

Department of Informatics, University of Zürich

BSc Thesis

Integration of Ongoing Integers into PostgreSQL

Timothy Pescatore

Matrikelnummer: 14-916-886

Email: timothy.pescatore@uzh.ch

September 15, 2018

supervised by Prof. Dr. M. Böhlen and Y. Mülle



University of
Zurich^{UZH}

Department of Informatics



Acknowledgements

First of all, I want to thank my supervisor Yvonne Mülle who supported me and gave valuable advice during the whole time. I also want to thank Prof. Dr. Michael Böhlen, head of the database technology group for giving me the opportunity for this thesis.

Abstract

Data that are associated with valid time interval are common in nowadays databases. They often store the time point *now* that represents the current time point and therefore changes its value over time. With *ongoing time points* like *now*, we face the problem, that the result of a query changes when time progresses. A new *ongoing approach* wants to calculate query results which remain valid over time. This was achieved with a new type of date, called *ongoing date*, that stays uninstantiated during query evaluation. With *ongoing dates* we can have a time interval whose duration changes as time passes by. If we do not want to refresh the duration results all the time, we need an *ongoing integer*, which holds different values for different time points to describe such a duration. This *ongoing integer* was implemented in this thesis into the widely-used PostgreSQL database system. This was achieved with a new data type called *ogint*. To integrate the new data type into the *ongoing approach* we implemented functions to measure the duration of an *ongoing interval* and additionally an addition and a maximum function. With the *ogint* we achieved a runtime for the duration function that was as fast as a *bind approach*, where we first need to convert *ongoing dates* to fixed dates. The major advantage to the *bind approach* is that we do not need to reevaluate our queries as time passes by.

Zusammenfassung

Daten, die einem Gültigkeitszeitintervall zugeordnet sind, sind in heutigen Datenbanken üblich. Sie speichern oft den Zeitpunkt *now*, der den aktuellen Zeitpunkt repräsentiert und somit seinen Wert mit fortschreitender Zeit ändert. Mit *ongoing Zeitpunkten* wie *now* haben wir das Problem, dass sich das Ergebnis einer Abfrage mit der Zeit ändert. Ein neuer *ongoing Ansatz* möchte Abfrageergebnisse evaluieren, die über die Zeit hinweg gültig bleiben. Dies wurde mit einem neuen Datentyp erreicht, der als *ongoing date* bezeichnet wird und während der Abfrageauswertung uninstantiiert bleibt. Mit *ongoing dates* können wir ein Zeitintervall haben, dessen Dauer sich mit der Zeit ändert. Daher brauchen wir einen *ongoing integer*, der verschiedene Werte für verschiedene Zeitpunkte enthält, um eine solche Dauer zu beschreiben. Dieser *ongoing integer* wurde in dieser Arbeit in das weit verbreitete PostgreSQL-Datenbanksystem implementiert. Dies wurde mit einem neuen Datentyp namens *ogint* erreicht. Um den neuen Datentyp in den *ongoing Ansatz* zu integrieren, haben wir Funktionen implementiert, um die Dauer eines *ongoing Intervalls* zu messen und zusätzlich die Addierung zu ermöglichen und das Maximum zu bestimmen. Mit dem *ogint* haben wir eine Laufzeit für die Duration-Funktion erreicht, die so schnell war wie der *bind Ansatz*, wobei wir zunächst die *ongoing dates* in fixe Daten umwandeln müssen. Der Hauptvorteil vom *bind Ansatz* ist, dass wir unsere Abfragen nicht im Laufe der Zeit erneuern müssen.

Contents

1	Introduction	8
1.1	Temporal Databases	8
1.2	Problem Statement	8
2	Ongoing Data Types	10
2.1	Preliminaries	10
2.2	Ongoing Dates	10
2.3	Ongoing Integers	12
3	Implementation	14
3.1	Ongoing Integers	14
3.2	Basic Function	16
3.2.1	Read and Write Function	16
3.2.2	Serialize and Deserialize	16
3.2.3	Sort and Search	17
3.3	Duration Function	17
3.4	Addition Function	19
3.5	Maximum Function	20
3.6	Special Points of Focus	23
4	Evaluation	25
4.1	Setup	25
4.1.1	Experimental Setup	25
4.1.2	Real-world Dataset	25
4.1.3	Synthetic Dataset	25
4.2	Tests and Results	25
4.2.1	Result size	26
4.2.2	Duration Function	26
4.2.3	Addition Function	29
4.2.4	Maximum Function	30
5	Conclusions and Future Work	31

List of Figures

- 2.1 Illustration of ongoing time points 11
- 3.1 Duration of an ongoing time interval 17
- 4.1 Comparing ongoing and bind approach 28
- 4.2 Duration function with synthetic dataset 28
- 4.3 Addition function with synthetic datasets 29
- 4.4 Maximum function with synthetic datasets 30

List of Tables

2.1 Notation. 10
2.2 Illustration of all ongoing integer segment types. 12

1 Introduction

1.1 Temporal Databases

Data that are associated with a valid time interval are common in nowadays databases. These intervals often contain the time point *now*, e.g. an employment contract which has no fixed ending date. In the paper of Mülle Y. et al. [4] a new type of date, a so-called *ongoing date*, was proposed to store a time point such as *now* which keeps uninstantiated while time progresses. In most databases the *now* time point is instantiated at the current time of the query, so it can be treated like a fixed time point [1]. The *ongoing dates* are not instantiated during the query evaluation and therefore allow, that results stay valid as time passes by. This can be useful to store tuples with a valid time, where it is crucial for the result at what time the query is executed.

Let's say we have a contract which is valid from 2018/1 until *now* and we query for the duration of the employment on the date 2018/2. The result of the query would return one month, but this result does not remain valid as soon as time progresses.

1.2 Problem Statement

When we want to know the result of the duration, not shown as a time interval but as an integer, we face the same problem as with the fixed time points where we need to refresh the result. The *ongoing integer* is a new approach where we describe the value of an integer for all reference time points. With such an *ongoing integer*, it is possible to describe the duration of an *ongoing time interval*, even though the result changes as time passes by. With the *ongoing integer* we can know the duration for any reference time.

Consider a company with a relational database that keeps track of their employment contracts. To identify the contract, it has a unique ID and it also stores the name of the employee. The validity of the contract is described by the attribute *vt* storing a time interval.

In the relation figure, we have two contracts for Tom and Max. The contract from Tom is running out on March 2018 and Max starts at the company in January 2018 with an open end. When query for the duration of Tom's contract, it does not matter when we do the query, the result keeps the same. Whereas for the duration of Max's contract, the time point our query refers to, is crucial. When we query before 2018/1 the result is always zero, afterwards the

ID	Name	VT
101	Tom	(2016/1,2018/3)
102	Max	(2018/1, <i>now</i>)

result is increasing with each time increment. The *ongoing integer* has the property to be built with segments each describing a different alteration of the integer, so that we can describe the development of the integer which first stays zero and afterwards starts growing.

In this thesis we will integrate the *ongoing integer* into the widely used and open source database system PostgreSQL. The new data type will be implemented with C into the PostgreSQL kernel. The implementation includes low level functions as for example the serialization which is necessary, so we can store our data type, up to high level functions which are aimed for queries and the usage with other ongoing data types e.g. the duration function.

Finally, we will evaluate the new data type, called *ogint*, with real-world and synthetic datasets. We measure the overhead of an *ogint* compared to a fixed integer and we will analyze how the number of segments grows in different functions as well as how the runtime changes with different input.

