Department of Informatics, University of Zürich

BSc Thesis

Temporal Integrity Constraints in Databases With Ongoing Timestamps

Jasmin Ebner

Matrikelnummer: 13-747-159

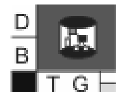
Email: jasmin.ebner@uzh.ch

August 22, 2016

supervised by Prof. Dr. Michael Böhlen and Yvonne Mülle

University of Zurich[UZH]

Department of Informatics



Acknowledgements

First of all, I want to thank my supervisor Yvonne Mülle who gave me advice and support during the whole time. A big thanks to Prof. Dr. Michael Böhlen, head of the database technology group who gave me the opportunity for this thesis. I also want to thank my family and friends who are supporting me all the time.

Abstract

Temporal database systems include time varying data. Like that, one has the ability to record when a tuple is supposed to be valid. This is necessary in many real-world use cases. An example is an insurance company which wants to know, when a customer is insured and under what conditions. As it is often not clear, how long a customer relation might finally last, it can be desirable to be able to store that the insurance contract is valid until *NOW*. In this case, ongoing timestamps are used. Database systems in general provide support for the two integrity constraints primary key and foreign key. These constraints can be adapted to temporal database systems. Then the constraints must be fulfilled independently at every point in time [6]. When working with temporal database systems that include ongoing timestamps, the constraints cannot only be violated when a relation is modified, this includes insertion, update and deletion of tuples, but also when time advances. We call those constraints *Ongoing Primary Key* and *Ongoing Foreign Key*. As none of the existing database systems currently supports temporal integrity constraints with ongoing timestamps, the objective of this thesis is to integrate these constraints into the kernel of the widely used DBS PostgreSQL. In this report the ongoing integrity constraints are defined, the implementation approach is explained and an overall evaluation of the solution is made.

Zusammenfassung

In temporalen Datenbanksystemen hat man die Möglichkeit, anzugeben, wann ein Tupel gültig ist. In vielen Anwendungen im täglichen Leben ist das nötig. Beispielsweise in einer Versicherungsgesellschaft, welche wissen möchte, zu welcher Zeit und mit welchem Angebot ein Kunde versichert ist. Oftmals ist dabei nicht klar, wie lange der Kunde schlussendlich bei der Versicherung bleibt. Daher kann es wünschenswert sein, im Gültigkeitsbereich anzugeben, dass das Tupel bis zum aktuellen Zeitpunkt gültig sein soll. Der Zeitpunkt passt sich dabei fortlaufend an die aktuelle Zeit an. In diesem Fall arbeitet das Datenbanksystem mit sogenannten *ongoing*, also fortlaufenden, Zeitstempeln. Jedes grössere Datenbanksystem implementiert die Integritätsbedingungen Primärschlüssel und Fremdschlüssel. Wenn man diese Bedingungen an temporale Datenbanken anpassen will, muss die Bedingung zu jedem Zeitpunkt an dem das Tupel gültig ist, erfüllt sein. Braucht man zusätzlich die oben beschriebenen fortlaufenden Zeitstempel, kann eine Verletzung der Bedingung nicht nur dann erfolgen, wenn man die Datenbank modifiziert, sondern auch wenn die Zeit voranschreitet. Man spricht in diesem Fall von *fortlaufenden Primär- und Fremdschlüsseln*. Da zur Zeit keines der Datenbanksysteme Support für diese fortlaufenden temporalen Bedingungen anbietet, ist das Ziel dieser Arbeit, den PostgreSQL Kernel zu erweitern, so dass die Integritätsschlüssel als Bedingungen auf einer Relation gesetzt werden können. Dieser Report definiert die beiden Schlüssel, zudem dokumentiert er die Implementation und gibt eine Auswertung der Lösung.

Contents

1 Introduction **9**
1.1 Problem Statement 9
1.2 Motivation 9
1.3 Thesis Outline 10

2 Related Work **11**

3 Ongoing Key Constraints **13**
3.1 Application Scenario 13
3.2 Time Domain 15
3.3 Ongoing Primary Key Constraint 17
3.3.1 Definition 17
3.3.2 Violation 19
3.3.3 Resolving Strategy 19
3.4 Ongoing Foreign Key Constraint 20
3.4.1 Definition 20
3.4.2 Violation 21
3.4.3 Resolving Strategy 22
3.5 Overview 23
3.6 Example 23
3.6.1 Illustration of the Time Domain 23
3.6.2 Illustration of the Ongoing Primary Key 25
3.6.3 Illustration of the Ongoing Foreign Key 26

4 Implementation Approach **29**
4.1 Internal Implementation 29
4.1.1 Creation of Ongoing Constraint Tables 29
4.1.2 Ongoing Primary Key Constraint 31
4.1.3 Ongoing Foreign Key Constraint 34
4.2 Implementation with User Defined Triggers 37
4.2.1 Creation of Ongoing Constraint Tables 37
4.2.2 Ongoing Primary Key Constraint 38
4.2.3 Ongoing Foreign Key Constraint 40

5 Experimental Evaluation **45**
5.1 Experiment Setup 45
5.1.1 Relations 45

5.1.2 Data Set 45
5.1.3 Test Case Selection 45
5.2 Test Results and Evaluation 49

6 Conclusions and Future Work 60

List of Figures

3.1 Violation of the OPK constraint when time advanced 14
3.2 Unlimited ongoing timestamp in upper bound 16
3.3 Limited ongoing timestamp in upper bound 16
3.4 Overlapping of two timeframes at least at one point in time 18
3.5 Fixed lower and unlimited ongoing upper bound. 24
3.6 Fixed lower and limited ongoing upper bound. 24
3.7 OPK Violation while trying to insert the red tuple. 25
3.8 No OPK Violation after updating old tuple 26
3.9 OFK violation, because the timeframe is not fully covered 27
3.10 OFK violation, because the timeframe is not fully covered 28

5.1 Measurements Test Case 3 54
5.2 Measurements Test Case 4.1 55
5.3 Measurements Test Case 4.2 56
5.4 Measurements Test Case 5 57
5.5 Measurements Test Case 6 57

List of Tables

3.1 Initial product relation 13
3.2 Initial customer relation 15
3.3 Overview of OPK and OFK violation cases 23
3.4 Initial available products 23
3.5 Initial customer relation 23
3.6 Initial available products 25
3.7 Initial customer relation 26

5.1 Overview Test Case 2 51
5.2 Overview - a few tuples to check vs. many tuples to check 51
5.3 Overview - fixed timestamps vs. ongoing timestamps 52
5.4 Overview Test Case 2 52
5.5 Overview Test Case 2.4 and Test Case 3 54

6.1 Products in referenced relation - the blue one should be deleted 62
6.2 Customer referencing a product to be deleted 62

1 Introduction

1.1 Problem Statement

In contrast to non-temporal database systems, temporal database systems include time varying data [3]. This means, that one can specify through a temporal attribute when a tuple is valid. This is necessary in many real-world use cases, like for example an insurance company which wants to know when a customer is insured and under what conditions. But often, it is not clear how long a customer relation will finally last and it can be desirable to simply say that the insurance contract is valid until *NOW*. In this case ongoing timestamps must be used. They will be defined in chapter 3.

