Department of Informatics, University of Zürich

BSc Thesis

Integration of Ongoing Time Points into PostgreSQL

Lukas Yu

Matrikelnummer: 14-720-866

Email: lukas.yu@uzh.ch

September 28, 2017

supervised by Prof. Dr. Michael Böhlen and Yvonne Mülle

University of Zurich[UZH]

Department of Informatics

Acknowledgements

I would first like to thank my supervisor Yvonne Mülle for her continuous support and great patience throughout my thesis work. I am also grateful to Prof. Dr. Michael Böhlen, head of the Database Technology Group for giving the opportunity to work on this thesis. Finally, very special thanks to my family and friends for all of the moral support.

Abstract

Ongoing dates, such as *now*, are nowadays widely used in many temporal databases. However, the ever nature of ongoing dates with passing time have a few implications. The instantiated value of the date, the validity of predicates, the result of functions and queries involving ongoing dates changes with passing time. To avoid this problem, the conventional *bind approach* instantiates all ongoing dates to a given reference date before further processing. This implies that any query result of the bind approach is only valid for that specific reference date. A newly proposed *ongoing approach* tries to remedy this issue by leaving any ongoing dates *uninitialized* during the process and generating a result that takes any reference date into consideration. A result for a specific reference date can be retrieved from this result with minimal computational effort and without re-evaluating the query, resulting in a drastic reduction in execution runtime. In this thesis, this novel approach is implemented into the kernel of the popular open-source database system *PostgreSQL* and evaluated against the conventional bind approach. With our novel ongoing approach we see a similar runtime compared to the *bind approach* for results at a single reference time. However, for every *subsequent* request at a different reference date, the ongoing approach can retrieve results from a previously *cached result* for almost no computational effort, making it greatly favorable when queries are evaluated at multiple reference dates.

Zusammenfassung

Fortlaufende Kalenderdaten wie *now* sind heutzutage weitgehend in temporale Datenbanken gebräuchlich. Weil jedoch solche Daten sich mit der Zeit ändern, bringen sie etliche Probleme mit sich mit. Der instanziierte Wert, die Gültigkeit Resultate von Prädikaten, Funktionen und Queries mit fortlaufende Daten verändern sich mit der Zeit. Um dieses Problem zu umgehen, instanziiert das konventionelle *Bind Approach* alle fortlaufende Daten zu einem gegebenen Bezugsdatum, bevor es weiter verarbeitet wird. Jedoch sind alle Resultate dieser Methode nur gültig für das Bezugsdatum. Ein neulich vorgestelltes *Ongoing Approach* umgeht dieses Problem, indem es die fortlaufenden Daten *uninstanziiert* lässt. Die neue Methode generiert Resultate, die alle Bezugsdaten in Betracht zieht und demzufolge auch für alle Bezugsdaten gültig ist. Resultate für ein spezifisches Bezugsdatum können mit minimalem Rechenaufwand aus diesem Zwischenresultat abgerufen werden. Die Query muss in diesem Fall nicht neu evaluiert werden, was in einer drastischen Reduktion der Laufzeit resultiert. In dieser Thesis wird diese neue Methode in den Kernel von *PostgreSQL*, ein populärer open-source Datenbanksystem, implementiert. Folglich wird es evaluiert und mit dem konventionellen *Bind Approach* verglichen. Mit der neuen Methode ist die Laufzeit vergleichbar mit dem *Bind Approach*. Jedoch kostet jede *nachfolgende* Abfrage mit dem *Ongoing Approach* zu einem unterschiedlichen Bezugsdatum kaum Rechenaufwand, was es zur bevorzugten Methode macht, falls eine Query zu mehreren Bezugsdaten abgefragt wird.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

Ongoing dates, such as *now*, are widely used in relations that include valid times. The most common form of such valid time attributes are date intervals with fixed dates as the start date and an ongoing date *now* as the end date that describes a tuple that has a known start point and an end date not known yet. Such tuples are therefore valid from the given start date to the current date, changing depending when the tuple is retrieved.

Clifford et al. [2] proposes a *now* date that is only instantiated by the time it is accessed. The point of such *ongoing dates* is that its instantiated value changes as time passes by to allow dates such as *now*. The ongoing date *now* instantiated at the reference date 2015-10-15 will return 2015-10-15, while instantiated at reference time 2017-08-11 will return 2017-08-11. While ongoing dates are useful to represent such dates, they can not be easily processed. To use them in queries with the conventional method, the *bind approach* proposed by *Anselma et al.* [1], they first have to be instantiated into fixed dates with a reference date. However, that means that the results are only valid at that specific reference date. It could be possible that the results are completely wrong for any other reference date. A newly proposed solution by *Y. Mülle et al.* [3] addresses this issue by leaving ongoing dates *uninitialized* and produces a result that is independent of reference dates. These results can be cached and instantiated at any desired reference date, even multiple times, without re-evaluating the query. In this thesis, this approach is referred as the *ongoing approach*.

To exemplify both the *ongoing* and the *bind approach* and illustrate the difference between them, consider a relation describing software bugs and keeping track of their status. Table 1.1 serves as an example with two entries. Its attributes are an *id* to uniquely identify the bug, a *description* and a date range as *valid time* to denote when the bug was recorded and resolved. The first entry **B1** describes a bug that was discovered 2015-01-01 and closed 2015-03-02. The second entry **B2** has been recorded on 2014-01-01 and has not been fixed yet, hence a *now* as its end date.

Bug ID	Description	VT
B1	Login broken	[2015-01-01, 2015-03-02)
B2	Crashes every 5 minutes	[2014-01-01, *now*)

Table 1.1: Example relation

As an example query, we want to know if the valid time interval of **B1** overlaps with **B2** at the reference dates 2014-10-14 and 2015-08-20. We start with joining the table 1.1 with itself and choose the tuple where **B1** and **B2** are joined together.

To achieve the goal with the conventional *bind approach*, any ongoing date involved has to be first *instantiated* at the reference date 2014-10-14. In this case, the valid time of **B2** becomes [2014-01-01, 2014-10-14). With only fixed dates in the valid time intervals, the regular *overlaps predicate* can be used to determine if the two date ranges overlap with each other, resulting in *false* in our example. For the second reference date 2015-08-20, the same procedure must be done again with the new reference date.

With the *ongoing approach*, the ongoing date *now* in bug B2 is left *uninstantiated*. To process ongoing dates, a special *generally valid overlaps predicate* is necessary. This new overlaps predicate takes two date ranges with ongoing dates as inputs and produces a result that holds information on whether they overlap or not at any reference date. The result is then instantiated at the first reference date 2014-10-14 to *false*. The same result is then instantiated at the other reference date 2015-08-20 to *true* without having to re-evaluate the query.

The main difference between the *bind* and the *ongoing approach* is that the result of the bind approach is only valid at a specific reference date. For any other reference date the query must be re-evaluated from scratch. On the other hand, the ongoing approach evaluates the query at any reference date, making its result valid regardless of reference date. The final result of multiple reference dates can be easily retrieved from a cached result of the ongoing approach for almost no computational cost.

In this thesis, an implementation of the new *ongoing approach* is presented. First, two new data types are introduced: the *ongoing date* for holding data about ongoing dates and the *ongoing boolean* to store the intermediate results of the ongoing approach. Then, *ongoing predicates* and *generally valid logical connectives* are added to make queries following the ongoing approach possibles. The implementation is then evaluated and compared against the bind approach. The results will be analyzed and used to determine if the new ongoing approach is applicable in real-world scenarios.

1.2 Thesis Outline

The remainder of this thesis is structured as follows. First, in chapter 2, the theory behind the ongoing approach, namely two data types and their respective functions, is provided as a foundation required to fully understand the implementation. Chapter 3 explains the implementation of the ongoing approach into the PostgreSQL kernel in detail, which is then evaluated against the bind approach in the chapter 4. In the chapter 5, the thesis is summarized with a short conclusion and future potential of the new approach is discussed.

2 Ongoing Data Types

In this chapter, the theory behind the *ongoing approach* to query ongoing dates is explained in detail to provide the necessary knowledge to understand the remainder of the thesis. Furthermore, relevant terms of the ongoing approach used throughout in this paper are defined and explained and exemplified with an application scenario.

2.1 Ongoing Date

An ongoing date is a date variable consisting of two dates, notated as a+b with a and b as separate fixed dates. It can be instantiated to a fixed date with a reference date. The instantiation of an ongoing date at a reference date can be defined with three cases:

- If the reference date falls between a and b, the ongoing date instantiates to the value of the reference date.
- If the reference date is earlier than or equal to a, it instantiates to a.
- If the reference date is later than or equal to b, it instantiates to b.

As an example, the ongoing date 2010–01–01+2020–01–01 with a reference date 2015–01–01 will instantiate to to the same date as the reference date, as it falls between the dates a and b. With a reference date of 2005–01–1, it will instantiate to the date a, 2010–01-01, as the reference is earlier than a.

Any ongoing date can be categorized into five types listed in the table 2.1. The notation of every type except for *capped* ongoing dates can be shortened. The short names of any ongoing dates will be primarily used throughout the thesis.

Type	Fixed	Now	Growing	Limited	Capped
Notation	a+a	$-\infty$+∞	a+∞	$-\infty$+b	a+b
Short form	a	*now*	a+	+b	a+b
Example	*2010-01-01*	*now*	*2010-01-01+*	*+2010-01-01*	*2010-01-01+2020-01-01*

Table 2.1: Ongoing Date Types

2.2 Ongoing Boolean

The ongoing approach evaluates predicates with ongoing dates at any reference date and therefore produces a result that can change with different reference dates. To actually store these results, a simple boolean value is not enough. The ongoing boolean is introduced as a new data type to represent results that either instantiates to *true* or *false* at a given reference date.

It is represented as a set of fixed date ranges to signify date intervals of reference dates at which it instantiates to *true*. As an example, the *ongoing boolean* **b**[[2010-01-01, 2011-01-01)] instantiates to *true* at any reference date from 2010-01-01 until 2011-01-01 and to *false* for any other reference date.

2.3 Generally Valid Predicates

The *generally valid predicates* are special functions that evaluate ongoing dates with ongoing booleans as results. These special predicates are necessary for using the ongoing approach because it is not required to instantiate any ongoing dates before evaluating them, keeping the results valid for any reference date. For example, the ongoing *less-than predicate* evaluates the two ongoing dates 2015-01-01 and *now* into the ongoing boolean **b**[[2015-01-02, ∞)]. The result of this ongoing less-than predicate tells that the fixed date 2015-01-01 is less-than (or earlier than) the ongoing date *now* for any reference date from 2015-01-02 and later. While any predicate that can evaluate dates can be theoretically transformed into an ongoing equivalent, the focus in this thesis lies on the *less-than* and *overlaps* predicate.