The thesis is organized as follows. Chapter 2 describes the *ongoing time points* and introduces the *ongoing integer* in a formal way. In chapter 3 the implementation of the new data type into the database system PostgreSQL is described as well as the algorithms used for the addition and maximum function. In chapter 4 the tests and their evaluation is described and chapter 5 concludes the thesis and points to future work.

2 Ongoing Data Types

In this chapter the ongoing data types *ongoing date* and *ongoing integer* are described formally. The focus is on the *ongoing integer* but because until now an *ongoing integer* only emerges when the duration of an *ongoing time interval* is measured, we first need to get a quick overview of *ongoing time points*.

2.1 Preliminaries

We assume a linearly ordered, discrete *time domain* \mathcal{T} with $-\infty$ as the lower limit and ∞ as the upper limit. A *time point* is an element of time domain \mathcal{T} . A *time interval* $[t_s, t_w)$ consists of an inclusive start point t_s and an exclusive end point t_w . *Fixed* data types consists of values that do not change as time passes by. *Ongoing* data types include values that change as time passes by.

In the table 2.1 are the symbols listed which are used for further description of ongoing and fixed data types.

Symbol	Meaning
\mathcal{T}	domain of fixed time points
Ω	domain of ongoing time points
Z	domain of ongoing integers
<i>now</i>	ongoing time point
$[t_s, t_e)$	fixed or ongoing time interval
$\ \cdot\ _{rt}$	bind operator

Table 2.1: Notation.

2.2 Ongoing Dates

We briefly introduce the ongoing time points and time intervals proposed by Mülle Y. et al. [4].

Definition 1 (Ongoing Time Point) *Let $rt \in \mathcal{T}$ be a reference time and $a, b \in \mathcal{T}$ with $a \leq b$. The ongoing time point $a+b$ is defined as*

$$\|a+b\|_{rt} = \begin{cases} a & rt \leq a \\ rt & a < rt < b \\ b & \text{otherwise} \end{cases}$$

An *ongoing time point* includes a time interval with a start and end time point called a and b , both are fixed. We can use the bind operator $\|\|_{rt}$ to instantiate the ongoing time point to a fixed *reference time*. When the reference time is within the time interval, the *bound operation* returns the same time point as the *reference time*. Otherwise it returns either the start point, when the *reference time* is below the starting point or the end point, if it is above the end point. The time interval is noted as $a+b$ where a and b are either a point in time or $\pm\infty$. The $+$ describes all time points between the start and the end point. The expression $a+b$ is read as, *not earlier than a, but not later than b*. With the constraint that b cannot be smaller than a . The two variables a and b can store either a time point or $\pm\infty$, with different combinations we can build five different types of *ongoing time points*, all different types are illustrated in the figure 2.1. When both variables store the same time point, we call it a *fixed ongoing time point*. If a is equal $-\infty$ it is called *limited* and if only b is ∞ it is referred to as *growing*. When both a and b hold $-\infty$ and ∞ it describes the time point now, since it includes all possible time points.

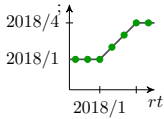
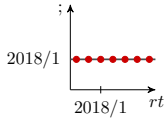
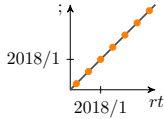
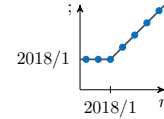
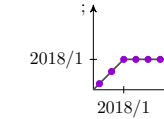
	Ongoing time point	Type			
		Fixed time point	Time point now	Growing time point	Limited time point
Notation	$a+b$	$a+a$	$-\infty+\infty$	$a+\infty$	$-\infty+b$
- as $a+b$	$a+b$	a	now	$a+$	$+b$
- short	$a+b$	a	now	$2018/1+$	$+2018/1$
Example	2018/1+2018/4	2018/1	2018/1	2018/1+	+2018/1
Semantics					

Figure 2.1: Illustration of ongoing time points $a+b$ [4].

As you can describe a time interval with fixed dates, e.g. from March 2018 until September 2018, we can also build an *ongoing time interval*. For instance, in the example from the introduction, we saw that the contract of Tom was valid from 2016/1 until 2018/3. With the *ongoing approach* we would use two fixed time points to describe the start and the end of this interval. When we look at the valid time interval of Max, we have a valid time from 2018/1 until *now*, with ongoing time points we would write it as $[(2018/1+2018/1), (-\infty+\infty))$, or in short form $[2018/1, now)$.

We can instantiate a ongoing time interval to a certain reference time. This instantiation means that we use the bind operator on both ongoing time points from the interval, so we get a fixed time interval as a result.

When we instantiate the latter interval to the two reference time points, 2017/1 and 2018/5, we would get, $\| [2018/1, now) \|_{2017/1} = \emptyset$ and $\| [2018/1, now) \|_{2018/5} = [2018/1, 2018/5)$. The first interval is empty because t_e is smaller than t_s and a negative time interval is senseless. The duration of an interval can be described as an integer, but because the interval changes depending on the reference time, we would need to calculate the duration for each instantiation. To avoid this recalculation, we use an *ongoing integer* which can store different integer values for different reference times.

2.3 Ongoing Integers

An *ongoing integer* is represented as a set of *ongoing integer segments* $s[(a \rightsquigarrow b, s) \text{ at } [rt_s, rt_e]]$. This segment describes the value of an integer at each point in time during the reference time interval from rt_s until rt_e . The variable a is the integer value at *reference time* rt_s and b is the value at the time point rt_e . During the time between rt_s and rt_e the value of a increments with s per reference time increase. The definition is as in[3]:

Definition 2 (Ongoing Integer Segment) *Let $a, b, s \in \mathbb{Z}$ be integers and $rt, rt_s, rt_e \in \mathcal{T}$ be reference times with $rt_s < rt_e$. Let $a - b = s \cdot (rt_e - rt_s)$. An ongoing integer segment $s[(a \rightsquigarrow b, s) \text{ at } [rt_s, rt_e]]$ is defined as*

$$\|s[(a \rightsquigarrow b, s) \text{ at } [rt_s, rt_e]]\|_{rt} = \begin{cases} a + s \cdot (rt - rt_s) & rt_s \leq rt < rt_e \\ \quad \wedge |a| \neq \infty \\ b - s \cdot (rt_e - rt) & rt_s \leq rt < rt_e \\ \quad \wedge |b| \neq \infty \\ \omega & \text{otherwise} \end{cases}$$

When the bind operator is applied on an ongoing integer segment, its value is instantiated and returns the integer value at the reference time. The value is calculated, as described in the definition, with the change value and the difference in time. Therefore, a segment is predominantly characterized by the incrementation value s and by its reference time interval. This means that from the reference time interval we know, whether the segment is long or short and s expresses what type of segment we have, constant, increasing or decreasing. They are shown graphically in the table 2.2.

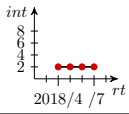
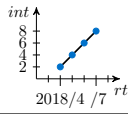
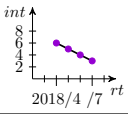
	<i>Ongoing Integer Segment Type</i>		
	Constant	Increasing	Decreasing
Notation	$s[(a \rightsquigarrow a, 0) \text{ at } [rt_s, rt_e]]$	$s[(a \rightsquigarrow b, +s) \text{ at } [rt_s, rt_e]]$	$s[(a \rightsquigarrow b, -s) \text{ at } [rt_s, rt_e]]$
Properties			
- a and b	$a = b$	$a < b$	$a > b$
Example	$s[(2 \rightsquigarrow 2, 0) \text{ at } [2018/4, 2018/7]]$	$s[(2 \rightsquigarrow 8, +2) \text{ at } [2018/4, 2018/7]]$	$s[(6 \rightsquigarrow 3, -1) \text{ at } [2018/4, 2018/7]]$
			

Table 2.2: Illustration of all ongoing integer segment types.

