInter.next < − tempRangeStackDiff
            end
            resultStack < − tempRangeStackInter
            copyOfTempStack < − copyOfTempStack.next
        end
        temporaryRangeStack < − resultStack
    end
    currentTuple < − fetch tuple from n.outerplan
    if currentTuple = null then
        samegroup < − false
    else
        samegroup < − execTuplesMatch(leftTuple, currentTuple)
    end
    if temporaryRangeStack ≠ null then
        storeTuple(resultHeapTuple, resultTupleSlot) temporaryRangeStack < −
          temporaryRangeStack.next
        return resultTupleSlot
    end
end
if n.done then
    return null
end
return resultTupleSlot
```

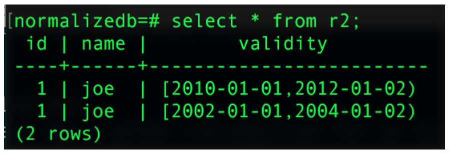
Algorithm 1: Executor function

5.3 Reference Query

In this section we show a reference query being transformed during its way to the final execution in the kernel. The database hosts relations r1 and r2 which has following data:



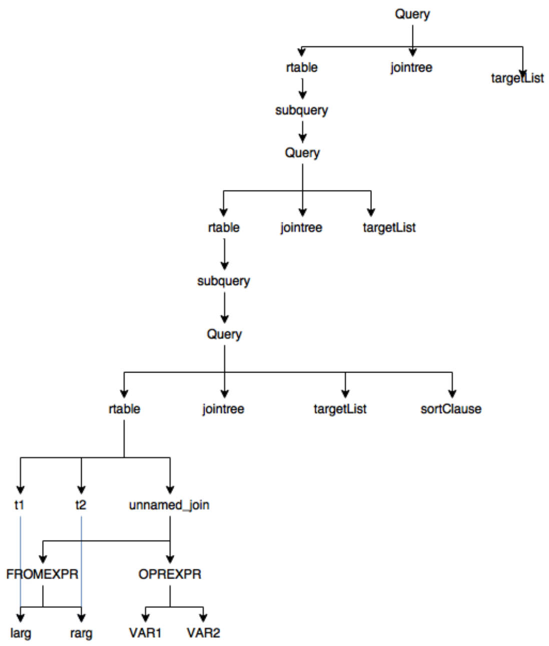
```
normalizedb=# select * from r1;
 id | name |         validity
----+------+-------------------------
  1 | joe  | [2000-01-01,2020-01-02)
(1 row)
```



```
normalizedb=# select * from r2;
 id | name |         validity
----+------+-------------------------
  1 | joe  | [2010-01-01,2012-01-02)
  1 | joe  | [2002-01-01,2004-01-02)
(2 rows)
```

Normalize query is a kind of `select` query which have a *normalize* keyword in the `from` clause of the query. For example relations `r1` and `r2` a normalize select would look like:
`SELECT * FROM (r1 NORMALIZE r2 USING ()) resultAlias;`

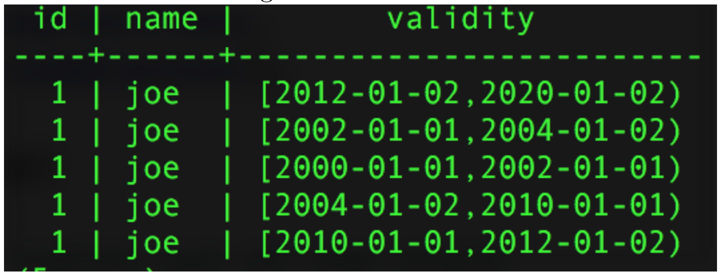
When the above query goes through the analyzer, the function `transformAdjustment` from file `analyze.c`, the `r1 NORMALIZE r2 USING ()` part of the query is transformed into the following query tree:



The above query (**only the normalize subquery**) needs to be re-written to an equivalent query which looks like:
```
SELECT r1.*, r1.Validity*r2.Validity
FROM r1 LEFT JOIN r2
```

```
ON r1&r2
WHERE r1.validity OVERLAPS r2.validity
AND <Expr>
ORDER BY r1.*
```

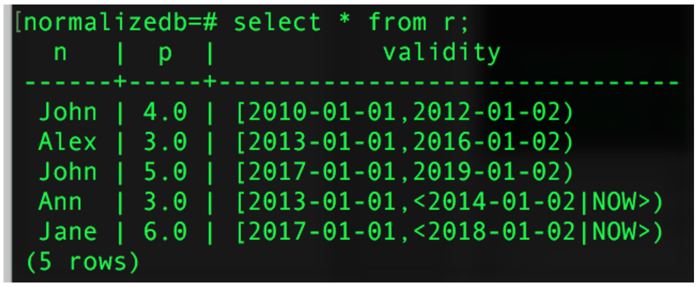
The result is then given as below:



```
 id | name |         validity
----+------+------------------------
  1 | joe  | [2012-01-02,2020-01-02)
  1 | joe  | [2002-01-01,2004-01-02)
  1 | joe  | [2000-01-01,2002-01-01)
  1 | joe  | [2004-01-02,2010-01-01)
  1 | joe  | [2010-01-01,2012-01-02)
```

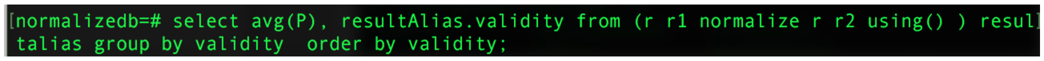
6 Results and Empirical Evaluation

We created a relation r which has three attributes n,p and validity. It has following rows:



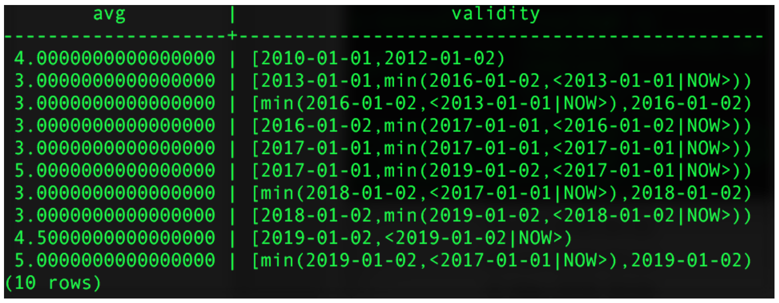
```
[normalizedb=# select * from r;
  n   |  p  |            validity
------+-----+-----------------------------
 John | 4.0 | [2010-01-01,2012-01-02)
 Alex | 3.0 | [2013-01-01,2016-01-02)
 John | 5.0 | [2017-01-01,2019-01-02)
 Ann  | 3.0 | [2013-01-01,<2014-01-02|NOW>)
 Jane | 6.0 | [2017-01-01,<2018-01-02|NOW>)
(5 rows)
```

We query the above relation with the following query:



```
[normalizedb=# select avg(P), resultAlias.validity from (r r1 normalize r r2 using() ) resul
 talias group by validity  order by validity;
```

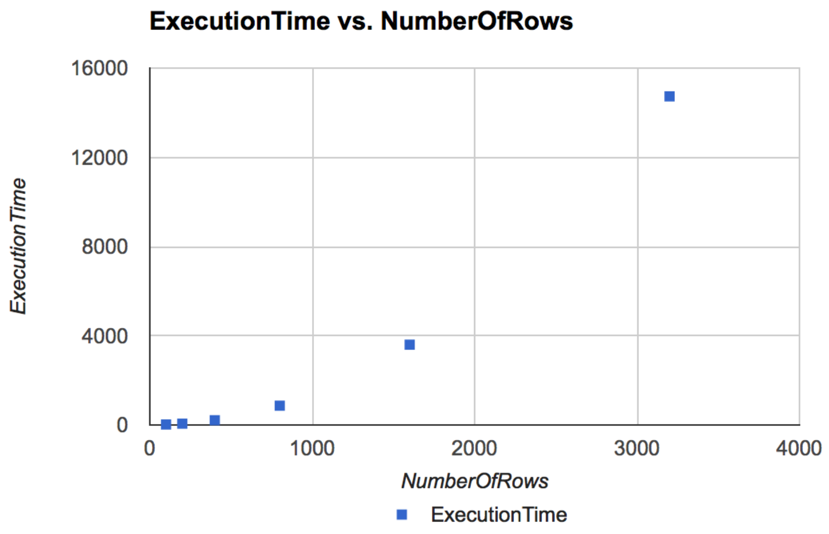
The result of the above query is as following:



```
       avg         |                    validity
-------------------+-----------------------------------------------
 4.0000000000000000 | [2010-01-01,2012-01-02)
 3.0000000000000000 | [2013-01-01,min(2016-01-02,<2013-01-01|NOW>))
 3.0000000000000000 | [min(2016-01-02,<2013-01-01|NOW>),2016-01-02)
 3.0000000000000000 | [2016-01-02,min(2017-01-01,<2016-01-02|NOW>))
 3.0000000000000000 | [2017-01-01,min(2017-01-01,<2017-01-01|NOW>))
 5.0000000000000000 | [2017-01-01,min(2019-01-02,<2017-01-01|NOW>))
 3.0000000000000000 | [min(2018-01-02,<2017-01-01|NOW>),2018-01-02)