2.4 Generally Valid Logical Connectors

Generally valid logical connectors like *AND*, *OR* and *NOT* for ongoing booleans are also different from logical connectors for simple booleans. Again, because the *instantiated value* of ongoing booleans can change depending on the reference date, results of generally valid logical connectors for ongoing booleans are ongoing booleans again. For example, the *generally valid conjunction* of the two ongoing booleans **b**[[2012-01-01, 2014-01-01)] and **b**[[2013-06-02, 2017-08-02)] would be another ongoing boolean **b**[[2013-06-02, 2014-01-01)].

These logical connectors for ongoing booleans are needed to combine generally valid predicate results to allow for more complex queries and to construct more generally valid predicates. The implementation of the *generally valid overlaps predicate* depends on the generally valid less-than predicate and the generally valid conjunction as an example.

3 Implementation

In order to use the ongoing approach, the *ongoing date*, the *ongoing boolean*, *generally-valid logical connectives* and an *overlaps predicate* must be implemented first. The widely used open-source database system *PostgreSQL* is used as a base for this implementation. The source code of PostgreSQL is written in the C programming language. Therefore any modification and addition are in the same language. In this section, the any changes made to the kernel are described in detail.

3.1 Ongoing Data Types

3.1.1 Ongoing Dates

```
typedef struct
{
    // a + b
    DateSingleADT ongoingLower; // a
    DateSingleADT ongoingUpper; // b
    bool isFixed;
} DateADT;
```

Listing 3.1: Implementation Ongoing Date

The built-in PostgreSQL *date* data type was only intended to hold fixed dates. The type "DateADT" was an alias for a 32-bit integer that stored the value of the fixed date. We extended the DateADT type to support ongoing dates in the form of a+b. We preserve a fixed date data type as the *DateSingleADT*. The former definition of "DateADT" is then replaced with a new ongoing date type struct, shown in listing 3.1. In addition to the two DateSingleADTs a and b contained in the DateADT struct, the bool *isFixed* indicates if the dates a and b are equal. Equal dates implies that it instantiates to the same date regardless at which reference date. This boolean value simplifies checks on whether it is fixed or not and is often used in algorithms.

The reason a new *date* data type with ongoing functionalities was not created entirely from scratch was the ability to use the built-in PostgreSQL *daterange* type, which stored two DateADTs as a range, without any large adjustments. With an implementation of a daterange type that also works with ongoing dates, date ranges with ongoing dates as range limits are possible. This advantage is used to store ongoing dateranges and also makes is possible to implement an generally valid overlaps predicate for dateranges.

The struct type is used to describe the five ongoing date types described in the previous chapter: *fixed*, *now*, *growing*, *limited*, and *capped*. The representation of each type as DateADT is illustrated in table 3.1.

Struct member	Fixed	Now	Growing	Limited	Capped
ongoingLower	2010-01-01	$-\infty$	2010-01-01	$-\infty$	2010-01-01
ongoingUpper	2010-01-01	∞	∞	2010-01-01	2018-01-01
isFixed	*true*	*false*	*false*	*false*	*false*

Table 3.1: DateADT examples

Along with the redefinition of the DateADT type to a struct, other changes were necessary to render the new ongoing date usable. To use ongoing dates in SQL statements, the **date_in()** and **date_out()** were adjusted. These functions are responsible for deserializing strings in SQL statements into ongoing date objects. For example, deserializing the string "2017-04-22+2018-06-04" to an instance of the DateADT struct and serialize the DateADT struct back to the string.
A trade-off decision was to include the *isFixed* boolean to the DateADT struct. Whether a DateADT is a fixed date or not can be implied from the two DateSingleADTs, so there is a small redundancy. This redundancy results in DateADTS requiring more memory space. However, the *isFixed* boolean is kept in the struct to simplify checks the fixed state, which is done in many algorithms. The benefit of reduction of the runtime was chosen over the benefit of reduction of memory space requirements.
Another decision was whether to extend the existing built-in *date* data type of PostgreSQL with ongoing features or to create a new data type separate from the built-in type. One advantage of the current implementation with the extension of the PostgreSQL date is that the built-in *daterange* data type, used to represent date intervals, can be used with the extended *date* type without any large modifications. The downside of this method is the loss of dateranges with non-ongoing dates. In the current implementation, ongoing booleans are persisted as daterange arrays, but only fixed dates are allowed as daterange bounds per definition of the ongoing boolean. Basically, the extended version of the *date* type is used for ongoing booleans when ordinary fixed dates would be enough. The consequence is greater memory usage for ongoing booleans and the necessity of checks on dates where only fixed dates are allowed. The method of extending the *date* data type was chosen over creating a separate ongoing date due to the simplicity of using the extended date for everything, as those can emulate simple dates just as well.

3.1.2 Ongoing Booleans

```
typedef struct OngoingBoolean
{
    DateSingleADT lower;
    DateSingleADT upper;
    struct OngoingBoolean *next;
} OngoingBoolean;
```

Listing 3.2: Implementation Ongoing Boolean

There are two different ways *ongoing booleans* are stored. The first one is the *OngoingBoolean* struct in C (see listing 3.2) and the other one a PostgreSQL *daterange array*. There is a reason why two seperate representation of the ongoing boolean are needed. While the OngoingBoolean struct is easy to modify and create, it can not be stored in the database. That is why the built-in *daterange array* data type is necessary to *persistently* save ongoing booleans.

The *OngoingBoolean* struct is implemented as a forward list of nodes, each node containing one date range represented with DateSingleADTs *lower* and *upper* as members of the struct. The *next* pointer points to the next OngoingBoolean node in the list or to *null* if the list ends. It is a series of non-overlapping date ranges ordered by time in ascending order. The reasoning behind these constraints is to allow the generally valid logical connectives for ongoing booleans to be implemented more efficiently, but those will be discussed later.

The list feature of the struct is crucial as the number of ranges is typically not known before the end of a logical connective algorithm. With a forward list like the *OngoingBoolean* struct, any additional node can be appended by setting the next pointer of the latest node to the node to be appended. The same can not be done with an array, as the memory required must be allocated beforehand and the size of the allocated memory can not be changed after.
To convert the OngoingBoolean struct into persistable daterange arrays and vice versa, two functions are defined: **deserialize_ongoingBoolean()** in algorithm 1 that transforms daterange arrays into OngoingBoolean structs, and **serialize_ongoingBoolean()** in algorithm 2 that reverses the process.

Algorithm 1: Deserialize ongoing booleans

Input : inputArray - Daterange[]
Output: outputBool - OngoingBoolean pointer

```
outputBool = null;
currentBool = null;
rangeArray = deserialize_array(inputArray);
foreach range_element ∈ rangeArray do
    bounds = range_deserialize(rangeElement);
    newNode = new OngoingBoolean(bounds.lower, bounds.upper);
    if outputBool = null then
        outputBool = newNode;
    else
        currentBool→next = newNode;
    currentBool = newNode;
end
return outputBool;
```

Algorithm 2: Serialize ongoing booleans

Input : inputBool - OngoingBoolean pointer
Output: outputArray - Daterange[]

```
if inputBool = null then return empty_array();
;
numElems = length(inputBool);
outputArray = new Array[numElems];
foreach i ∈ range(0, numElems) do
    lowerBound = create_dateADT(inputBool→lower, inputBool→lower);
    upperBound = create_dateADT(inputBool→upper, inputBool→upper);
    outputArray[i] = make_daterange(lowerBound, upperBound);
    ongoingBool = ongoingBool→next;
end
return outputArray;
```

3.2 Generally Valid Operations

3.2.1 Logical Connectives for Ongoing Booleans

We implemented the three logical connectives for ongoing booleans: *NOT*, *AND*, and *OR*. They are used to combine ongoing booleans to enable more complex queries. There are two functions defined for each connector, one being the *actual algorithm* that is only callable from within the C-code and the other being a function callable from the PostgreSQL client. The functions accessible through the PostgreSQL client via queries take PostgreSQL *daterange arrays* as arguments and are responsible for properly *serializing* and *deserializing* input arrays. The internal function in C will take those deserialized OngoingBooleans and calculate the results of logical connectives.

For example, the generally valid logical connective *NOT* can be called with **gv_logical_not(daterange[])** in a query. It *deserializes* the daterange array into OngoingBooleans, calls the *internal* function **gv_internal_logical_not(OngoingBoolean input)**, *serializes* the result back to a daterange array and returns it. This separation is done in order to keep algorithms for OngoingBooleans *chainable* without serializing to daterange arrays and deserializing to OngoingBooleans again in between function calls. An example of a use case of the ability to chain OngoingBoolean logical connectives is the overlaps predicate of the DateADT type. The conjunction of OngoingBooleans is used three times for one overlaps function call and it would create a massive computational overhead to *serialize* and *deserialize* between OngoingBooleans and daterange array. Another advantage of modularizing logical connectives and predicates is the improved readability and maintainability of the source code. Another example is illustrated in algorithm 3.

When using the generally valid logical connectives, is crucial that any input OngoingBoolean for the functions comply with the constraints of *non-overlapping* date ranges and *order* by ascending date. This is done to greatly simplify the algorithms and results in a lower runtime. However, as the algorithms are constructed with these limitations in mind, it will break with invalid ongoing booleans as input. The generally valid predicates and logical connectives for ongoing booleans in this implementation will always produce valid ongoing booleans if the inputs were also valid.

Algorithm 3: Separation of Serialization and Algorithm Logic

```
1  // Callable function from SQL statement
2  Function gv_logical_not(daterange[])
3      ongoingBoolInput = deserialize_ongoingBoolean(daterange[]);
4      ongoingBoolResult = gv_internal_logical_not(ongoingBoolInput);
5      daterangeResult = serialize_ongoingBoolean(ongoingBoolResult);
6      return daterangeResult;

7  // Internal algorithm without serialization/deserialization
8  Function gv_internal_logical_not(ongoingBoolInput)
9      [[ Not algorithm ]];
10     return negation;

11 // The separation between serialize/deserialize and algorithm allows to use algorithms
       back to back without de-/serializing.
12 Function double_negate(ongoingBooleInput)
13     negated = gv_internal_logical_not(ongoingBoolInput);
14     double_negated = gv_internal_logical_not(negated);
15     return double_negated;
```

Algorithm 4: Negation

Input : inputBool - OngoingBoolean pointer
Output: outputBool - OngoingBoolean pointer

```
1  outputBool = null;
2  currentBool = null;
3  if inputBool = null then return b[(MINVAL, MAXVAL)];
4  if inputBool → lower ≠ MINVAL then
5  |  newNode = b[(MINVAL, inputBool→lower)];
6  |  outputBool = newNode;
7  |  currentBool = newNode;
8  end
9  while inputBool ≠ null do
10 |  if inputBool → upper = MAXVAL then break ;
11 |  currentBool = new OngoingBoolean();
12 |  currentBool→lower = inputBool→lower;
13 |  inputBool = inputBool→next;
14 |  if inputBool ≠ null then
15 |  |  currentBool→upper = inputBool→lower;
16 |  else
17 |  |  currentBool→upper = MAXVAL;
18 |  end
19 |  if outputBool = null then
20 |  |  outputBool = newNode;
21 |  else
22 |  |  currentBool→next = newNode;
23 |  currentBool = newNode;
24 |  inputBool = inputBool→next;
25 end
26 return outputBool;
```

The first generally valid connective is a *logical negator* that *flips* an OngoingBoolean. In essence, it will produce an OngoingBoolean as a result that instantiates to *false* if the input OngoingBoolean instantiates to *true* and vice versa at the same reference date. The algorithm is illustrated in algorithm 4.