An ongoing integer contains only segments which are one of these three types. When we imagine an ongoing integer graphically, it can be seen as a concatenation of segments as they are illustrated in in table 2.2. Each end point of such a line segment must be the start point of another one. Every time there is an edge in the line, a new segment must start. We define an ongoing integer as in [3]:

Definition 3 (Ongoing integer) *Let \mathcal{T} be the time domain of the reference time. An ongoing integer \mathbf{i} is a set of ongoing integer segments $\{s[(a_j \rightsquigarrow b_j, s_j) \text{ at } [rt_{js}, rt_{je}]]\}$, such that*

1. $(\bigcup_{j=1}^{|\mathbf{i}|} [rt_{js}, rt_{je})) = \mathcal{T}$ and
2. $\forall s_j, s_k \in \mathbf{i} (s_j = s_k \vee ([rt_{js}, rt_{je}) \cap [rt_{ks}, rt_{ke}) = \emptyset)$

The first property guarantees that an ongoing integer returns a value for each possible reference time point. Additionally, the second property guarantees that it instantiates to *exactly one* integer value for *each reference time*.

Now let's look at an example to demonstrate how we can use an ongoing integer when we measure the duration of an ongoing time interval. We use the contract from Max as an example where the valid time interval was $[2018/1, now)$. When we instantiate this ongoing interval to any reference time before 2018/2 the resulting interval would be empty. With a reference time greater than 2018/1 the interval would increase with every time increment. This behavior can be expressed with an ongoing integer made out of two segments. One is constant, holding the value 0 and the second one is incremental starting at the value 0. So the duration of this ongoing time interval described by an ongoing integer would be $s[(0 \rightsquigarrow 0, 0) \text{ at } [-\infty, 2018/1)], s[(0 \rightsquigarrow \infty, 1) \text{ at } [2018/1, \infty)]$.

3 Implementation

This chapter describes and explains the implementation of the ongoing integer into the kernel of PostgreSQL. Starting with the structure of the data type itself, we will then look at the basic functions, reading the data type in, printing it out and the underlying serialization. These are needed so we can start working with it. Furthermore, we implemented the duration function, which is essential, because we can measure the duration of an *ongoing* time interval and therefore can integrate the ongoing integer into the *ongoing approach*. It is also the only function until now, which has an ongoing integer as output but not as input. Additionally, the addition and the maximum function were implemented to test the runtime of the functions and how much overhead is created when using ongoing integers.

3.1 Ongoing Integers

The decision what to include in the new data type, was based on three aspects. First, the amount of memory usage, since we do not want to waste any memory space of the database. Second, any information must not be missing, so all necessary operations on the *ongoing integer* are applicable. Third, aesthetics and readability, since ongoing integers are also planned to be returned uninstantiated to the user, the reader should be able to read them easily and get the important information quickly.

In the previous chapter we used the notation $s[(a \mapsto b, s) \text{ at } [rt_s, rt_e)]$ to describe an *ongoing integer segment*. An ongoing integer segment consists of three integer values and two fixed dates. This is a lot of information, which is presented to the reader and as soon as we have several segments together, it gets difficult to conceive the nature of the ongoing integer. Below we have an example of an ongoing integer with only two segments:

$$s[(0 \mapsto 0, 0) \text{ at } [-\infty, 2018/1)], s[(0 \mapsto \infty, 1) \text{ at } [2018/1, \infty)],$$

Even with only two segments, it gets difficult to read because the expression is quite long. Looking at this specific ongoing integer, there are four zeros in total, where two of them describe the same integer value for the same reference time. The date 2018/1 appears also twice, because the end point of the reference time interval is exclusive and the start point is inclusive. Therefore, a date and a value will be listed twice in an ongoing integer, for every consecutive pair. This redundancy is not necessary for a practical manner, since we know that an ongoing integer is defined for the whole time domain \mathcal{T} and therefore the start point of an segment must be the end point of the precedent segment.

The simplest way to find improvement in all three aspects, memory usage, complete information and readability, is to reduce or eliminate the redundancy. This would help to lower the

memory usage, without any loss of information plus the readability could increase, since the reader needs to filter out less, when looking at a string representation of an ongoing integer. Before we look at an example we need first to recall a few points from the definition about an ongoing integer. We know from the definition, that an ongoing integer is defined for the whole time domain \mathcal{T} . Therefore, rt_s from the first segment of an ongoing integer will always be equal to $-\infty$ and rt_e from the last segment will always be equal ∞ ¹. The second point guarantees us, that there is exactly one integer for each point in time and since rt_e of a segment is equal to rt_s from its subsequent segment, b will be equals to a of its subsequent segment. So these four variables contain redundancy and this is true for all segments except the last one, but we will come to that after the example.

Let's look now at the example from before. We wrote it as:

$s[(0 \rightsquigarrow 0, 0) \text{ at } [-\infty, 2018/1)], s[(0 \rightsquigarrow \infty, 1) \text{ at } [2018/1, \infty)]$

Even though there are only two segments, one which is constant and the other one increasing, it is not so easy to see it with a quick glance. To improve the readability, we eliminate rt_e and b from each segment and reorder the segment, so we get: $a(rt_s)s$.

The reading flow is from left to right as: *what value at what time does change with what amount*, e.g. zero at 2018/1 increasing by one. Finally, the ogint presented as a string would be written as:

$0(-\infty)0, 0(2018/1)1$

This is a lot shorter than before and still holds all the necessary information. As mentioned before, we do not see the last rt_e and the last value from the last segment. This is the only missing information, but since the last date is always ∞ and the last value is either equal to the start value of the last segment or $\pm\infty$, depending on the change value s , we can know this values with very little interpretation. In listing 1 is the actual implementation of the *ogint* data type in the PostgreSQL kernel written in C code.

We store the set of ongoing integers as a list. Therefore, a fourth value is stored in the *struct*, which is a pointer to the next *ogint* of the ordered set of segments. The decision to couple the elements as a list, was made out of two reasons. First, the number of elements in an ongoing integer is not always known when we create a new one. We will look at the reason why in this chapter of the maximum and addition function. In a list we can easily add new nodes to the last node. Second, in a set of ongoing integer segments we orient us on the dates, and not on the position of the segment in the set. Therefore, when we want to know a value at a certain date, we will need to travel from the start on through each element and check the dates. Because the number of segments in a ongoing integer is rather small, we will not lose too much time going through the list.

¹When $\pm\infty$ is mentioned in the implementation part we refer to the maximum and minimum value of an `int32`, since PostgreSQL uses this as an internal representation of $\pm\infty$

```

1 typedef struct ogint{
2     // value(date)change
3     int32 value;
4     DateSingleADT date;
5     int32 change;
6     struct ogint *next;
7 } ogint;

```

Listing 1: New data type ogint

3.2 Basic Function

There are a few basic functions which are needed when integrating a new data type into the PostgreSQL kernel. For the implementation of this thesis, we chose to implement only those functions, which were necessary, so we could start working with the ogint. Therefore, only the functions to read the data type in and print it out, as well as the functions which are needed, such that the ogint can be stored in the database, were implemented. Other functions which were not implemented will be discussed in the section 3.2.3.