 3.0000000000000000 | [2018-01-02,min(2019-01-02,<2018-01-02|NOW>))
 4.5000000000000000 | [2019-01-02,<2019-01-02|NOW>)
 5.0000000000000000 | [min(2019-01-02,<2017-01-01|NOW>),2019-01-02)
(10 rows)
```

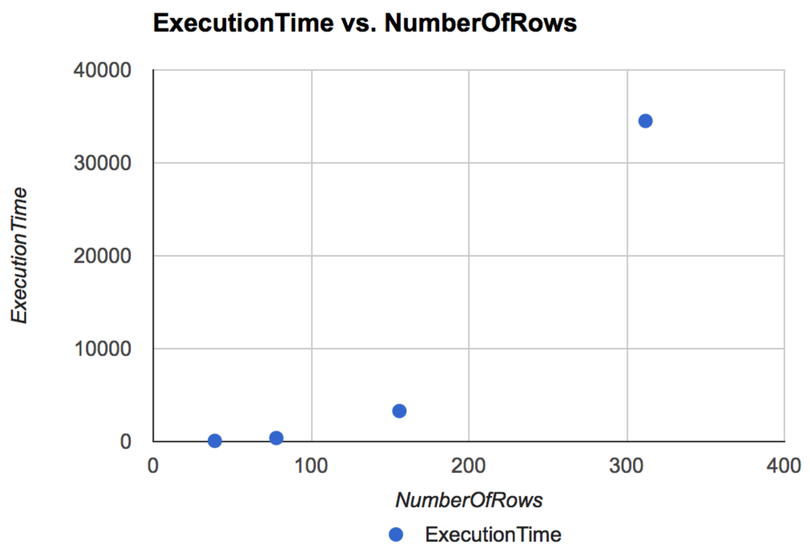
For the performance evaluation of normalize we created two relation T1 and T2. T1 with dynamic number of rows and T2 contains a single row spanning entire rows in T1. The query we used for this test was:
`SELECT count(*) FROM (t2 NORMALIZE t1 using()) normalias;`
. *The number of rows containing NOW in this case were zero.* This tested a worst case scenario for the algorithm where it had to go through each entry creating multiple stack entries in the executor function. The time spent for different number of rows is shown in the table below followed by its graph. The result is as following:

Number of rows	Execution time in ms
100	21.393
200	60.023
400	211.672
800	865.637
1600	3599.63
3200	14733.274



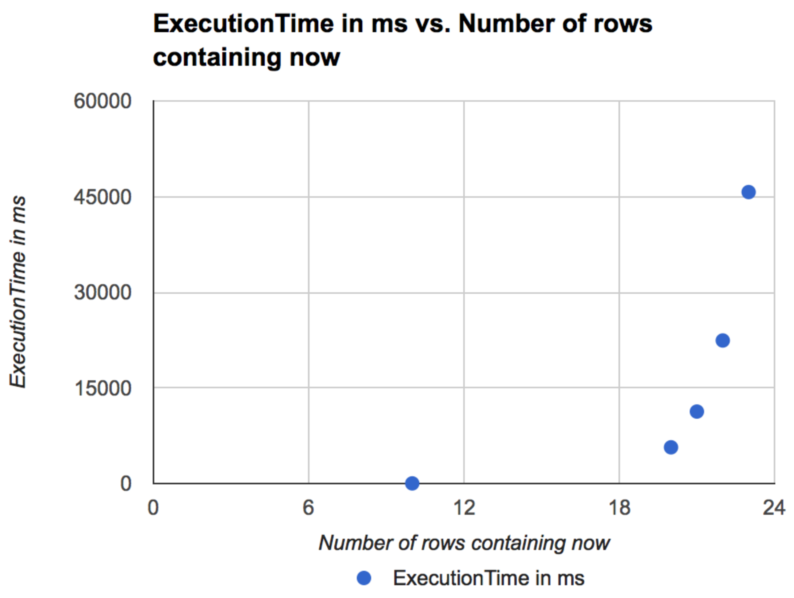
Next in the performance evaluation is a self normalization case. We test the query `SELECT count(*) FROM (t1 ts NORMALIZE t1 te using()) normalias;`. The query is timed against a relation containing different number of tuples as seen in the table below to find out the relation between execution time and size of the table. The server ran out of memory at 625 rows in relation, crashing the postgres server process:

Number of rows	Execution time in ms
39	75
78	384
156	3288
312	34517
625	Memory overFlow



Next in the performance evaluation is a case containing rows with *NOW*. The query is:
`SELECT count(*) FROM (t2 NORMALIZE t1 using()) normalias;`
It is run against following number of tuples in T1. Every additional tuple containing *now* doubles the execution time and the server eventually crashes at 25 tuples. :

Number of rows containing now	Execution time in ms
10	22.76
20	5681