As an example to show the functionality of the negator algorithm, consider the ongoing boolean **b**[[2010-01-01, 2015-01-01), [2018-01-01, 2020-01-01)] as input parameter. The algorithm will check if the first date range starts at *MINVAL*, the minimum value of a date, at line 4. If this is not the case, like in the example, it will add a date range with *MINVAL* as start date and the start date of the first input date range as the end value to the result. In our example, it would be [*MINVAL*, 2010-01-01). After the initial check, it will iterate through the remaining input ranges and add an range for each gap between them in a while loop starting at line 9. In the example, an range [2015-01-01, 2018-01-01) is appended to our result array. At line

14, if the last date range of the input array does not end with *MAXVAL*, the maximum value of a date, it will add another one to the result with the ending date of the last input element as the start date and *MAXVAL* as ending date. In the case in the example, [2020-02-02, *MAXVAL*). The returned result is the ongoing boolean **b**[[$-\infty$, 2010-01-01), [2015-01-01, 2018-01-01), [2020-01-01, ∞)].

Algorithm 5: Conjunction algorithm

Input : input1, input2 - OngoingBooleans pointers
Output: outputBool - OngoingBoolean pointer

```
1  outputBool = null;
2  currentBool = null;
3  while input1 ≠ null ∧ input2 ≠ null do
4  |   while input1 → lower ≥ input2 → upper ∨ input2 → lower ≥ input1 → upper
   |    do
5  |   |   if input1 → lower ≥ input2 → upper then input2 = input2→next ;
6  |   |   else input1 = input1→next ;
7  |   |   if input1 = null ∨ input2 = null then return outputBool ;
8  |   end
9  |   newNode = new OngoingBoolean(
10 |       max(input1→lower, input2→lower),
11 |       min(input1→upper, input2→upper)
12 |   );
13 |   if outputBool = null then outputBool = newNode ;
14 |   else current→next = newNode ;
15 |   current = newNode;
16 |   if input2 → upper < input1 → upper then input2 = input2→next ;
17 |   else if input1 → upper < input2 → upper then input1 = input1→next ;
18 |   else
19 |   |   input1 = input1→next;
20 |   |   input2 = input2→next;
21 |   end
22 end
23 return outputBool;
```

The next logical connector is the *conjunction* of two ongoing booleans. It takes two ongoing booleans as input and calculates a new ongoing boolean that instantiates to *true* at reference dates where both input OngoingBooleans instantiate to *true*. The overlapping parts are added into a new OngoingBoolean forward list to be returned as the result. The logic behind this algorithm is taken from the paper presentig the *ongoing approach* [3]. However, it was modified and tested to fit in this implementation. The algorithm is illustrated in algorithm 5.
To exemplify the *conjunction* algorithm, consider the two ongoing booleans **b**[[2000-01-01,

2010-01-01)] and **b**[[2005-01-01, 2012-01-01)]. It will pass the first while loop in line 3, but fail the loop in line 4. Then, a new OngoingBoolean is created with an daterange set as the overlapping part of the two inputs in line 9 and its value is assigned to the outputBool pointer variable, which will be returned. The input2 pointer advances to the next node in line 13, in this case to *null*. At the end of the while loop, it will fail the condition in line 3, as input2 is now assigned to *null*. It breaks out of the loop and returns the outputBool variable, which contains the conjunction of the two input OngoingBooleans. The final results is the ongoing boolean **b**[[2005-01-01, 2010-01-01)].

Algorithm 6: Disjunction algorithm

Input : input1, input2 - OngoingBooleans pointer
Output: outputBool - OngoingBoolean pointer

```
1  outputBool = null;
2  currentBool = null;
3  while input1 ≠ null ∧ input2 ≠ null do
4      if input2 → lower < input1 → lower then swap(input1, input2) ;
5      newNode = new OngoinBoolean();
6      newNode→lower = input1→lower;
7      while input1 ≠ null ∧ input2 ≠ null] do
8          if input2 → lower ≤ input1 → upper then
9              if input2 → upper > input1 → upper then
10                 input1 = input1→next;
11                 swap(input1, input2);
12             else
13                 input2 = input2→next;
14             end
15             if input2 = null then
16                 newNode→upper = input1→upper;
17                 input1 = input1→next;
18             end
19         else
20             newNode→upper = input1→upper;
21             input1 = input1→next;
22             break;
23         end
24     end
25     if outputBool = null then
26         outputBool = newNode;
27     else
28         currentBool→next = newNode;
29     currentBool = newNode;
30 end
31 while input1 ≠ null ∨ input2 ≠ null do
32     if input1 ≠ null then
33         currentBool→next = input1;
34     else
35         currentBool→next = input2;
36     currentBool = currentBool→next;
37 end
38 return output;
```

The last algorithm for OngoingBooleans is the *disjunction* of two OngoingBooleans illustrated in algorithm 6. It takes two OngoingBooleans as input. The algorithm works with two pointers, each pointing to the first element of the two input OngoingBoolean lists. The first while loop repeats if both input lists are not at the end yet. Inside the loop, it checks if the OngoingBoolean node the input1 points to starts before or at the same date as the one input2 points to. If this is not the case, the two pointers get swapped to make sure the OngoingBoolean of input2 does not start earlier. From here, the start point of the a new OngoingBoolean node is set to the lower value of input1. The algorithm then enters the inner while loop, in which the input1 is compared to the other set. Depending on the position of the daterange from input2, different actions are taken:

- **Line 9**
  If the daterange of input2 is fully contained in the daterange of input1, the pointer of input2 is set to the next value in the list, as the current value does not change the result. The inner loop is repeated.

- **Line 12**
  If the daterange of input2 is starting inside the daterange of input1 or daterange input2 is extending daterange input1 respectively, the input1 pointer is set to the next element in the list and the pointers are swapped again. The inner loop is repeated.

- **Line 15**
  If input2 is pointing to null, the end of the list is reached and the upper value of the current result node is set to to the upper value of the input1 daterange. The input1 pointer is set to the next element in the list. It will break out of the inner while loop, as one of the pointers is set to null.

- **Line 19**
  If the daterange of input2 is fully outside or has a start point later than the end point of the daterange of input1 respectively, the end point of the current result node is set to the upper value of the input1 daterange. The input1 pointer is set to the next element in the list and the inner while loop is repeated.

Once outside the inner loop at line 24, the current result node is appended to the result OngoingBoolean list. The outer loop is repeated until at least one of the input lists is at the end. The last while loop at line 27 appends any leftover dateranges to the result list that may exist in one of the input lists.

To exemplify the disjunction algorithm, consider the two ongoing booleans **b**[[2009-01-01, 2016-01-01)] as input1 and **b**[[2008-01-01, 2010-01-01), [2018-01-01, 2020-01-01)] as input2. Inside the first while loop at line 3, it will make sure that the daterange of input1 starts earlier than the daterange of input2, swapping both pointers if it is not the case. In our example, input1 points at the daterange [2009-01-01, 2016-01-01), which starts later than input2 [2008-01-01, 2010-01-01). Thus both pointers are swapped including any following dateranges. If the daterange of input1 starts first, it is certain that the new daterange of the result starts at the

same time as the daterange of input1. In line 6, the lower date for the new OngoingBoolean is set. It will then enter the inner loop at line 7, where the position of input2 daterange is compared to the position of input1 daterange. The input2 daterange [2009-01-01, 2016-01-01) starts inside the input1 daterange [2008-01-01, 2010-01-01), but extends after the end of input1, so it will enter the if case in line 9. In there, the next daterange of input1 is assigned to it and both inputs are swapped again in order to ensure that input1 starts first. From there, the inner loop at line 7 is repeated with swapped inputs. Now, the input2 daterange [2018-01-01, 2020-01-01) is completely outside of the input1 daterange [2009-01-01, 2016-01-01). In this case, the algorithm will go to line 19, where the end date of the new OngoingBoolean node is set to the end date of input1. It will then advance input1 to the next daterange and break out of the inner loop at line 7 and append the new node to the result. Because input1 now points to null, the outer loop at line 3 is exited. At the end in the while loop in line 31, the last daterange left is in input2, which will be copied and appended to the result as it is. The final returned result is the ongoing boolean **b**[[2008-01-01, 2016-01-01), [2018-01-01, 2020-01-01)].

3.2.2 Less-than Predicate

The ordinary less-than predicate for fixed dates can not be used on ongoing dates, as they are only intended to work with fixed dates and return an fixed boolean. A *generally-valid less-than predicate* has to be implemented in order to be able to process ongoing dates. This new generally-valid predicate takes two ongoing dates as input and computes an ongoing boolean, showing for which reference dates the two ongoing dates would overlap with each other.
The algorithm for the generally-valid less-than predicate $a+b < c+d$ can be broken down to five cases illustrated in algorithm 7.

Algorithm 7: Implementation less-than predicate

Input : Two ongoing dates: $a+b$, $c+d$
Output: output: OngoingBoolean
1 **if** $b < d \wedge b < c$ **then return b**[$(-\infty, \infty)$] ;
2 **if** $a < c \leq d \leq b$ **then return b**[$(-\infty$, c)] ;
3 **if** $c \leq a \leq b < d$ **then return b**[[b+1, $\infty)$] ;
4 **if** $a < c \leq b < d$ **then return b**[$(-\infty$, c), [b+1, $\infty)$] ;
5 **else return b**[] ;

This predicate can be used with the SQL function **gv_date_lt(date1, date2)**. For instance, the predicate **gv_date_lt('2000-01-01+2010-01-01', '2005-01-01+2020-01-01')** goes into line 4 and evaluates to the ongoing boolean **b**[$(-\infty$, 2005-01-01), [2010-01-02, $\infty)$]

3.2.3 Overlaps Predicate

A generally valid *overlaps* predicate for ongoing dates is also implemented. It builts upon the previously defined generally valid *less-than* overlaps and the generally valid logical *conjunction* connector. The implementation of the generally valid *overlaps* predicate is illustrated in algorithm 8.