Database systems in general provide support for the two integrity constraints primary key (that ensures that no more than one tuple with the same primary key attributes exist) and foreign key (that points to such a primary key). For every modification on a relation, it must be ensured, that no violation occurs. Modifications do in this report always include insertions, updates and deletions of tuples. These constraints can be adapted to temporal database systems. In this case the constraints must fulfil the conditions for the whole valid time of the tuple [6]. But when working with temporal database systems that include ongoing timestamps, the constraints cannot only be violated when a relation is modified, but also when time advances. In this case, the constraints will be named *Ongoing Primary Key* and *Ongoing Foreign Key*. For simplicity, these terms are abbreviated to OPK respectively to OFK in this report.

Currently, widely used database systems like PostgreSQL, MySQL and Oracle provide no native support for temporal integrity constraints. Therefore the object of this thesis is, to integrate a solution for temporal integrity constraints with ongoing timestamps into one of these systems, namely the PostgreSQL kernel.

1.2 Motivation

Ongoing timestamps and with them the ability to implement ongoing integrity constraints bring several benefits. As mentioned, ongoing timestamps are necessary if it is not clear, how long a tuple will be valid in the future. Imagine that one would have to adapt the valid time of these tuples manually and steadily to the current time. This would be an extremely costly approach. But as ongoing timestamps do this in an automated way, a lot of work can be saved.

When using ongoing integrity constraints, the database system can be used in a more flexible way. Relations that have these constraints can not only record the current state of a system but also its history and a potential future. Imagine, a relation records all customers and the products they are insured with. The product is stored by a number that serves as the - either non-temporal or ongoing - primary key attribute and has an additional attribute that defines

the premium of the product. When that premium does change, the tuple in the non-temporal primary key relation must be updated and the product must be changed. All information about the product before the update are lost. It is not possible to insert a new tuple for the product, as this would lead to a primary key violation. But in temporal databases where an additional attribute specifies when a tuple is supposed to be valid, another tuple for the same product can be added - presupposed, their valid times do not overlap. This way, the history is recorded.

As mentioned, widely used DBMS do not yet support these ongoing constraints. If someone would like to work with them, he or she is currently supposed to work out an own solution. This may happen through adding triggers to a relation that check the constraints and abort the statement if necessary. Not only is this connected to more work, but the user is also responsible for the correctness of the implemented constraint logic themselves. Additionally, the logic must be replicated for every new relation. This makes the user defined approach more error-prone. Therefore, simplicity, convenience and stability are additional reasons for the integration of the ongoing constraints into the PostgreSQL kernel made in this thesis.

As the ongoing integrity constraints and with it the recording of history are a real-world use-case, it may be surprising that currently big DBMS do not support them. A possible reason could be, that the handling of temporal data in general is still relatively new in SQL, as it is only available with SQL:2011 [2]. Furthermore, using ongoing timestamps instead of fixed ones leads indeed to more flexibility but also to additional challenges. The support for the variable *NOW* as an ever increasing timestamps first needs to be implemented into the DBMS[1]. In this thesis, the implementation of the ongoing integrity constraint was possible because the support for the variable *NOW* has already been researched and implemented by Yvonne Mülle.

1.3 Thesis Outline

The remainder of this thesis is structured as follows. In chapter 2 an overview to related work of this topic is given. In chapter 3 an application scenario is provided that will be used in the whole thesis. Besides, relevant terms are defined, the violation cases of the ongoing integrity constraints are identified and the constraints are illustrated on the application scenario. Chapter 4 describes the implementation approach that was used to provide native support for the two ongoing integrity constraints as well as an approach using user defined triggers. The implementation is then evaluated in chapter 5. A conclusion as well as open issues for future work are provided in chapter 6.

2 Related Work

There have been several researches about temporal integrity constraint checking and about how to handle those constraints. Some of them are mentioned here. All approaches mentioned do not support integrity checking for ongoing timestamps but only for fixed ones.

Temporal primary key (TPK) and temporal foreign key (TFK) constraints have been recently added to the temporal features of the SQL:2011 standard [2]. In SQL:2011, tables that contain an application-time period provide support for valid time.

The TPK constraint in SQL:2011 forbids overlapping application-time periods for the specified key attribute. The TFK constraint in SQL:2011 ensures that the application-time of a period of a row in the child table must be contained in one application-time period of the parent table or in the union of two or more contiguous application-time periods in the parent table. Thereby, the child table is the table with the TFK constraint that references the parent table, that is the one with the TPK constraint.

Li et al. provide a database independent approach to check if a temporal key violation occurs [6]. This means, the underlying database management system does not have to be changed. Instead, Li et al. propose implementing TPK and TFK constraints on top of a database system with user defined triggers. Like this, the constraints do only have to be checked when a modification on the relation is made.

The approach is based on a B^+-tree. For a relation with a TPK constraint, a B^+-tree index is declared, with an index key of the key attributes coupled with a date attribute and a date type. The date type indicates if the date is the start or end date of a time interval. For every tuple in the relation, two unique indexes are created, one containing the start and one containing the end date.

Li et al. provide an algorithm to check if the TPK constraint is still fulfilled when inserting a new tuple into the relation. The algorithm can also be adapted to check the TPK constraint when a tuple is updated. The algorithm first searches the B^+-tree to find an index entry that has the same key attributes as the key attributes of the tuple to be inserted and whose date value is between the start and the end date of the tuple to be inserted. When such a tuple is found, then it means that there does for sure exist a tuple in the relation whose time interval overlaps with the time interval of the tuple to be inserted and the insertion must be rejected.

But this check does not verify that there does not exist a tuple in the relation that has the same key attributes and whose time interval does contain the whole time interval of the tuple to be inserted. To verify that, the algorithm searches the B^+-tree to find the last index entry (in time) whose date value is less than the start date of the tuple to be inserted and whose key attributes are the same as the ones of the tuple to be inserted. If such an entry it found and its date type is a start date then it means, that the corresponding end date is greater than the end date of the tuple to be inserted. In this case also a violation occurs and therefore the insertion

must be rejected. Otherwise the tuple can be inserted.

Li et al. extend the algorithm to TFK checking. When a tuple into a relation with a TFK is inserted, the algorithm searches the B^+-tree to find the last index entry in time whose date value is less than the start date of the tuple to be inserted. It then iterates the B^+-tree until an index entry is found, whose date value is greater than the end date of the tuple to be inserted. If such an entry is found, it must also be assured, that no gaps between the found entries exist. Because otherwise, a violation of the TFK constraint would occur.