3.2.1 Read and Write Function

As a user you need to be able to insert an *ogint* into and retrieve it from a relation. This is done with the *ogint_in* and a *ogint_out* function. These two functions are aimed for the user and work mainly with strings. In the reading function the input string needs to be checked for the correct syntax, e.g. are the ogint separated by commas, are the braces in correct order? Second, we check for correctness of the values, are the values of two consecutive segments precisely calculated, and were correct dates inserted? Third, we check for the correct order of the dates. The write function just needs to convert the integer values back into a correct string.

3.2.2 Serialize and Deserialize

The database needs to store the newly read in data type, an internal representation for the kernel. The ogint is a new data type and therefore we needed to implement the serialization function by our self. Within the kernel, any data type is stored as a sequence of bytes. The problem is that the length of a list can vary. Therefore, an extra data type needed to be constructed which is called *varlena-header* and stores the information, how many bytes, including the header itself, belong to the stored data type. When an ogint is stored, we write after the header all ogint as a sequence of bytes. This means that we use first four bytes to store the value, then use four bytes to store the date, which is also an int32, then another four bytes for the change value and finally the last four bytes for the pointer. The order is important since we need to know which byte belong to what value when we want to deserialize the ogint.

3.2.3 Sort and Search

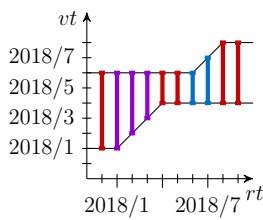
For a better control of ongoing integers in the PostgreSQL database, more functions would be needed to be implemented. Sorting is essential for a database, therefore are search and sorting functions, e.g. the binary search, essential for a proper usability of the data stored. Sorting is built upon the comparative operators and which are executed always between two entities. Comparing by itself is not a big issue but we need to find useful criteria which we can compare. An ongoing integer stores three different values, comparing two of them can be done in many ways. A simple way would be to compare the fixed value of an ogint at a specific reference date, but for this we only need to instantiate the ogint and can then use existing comparing functions. Since the ogint is defined over time, we could use the value of a certain time interval for comparison. For such a comparison we would need to calculate the *integral value* for this time interval. Of course, many other criteria could be used for comparison, e.g. number of segments, change value at a fixed date or the dates themselves. Future applications and users will decide what is necessary and useful.

Hashing is needed for many tasks, e.g. to build hash tables for the purpose of searching. But since we have structs which build a list and have three different values stored in them, there is not only one value we can hash. One approach would be to build a hash function which uses the values, change, date and value which are all int32 and a additional coefficient to guarantee, that two different ogint have the same hash value.

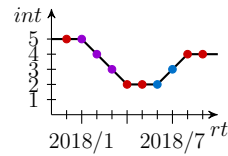
Additionally, a send and receive function would also be useful for PostgreSQL, but due to the limitation of time we focused only on the most necessary functions.

3.3 Duration Function

The duration function is an important function, since it integrates the ogint into the ongoing approach because it uses the ongoing dates as an input. The duration is measured as the difference between two fixed time points. Since we measure the duration of an ongoing time interval we get an integer value for each point in time, therefore we need an ongoing integer. For a better intuitive understanding, such an ongoing time interval and the resulting duration are illustrated in the figure 3.1.



(a) Ongoing time interval.



(b) Duration.

Figure 3.1: Evaluation of duration function *dur* on ongoing time interval [2018/1+2018/4, 2018/6+2018/8).

As mentioned in section 2.2, are time intervals, where t_e is smaller or equal t_s , empty. Therefore, if the interval would be negative, the duration is 0. The duration from the figure 5 would be described as an ongoing integer with five segments. The number of resulting segments varies, depending on the ongoing time points of the ongoing interval. It can go from one segment up to five. A general definition of the duration function is the following as in [3].

Definition 4 (Duration $dur : \Omega \times \Omega \rightarrow Z$) Let $[a+b, c+d) \in \Omega \times \Omega$ be an ongoing time interval. Let $c' = \max^F(a, c)$ and $d' = \max^F(b, d)$. The duration function dur is defined as

$$\begin{aligned}
& dur([a+b, c+d)) \\
&= \{ \mathbf{s}[(c' - a \looparrowright c' - a, 0) \text{ at } [-\infty, a)], \\
&\quad \mathbf{s}[(c' - a \looparrowright \max^F(c' - b, 0), -1) \text{ at } [a, \min^F(b, c')]], \\
&\quad \mathbf{s}[(\max^F(c' - b, 0) \looparrowright \max^F(c' - b, 0), 0) \text{ at } [\min^F(b, c'), \max^F(b, c')]], \\
&\quad \mathbf{s}[(\max^F(c' - b, 0) \looparrowright d' - b, +1) \text{ at } [\max^F(b, c'), d')], \\
&\quad \mathbf{s}[(d' - b \looparrowright d' - b, 0) \text{ at } [d', \infty)] \}
\end{aligned}$$

This definition describes all five segments which can occur when measuring the duration of an ongoing time interval. The implementation from this thesis of the duration function took a different approach than converting this definition into C code. Instead of implementing a general valid duration function, many different functions were written each corresponding to a different case. To understand the reasons behind this decision, we need to recall a few points. First, there are only four different kind of *ongoing time points*, so the number of possible combinations of two dates in an interval is limited. Second, the number of segments in an resulting ongoing integer are at least one and at most five. The chosen approach for this function goes as follows: first determine what case it is from all possible ones and then, depending on the case, build the resulting *ogint*. This led to a very simple but verbose code. So, the duration function has an if-else statement resulting in 25 cases and almost each case has its own function, e.g. for the case of two fixed ongoing time points, we only need to calculate the duration between those two dates and can then directly build a single segment *ogint* with the calculated value and the change attribute is zero. This was done mainly because of two reasons.

First, when measuring the duration between two fixed *ongoing dates* the result is a single segment *ogint*. So, when using the approach from the definition we would build first five segments and then collapsing them back into one. This is highly time consuming.

Second, the *DateADT* can contain the *infinity*-values which always need a special treatment since it is possible to have an integer overflow. This is because the *infinity*-values are represented as the maximum and minimum values of an *int32*. To guarantee that there is no overflow, a lot of if statement would be necessary and to determine the case in advance there are at most five if statements.

3.4 Addition Function

Adding two *ongoing integers* is not very different from adding fixed integers, since it is just an addition of two integers at every possible reference time. Since an ongoing integer describes all values from its start point until its end point through the integer value and the change value, we only need to add the values of the edge points when a new segment starts. So, every time we get to a new ogint node, we add the corresponding value of the other ogint at the same reference time to its own value and we add their change values together. In the algorithm 1 is the algorithm used to calculate the addition of two ogint.

Let's apply algorithm 1 to an example. We use a two ogint which both have two segments.

```

Procedure: Addition  $i_1 + i_2$ 
Input:  $i_1, i_2 \in Z$ : two ongoing integers
Output:  $i_r \in Z$ : ongoing integer.
1  $i_r = []$ ;
2  $s_j \leftarrow i_1.first$ ;
3  $s_k \leftarrow i_2.first$ ;
4  $change_j \leftarrow s_j.change$ ;
5  $change_k \leftarrow s_k.change$ ;
6 while  $s_j \neq \omega \wedge s_k \neq \omega$  do
7   if  $s_j.date < s_k.date$  then
8      $s_r \leftarrow add(s_j, s_k, change_k)$ ;
9      $i_r.append(s_r)$ ;
10     $change_j \leftarrow s_j.change$ ;
11     $s_j \leftarrow i_1.next$ ;
12  else if  $s_k.date < s_j.date$  then
13     $s_r \leftarrow add(s_k, s_j, change_j)$ ;
14     $i_r.append(s_r)$ ;
15     $change_k \leftarrow s_k.change$ ;
16     $s_k \leftarrow i_2.next$ ;
17  else
18     $s_r \leftarrow s_k + s_j$ ;
19     $i_r.append(s_r)$ ;
20     $change_k \leftarrow s_k.change$   $change_j \leftarrow s_j.change$ ;
21     $s_j \leftarrow i_1.next$   $s_k \leftarrow i_2.next$ ;
22 end
23  $buildTail(i_r, s_j, s_k, change_j, change_k)$ ;
24  $cleanUp(i_r)$ ;
25 return  $i_r$ ;

```

Algorithm 1: Algorithm to calculate the addition of two ongoing integers.