Algorithm 8: Implementation less-than predicate

Input : Two dateranges: range1, range2
Output: OngoingBoolean
1 **return** range1.start $<$ range2.end $\wedge$ range2.end $<$ range1.start $\wedge$ range1.start $<$ range1.end $\wedge$ range2.start $<$ range2.end;

This predicate can be used with the SQL function **gv_date_overlaps(daterange1, daterange2)**. For instance, the predicate **gv_date_overlaps(daterange(2000-01-01, *now*), daterange(2005-01-01, 2020-01-01)** evaluates to the ongoing boolean **b**[[2005-01-02, ∞)]

3.2.4 Bind Operator

Another addition to the new DateADT type are the **date_bind()** and the **daterange_bind()** functions. These can be called from a PostgreSQL client via a SQL query and is responsible to *instantiate* an Ongoing Date at a reference date. For example, the expression **date_bind(2010-01-01+, 2015-01-01)**, where the first argument is the ongoing Date and the second date is the reference date to instantiate the ongoing date at, would evaluate to 2015-01-01. The **daterange_bind()** function instantiates the two dates inside a given daterange to a reference date similar to **date_bind()**. These functions were implemented to allow us to write queries following the *bind approach*, where the ongoing date variables are instantiated to a specific reference date before it is further processed.
The same is implemented for ongoing booleans with the **bind_ongoing_boolean()** function. It takes an ongoing boolean and a reference date as arguments and *instantiates* the ongoing boolean to the reference date, resulting in either *true* or *false*. For example, in a SQL statement, the function **call bind_ongoing_boolean([[2008-01-01, 2014-01-01)], '2010-01-01')** evaluates to *true*, whereas **bind_ongoing_boolean([[2008-01-01, 2014-01-01)], '2010-01-01')** evaluates to *false*.
The **date_bind()** is necessary for the implementation of the bind approach, as the ongoing dates must be *instantiated* before the query evaluation. The **bind_ongoing_boolean()** is necessary for the ongoing approach to instantiate the results at given reference times into fixed booleans.

3.3 PostgreSQL Kernel Adjustments

In addition to all the changes described above, some smaller changes had to be done in order to render the implementation useful. The first change is to the file *pg_type.h*. In this file, data types are declared that can be used in PostgreSQL. The only line that required a change was the *date* data type declaration because it was extended with ongoing features. The number that depicts the size of the data type was raised from 4 bytes, a single 4-byte integer, to 9 bytes, two 4-byte integers and one boolean. The new OngoingBoolean type does not need its own declaration because it is only a date struct used internally in C-code to store and modify data from daterange arrays.

To make C-functions callable within SQL queries, they have to be declared in the *pg_proc.h* file. Each function is declared with a unique *OID*. The OID number is an identifier for objects in Postgres like data types or functions for example. Every object has to be assigned to a unique OID, regardless of the object type. For the newly added lines, free OID numbers from the 6000 to 6999 are used to avoid any OID duplicates with built-in objects. A sample PostgreSQL function declaration is illustrated in table 3.2.

Parameter	**Value**	**Description**
OID	6001	An arbitrary, but unique OID
Internal name	gv_date_lt	Name of the internal C-function
External name	gv_date_lt	SQL function name
Input types	1082, 1082	Two ongoing dates
Return type	3913	Ongoing boolean

Table 3.2: Sample declaration of a PostgreSQL function

4 Evaluation

In this section, we evaluate the *ongoing approach* on real-world and synthetic datasets. We compare its execution *runtime* and *result size* with the ones of the *bind approach*. We use self-joins whose predicate contains the overlaps predicate as queries. With a wide range of different datasets, it can be easily observed how both approaches behave in different circumstances. In addition to a comparison between approaches, different parts of the execution of the generally valid *overlaps* and their impact on runtime is evaluated and analyzed. The overlaps predicate for ongoing dates can be broken down into generally valid less-than predicates and generally valid conjunctions for ongoing booleans. This test is done to compare how different parts of the overlaps predicate affect the overall runtime.
The evaluation results show that the ongoing approach execution runtime is never significantly worse than the bind approach for results at a single reference date. With the *End set*, the runtime of the ongoing approach was even consistently shorter across all instantiation dates and dataset sizes. In terms of results size, the result size of the ongoing approach was always the same as the worst case of the bind approach in the test queries.

4.1 Setup

4.1.1 Datasets

To evaluate the ongoing approach, it was tested with both data from a *real-world* application and from synthetically created data. All datasets contain date ranges with a fixed date as start point and either a fixed date or an ongoing growing date as end point. This format of date ranges is used because real-world data usually follow this pattern of a known fixed start point and an end point that is not known. It is also favorable for the evaluation, as growing dateranges overlap with every subsequent non-empty daterange, increasing execution time and result size, magnifying differences between approaches for a more discernible comparison.

Real World Data

The *real-world* data set is a public dataset of bug reports from the Eclipse project [4]. The data, initially formatted in a XML schema, is converted into a csv table with the following schema:

(id integer, product text, component text, vt daterange)

Each tuple has a unique integer *id*, a *product* text attribute to show for which product the bug was logged, a *component* text attribute to show for which part of the product it was registered and a *valid time* daterange which signifies the date interval the bug was open.

In total, this dataset contains 165547 tuples. To test the ongoing approach with different *set sizes*, the original set was cut down into smaller pieces. The first *10k*, *20k*, ... , *90k* tuples of the original dataset ordered by the start point of vt in descending order are saved as smaller datasets in order to include only the most recent tuples. With increasing dataset size, the time range in which dateranges exists becomes larger, but the density of ranges stays approximately the same. With this method, we can simulate datasets with different length of histories and ultimately show the consequence of datasets that have been collecting data for a longer time. It should also be noted that *growing dateranges* (unresolved bugs) tend to be at the end of the history, as it is more probable that a bug has been closed over a longer period of time. This dataset was chosen because it contained real-world data, making evaluation results more relevant to real-world applications.

Synthetic Data

The *synthetic* data was created with a C program and has the following schema:

(id integer, vt daterange)

Starting points of tuples from the synthetic datasets fall within the range [2010-01-01, 2020-01-01]. The end point of fixed date ranges is set to 183 days (6 months) after the starting point, while ongoing tuples have a growing ongoing date as end point. All sets consist of 20% ongoing tuples, while the rest are fixed ones.

There are 3 types of synthetic datasets: *Start*, *End* and *Random*. For the *Start set*, all ongoing tuples start in the first 20% of the range while the fixed ones populate the remaining 80%. In the *End set*, all ongoing tuples start in the ending 20% of the range and every fixed tuple start before that. In the *Random set* all tuples, growing or fixed, are randomly distributed within the range. As with the *Eclipse set*, there are also various sizes with synthetic data sets. For each type of synthetic datasets, multiple sets are created with sizes *10k*, *15k*, ... , *35k*. However, note that unlike the aforementioned Eclipse dataset, the range where tuples can start stays the same for synthetic datasets, effectively increasing the density of dateranges. This should show a contrast between adding tuples by extending the history like the eclipse dataset and adding tuples by increasing the density in the same date range.

4.2 Methods

The two approaches are evaluated based on two criteria: *execution runtime* and *number of rows* in the result. The test queries are always based on self-joining the datasets and filtering by checking if any joined dateranges overlap with another.

4.2.1 Instantiation Dates for Bind Approach

The *bind approach* requires to instantiate the ongoing dates at a reference time before the query is evaluated. To observe the impact of different *instantiation date* on the two criteria,

multiple queries with different instantiation dates are evaluated. No instantiation dates are required in order to execute the query for the ongoing approach, due to the result that remains valid for all reference dates. The instantiation dates used for each set are listed in table 4.1.

Eclipse bugs	Start set	Random set	End set
2001-06-01	2009-01-01	2009-01-01	2010-01-01
2002-06-01	2010-01-01	2010-01-01	2018-01-01
2003-06-01	2011-01-01	2011-01-01	2018-06-01
2004-06-01	2012-01-01	2012-01-01	2019-01-01
2005-06-01	2013-01-01	2013-01-01	2019-06-01
2006-06-01	2014-01-01	2014-01-01	2020-01-01
2007-06-01	2015-01-01	2015-01-01	
2008-06-01	2016-01-01	2016-01-01	
2009-06-01	2017-01-01	2017-01-01	
2010-06-01	2018-01-01	2018-01-01	
2011-06-01	2019-01-01	2019-01-01	
2012-06-01	2020-01-01	2020-01-01	

Table 4.1: Binding dates of tests

The *instantiation dates* are chosen to cover the whole range of the dataset. To achieve this, one instantiation date must be earlier or equal the minimum start date and another instantiation date must be later or equal the maximum end date of any fixed tuple in the dataset. Any other instantiation date outside of the range between those two dates would not yield any different results for the generally valid overlaps predicate. To observe the behavior with instantiation dates in between the range, additional dates are added for each year except for the End dataset. For the *Eclipse set*, the ongoing dates are instantiated at every year from 2001-06-01 to 2012-06-01. These dates are mid year because the maximum date of the tuples of the eclipse dataset only goes up to 2011-05-08. For the *Start* and *Random set*, the ongoing dates are also instantiated at every year, but from 2010-01-01 to 2020-01-01. And finally for the *End set*, the ongoing dates only get instantiated at the very start in 2010-01-01 and in the ending 20% of the range. Any instantiation date before the ending 20% of the range would yield the same results for the End set, as all ongoing tuples would instantiate to empty ranges.

4.2.2 Queries

This section describes the *SQL queries* with which we evaluate our implementation. In total, 425 test queries are executed to account for every combination of dataset, size, query approach type (ongoing or bind), and instantiation dates for the bind approach. In order not to write out every one of the queries, they contain placeholders to shorten the list down to four queries.

Queries for Synthetic Sets

The *#dataset* placeholders are replaced with the dataset names of *Start*, *End* or *Random* set with each one having 6 different sizes as described in the setup section. The *#instdate*

placeholders for the query of the bind approach are replaced with the fixed reference dates stated in table 4.1.