There is also a report from IBM written by Nicola and Sommerlandt that shows how temporal integrity constraints can be handled in the DBMS DB2 using SQL [4]. For the TPK constraint, DB2 provides native support. The user has the possibility to extend the primary key by a temporal attribute (in DB2 this is a period) and the key word *WITHOUT OVERLAPS*. Like this, DB2 ensures, that no two tuples with the same non-temporal key attribute values exist whose periods do overlap in time.

For the TFK constraint, DB2 does not provide native support. Nicola and Sommerlandt show two different approaches to check the TFK constraint, one writing user defined triggers and one writing stored procedures. The alternative of stored procedures provides the advantage that it may be more efficient to check the TFK constraint when many rows in a relation are modified.

As it can be seen, there are several researches about the topic of *temporal integrity constraints*. The temporal integrity constraints were handled in different database systems either by writing user defined triggers that do not change the underlying database system or by providing native support for the constraints. But none of the researches defines the temporal integrity constraints using ongoing timestamps. Especially a possible implementation approach is missing.

In this report, the temporal integrity constraints will be extended to be able to work with ongoing timestamps. Therefore, the integrity constraints using ongoing timestamps first have to be defined. The constraints are integrated into the PostgreSQL kernel to provide native support. Native support for temporal integrity constraints is also provided in the approach used in SQL:2011 [2] and for the TPK also in the approach used in DB2 [4]. The other approaches are based on user defined triggers.

In this thesis, the integration into the PostgreSQL kernel is based on internal triggers, which check if a violation of the ongoing integrity constraints occurs when a modification statement is executed and, if so, reject the statement. Internal means, that the triggers are directly integrated into the PostgreSQL kernel. Triggers for the constraint checking are also written in the approach of Li et al. [6] and for the TPK in the approach of Nicola and Sommerlandt [4]. But the triggers they provide are user defined and not internal.

3 Ongoing Key Constraints

In this chapter an application scenario is provided that will be used in the whole thesis. Additionally, relevant terms are defined. This includes the definition of the *time domain* and with it the definition of *ongoing timestamps*, as well as *ongoing primary keys (OPK)* and *ongoing foreign keys (OFK)*. To exemplify, the time domain and the constraints will be illustrated on the application scenario at the end of the chapter.

3.1 Application Scenario

To exemplify the constraints, the application scenario of an insurance company - let's call it Avivo - is given. Avivo stores all information about the products they offer in a relation called *Product* and all information about their customers in a relation called *Customer*.

A tuple in the products relation consists of multiply attributes. First it has an *id* by which the product is identified. It does also have a *name*, because it is easier to distinguish products by a name than by a number. Then every tuple has a *premium*. For simplicity, this premium is independent of the insured sum of a customer. The last attribute on the tuple is the *timeframe* attribute. It indicates from when to when the tuple is valid.

The customer relation contains all customers that are insured with the insurance company. A tuple in this relation has a *customerId* to identify the customer. As the insurance company needs to know what product the customer is insured with, the product is referenced on the customer tuple through the *productId*. Then every tuple does also include the attribute *insuredSum*, which indicates up to what sum the customer is insured. Also in the customer relation the *timeframe* attribute indicates from when to when the tuple is valid.

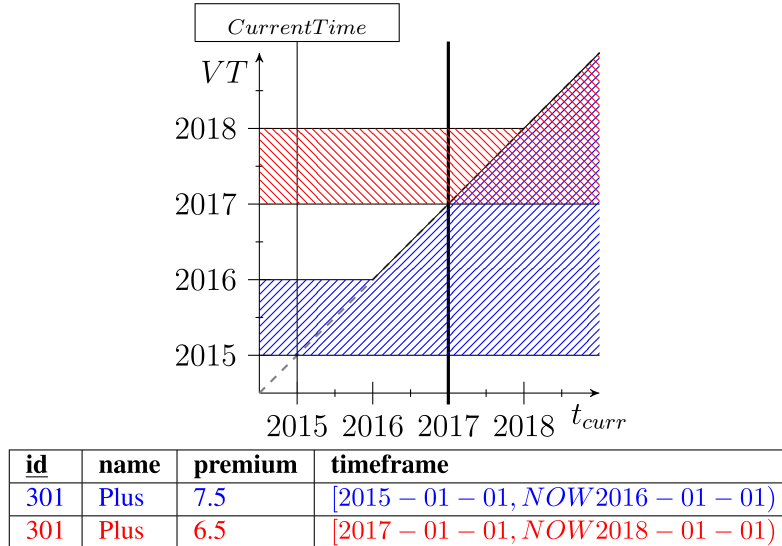
On the *id* of the product relation an OPK is created. This means, it is not allowed that more than one tuple with the same *id* is valid at any point in time. To illustrate, it is assumed, that the product relation is in the state shown in table 3.1.

id	**name**	**premium**	**timeframe**
300	Standard	4.5	$[2015-01-01, NOW2016-01-01)$
301	Plus	7.5	$[2015-01-01, min2017-01-01NOW2016-01-01)$
301	Plus	6.5	$[2017-01-01, NOW2018-01-01)$

Table 3.1: Initial product relation

Table 3.1 shows that Avivo (the insurance company) currently offers two products, "Standard" and "Plus". The standard product has a fixed premium of 4.5% that the company does not plan to change in the near future. As it is the only tuple with the *id* 300, the OPK constraint is fulfilled.

The plus product on the other hand is stored two times in the relation. First, the product has a premium of 7.5%. But the product has not been profitable enough for the company because not enough customers were interested in it. Therefore, Avivo decides to lower the premium to 6.5% at the beginning of 2017. In this case, the product "Standard" with the *id* 301 is stored two times in the relation, but as one of them is valid until 2017 and one from 2017 on, the OPK constraint is fulfilled.



id	name	premium	timeframe
301	Plus	7.5	$[2015-01-01, NOW 2016-01-01)$
301	Plus	6.5	$[2017-01-01, NOW 2018-01-01)$

Figure 3.1: Violation of the OPK constraint when time advanced

Assume the current year is 2016 and the timeframe of the product "Plus" with a premium of 7.5% is valid longer than only until 2017. Then assume a new tuple for the product "Plus" with a premium of 6.5% should be inserted into the relation. This new tuple is valid from 2017 on. As it can be seen in Figure 3.1, at the current time, no violation of the OPK constraint occurs. The constraint is fulfilled, as long as the current time is less than 2017, but afterwards, the constraint will be violated, because from then on, for the product "Plus" with *id* 301, two tuples will be valid. Therefore, the new tuple should not be allowed to be inserted into the relation, as it would lead to a postponed violation. This means, that a violation of the constraint can not only occur when a modification on a relation is made, but also when time advances.

So the OPK on the product's *id* makes sense, because it ensures that there does not exist multiply information about one single product for any point in time. If this constraint would not exist then it could happen that for some point in time, on one product, multiply premiums are recorded and it would be unclear, what premium the customers do have to pay.