Input 1	10($-\infty$)0	10(2018/8)1
Input 2	4($-\infty$)0	4(2018/7)2

In the first iteration we get into the else clause of line 16, because the dates of the first segments are equal. So, we just add their values and changes together and create the resulting ogint.

Ogint 1	10($-\infty$)0	10(2018/7)2
Ogint 2	4($-\infty$)0	4(2018/8)1
Result	14($-\infty$)0	

In the second iteration we get to the if clause at line 6, since $2018/7 < 2018/8$. The add function the value of the second ogint at the date 2018/7 is calculated. Since we already point to the its second segment, we need the change value of the first segment from the second ogint.

Ogint 1	10(2018/7)2
Ogint 2	4(2018/8)1
Result	14(2018/4)2

In the third iteration, we go out of the while loop, because one of the two ogint is at its end. The function buildTail() runs through the rest of the segments of the second ogint and adds the corresponding value of the other ogint to its segments.

Ogint 1	
Ogint 2	4(2018/8)1
Result	16(2018/6)3

Because we do not compare the change attributes of the resulting ogint in the while loop starting at line 5, it can happen, that we build two segments with the same change value even though they are consecutive. So, we run through the resulting ogint and eliminate the second segment if we find such a pair.

3.5 Maximum Function

When applying the maximum operation for two ongoing integers we want to know which the highest value at each reference time is. In the definition, is the F -superscript for operations on fixed data types used. For instance, \min^F is the standard minimum function over fixed arguments, i.e., $\min^F(j, k) = j$ if $j < k$ and $\min^F(j, k) = k$ otherwise

Definition 5 (Maximum $\max : Z \times Z \rightarrow Z$) Let $\mathbf{i}_1, \mathbf{i}_2 \in Z$ be two ongoing integers. The maximum function \max is defined as

$$\forall rt \in \mathcal{T}(\|\max(\mathbf{i}_1, \mathbf{i}_2)\|_{rt} \equiv \max^F(\|\mathbf{i}_1\|_{rt}, \|\mathbf{i}_2\|_{rt}))$$

We implemented the max function for ongoing integers as given in algorithm 2. The complete maximum function is divided in three parts such that the function does not get too verbose. The first part

creates the first segment of the resulting ogint. This is because the first segment of an ogint contains special values like $\pm\infty$, which need to be treated differently. With the first part we also guarantee that the two input ogints and the resulting ogint have the necessary requirements, which are listed in the next paragraph, for the second part, that is described by algorithm 2. The second part builds the maximum of two ogints as long as both did not reach their last segment. Then starts the third part, where we have to use different calculations for the comparison, because we often access the values of the next ogint for comparison, what would lead to a segmentation fault. In this section we will focus on the main part which is the middle part and uses recursion.

The input for the function has the requirement that the first input must be the ongoing integer which is the current maximum value, we refer to it as *high* ogint. The other one is called *low* ogint. The *max* ogint is the resulting ogint and is either equal to the *high* one or its date is later than the *high* ones. We will come to that later in this section. After the first part we built an artificial segment such that both segments have the same date. This will be discussed in the next section

The basic idea of the algorithm is to build in each recursion step at least one segment of the resulting *max* ogint and that the *low* pointer always points to the segment which has the closest date compared to the *high* one but its date is always smaller than the date from the *high* one.

When we look at algorithm 2, we see that in line 5, we check for the result of the *intersection* function. This function returns true if the *low* ogint reaches a higher value than the *high* ogint, before the date of the next *high* segment. Note that when we say, a higher value than, we compare the two ogint always at the same reference time. So if the *low* ogint becomes the *high* ogint within the time interval of the current *high* segment, it returns true and false otherwise. When the return value is a *false*, we can add the segment from the current *high* ogint to the resulting *max* ogint.

When the *intersection* function returns *true* we need to distinguish between two cases. This is done by the function *cleanCrossing*. What we have to check for is, because we have only discrete values, it is possible that the two *crossing* segments never have the same value. For example, the ogint $0(2018/1)3$ and $9(2018/1) - 1$, will have on the date 2018/3 the values 7 and 6 and on 2018/4 it will be 6 and 9. So here we would need to build a extra segment for one time increment because from 7 to 9 the change is 2 and not 3 as in the first ogint. This is done in the algorithm by the function *buildBridge*. The other two build- functions just use the values from the new *high* ogint to build the segment at the correct date.

The algorithm advances either in the else clause in line 18 where the *high* ogint jumps to its next segment or in the *intersection* function where the *low* ogint travels through the segments. Even though it is possible, that in a iteration the two ogint do not advance with their segments, it is guaranteed, that they will in the next iteration, because after an *intersection* the two values must diverge. This means that in the next iteration either progress is made in the *intersection* or there is no intersection and the *high* ogint advances.

For a better understanding, let's look at an example of the algorithm with three iterations. The pointer name tells what segment was called as high or as low in this iteration. The arrow tells us where the low pointer will travel in this iteration. Above the table we will write the output of the intersection function. In the lowest row we write the segments which are added to the result in this iteration. The three points symbolize, that we do not start at the beginning of an ogint.

```

Procedure: Maximum  $max(i_1, i_2, i_r)$ 
Input:  $i_1, i_2 \in Z, i_r$ :  $i_r$  is not a complete ongoing integer
1  $high \leftarrow i_1$ ;
2  $low \leftarrow i_2$ ;
3  $tmp$ ;
4 if  $high = \omega \vee low = \omega$  then return  $\triangleright$  termination of the recursion ;
5 if  $intersection(high, low, max)$  then  $\triangleright$  check if high becomes low until the next segment
6   if  $cleanCrossing(high, low)$  then  $\triangleright$  check if their values become the same
7      $tmp \leftarrow buildSegFromCrossing(high, low)$ ;
8      $i_r.append(tmp)$ ;
9      $max(low, high, i_r)$  ;
10  else
11     $tmp \leftarrow buildBridge(high, low)$   $\triangleright$  build "bridge"-segment for 1 time increment
12     $i_r.append(tmp)$ ;
13     $tmp \leftarrow buildSegAfterBridge(high, low)$ ;
14     $i_r.append(tmp) max(low, high, i_r)$ ;
15  ;
16 else
17    $high \leftarrow i_1.next$ ;
18    $i_r.append(high)$ ;
19    $max(high, low, i_r)$  ;

```

Algorithm 2: Algorithm which describes the second part of the complete maximum function.

First iteration: `intersection()` -> *false*

Pointer to ogint 1	...	high		
Ogint 1	...	$10(2018/2)0$	$10(2018/5) - 2$	$0(2018/10)0$
Ogint 2	...	$2(2018/1)0$	$2(2018/3)2$	$12(2018/8)0$
Pointer to ogint 2		low		
Segment added to result			$10(2018/5) - 2$	

The `intersection` function returned *false*, so the *high* ogint jumps to the next segment and we add it to the max ogint.