The query of the ongoing approach (see listing 4.1) performs a self-join with the specified dataset and calls the overlaps predicate function **gv_date_overlaps()** for each joined tuple. The return value of the function is a daterange array, the serialized interpretation of an ongoing boolean. The first join condition filters out any joined tuples with the same **id** with trivial overlaps results. The second condition counts the elements in the result array and discards the result row if the array is emptyl. An empty ongoing boolean as a result means that there does not exist a reference date at which the two dateranges overlap. The **cardinality()** function takes care of counting the elements in an array.

```
SELECT r.vt, s.vt, gv_date_overlaps(r.vt, s.vt) AS rt
FROM #dataset r
JOIN #dataset s
ON r.id != s.id
AND cardinality(gv_date_overlaps(r.vt, s.vt)) > 0
```

Listing 4.1: Ongoing approach for synthetic sets

The query of the bind approach (see listing 4.2) binds all ongoing dates at a given reference date before the dataset is joined with itself. It ensures that the joined table only consist of date ranges with fixed dates. Without any ongoing dates, the operator for ordinary overlaps predicate for fixed dateranges (&&) is used to filter out non-overlapping tuples.

```
SELECT r.vt, s.vt
FROM (
    SELECT id, daterange_bind(vt, #instdate) as vt
    FROM #dataset
) r
JOIN (
    SELECT id, daterange_bind(vt, #instdate) as vt
    FROM #dataset
) s
ON r.id != s.id
AND r.vt && s.vt
```

Listing 4.2: Bind approach for synthetic sets

Query for Eclipse Set

The *#eclipseset* placeholders of the queries in listings 4.3 and 4.4 are to be replaced with the names of the *Eclipse set* for each size. The *#binddate* placeholders for the query of the bind approach are to be replaced with the reference dates listed in table 4.1.

The only difference between the queries for the synthetic datasets and for the Eclipse set are the additional join conditions for the product and component attribute of the Eclipse queries. The reason behind the product and component matching is to mimic a realistic use case.

```
SELECT r.vt, s.vt, gv_date_overlaps(r.vt, s.vt) AS rt
FROM #eclipseset r
JOIN #eclipseset s
ON r.id != s.id
AND r.product = s.product
AND r.component = s.component
AND cardinality(gv_date_overlaps(r.vt, s.vt)) > 0
```

Listing 4.3: Ongoing approach for Eclipse set

```
SELECT r.vt, s.vt
FROM (
    SELECT id, daterange_bind(vt, #bind_date) as vt
    FROM #eclipseset
) r
JOIN (
    SELECT id, daterange_bind(vt, #bind_date) as vt
    FROM #eclipseset
) s
ON r.id != s.id
AND r.product = s.product
AND r.component = s.component
AND r.vt && s.vt
```

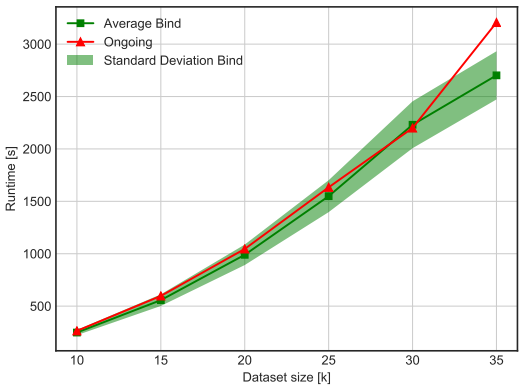
Listing 4.4: Bind approach for Eclipse set

4.3 Results

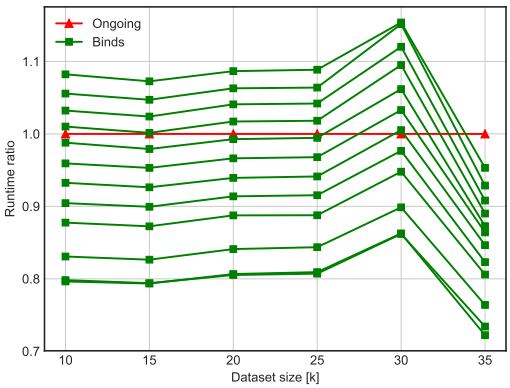
In this section the evaluation results are discussed. For the results, we had some expectations on what the test results show. First, for the result size, we expect that different instantiation dates of the bind approach yield different result sizes. On the basis of the fact that all datasets only contain fixed and growing tuples, it is certain that with a growing reference date, the number of overlapping tuples can not become smaller. With our instantiation dates illustrated in table 4.1, we expected result sizes to grow with later instantiation dates. Along with increasing result sizes, the execution runtime should also increase.

For each dataset, the results are shown in four graphs. The runtime and result size each having two graphs. One showing the absolute values of the measurements and the other showing the ratio of the results of the binding approach compared to the ongoing approach. To prevent the graphs with the absolute values from being illegible, they show the binding approach values with an average line with an area that show the standard deviation from the average. The graphs with the ratios show all results at different instantiation dates with a separate line. The bind approach lines are not labeled with their respective instantiation dates as they provide no useful or additional insight. The later the instantiation date, the longer the execution time and the greater the result size will be for any result of the ongoing approach.

4.3.1 Start set

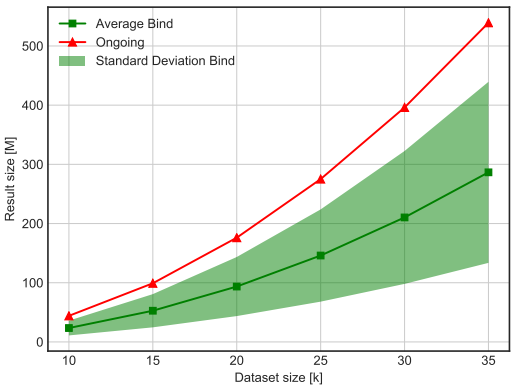


(a) Absolute runtime

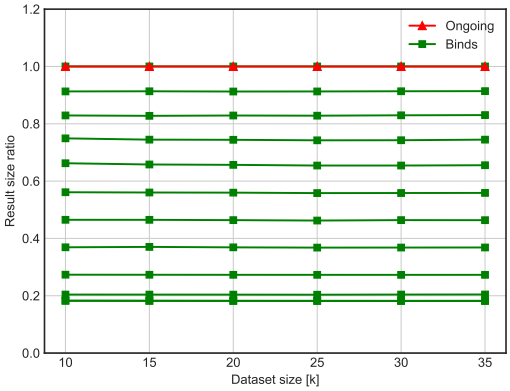


(b) Relative runtime

Figure 4.1: Start set runtime



(a) Absolute result size

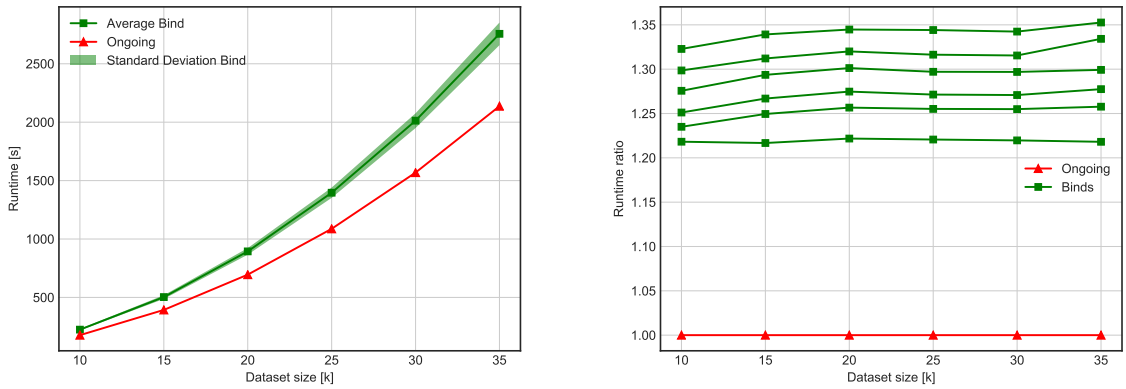


(b) Relative result size

Figure 4.2: Start set result size

From the results from figure 4.2b we can see that the result size of the ongoing approach is always the worst case of the bind approach. This is expected, as the result of the ongoing approach holds information about overlaps for all reference dates. As with the runtime, figure 4.1b shows that the execution runtime of the ongoing approach tend to get worse with growing numbers. At the size of *35k* tuples, the ongoing approach is worse than all instiation dates of the bind approach. This can be explained with the ongoing growing tuples at the start of the range, as they will overlap with every following tuples if evaluated with the ongoing approach. The bind approach on the other hand will instantiate any growing dateranges to a fixed daterange, therefore limiting overlaps.

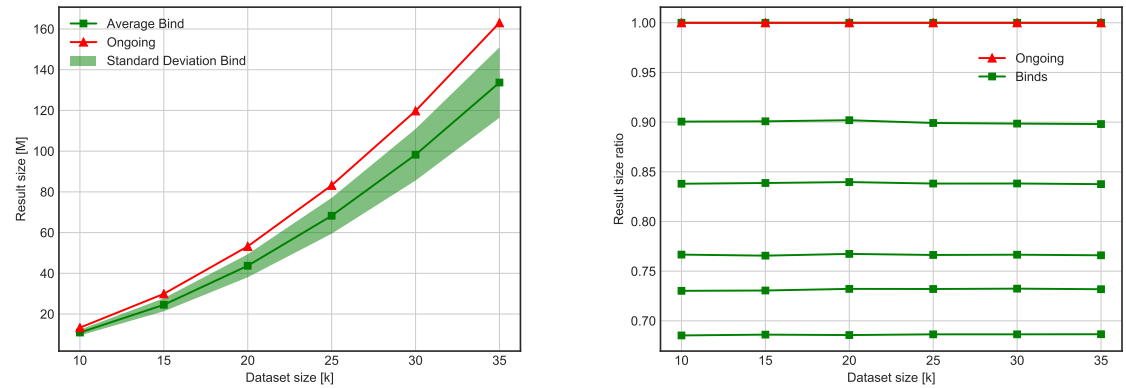
4.3.2 End set



(a) Absolute runtime

(b) Relative runtime

Figure 4.3: End set runtime



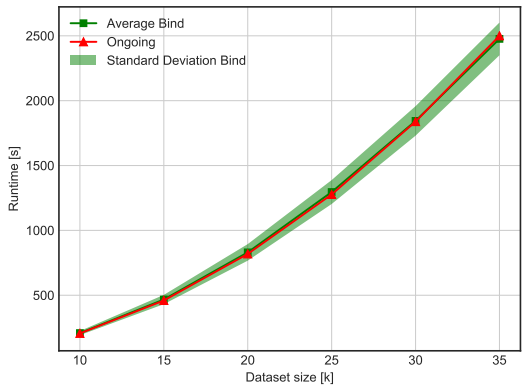
(a) Absolute result size

(b) Relative result size

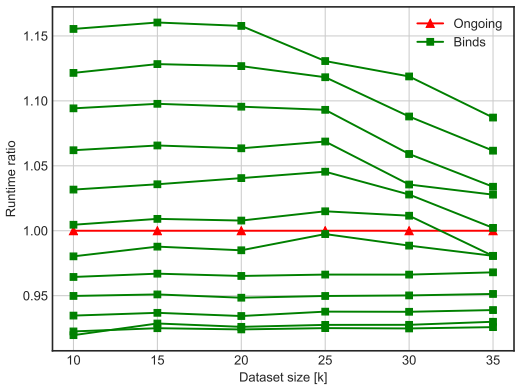
Figure 4.4: End set result size

In terms of result size in figure 4.4, the same can be said as with the Start set: The result size of the ongoing approach is the same as the worst case of the bind approach.
With the *End set*, it can be observed from figure 4.3b the ongoing approach is significantly below the bind approach regardless of the reference date used consistently across all dataset sizes. This can be explained with the ongoing tuples mostly being at the end of the range. The ongoing tuples still overlap with any following tuples with the ongoing approach, but unlike the Start set, every fixed tuple are placed before the ongoing tuples, which means that the ongoing tuples only overlap minimally with the fixed tuples.