On the other hand, on the *productId* of the customer relation an OFK is created that references the *id* in the product relation. This means, on a customer relation, there can only exist tuples having a *productId* for that actually one or more tuples in the product relation exist with the same *id* and the whole timeframe of the customer tuple must be covered by timeframes of the tuples from the product relation with the same *id* as the *productId*. To illustrate, it is assumed, that the customer relation is in the state shown in table 3.2.

customerId	productId	insuredSum	timeframe
C-767	300	1,000	$[2015-01-01, NOW 2016-01-01)$
C-769	300	2,000	$[2015-01-01, 2016-01-01)$
C-843	300	2,000	$[2016-01-01, min 2019-01-01 NOW 2018-01-01)$
C-900	301	4,000	$[2015-01-01, min 2018-01-01 NOW 2015-01-01)$

Table 3.2: Initial customer relation

Table 3.2 shows that Avivo currently insures four customers. In the current state of the relation, no violation occurs because all *productId*'s (300 and 301) listed in the customer relation do exist in the product relation. Furthermore, all timeframes listed in the customer relation are fully covered by tuples with the same key attribute values in the products relation.

The OFK constraint on the *productId* on the customer relation makes sense, because it ensures that for every point in time when the customers are insured, they are insured with products that do actually exist.

It would also make sense to create an OPK constraint on the *customerId* of the customer, because it should not be allowed that two customers with the same *customerId* exist. But for simplicity, this constraint is omitted in this thesis on the customer relation.

Summed up, the relations have the schemas:

- Product(id, name, premium, timeframe)
- Customer(customerId, productId *references Product(id)*, insuredSum , timeframe)

In the product schema, the underlined attribute (that could for another use case also be more than one) refers to the key attribute of the OPK. The timeframe is automatically a part of the ongoing integrity constraints. But as it does not belong to the key attributes that identify the tuples, it is not underlined.

In the customer schema it is indicated by "productId *references Product(id)*" that the *productId* references the *id* of the product relation.

3.2 Time Domain

In the next step relevant terms are defined. The first definition that is given, is the one of the *time domain* and with it the definition of *ongoing timestamps*.

Definition 1 (Unlimited ongoing timestamp)
Let $\mathcal{T}$ be the set of fixed timestamps and $t \in \mathcal{T}$.
An unlimited ongoing timestamp is a timestamp '$NOW\ t$' that is equal to t as long as the current time is less than t and afterwards equal to the current time.

Assume a timeframe [2015-01-01, *NOW* 2017-01-01) is given (cf. Figure 3.2). Then the tuple containing this timeframe will be valid from 2015 and as long as the current time is less than 2017, it will be valid until 2017. Afterwards, this means from 2017 on, it will be valid until the current time.

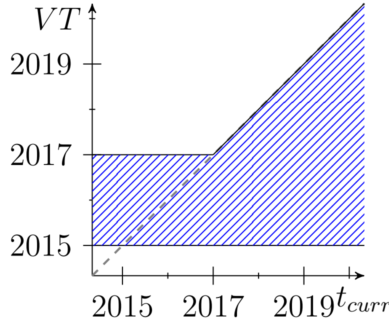


Figure 3.2: Unlimited ongoing timestamp in upper bound

Definition 2 (Limited ongoing timestamp)
Let $\mathcal{T}$ be the set of fixed timestamps and $t_1, t_2 \in \mathcal{T}$.
Let $t_1 > t_2$.
A limited ongoing timestamp is a timestamp '$min\ t_1\ NOW\ t_2$' that is equal to t_2 as long as the current time is less than t_2 and afterwards, it is equal to the current time but only as long as the current time is less than t_1, afterwards, it is equal to t_1.

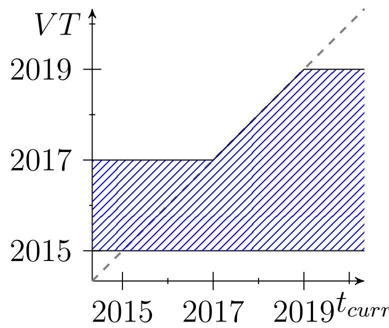


Figure 3.3: Limited ongoing timestamp in upper bound

Assume a timeframe [2015-01-01, min 2019-01-01 *NOW* 2017-01-01) is given (cf. Figure 3.3). Then the tuple containing this timeframe will be valid from 2015 and as long as the current time is less than 2017, it will be valid until 2017. Afterwards it will be valid until the current time, but only as long as the current time is less than 2019, then it will be valid until 2019.

Combining the fixed timestamps, as well as the unlimited and limited ongoing ones, the time domain results:

Definition 3 (Time domain)
Let $\mathcal{T}$ be the set of fixed timestamps.
Then the time domain Ω^v consists of the set of fixed timestamps, as well as the unlimited and the limited ongoing timestamps:

$$\Omega^v := \mathcal{T} \cup \{NOW\ t \mid t \in \mathcal{T}\} \cup \{min\ t_1\ NOW\ t_2 \mid t_1, t_2 \in \mathcal{T} \wedge t_2 < t_1\}$$

The elements defined in the time domain are used as lower and upper bounds in the temporal attributes that define the valid time of a tuple. For the PostgreSQL integration this attribute is of the range type *daterange* with an inclusive lower and an exclusive upper bound and this temporal attribute will be referred to as timeframe.

Definition 4 (Timeframe)
Let $t_1, t_2 \in \Omega^v$.
A timeframe is a closed-open interval of $\Omega^v \times \Omega^v$*:*

$$timeframe := [t_1, t_2)$$

3.3 Ongoing Primary Key Constraint

This section defines the term *Ongoing Primary Key Constraint*. First the definition is given, then the different cases in which violations are possible are listed and the resolving strategy is explained. The resolving strategy is the strategy used to prevent the occurrence of the violations.

3.3.1 Definition

To understand the principle of OPK constraints, a definition of a non-temporal primary key is given first.

Definition 5 (Non-temporal primary key)
Let R *be a relation of the schema* $sch(R) = (B_1, ..., B_n)$ *and* $B = \{B_1, ..., B_n\}$ *the set of attributes.*
Let $A \subseteq B$ *be the set of key attributes.*
The primary key on R *ensures that there do not exist two tuples with the same set of key attributes* A *in* R*:*

$$\forall r \in R : (\nexists\, r' \in R \setminus \{r\} : r'.A = r.A)$$

This means that a primary key constraint indicates that a set of columns *A* can be used as a unique identifier for rows [5]. Therefore, in relations having a primary key constraint no more than one tuple can have the same primary key attribute values.

This constraint can be adapted to temporal database systems, but then the constraint must be fulfilled independently at every point in time [6]. This means that there can indeed be more than one tuple with the same key attributes in a relation, as long as their temporal attributes do not overlap in time [6].

In temporal database systems using ongoing timestamps, the meaning of *overlap* is not unambiguous. As in this thesis a statement is always rejected if a violation occurs at least at one point in time, an OPK violation does already occur, if two tuples overlap at least at one point in time.