21	11289
22	22433
23	45710
25	memory overflow



7 Conclusion

This project was a great introduction for us to get into postgresql development. We went through the most structural components of the kernel. We learned the following basic tasks when it comes to postgresql kernel development:

- Basic workflow of how a client-server architecture based dbms works. How to set up development environment for such an environment and debugging individual queries through the development process.
- Adding new keywords to the grammar
- Parsing a query and how to deal with particular keywords
- Extending existing data types in the kernel
- Working with temporal primitives

- Executing a query by writing a custom executor function
- How an adjustment state is used to operate on intermediate tuples and how to effectively modify those tuples to produce custom results.

We did not implement alignment in the executor with the `Daterange` type. This implementation can be further improved using better way to execute the normalize query. Currently the performance of normalize query is not adequate for practical usage in production environments, but it serves a good proof of concept of using normalize to query data with time domain. It shows that queries can be simplified with easy to use predicates.

References

[1] Yvonne Mülle, Generally valid queries on databases with ongoing timestamps, Working paper.

[2] `http://www.ifi.uzh.ch/dbtg/research/align.html`

[3] PostgreSQL development environment: *postgresql in eclipse*. `https://wiki.postgresql.org/wiki/Working_with_Eclipse`

[4] Clifford, James and Dyreson, Curtis and Isakowitz, Toma s and Jensen, Christian S. and Snodgrass,RichardThomas. On the Semantics of NOW in Databases. *ACMTrans.Data- base Syst.*, 1997

[5] A.Dignös, M.Böhlen, and J. Gamper. Temporal alignment. *In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 433444. ACM, 2012.