4.3.3 Random set

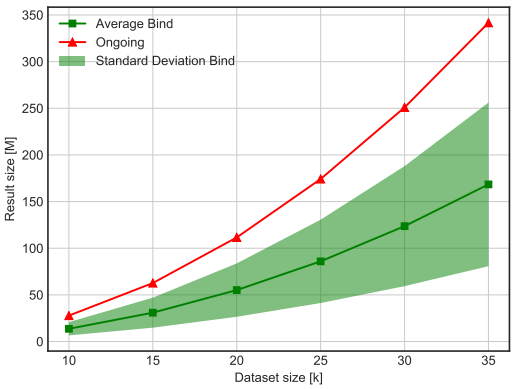


(a) Absolute runtime

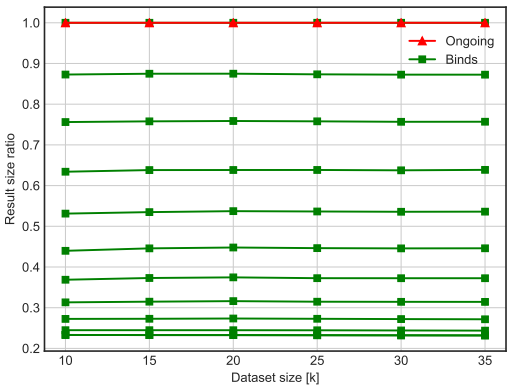


(b) Relative runtime

Figure 4.5: Random set runtime



(a) Absolute result size

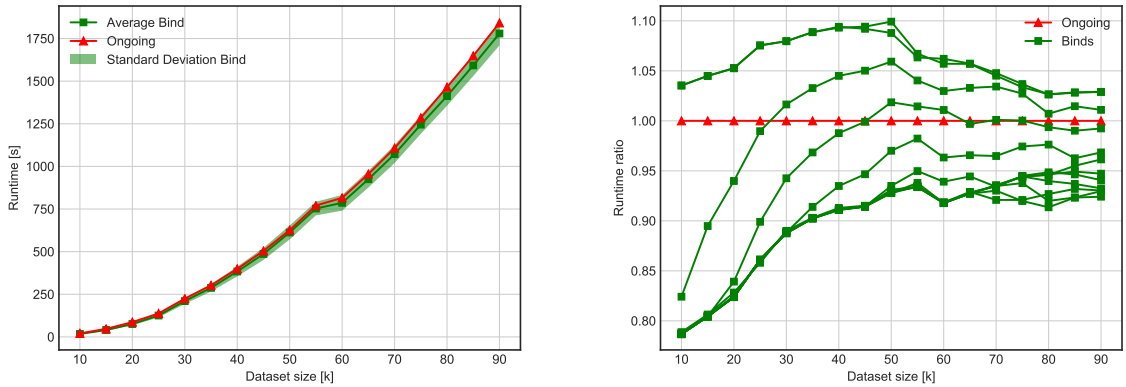


(b) Relative result size

Figure 4.6: Random set result size

From figure 4.5a we can see that the runtime of the ongoing approach is practically the same as the average of results of the bind approach with the *Random set*. However, in figure 4.5b it can be seen that for higher dataset sizes, some bind approach results tend to get faster compared to the ongoing approach. This effect can probably be explained by the same argument as in the Start set, but to a lesser extent.

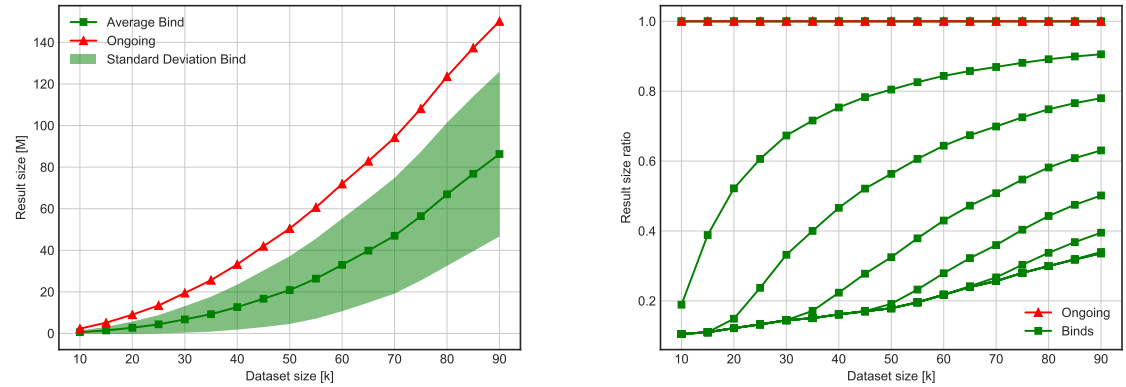
4.3.4 Eclipse set



(a) Absolute runtime

(b) Relative runtime

Figure 4.7: Eclipse set runtime



(a) Absolute result size

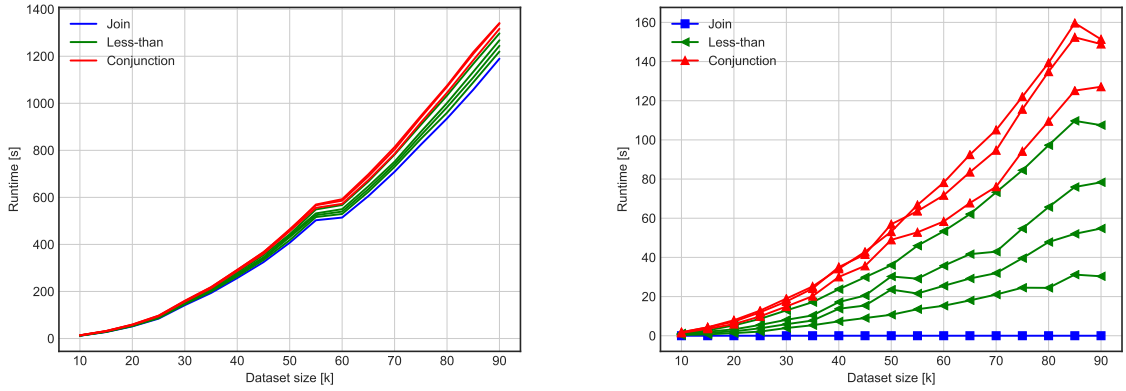
(b) Relative result size

Figure 4.8: Eclipse set result size

Note that for the *Eclipse set*, there is a slight "bump" in the lines between the sizes *55k* and *60k* in figure 4.7a due to a changed query plan. It does not influence the ratio values, as both the ongoing and bind approach queries are affected. From the results from 4.7b we can see that the ratios of the bind approach compared to the ongoing approach are stabilizing with growing sample size. On average, the bind approach has a slightly shorter runtime than the ongoing approach. In terms of result size, the ongoing approach is still the same as the worst case of any bind approach results. However, unlike with the synthetic datasets, the ratio of results of the bind appraoch in figure 4.8b changes with different dataset sizes. This is due to the prolonging of the history with increasing dataset size with the *Eclipse set* instead of increasing the daterange density within a fixed range with the synthetic sets.

4.4 Analysis of Runtime of Overlaps

To fully understand the *runtime overhead* of the overlaps predicate, we split it up into different parts and measured the runtime for each one. With new insights on how much each part contributes to the total runtime, we hope to discover potential improvements. The eclipse dataset was chosen as test data because it provided the highest number of samples and is based on real-world data. The implementation of the overlaps predicte was defined in algorithm 8 in the chapter 3.
The first step is the *self-join* of the dataset with join conditions and deserializing of dateranges. The following four steps are the computations of the *generally valid less-thans*. The last three steps are the *generally valid logical conjunction connectives*, with a total of eight steps. For each step, a new test function is defined, which executes the predicate as usual until its step is finished. It will stop immediately and return nothing.



(a) Absolute runtime

(b) Relative runtime

Figure 4.9: Runtime breakdown of overlaps predicate

The obvious thing to notice in figure 4.9a is that the first step, the *join* with itself, is responsible for almost the entire runtime. However, the actual overlaps predicate evaluation is still responsible for approximately 10% of the runtime on average, which is not an unsignificant amount.
To visualize the other step in greater detail, the figure 4.9b shows the same values but measured after finishing joining. From there it can be seen that the ongoing less-than predicates on the dateranges require more time to finish than the ongoing logical conjunction on the ongoing booleans. Another interesting observation is that the three ongoing logical conjunction connectors vary significantly in runtime. With a dataset size of *90k*, the first and second conjunctions take about the same time, but the third one is immensely faster than the other two. 11% of the runtime of the second conjunction. A similar trend can be observed with smaller dataset sizes. One possible explanation would be the sequence in which the conjunctions are evaluated. In the current implementation, the leftmost and the rightmost conjunctions of the formula are first evaluated. The results of the two conjunctions are then evaluated again with the middle conjunction. The difference in runtime between the first two and third conjunction

might be the input, where the first two gets ongoing booleans from results of less-than predicates, whereas the input of the third conjunction are results of ongoing conjunctions, which could be easier to evaluate.

4.5 Discussion

In conclusion, the results of our tests show that show that the ongoing approach is never significantly worse than the bind approach. For the best case, *End set*, the runtime of the ongoing approach was consistently under the runtime of the bind approach across all instantiation dates and dataset sizes. Based on the tests with the real-world data, the *Eclips set*, the runtime for the ongoing approach was only slightly longer than the average of the bind approach. However, if *multiple results* are desired at different reference dates, the bind approach must re-evaluate the query, while the ongoing approach can instantiate the result from a previously cached intermediate result. This is possible due result of the ongoing approach being valid for all references dates. In such scenarios, even with desired results at only two different reference dates, the ongoing approach is favorable in every tested case in terms of runtime. This advantage is amplified with each subsequent query with a different reference date.
In terms of result size, results of the ongoing approach is always the same as the worst case scenario of the bind approach with datasets with only fixed and growing tuples, which is the case in most of real-world application scenarios. This was expected, as a single result of the ongoing approach must contain any information retrievable by the bind approach. However, analogous to runtime, the more requested results at different reference dates, the greater the combined result sizes of the bind approach becomes while the result of the ongoing approach always stays the same size.
One interesting thing to notice is that despite our expectations, the ongoing approach can be faster than the bind approach. We expected that with a result size that is always the worst case of the bind approach and a complex overlaps predicate to evaluate, the ongoing approach would always have a longer execution runtime. A possible explanation would be that with the bind approach, instantiating the ongoing dates before the overlaps evaluation is responsible for a unexpectedly large runtime overhead.