Definition 6 (Overlapping of closed-open timeframes in temporal databases with ongoing timestamps)
Let $T_1 = [a, b)$ and $T_2 = [c, d)$ be two timeframes and $a, b, c, d \in \Omega^v$.
Let $\cap^t$ be the intersection at the current time t.
Then,

$$T_1 \; overlaps^t \; T_2 \Leftrightarrow \exists t \in \mathcal{T} : (T_1 \cap^t T_2 \neq \varnothing)$$

For a better understanding of the meaning of *at least at one point in time* an example is given. Assuming the current year is 2016. In a relation two tuples are stored (cf. Figure 3.4). In the year 2016, the tuples' timeframes would actually not have any common intersection. This means they would not overlap when only observing the current date. But when observing the whole time span, it becomes clear, that from 2018 on and therefore, at least at one point in time, the timeframes do have a common intersection and do therefore fulfil the condition of overlap defined in Definition 6.

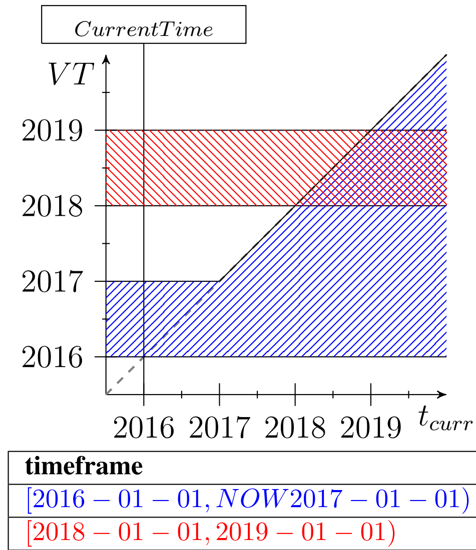


Figure 3.4: Overlapping of two timeframes at least at one point in time

After having seen the time domain used in this thesis and the meaning of overlap, this leads to the final definition of ongoing primary key constraints.

Definition 7 (Ongoing primary key)
Let R be a relation of the schema $sch(R) = (B_1, ..., B_n, T)$, $B = \{B_1, ..., B_n\}$ the set of non-temporal attributes and T the timeframe.
Let $A \subseteq B$ be the set of key attributes.
The ongoing primary key on R ensures that there do not exist two tuples with the same set of key attributes A and whose timeframes do overlap at any point in time:

$$\forall r \in R : (\nexists r' \in R \setminus \{r\} : r'.A = r.A \wedge r'.T \; overlaps \; r.T \,)$$

3.3.2 Violation

When dealing with OPKs there exist two situations in which a violation of the constraint could occur. This is either by inserting or by updating tuples. Because there can never result a new intersection of timeframes but only new gaps by deleting a tuple, it is not possible to cause an OPK violation by deleting a tuple in a relation.

OPK-Violation 1 (Insert)
Let R be a relation of the schema $sch(R) = (B_1, ..., B_n, T)$, $B = \{B_1, ..., B_n\}$ the set of non-temporal attributes and T the timeframe.
Let $A \subseteq B$ be the set of key attributes.
When inserting a new tuple r' into R a violation occurs, if there does already exist a tuple in the relation with the same key attributes and a timeframe that overlaps the timeframe of the tuple to be inserted.

$$\exists r \in R : (r.A = r'.A \wedge r.T \; overlaps \; r'.T)$$

OPK-Violation 2 (Update)
Let R be a relation of the schema $sch(R) = (B_1, ..., B_n, T)$, $B = \{B_1, ..., B_n\}$ the set of non-temporal attributes and T the timeframe.
Let $A \subseteq B$ be the set of key attributes.
When updating a tuple in R from r' to r'' a violation occurs, if there does already exist a tuple in the relation (except the tuple to be updated itself) with the same key attributes as the new key attributes of the tuple to be updated and a timeframe that overlaps the new timeframe of the tuple to be updated.

$$\exists r \in R \setminus \{r'\} : (r.A = r''.A \wedge r.T \; overlaps \; r''.T)$$

3.3.3 Resolving Strategy

To ensure that no violation occurs, a resolving strategy is needed. In this thesis the resolving strategy is to reject a statement, if at least at one point in time, one of the two OPK-Violation cases can occur.

This resolving strategy has the advantage that it is simple and efficient. This is, because the database system needs to check only when a modification that can lead to a violation is made, whether the constraint is fulfilled at every point in time. Like this, no postponed violation of these constraints can occur when time advances.

But due to the simplicity, this resolving strategy is also restricted. Statements will always be rejected, no matter at what point in time the violation occurs. That means, the modifications are also rejected if the violation would only occur in the past or far away in the future.

Another possibility would be, to only reject a statement, if a violation occurs at the current time. But then, it would be necessary to ensure steadily that no postponed violation occurs when the time advances. This approach would be very costly.

3.4 Ongoing Foreign Key Constraint

Next step is to define the *Ongoing Foreign Key Constraint*. Again, first the non-temporal constraint is defined, then the definition is adapted to temporal databases using ongoing time-stamps and the cases in which a violation can occur are discussed. Also this constraint will be illustrated on the application scenario in section 3.6.

3.4.1 Definition

A non-temporal foreign key can be described as follows.

Definition 8 (Non-temporal foreign key)
Let R and S be two relations with schemas $sch(R) = (B_1, ..., B_n)$ and $sch(S) = (C_1, ..., C_m)$.
Let $B = \{B_1, ..., B_n\}$ be the set of attributes of R and $C = \{C_1, ..., C_m\}$ be the set of attributes of S.
Let $D \subseteq C$ be the set of primary key attributes of relation S and $A \subseteq B$ be the set of foreign key attributes of relation R, referencing D.
Then, the foreign key on R ensures that for every tuple r in R a tuple s in S exists, where the foreign key attributes in r are the same as the primary key attributes in s:

$$\forall r \in R : (\exists s \in S : r.A = s.D)$$

This means, a foreign key is a constraint that specifies that the values in a set of columns must match the values appearing in some row of another table (having a PK constraint) [5]. Therefore, for every foreign key tuple, there must exist exactly one corresponding tuple in the primary key relation. A tuple is corresponding if the key attributes of the PK and FK tuple are the same.