Second iteration: `intersection()` -> *true*

Pointer to ogint 1			high	
Ogint 1	...	$10(2018/2)0$	$10(2018/5) - 2$	$0(2018/10)0$
Ogint 2	...	$2(2018/1)0$	$2(2018/3)2$	$12(2018/8)0$
Pointer to ogint 2	...	low	->	
Segment added to result			$8(2018/6)2$	

Within the `intersection` function the low ogint went to its next segment and detected a coming intersection. On the date 2018/6 both ogint will have the value 8 so the `cleanCrossing` function returns *true* and the new segment is added to the max ogint. Now the high and low ogint will be swapped.

Third iteration: `intersection()` -> *false*

Pointer to ogint 1			low	
Ogint 1	...	$10(2018/2)0$	$10(2018/5) - 2$	$0(2018/10)0$
Ogint 2	...	$2(2018/1)0$	$2(2018/3)2$	$12(2018/8)0$
Pointer to ogint 2			high	
Segment added to result				$12(2018/8)0$

In the third iteration both ogint did not advance, but in the next iteration we have again progress, because we swapped *high* and *low* so we know, that there is no intersection before one of both ogint will go to its next segment.

3.6 Special Points of Focus

In both, the maximum and addition function, dealing with the start and the end of the lists, lead to an extra overhead, because they needed a special treatment.

The first segment can contain values of $\pm\infty$ which will lead to an overflow in the addition, when we would add a positive value to ∞ . The solution was to add only the change values to each other and then it will be decided what value to store depending on whether the resulting change value is greater or smaller than zero. Since ∞ can only have a negative change value and vice versa for $-\infty$. If it becomes zero, we just add the value of the next segment together.

In the maximum function it does not makes sense to compare ∞ with ∞ . Another problem with the

maximum function at the beginning was that we normally calculate the value of a segment at a specific reference time with its value, date and change, but each first segment contains the date $-\infty$ so we need to access the date from the next segment and calculate it from there.

Because we do not know how many times an ogint crosses the other one within its first segment and we need a special calculation for the comparison, we get fast many different cases to cover. Therefore, we want to get past the first part of the function and come to the algorithm described before, as fast as possible. To achieve this, we checked which one of both second segment date was earlier and then built a artificial segment as this date for the other ogint. Like this both ogint are not at their first segment anymore and we can use the algorithm from before.

When one ogint reached its last segment, we get to the last part, where we just travel through the segments of the other ogint, while we check whether it crosses this segment or not.

4 Evaluation

In this chapter we will evaluate the newly implemented data type *ogint* and the additional functions on runtime performance and how the number of segments evolve within the different functions. For the evaluation of the runtime performance, we will compare the duration function with the bind approach where we use fixed time intervals to measure the duration. For the addition and the maximum, we want to know how the runtime increment correlates to the increasing number of segments.

4.1 Setup

4.1.1 Experimental Setup

The experiments were carried out on a Linux virtual machine with 64 GB RAM. The machine the VM was running on has two 2.40 GHz XEON(R) CPU. The client and the database server ran on the same server. The runtime of a query was measured with the internal explain analyze function of PostgreSQL.

4.1.2 Real-world Dataset

One dataset used for the tests comes from the Mozilla defect tracking set [2]. The relation contains 394878 tuples and the attributes used for the tests were the ID which is an integer and the valid time which is of type *daterange*. A *daterange* is the data type for an ongoing time interval and it contains two ongoing time points which are of type *date* in this the PostgreSQL database used.

4.1.3 Synthetic Dataset

Data from a real application is a good reference on how big ongoing integers could become and how fast the functions run when used in a real-world environment. To test the functions specifically for the runtime performance with different numbers of segments in the input and output, we build synthetic datasets with *ogints* of different length. The *ogints* were chosen in such a way, that they either had a rather short or long output, when used for the maximum and addition function. This is done, simply so we can see the correlation between number of ongoing integer segments and the runtime of the implemented functions. We will elucidate on what we focused while building the synthetic datasets in the next section. All synthetic datasets were build with 10 different tuples. In the tests those 10 tuples were inserted up to 10'000 times, so the resulting set had 100'000 tuples. This was done to simplify the comparison of the runtime with the duration function using the Mozilla dataset.

4.2 Tests and Results

In this section we will first analyze how the number of elements of the output can vary for the different functions. We will then look at the structure and the results of the tests from each function separately.

4.2.1 Result size

The number of segments from the output from each function depends strongly on the kind of input. For the duration function we know quite well how many segments the output will have depending on the input. For each possible case the number of segments can be determined beforehand. There is a limit to the possible combinations of two ongoing time points, since we have a limit of different types, fixed, limited, growing, now and a general ongoing time point. The minimum of segments in all those combinations is one and the maximum is five.

For the addition and maximum function this is a little bit different. To describe the number of segments within an ongoing integer we use the notation of: $|\mathbf{i}|_{seg}$

When we add to ogint together the maximal number of segments which can occur is:

$$\text{Maximum of } |\mathbf{i}_r|_{seg} = |\mathbf{i}_1|_{seg} + |\mathbf{i}_2|_{seg} - 1$$

This is, because for each segment from the input, we need to build one in the output, since at every with every segment a shift in the change value happens. When all segments start on a different date, we will get as many segments in the output as in the input combined. But because the first segment always start at $-\infty$ the first segments from the input will merge, therefore the minus one.

The minimum is one segment but only for a *very special* case. This is possible when two ogint have the same date in each segment and the addition of their change values is equal to the change value of the preceding result-segment.

While adding two ogints, for each segment which have the same date value, two segments will be merged together to one. So, the result size for the addition depends on whether the input data is more likely to have similar dates or not.

The result size of the maximum function depends on how often they alternate in being the maximum. When they never alternate and the ogint with less segments is the maximum we get the minimum of the possible result size. So, the minimum is:

$$\text{Minimum of } |\mathbf{i}_r|_{seg} = \min(|\mathbf{i}_1|_{seg}, |\mathbf{i}_2|_{seg})$$

But for each time the maximum ogint alternates we get an additional segment for the result. When the discrete values not match on a certain date we get even two segments, since we need to build an extra segment to "bridge" to the new maximum. So the maximum of the result size is:

$$\text{Maximum of } |\mathbf{i}_r|_{seg} = \max(|\mathbf{i}_1|_{seg}, |\mathbf{i}_2|_{seg}) + 2 * \min(|\mathbf{i}_1|_{seg}, |\mathbf{i}_2|_{seg})$$

4.2.2 Duration Function

To test the duration function, we used both, synthetic and real-world datasets. Unlike the addition and maximum function, we could compare the duration function with the bind approach, since the input are not ogints. To compare the bind with the ongoing approach, we used the real-world dataset, to see how they would perform in a real-world environment. The bind approach has to use the bind operator for the upper and the lower *date* attribute which are stored in the *daterange* data type. Then we can subtract the resulting fixed dates from each other. This means that with the bind approach the result will be a fixed integer and not an ongoing integer. The query used is shown in listing 2. To compare the two different

results we just look at, after how many refreshments of the bind approach, the result of the ongoing approach becomes faster. In the queries, the *vt* attribute is the valid time of type *daterange* with the *lower()* and *upper()* command we can access the single *dates*. 2018/1/1 was chosen as reference time point for the bind approach.

```
1  select
2  (date_bind(upper(vt), '2018-01-01')
3  -
4  date_bind(lower(vt), '2018-01-01'))
5  from mozillaset;
```

Listing 2: Bind approach

```
1  select date_dur(lower(vt), upper(vt))
2  from mozillaset;
```

Listing 3: Ongoing approach

The results from these two queries are illustrated in the figure 4.1. We can see, that the ongoing approach is as fast as the bind approach. The duration in the bind approach is measured with a single subtraction of the dates, which is fast operation. In the bind approach we need to do the bind operation twice. This is where the bind approach loses time. The absence of the bind operation in the duration function, makes up the overhead of comparison and building ogint.