5 Conclusions and Future Work

The goal of this thesis was to implement the newly proposed *ongoing approach* by *Y. Mülle et al* [3] into the PostgreSQL kernel and evaluate the new method by comparing it to the conventional *bind approach*. The results from queries following the ongoing approach are *generally valid*, meaning its valid state remains with passing time. These intermediate results can be cached and retrieved without re-evaluating the query, even at different reference date, resulting in minimal computational effort requirements.
The ongoing approach was built upon the PostgreSQL database kernel, which was extended with additional data types and their respective functions. With only two new data types, the *ongoing date* and *ongoing boolean*, a generally valid *overlaps predicate* for ongoing date ranges was usable in SQL queries.
Results from the evaluation with an extensive range of datasets and different reference dates have shown that the ongoing approach was never significantly worse compared to the bind approach for the first time a query with overlaps is evaluated. However, with each subsequent query with a different reference date, the computational effort to retrieve the answer from the *cached intermediate result* of the ongoing approach is next to nothing, while the bind approach takes substantially longer to re-evaluate. In terms of result size, the number of result tuples from a query following the ongoing approach were always as high as the worst case of a query of the binding approach, which was expected. However, analogous to the execution runtime, the result size of the bind approach increases with the number of requested results at different reference dates, whil the the result size of the ongoing approach stays the same. In conclusion, the ongoing approach is clearly the favorable method to process ongoing dates if the same query is evaluated at multiple reference dates.

For the future, we would like to see further development of the *ongoing approach*. Looking at the runtime results of breaking the ongoing overlaps predicate in parts, the obvious next challenge is to optimize the join operation, as it took the highest percentage of the entire runtime. However, the actual overlaps predicate consisting of ongoing less-thans and ongoing *ANDs* taking as much as 10% of the runtime, some improvements could be made by optimizing these functions.
In this thesis, evaluation results have shown that it is definitely applicable in real-world scenarios and has a massive runtime advantage compared against the bind approach in certain applications. However, most of the commonly used database systems do not even support ongoing dates or something similar, let alone ongoing predicates. An integration of an implementation into the an official release of a database system could be very useful to provide the functions and advantages of the ongoing approach out-of-the-box.

Bibliography

[1] L. Anselma, B. Stantic, P. Terenziani, and A. Sattar. Querying Now-Relative Data. *Journal of Intelligent Information Systems*, 41(2):285-311, 2013.

[2] Clifford, James and Dyreson, Curtis and lsakowitz, Tomás and Jensen, Christian S. and Snodgrass, Richard Thomas. On the Semantics of Now in Databases. *ACM Transactions on Database Systems*, 1997.

[3] Mülle, Yvonne and Böhlen, Michael H. Generally Valid Queries in Databases with Ongoing Time Points. *to be published.*

[4] Ahmed Lamkanfi and Javier Perez and Serge Demeyer. The Eclipse and Mozilla Defect Tracking Dataset: a Genuine Dataset for Mining Bug Information. *MSR '13: Proceedings of the 10th Working Conference on Mining Software Repositories, May 18-–19, 2013. San Francisco, California, USA*, 2013

Appendix

CD Contents

- **Abstract.txt**
  The abstract in English
- **Zusfsg.txt**
  The abstract in German
- **Bachelorarbeit.pdf**
  The thesis in PDF format
- **PostgreSQL-9.4.11.zip** A compressed zip file of the PostgreSQL source code. A shell script is included for installing and starting.

Code Snippets

In this section, all important changes to the PostgreSQL source code are listed.

Listing 1: date.c.h

```
/* date_in()
* Given date text string, convert to internal date format.
*/
Datum
date_in(PG_FUNCTION_ARGS)
{
  // Possible input dates (with example dates): 2010-01-01,
     2010-01-01+, +2010-01-01, 2010-01-01+2011-01-01, -
     infinity+infinity
  char *input = PG_GETARG_CSTRING(0);
  DateADT *result = palloc(sizeof(DateADT));

  if (strchr(input, '+') != NULL) {
    result->isFixed = false;
    if (input[0] == '+') {
      // Limited time point
      input++; // Remove plus char
      result->ongoingLower = DATEVAL_NOBEGIN;
      result->ongoingUpper = parse_single_date(input);
    } else if (input[strlen(input)-1] == '+') {
      // Growing time point
      input[strlen(input)-1] = 0; // Remove plus char
      result->ongoingLower = parse_single_date(input);
      result->ongoingUpper = DATEVAL_NOEND;
    } else {
      // Capped time point
      char *split = strtok(input, "+"); // Get first date
      if (split != NULL) {
        result->ongoingLower = parse_single_date(split);
      }
      split = strtok(NULL, "+"); // Get second date
      if (split != NULL) {
        result->ongoingUpper = parse_single_date(split);
      }
```

```
    }
    // Check if ongoingDate is a fixed date
    if (result->ongoingLower == result->ongoingUpper) {
      result->isFixed = true;
    }
  } else if (strcmp(input, "now") == 0) {
    // now variable
    result->isFixed = false;
    result->ongoingLower = DATEVAL_NOBEGIN;
    result->ongoingUpper = DATEVAL_NOEND;
  } else {
    result->isFixed = true;
    result->ongoingLower = result->ongoingUpper =
       parse_single_date(input);
  }

  // Check if the lower is smaller than upper date
  if (result->ongoingUpper < result->ongoingLower) {
    elog(ERROR, "Upper bound is before lower bound");
  }

  PG_RETURN_DATEADT(result);
}

DateSingleADT
parse_single_date(char* str) {
  DateSingleADT date;
  fsec_t    fsec;
  struct pg_tm tt,
  *tm = &tt;
  int      tzp;
  int      dtype;
  int      nf;
  int      dterr;
  char     *field[MAXDATEFIELDS];
  int      ftype[MAXDATEFIELDS];
  char    workbuf[MAXDATELEN + 1];

  dterr = ParseDateTime(str, workbuf, sizeof(workbuf),
  field, ftype, MAXDATEFIELDS, &nf);
  if (dterr == 0)
    dterr = DecodeDateTime(field, ftype, nf, &dtype, tm, &fsec
       , &tzp);
  if (dterr != 0)
```

```
    DateTimeParseError(dterr, str, "date");

  switch (dtype)
  {
    case DTK_DATE:
      break;

    case DTK_CURRENT:
      ereport(ERROR,
        (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
        errmsg("date/time value \"current\" is no longer
           supported")));

      GetCurrentDateTime(tm);
      break;

  case DTK_EPOCH:
    GetEpochTime(tm);
    break;

  case DTK_LATE:
    DATE_NOEND(date);
    PG_RETURN_DATESINGLEADT(date);

  case DTK_EARLY:
    DATE_NOBEGIN(date);
    PG_RETURN_DATESINGLEADT(date);

  default:
    DateTimeParseError(DTERR_BAD_FORMAT, str, "date");
    break;
  }

  if (!IS_VALID_JULIAN(tm->tm_year, tm->tm_mon, tm->tm_mday))
    ereport(ERROR,
      (errcode(ERRCODE_DATETIME_VALUE_OUT_OF_RANGE),
      errmsg("date out of range: \"%s\"", str)));

  date = date2j(tm->tm_year, tm->tm_mon, tm->tm_mday) -
     POSTGRES_EPOCH_JDATE;

  return date;
}
```

```
/* date_out()
* Given internal format date, convert to text string.
*/
Datum
date_out(PG_FUNCTION_ARGS)
{
  DateADT    *date = PG_GETARG_DATEADT(0);
  char       *result;
  DateSingleADT lower = date->ongoingLower;
  DateSingleADT upper = date->ongoingUpper;

  if (date->isFixed) {
    char *buffer;
    buffer = (char*) compose_single_date(lower);
    result = pstrdup(buffer);
  } else {
    char buf[MAXDATELEN*2 + 1]; // Buffer double the size of a
        single date
    if (lower == DATEVAL_NOBEGIN && upper == DATEVAL_NOEND) {
      strcpy(buf, "now");
    } else if (lower == DATEVAL_NOBEGIN) {
      // limited date
      strcpy(buf, "+");
      strcat(buf, (char*) compose_single_date(upper));
    } else if (upper == DATEVAL_NOEND) {
      // growing date
      strcpy(buf, (char*) compose_single_date(lower));
      strcat(buf, "+");
    } else {
      // capped date
      strcpy(buf, (char*) compose_single_date(lower));
      strcat(buf, "+");
      strcat(buf, (char*) compose_single_date(upper));
    }
    result = pstrdup(buf);
  }

  PG_RETURN_CSTRING(result);
}
```

Listing 2: gv_predicates.h

```
OngoingBoolean *
gv_internal_date_lt(DateADT *date1, DateADT *date2){
```

```
  OngoingBoolean *first = NULL, *second = NULL;

  if (date1->ongoingUpper < date2->ongoingLower) {
    first = make_ongoing_bool(DATEVAL_NOBEGIN, DATEVAL_NOEND);
  } else if (date1->ongoingLower < date2->ongoingLower &&
     date2->ongoingUpper <= date1->ongoingUpper) {
    first = make_ongoing_bool(DATEVAL_NOBEGIN, date2->
       ongoingLower);
    } else if (date1->ongoingLower < date2->ongoingLower &&
       date2->ongoingLower <= date1->ongoingUpper && date1->
       ongoingUpper < date2->ongoingUpper) {
    first  = make_ongoing_bool(DATEVAL_NOBEGIN, date2->
       ongoingLower);
    second = make_ongoing_bool(date1->ongoingUpper + 1,
       DATEVAL_NOEND);
  } else if (date2->ongoingLower <= date1->ongoingLower &&
     date1->ongoingUpper < date2->ongoingUpper) {
    first  = make_ongoing_bool(date1->ongoingUpper + 1,
       DATEVAL_NOEND);
  }

  if (second) { first->next = second; }
    return first;
}

OngoingBoolean *
gv_internal_date_overlaps(DateADT *firstLower, DateADT *
   firstUpper, DateADT *secondLower, DateADT *secondUpper) {
  OngoingBoolean *a = gv_internal_date_lt(firstLower,
     secondUpper);
  OngoingBoolean *b = gv_internal_date_lt(secondLower,
     firstUpper);
  OngoingBoolean *c = gv_internal_date_lt(firstLower,
     firstUpper);
  OngoingBoolean *d = gv_internal_date_lt(secondLower,
     secondUpper);

  OngoingBoolean *x = gv_internal_logical_and(a, b);
  OngoingBoolean *y = gv_internal_logical_and(x, c);
  return gv_internal_logical_and(y, d);
}
```