When using temporal databases, we have the difference that there must exist one corresponding tuple in the OPK relation at every point in the valid time of a referencing tuple. In other words, it is possible that an OFK tuple points to multiple OPK tuples when its timeframe is covered by several tuples' timeframes. The important thing is that the whole timeframe is covered. Because otherwise, for some time span of a timeframe no corresponding tuple would exist and a violation would occur. The OFK constraint can therefore be defined as follows:

Definition 9 (Ongoing foreign key)
Let R and S be two relations with the schema $sch(R) = (B_1, ..., B_n, T)$ and $sch(S) = (C_1, ..., C_m, T)$.
Let $B = \{B_1, ..., B_n\}$ be the set of non-temporal attributes of R and $C = \{C_1, ..., C_m\}$ be the set of non-temporal attributes of S.
Let $D \subseteq C$ be the set of ongoing primary key attributes of relation S and $A \subseteq B$ be the set of ongoing foreign key attributes of relation R, referencing D.
Let T be the timeframe attribute.
Then, the ongoing foreign key on R ensures that for every tuple r in R a set of tuples $\{s_0, ...s_k\}$ in S exists, where for every tuple in the set, the ongoing primary key attributes are the same as

the ongoing foreign key attributes in r *and when subtracting all timeframes of the set* $\{s_0, ...s_k\}$ *from the timeframe form* r *the difference is empty:*

$$\forall r \in R : (\exists\{s_0, ...s_k\} \subseteq S : r.A = s_0.D \wedge ... \wedge r.A = s_k.D \ \wedge \\ ((((r.T - s_0.T) - s_1.T) - ...) - s_k.T) = \varnothing)$$

3.4.2 Violation

When dealing with OFKs there are two main scenarios in which a violation can occur. To begin with, it is possible when modifications on the OFK relation are made, but it is also possible when the OPK relation is modified. All cases are shown in the following. The violation cases depend on the assumptions that there exist two relations. The first relation S contains an OPK constraint and is called the referenced relation, whereas the second relation R contains an OFK constraint and is referencing the first. The schemas listed in the following are used to describe the violation cases (with T being the timeframe attribute):

$$sch(S) = (C_1, ..., C_m, T), OPK\ attributes\ D \subseteq C,$$

$$sch(R) = (B_1, ..., B_n, T), OFK\ attributes\ A \subseteq B\ referencing\ D$$

OFK-Violation 1 (Insert into ofk relation)
When inserting a new tuple r' *into* R*, a violation occurs, if there does not exist a tuple or a set of tuples in the referenced relation that*

1. *have the same key attribute values and*
2. *whose timeframes completely cover the timeframe of the tuple to be inserted.*

$$\nexists\{s_0, ..., s_k\} \subseteq S : (r'.A = s_0.D \wedge ... \wedge r'.A = s_k.D \ \wedge \\ ((((r'.T - s_0.T) - s_1.T) - ...) - s_k.T) = \varnothing)$$

OFK-Violation 2 (Update in ofk relation)
When updating a tuple in R *from* r' *to* r''*, a violation occurs, if there does not exist a tuple or a set of tuples in the referenced relation that*

1. *have the same key attribute values as the new key attributes of the tuple to be updated and*
2. *whose timeframes completely cover the new timeframe of the tuple to be updated.*

$$\nexists\{s_0, ..., s_k\} \subseteq S : (r''.A = s_0.D \wedge ... \wedge r''.A = s_k.D \ \wedge \\ ((((r''.T - s_0.T) - s_1.T) - ...) - s_k.T) = \varnothing)$$

As it can be seen, the origin values of the tuple to be updated are irrelevant. It is only important that the timeframe value of the updated tuple is covered by corresponding tuples in the referenced relation.

OFK-Violation 3 (Update in opk relation)
Let $\cap$ be the intersection over all points in time.
When updating a tuple in S from s' to s'', a violation occurs if either the key attributes changed and the origin tuple was referenced (3.1) or if the timeframe attribute, but not the key attributes, changed (3.2). In this case (3.2), a violation occurs, if the origin tuple was referenced and the intersection of the referencing tuple's timeframe with the timeframe of the updated tuple is not the same as the intersection of the referencing tuple's timeframe with the origin timeframe of the tuple to be updated.

$$s'.D \neq s''.D \wedge \exists r \in R : (r.A = s'.D \wedge r.T \ \ overlaps \ \ s'.T) \tag{3.1}$$

$$s'.D = s''.D \wedge s'.T \neq s''.T \wedge \exists r \in R : (r.A = s'.D \wedge r.T \ \ overlaps \ \ s'.T \wedge (r.T \cap s'.T) \neq (r.T \cap s''.T)) \tag{3.2}$$

This definition is based on the assumption that only one modification per transaction is allowed. It is therefore not possible, to insert another tuple into the referenced relation in the same transaction that would cover the missing part of the timeframe. Therefore, the intersection of the origin and the new tuple's timeframe must be the same.

OFK-Violation 4 (Delete on opk relation)
When deleting a tuple s_0 in S, a violation occurs, if the tuple to be deleted is actually being referenced by a tuple in the referencing relation. This means, if there does exist a tuple in the referencing relation that has the same key attribute values and whose timeframe overlaps with the timeframe of the tuple to be deleted.

$$\exists r \in R : (r.A = s_0.D \wedge r.T \ \ overlaps \ \ s_0.T \ )$$

When inserting a tuple into a relation, which does only have an OPK constraint no OFK violation can occur. Only a violation of the OPK is possible. The reason is that when a new tuple is inserted, it cannot be that a timeframe of a tuple in the referencing relation, whose timeframe was fully covered before the insertion, is not fully covered any more after the insertion into the relation with the OPK constraint. The same is true when a tuple in the relation with an OFK constraint is deleted. Therefore all possible violation cases have been identified.

3.4.3 Resolving Strategy

To ensure that no violation occurs, in this thesis, a statement that leads to one of the four OFK-Violation cases described above, is rejected, if at least at one point in time, a violation can occur. The advantages and disadvantages of this resolving strategy are the same as the ones mentioned in section 3.3.3.

3.5 Overview

Table 3.3 sums up all violation cases seen in sections 3.3 and 3.4. For a better understanding a reading example is given. The first row can be read like this: "When on a relation with only an *OPK constraint* a tuple is *inserted*, then this can lead to an *OPK violation*, but not to an *OFK violation*." Besides, it is assumed that the relation with the OPK constraint is referenced by the relation with the OFK constraint.

Relation	Modification	OPK violation possible	OFK violation Possible
only with an OPK constraint	insert	✓	✗
	update	✓	✓
	delete	✗	✓
only with an OFK constraint	insert	✗	✓
	update	✗	✓
	delete	✗	✗

Table 3.3: Overview of OPK and OFK violation cases

3.6 Example

In this section the definitions seen before are illustrated. All of the examples are based on the application scenario seen in section 3.1. The initial relations are once more shown in table 3.4 and 3.5.

id	name	premium	timeframe
300	Standard	4.5	$[2015-01-01, NOW2016-01-01)$
301	Plus	7.5	$[2015-01-01, min2017-01-01NOW2016-01-01)$
301	Plus	6.5	$[2017-01-01, NOW2018-01-01)$

Table 3.4: Initial available products

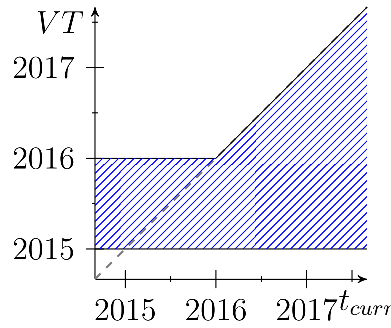
customerId	productId	insuredSum	timeframe
C-767	300	1,000	$[2015-01-01, NOW2016-01-01)$
C-769	300	2,000	$[2015-01-01, 2016-01-01)$
C-843	300	2,000	$[2016-01-01, min2019-01-01NOW2018-01-01)$
C-900	301	4,000	$[2015-01-01, min2018-01-01NOW2015-01-01)$

Table 3.5: Initial customer relation

3.6.1 Illustration of the Time Domain

As seen above, limited and unlimited ongoing timestamps as well as fixed ones can be used as the lower and upper bound of a timeframe. When combining them, different forms of timeframes arise. To get an overview, two of them will be explained below.