The results show, that when we are using ongoing dates, there is no overhead compared to the bind approach, when measuring the duration of an ongoing interval.

Because we do not only want to know how the duration function performs with the one real-world dataset we used, but also what are the possible upper and lower boundaries of the runtime of the function, we did also tests with synthetic datasets. e used different synthetic datasets where each holds a certain type of ongoing time points. With those datasets we want to see how large the difference is between measuring the duration of e.g. two fixed ongoing time points where the result is an ogint with only one segment and two ongoing time points, where the resulting ogint holds five segments. We decided to use different types of combinations to see how large the difference between short and long resulting ogints are, so we can have a upper and lower boundary for the duration function.

We used six different synthetic datasets with the most common combination of ongoing time points, the most expensive and the cheapest ones. Common combinations are e.g. two fixed time points and a fixed and open one. These combinations were the biggest part of the Mozilla dataset.

The query used, was the same as in the query of listing 3. The legend says what types of ongoing time points were used for the dataset. The Zero dataset holds ongoing time intervals that result in an

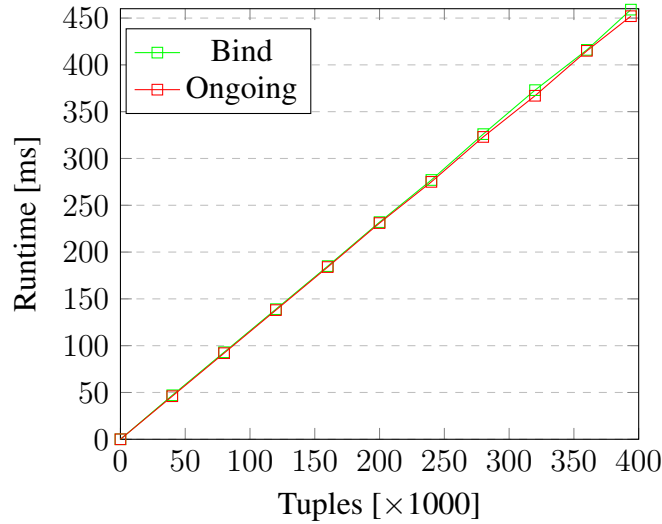


Figure 4.1: Comparison of ongoing and bind approach. Duration function with Mozilla dataset

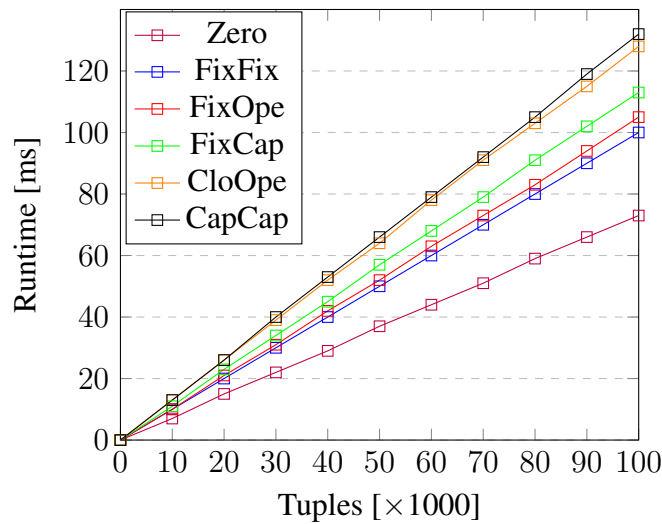


Figure 4.2: Duration function with synthetic dataset

empty interval when instantiated and therefore return a single segment ogint with the value 0. The results from the different sets return ogint with a different amount of segment. The Zero and FixFix dataset return one segment, the FixOpe returns 2 and FixCap and CloOpe returned each 3. The set CapCap returns 5 segments and this is also the maximum number of possible segments for the duration function. We see clearly that the more segments the result has, the bigger the runtime. But the runtime from two captured ongoing time points is roughly 30% slower than the one from two fixed time points even though it has five times the number of segments in the result. The results of the runtime from the Zero set were around 30% faster than the ones from the FixFix set even though it has the same number of segments in the result. Therefore, not only the building of segments is responsible for the runtime increment but also the calculation of its values. In the Zero set we do not calculate the value, we go through if statements and then build directly a segment with the value 0.

When we compare the runtime of the Mozilla set with the synthetic ones, we see that with 100'000 tuples the runtime of the Mozilla set is at 115ms and the average of the synthetic sets is 109ms, and without the zero set the average is 116ms. Therefore, the Mozilla dataset is a good reference for the runtime of the duration function and other sets should not differ too much in the runtime performance.

4.2.3 Addition Function

For the addition function we used only synthetic datasets for the tests, because our focus was not on the comparison with a bind approach but rather on how much the performance changes with different input. So we can better estimate how runtime performs when the number of segments increases. For these tests we made two sets of synthetic datasets. In each set we used five different datasets containing ogints with only one and up to five segments. We chose five segments because this is the maximum number of segments which can result in a duration function. So each dataset with a certain size of ogint in it, has a corresponding dataset in the other set. The difference between them is the size of the resulting ogint. Whereas one set of datasets return ogint with the same number of segments as the input ogint has, does the other one return an ogint with the maximum of possible segments. The queries were only simple select query of the form: `select add(ogint1 ,ogint2) from table ;`

The results are shown in the figure 4.3

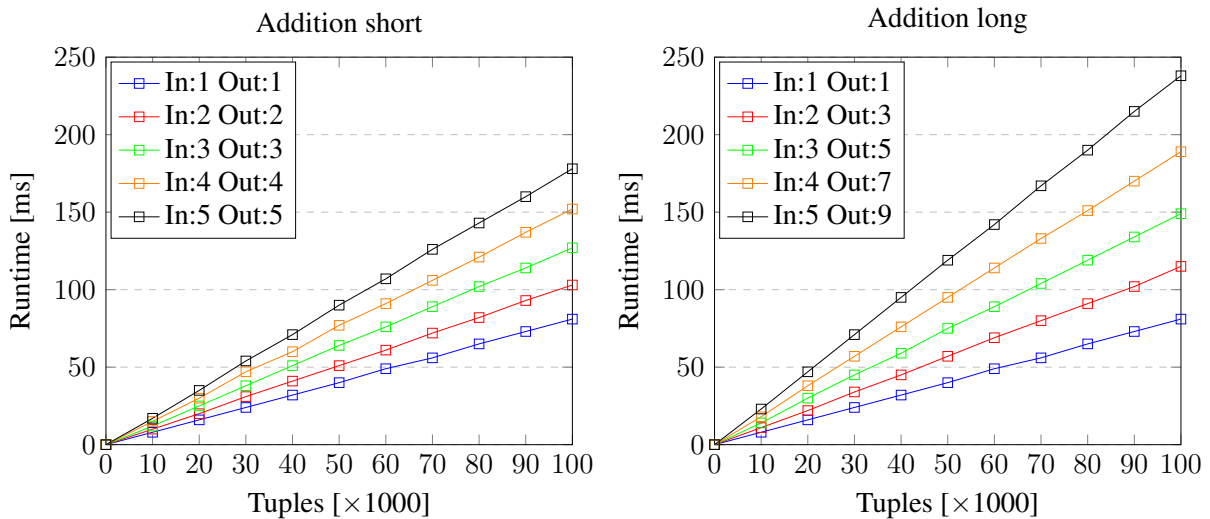


Figure 4.3: Addition function with synthetic datasets

When we compare the runtime of each dataset with its corresponding dataset from the other diagram, we see that not all change with the same amount. The two datasets with only one segment, have of course no change in the runtime, since the number of segments of the output is the same. For the others, the output has almost double the number of segments as in the corresponding short dataset. The runtime increments from short to long, for the *five segment*-dataset is around 30% whereas for the *two segment*-dataset, it is only around 10%. This is because the relative amount of additional segments, which are built, increases with the amount of input segments. This means, that compared to the short dataset, in the long dataset the two segment ogint had to build only 1 more segment which is 50% from the input, whereas the ogint with five input segments needed to build 4 additional segments which is

equal to 80% of the number of segments from the input.