Listing 3: gv_ongoing_bool.h

```
OngoingBoolean*
deserialize_ongoingBoolean(ArrayType *array) {
  // ArrayType variables
  int nitems;
  Datum *datumElements;
  bool *nulls;
  int i; // for loop

  // RangeType variables
  OngoingBoolean *first = NULL, *latest = NULL, *current;
  RangeBound lowerBound, upperBound;
  bool empty;
  TypeCacheEntry *typcache = lookup_type_cache(DATERANGE_OID,
     TYPECACHE_RANGE_INFO);

  deconstruct_array(
    array, DATERANGE_OID, DATERANGE_SIZE, false,
       DATERANGE_ALIGNMENT,
    &datumElements, &nulls, &nitems);

  for (i=0; i<nitems; i++) {
    if (nulls[i]) { continue; }

    range_deserialize(typcache, (RangeType*) datumElements[i],
      &lowerBound, &upperBound, &empty);

    // Only using ongoingLower attribute as the dates should
       be fixed
    current = make_ongoing_bool(
      ((DateADT*) lowerBound.val)->ongoingLower,
      ((DateADT*) upperBound.val)->ongoingLower
    );

    // Link current to the latest or to the returned pointer
    if (latest) {
      latest->next = current;
    } else {
      first = current;
    }
    latest = current;
  }

  if (latest) {
    latest->next = NULL;
```

```
  }

  return first;
}

ArrayType*
serialize_ongoingBoolean(OngoingBoolean *ongoingBoolList) {
  ArrayType *result;
  TypeCacheEntry *typcache = lookup_type_cache(DATERANGE_OID,
     TYPECACHE_RANGE_INFO);
  DateADT *lowerDate, *upperDate;
  Datum *elems;
  int counter = 0;
  int i; // for loop

  // empty list
  if (!ongoingBoolList) {
    return construct_empty_array(DATERANGE_OID);
  }

  OngoingBoolean *boolCounter = ongoingBoolList;
  while (boolCounter) {
    counter++;
    boolCounter = boolCounter->next;
  }

  elems = palloc(sizeof(Datum)*counter);
  for (i=0; i<counter; i++) {
    lowerDate = palloc(sizeof(DateADT));
    upperDate = palloc(sizeof(DateADT));
    RangeBound lowerBound;
    RangeBound upperBound;

    lowerDate->ongoingLower = ongoingBoolList->lower;
    lowerDate->ongoingUpper = ongoingBoolList->lower;
    lowerDate->isFixed = true;
    upperDate->ongoingLower = ongoingBoolList->upper;
    upperDate->ongoingUpper = ongoingBoolList->upper;
    upperDate->isFixed = true;

    lowerBound.val = (Datum) lowerDate;
    lowerBound.infinite = false;
    lowerBound.inclusive = true;
    lowerBound.lower = true;
```

```
    upperBound.val = (Datum) upperDate;
    upperBound.infinite = false;
    upperBound.inclusive = false;
    upperBound.lower = false;

    elems[i] = (Datum) make_range(typcache, &lowerBound, &
       upperBound, false);
    ongoingBoolList = ongoingBoolList->next;
  }

  result = construct_array(elems, counter,
  DATERANGE_OID, DATERANGE_SIZE, false, DATERANGE_ALIGNMENT);

  return result;
}

bool
gv_internal_bind_ongoing_boolean(OngoingBoolean *list, DateADT
    *ref) {
  if (!ref->isFixed) {
    elog(ERROR, "Reference date is not fixed");
  }
  DateSingleADT fixedDate = ref->ongoingLower;

  while (list) {
    if (list->upper < fixedDate) {
    return false;
  } else if (list->lower <= fixedDate) {
    return true;
  }
    list = list->next;
  }
  return false;
}


OngoingBoolean *
gv_internal_logical_not(OngoingBoolean *input) {
  OngoingBoolean *current, *first = NULL, *latest = NULL;

  if (!input) {
    return make_ongoing_bool(DATEVAL_NOBEGIN, DATEVAL_NOEND);
  }
```

```
  if (input->lower != DATEVAL_NOBEGIN) {
    current = make_ongoing_bool(DATEVAL_NOBEGIN, input->lower)
       ;
    first = current;
    latest = current;
  }

  for (;;) {
    if (input->upper == DATEVAL_NOEND) { break; }

    current = palloc(sizeof(OngoingBoolean));
    current->lower = input->upper;

    if (latest) {
      latest->next = current;
    } else {
      first = current;
    }
    latest = current;

    if (input->next) {
      input = input->next;
      current->upper = input->lower;
    } else {
      current->upper = DATEVAL_NOEND;
      break;
    }
  }

  if (latest) { latest->next = NULL; }
    return first;
}

OngoingBoolean *
gv_internal_logical_and(OngoingBoolean *input1, OngoingBoolean
    *input2) {
  OngoingBoolean *current, *first = NULL, *latest = NULL;

  while (input1 && input2) {
    while (input1->lower >= input2->upper || input2->lower >=
       input1->upper) {
      if (input1->lower >= input2->upper) {
        input2 = input2->next;
      } else {
```

```
        input1 = input1->next;
      }
      if (!input1 || !input2) {
        if (latest) { latest->next = NULL; }
        return first;
      }
    }

    current = make_ongoing_bool(
      max(input1->lower, input2->lower),
      min(input1->upper, input2->upper)
    );

    if (latest) {
      latest->next = current;
    } else {
      first = current;
    }
    latest = current;

    if (input2->upper < input1->upper) {
      input2 = input2->next;
    } else if (input1->upper < input2->upper) {
      input1 = input1->next;
    } else {
      input1 = input1->next;
      input2 = input2->next;
    }
  }

  if (latest) { latest->next = NULL; }
  return first;
}

OngoingBoolean *
gv_internal_logical_or(OngoingBoolean *input1, OngoingBoolean
   *input2) {
  OngoingBoolean *current, *first = NULL, *latest = NULL;

  // Each iteration creates a new OngoingBoolean
  while (input1 && input2) {
    // Swap pointers so the input1 always starts first
    if (input2->lower < input1->lower) {
      swap(&input1, &input2);
```

```
    }

    current = palloc(sizeof(OngoingBoolean));
    current->lower = input1->lower;

    // Join a group of continuously overlapping intervals
    while (input1 && input2) {
      if (input2->lower <= input1->upper) {
        // input2 starting inside input1
        if (input2->upper > input1->upper) {
          // input2 is extendind input1. Repeat loop swapped.
          input1 = input1->next;
          swap(&input1, &input2);
        } else {
          // input2 is countained inside input1
          input2 = input2->next;
        }
        if (!input2) {
          current->upper = input1->upper;
          input1 = input1->next;
        }
      } else {
        // End of overlapping intervals
        current->upper = input1->upper;
        input1 = input1->next;
        break;
      }
    }

    if (latest) {
      latest->next = current;
    } else {
      first = current;
    }
    latest = current;
  }

  // Add any remaining intervals
  while (input1 || input2) {
    current = palloc(sizeof(OngoingBoolean));

    if (input1) {
      memcpy(current, input1, sizeof(OngoingBoolean));
      input1 = input1->next;
```

```
    } else {
      memcpy(current, input2, sizeof(OngoingBoolean));
      input2 = input2->next;
    }

    if (latest) {
      latest->next = current;
    } else {
      first = current;
    }
    latest = current;
  }

  if (latest) { latest->next = NULL; }
  return first;
}
```

Listing 4: pg_proc.h

```
DATA(insert OID = 6000 ( gv_bind_ongoing_boolean  PGNSP PGUID
   12 1 0 0 0 f f f f f f i 2 0 16 "3913 1082" _null_ _null_
   _null_ _null_ gv_bind_ongoing_boolean _null_ _null_ _null_
   ));
DESCR("negation operator for ongoing booleans");
DATA(insert OID = 6001 ( gv_logical_not  PGNSP PGUID 12 1 0 0
   0 f f f f f f i 1 0 3913 "3913" _null_ _null_ _null_ _null_
    gv_logical_not _null_ _null_ _null_ ));
DESCR("negation operator for ongoing booleans");
DATA(insert OID = 6002 ( gv_logical_and  PGNSP PGUID 12 1 0 0
   0 f f f f f f i 2 0 3913 "3913 3913" _null_ _null_ _null_
   _null_ gv_logical_and _null_ _null_ _null_ ));
DESCR("conjunction operator for ongoing booleans");
DATA(insert OID = 6003 ( gv_logical_or  PGNSP PGUID 12 1 0 0 0
    f f f f f f i 2 0 3913 "3913 3913" _null_ _null_ _null_
   _null_ gv_logical_or _null_ _null_ _null_ ));
DESCR("disjunction operator for ongoing booleans");

DATA(insert OID = 6004 ( gv_date_lt  PGNSP PGUID 12 1 0 0 0 f
   f f f f f i 2 0 3913 "1082 1082" _null_ _null_ _null_
   _null_ gv_date_lt _null_ _null_ _null_ ));
DESCR("less than operator for DateADT");
DATA(insert OID = 6005 ( gv_date_overlaps  PGNSP PGUID 12 1 0
   0 0 f f f f f f i 2 0 3913 "3912 3912" _null_ _null_ _null_
    _null_ gv_date_overlaps _null_ _null_ _null_ ));
```

```
DESCR("ovelaps operator for DateADT ranges");
DATA(insert OID = 6006 ( date_bind  PGNSP PGUID 12 1 0 0 0 f f
    f f f f i 2 0 1082 "1082 1082" _null_ _null_ _null_ _null_
    date_bind _null_ _null_ _null_ ));
DESCR("binds an ongoing Date to a reference time");
DATA(insert OID = 6007 ( daterange_bind  PGNSP PGUID 12 1 0 0
   0 f f f f f f i 2 0 3912 "3912 1082" _null_ _null_ _null_
   _null_ daterange_bind _null_ _null_ _null_ ));
DESCR("binds a daterange to a reference time");
```

Listing 5: pg_type.h

```
DATA(insert OID = 1082 ( date     PGNSP PGUID  9 f b D f t
   \054 0  0 1182 date_in date_out date_recv date_send - - - d
    p f 0 -1 0 0 _null_ _null_ _null_ ));
DESCR("date");
```