Type 1 $[T, NOWt)$ *Fixed lower and unlimited ongoing upper bound*

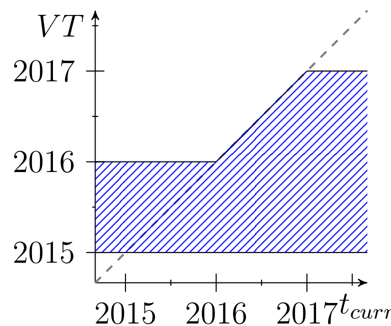


id	name	premium	timeframe
300	Standard	4.5	$[2015-01-01, NOW2016-01-01)$

Figure 3.5: Fixed lower and unlimited ongoing upper bound.

Type 1 represents a timeframe for which it is unknown how long it will be valid in the future. Therefore it is ongoing and unlimited. Figure 3.5 shows the valid time of the standard product from the application scenario. The insurance company offers the standard product since the beginning of 2015 and it will for sure be available until the beginning of 2016. But the company assumes that afterwards the product will still be available. Therefore, the upper bound is ongoing.

Type 2 $[T, min\ t_1\ NOW\ t_2)$ *Fixed lower and limited ongoing upper bound*



id	name	premium	timeframe
301	Plus	7.5	$[2015-01-01, min2017-01-01NOW2016-01-01)$
301	Plus	6.5	$[2017-01-01, NOW2018-01-01)$

Figure 3.6: Fixed lower and limited ongoing upper bound.

Type 2 stands for a timeframe that is valid at least until some point in time but also no longer than until another point in time. Therefore it is ongoing and limited. The graph in Figure 3.6 shows the valid time of the plus product to the time, when the product has the premium of 7.5%. Because after 2017 the premium will be decreased to 6.5%, the valid time of the tuple is limited to the top.

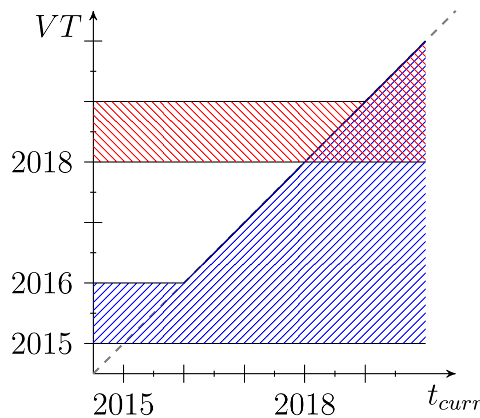
3.6.2 Illustration of the Ongoing Primary Key

In this section, the OPK will be illustrated with an example. For this scenario only the *Product* relation is relevant. Table 3.6 represents the current state of the product relation.

id	name	premium	timeframe
300	Standard	4.5	$[2015-01-01, NOW 2016-01-01)$
301	Plus	7.5	$[2015-01-01, min 2017-01-01 NOW 2016-01-01)$
301	Plus	6.5	$[2017-01-01, NOW 2018-01-01)$

Table 3.6: Initial available products

After some time, the company decides to add a new product that people with a lower budget could be interested in. They call it "Basic". They decide to make it available from the beginning of 2018 on. Because it is a new product, there will not be any OPK violation.



id	name	premium	timeframe
300	Standard	4.5	$[2015-01-01, NOW 2016-01-01)$
301	Plus	7.5	$[2015-01-01, min 2017-01-01 NOW 2016-01-01)$
301	Plus	6.5	$[2017-01-01, NOW 2018-01-01)$
302	Basic	3.0	$[2018-01-01, NOW 2018-01-01)$
300	Standard	5	$[2018-01-01, NOW 2019-01-01)$

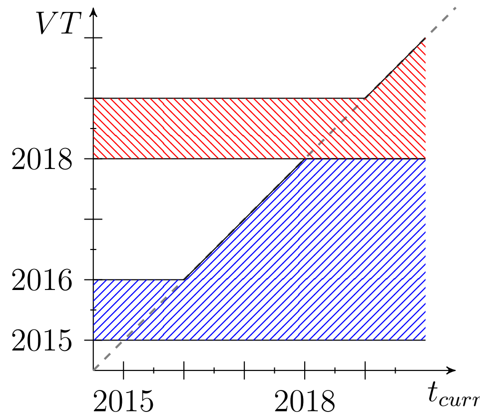
Figure 3.7: OPK Violation while trying to insert the red tuple.

Due to this change, Avivo thinks it is necessary to make the premium of the standard product a little more expensive. From the beginning of 2018 on and at least for one year, the premium will be 5% instead of 4.5%.

They decide to insert the tuple (300, Standard, 5, $[2018-01-01, NOW 2019-01-01)$) into *product*. But because the tuple (300, Standard, 4.5, $[2015-01-01, NOW 2016-01-01)$) is still present in the relation, a violation occurs. Figure 3.7 illustrates the situation with the blue area being the timeframe of the tuple with premium 4.5% and the red area being the timeframe of the tuple to be inserted. The part where the lines overlap represents the violation.

Therefore the company has to update the standard tuple with premium 4.5% to (300, Standard, 4.5, $[2015-01-01, min 2018-01-01 NOW 2016-01-01)$) and can then insert

the new tuple. Like this, no violation occurs (cf. Figure 3.8) and the updated relation leads to the table seen in Figure 3.8.



id	name	premium	timeframe
300	Standard	4.5	$[2015-01-01, min2018-01-01NOW2016-01-01)$
301	Plus	7.5	$[2015-01-01, min2017-01-01NOW2016-01-01)$
301	Plus	6.5	$[2017-01-01, NOW2018-01-01)$
302	Basic	3.0	$[2018-01-01, NOW2018-01-01)$
300	Standard	5	$[2018-01-01, NOW2019-01-01)$

Figure 3.8: No OPK Violation after updating old tuple

3.6.3 Illustration of the Ongoing Foreign Key

For the illustration of ongoing foreign keys, both relations are necessary. Assume relation *product* is in the state presented in Figure 3.8. Table 3.7 shows the *customer* relation from the application scenario. In this section an example for some violation cases mentioned in section 3.4 will be given. The cases are chosen, so that the interaction of the referenced and the referencing relation becomes clear.