So, the main cause for the runtime increment is each segment which needs to be built and checked.

4.2.4 Maximum Function

The tests for the maximum function were built quite similarly as for the addition function. We also used two sets of synthetic datasets with different ogints in it. The goal was also to show how the runtime changes with a larger output¹. The query used was also of the form:

```
select max(ogint1,ogint2) from table ;
```

In the legend the average number of segments from the input and the output are listed.

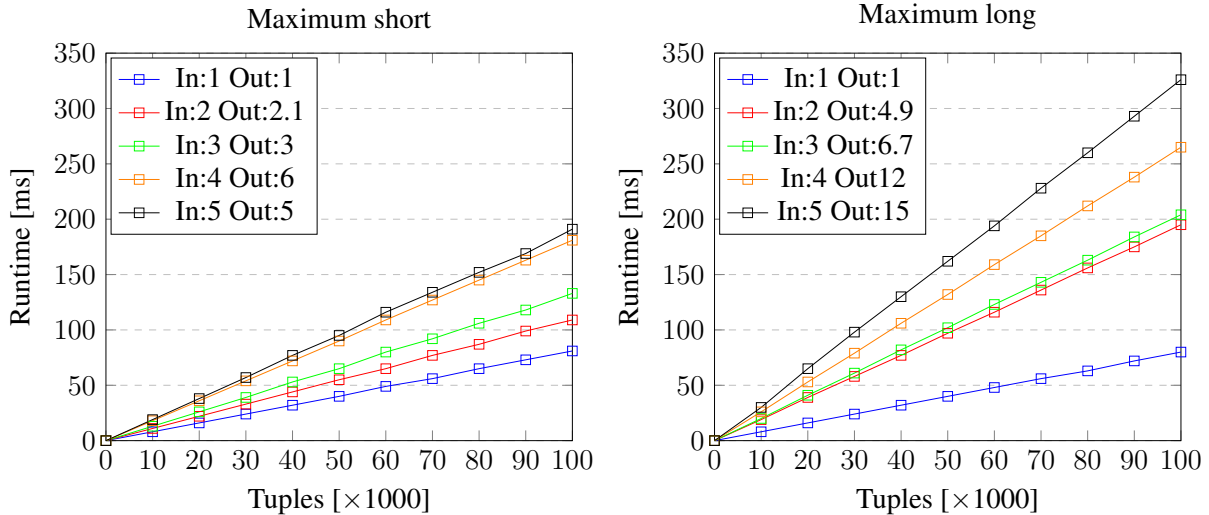


Figure 4.4: Maximum function with synthetic datasets

We see that the runtime increase is more than with the addition function. As mentioned in the result size section, we know that the maximum number of resulting segments is $\max(|\mathbf{i}_1|_{seg}, |\mathbf{i}_2|_{seg}) + 2 * \min(|\mathbf{i}_1|_{seg}, |\mathbf{i}_2|_{seg})$, since our input ogint always had the same amount of segments it is just $3 * |\mathbf{i}|_{seg}$. In the short dataset, the runtime of the maximum function was similar to the one of the addition, whereas in the long set they differ a lot. Obviously, the larger number of resulting segments lead to the time increase.

The synthetic data was built for the purpose to reach the maximum of possible segments, so we could get a upper boundary for the runtime. This was achieved by ogints which switched with being the maximum in each segment. Such ogints are very special and it will depend highly on the data used how many times two ogints will switch for the maximum.

¹Our implemented of the maximum function returns ongoing integers with too many segments for a special case. Because of a time bottleneck, the bug could not be resolved. The results can still be used to show a tendency how the results evolve, since only the results of two datasets are affected.

5 Conclusions and Future Work

The goal of this thesis was to integrate a new data type of an ongoing integer into the widely-used database system PostgreSQL. This new data type came from a new *ongoing* approach to deal with so-called *ongoing time points* as for example *now*. This new approach is in comparison with the *bind* approach where we use the current time stamp when we face a *now* in our database. Since we also want to measure the duration of an *ongoing time interval* a new data type was needed to describe such a result. To have a correct result an *ongoing integer* was invented by [3], so one can display and work with those results.

In this thesis the *ongoing integer* which is a set of consecutive *ongoing integer segments* was implemented as a *ogint* struct in C code. This *ogint* is a simplified version of an *ongoing integer segment* which also holds a pointer to its successor, such that it is built as a list of segments. The most basic functions to be able to handle this new data type were implemented. Since one of the needs was to measure the duration of an *ongoing interval*, accordingly such a function was implemented and compared to the current *bind approach*. To further test the new data type for overhead two aggregation functions, the addition and the maximum, were implemented and tested as well.

The implemented duration function had a runtime which was equal to the current *bind approach*, where the dates of the ongoing interval needed to be bound to a fixed date and then were subtracted. Because with ongoing data types we do not need to refresh our results with every time increment it becomes faster than the *bind approach* with every additional refreshment cycle. The tests with the addition and maximum function showed that the main contributor to a runtime increase is each segment from the input and the output. The minimum number of segments for the addition from a resulting *ogint* is 1 and the maximum is $|\mathbf{i}_r|_{seg} = |\mathbf{i}_1|_{seg} + |\mathbf{i}_2|_{seg} - 1$. For the maximum function the numbers of segments from the result is bound by the minimum of $|\mathbf{i}_r|_{seg} = \min(|\mathbf{i}_1|_{seg}, |\mathbf{i}_2|_{seg})$ and a maximum of $|\mathbf{i}_r|_{seg} = \max(|\mathbf{i}_1|_{seg}, |\mathbf{i}_2|_{seg}) + 2 * \min(|\mathbf{i}_1|_{seg}, |\mathbf{i}_2|_{seg})$.

The next steps to have a full functioning data type would be to implement more functions, so we can work properly with the *ogint*. As in the thesis already mentioned are comparison and hashing function an important feature for a data type in a database. Also, additional functions as for example the subtraction and the minimum are necessary to have a closure for the *ogint*.

The ongoing integer complements the *ongoing approach* and together with other *ongoing data types* it can help to solve the problem of results that do not remain valid as time passes by.

Bibliography

- [1] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the semantics of “now” in databases. *ACM Transactions on Database Systems (TODS)*, 22(2):171–214, 1997.
- [2] A. Lamkanfi, J. Perez, and S. Demeyer. The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information. In *MSR '13: Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 203–206, 2013.
- [3] Y. Mülle and M. H. Böhlen. Durations in Ongoing Databases. to be published.
- [4] Y. Mülle and M. H. Böhlen. Query Results over Ongoing Databases that Remain Valid as Time Passes By. to be published.