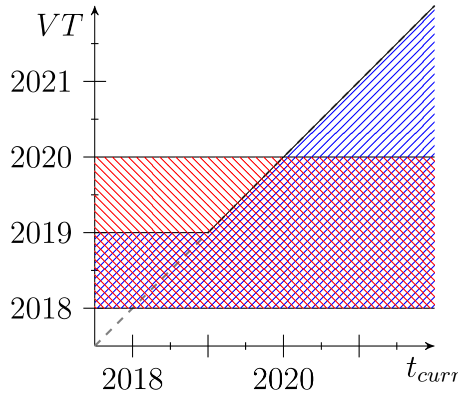
customerId	productId	insuredSum	timeframe
C-767	300	1,000	$[2015-01-01, NOW2016-01-01)$
C-769	300	2,000	$[2015-01-01, 2016-01-01)$
C-843	300	2,000	$[2016-01-01, min2019-01-01NOW2018-01-01)$
C-900	301	4,000	$[2015-01-01, min2018-01-01NOW2015-01-01)$

Table 3.7: Initial customer relation

The first case that will be illustrated is to insert a new tuple into the customer relation. The prospective customer gets the customerId C-901 and is interested in the insurance product 300 (standard) from 2018 on and for at least two years. The company tells him that the standard product can only be fixed insured until the beginning of 2019. Because an insurance contract of two years would lead to the violation shown in Figure 3.9, where the blue area represents

the product's timeframe and the red area the customer's timeframe. The part where the red area is not overlapping with the blue area shows the violation.

Avivo instead offers the customer to record on his data, that he would like to continue the insurance with the product 300 for one more year, if it is still available. As the person agrees, the tuple Customer(C-901, 301, [2018-01-01, min2020-01-01 NOW 2019-01-01]) is stored into the database.



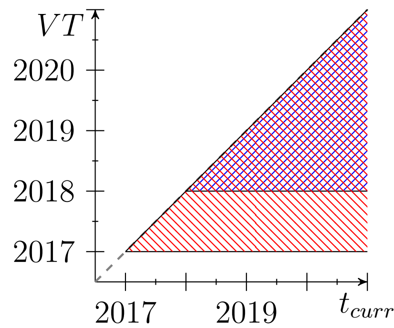
customerId	productId	insuredSum	timeframe
C-767	300	1,000	$[2015-01-01, NOW2016-01-01)$
C-769	300	2,000	$[2015-01-01, 2016-01-01)$
C-843	300	2,000	$[2016-01-01, min2019-01-01NOW2018-01-01)$
C-900	301	4,000	$[2015-01-01, min2018-01-01NOW2015-01-01)$
C-901	300	1,000	$[2018-01-01, 2020-01-01)$

id	name	premium	timeframe
300	Standard	4,5	$[2015-01-01, min2018-01-01NOW2016-01-01)$
301	Plus	7.5	$[2015-01-01, min2017-01-01NOW2016-01-01)$
301	Plus	6.5	$[2017-01-01, NOW2018-01-01)$
302	Basic	3.0	$[2018-01-01, NOW2018-01-01)$
300	Standard	5	$[2018-01-01, NOW2019-01-01)$

Figure 3.9: OFK violation, because the timeframe is not fully covered

An already registered customer (C-767) learns about the new product "Basic". He asks the company to change his insurance product to "Basic" for the next year, this would be 2017. As the basic product comes only in effect in 2018, the customer's request would lead to another violation (Figure 3.10).

The last example shows what happens when a product is deleted. Assume, Avivo decides to the delete the product "Plus" that includes the tuples (301, Plus, 7.5, $[2015-01-01, min2017-01-01NOW2016-01-01)$) and (301, Plus, 6.5, $[2017-01-01, NOW2018-01-01)$). This is of course not possible, because at least one customer's (C-900) insurance contract depends on that product. The deletion can not be performed.



customerId	insuredSum	productId	timeframe
C-767	300	1,000	$[2015-01-01, 2017-01-01NOW2016-01-01)$
C-767	300	1,000	$[2017-01-01, NOW2017-01-01)$
C-769	300	2,000	$[2015-01-01, 2016-01-01)$
C-843	300	2,000	$[2016-01-01, min2019-01-01NOW2018-01-01)$
C-900	301	4,000	$[2015-01-01, min2018-01-01NOW2015-01-01)$

<u>id</u>	name	premium	timeframe
300	Standard	4,5	$[2015-01-01, min2016-01-01NOW2018-01-01))$
301	Plus	7.5	$[2015-01-01, min2017-01-01NOW2015-01-01)$
301	Plus	6.5	$[2017-01-01, NOW2018-01-01)$
302	Basic	3.0	$[2018-01-01, NOW2018-01-01)$
300	Standard	5	$[2018-01-01, NOW2019-01-01)$

Figure 3.10: OFK violation, because the timeframe is not fully covered

4 Implementation Approach

In this chapter two implementation approaches are provided to support the ongoing integrity constraints. On the one hand, the PostgreSQL kernel was adapted. On the other hand, user defined triggers were written that verify the constraints.

The version of PostgreSQL used is 9.4, it was released at the end of 2014 and is written in C. The version provided to me by my supervisor did already include some adapted functions to work with ongoing timestamps. These include among others the range functions overlap, intersection and difference. These were used in the implementation.

4.1 Internal Implementation

First, the internal approach is provided. The implementation of both constraints (OPK and OFK) rely on the approach that was used to implement the non-temporal foreign key constraint. That means that for every statement that could possibly lead to a violation (cf. sections 3.3 and 3.4) an internal trigger is written that checks if a violation results and if so, rejects the statement. The implementation of the non-temporal PK constraint is based on a B^+-tree. The PK will automatically create a unique B^+-tree index on the columns of the key attributes [5]. But as there does not exist an underlying order in the primary keys depending on ongoing timeframes, a B^+-tree is not a good approach for the OPK constraint. When a tuple has an ongoing timestamp, it is not clear where this tuple should be placed in the tree.

In the following first the part is reviewed that is called, when tables with OPK respectively OFK constraints are created. Then the constraint triggers are discussed. The main files (or excerpts of them) that were adapted in the kernel can be found in the appendix.

4.1.1 Creation of Ongoing Constraint Tables

In this section it will be described, how a new table with OPK respectively OFK constraint can be created and what conditions must be fulfilled to do so. The syntax for creating these tables is defined in the parser file *gram.y*. The relevant changes of the file can be found in the appendix in Listing 6.1. With the actual implementation, it is only possible to add an ongoing integrity constraint when creating a new table. A later adding of the constraint with *alter table add constraint* is currently not possible.

First the creation of a table with an OPK will be exposed. To create such a table, the user must call a statement of the form:

Listing 4.1: Create Table with OPK constraint

```
CREATE TABLE opk_table_name (set_of_attributes,
```

```
timeframe daterange,
PRIMARY KEY ONGOING (set_of_pk_attribute_names));
```

where *set_of_attributes* stands for a list of attributes having a name and a data type and *set_of_pk_attribute_names* references some of these attributes. So *set_of_attributes* could for example be *id integer, name varchar, premium decimal* and *set_of_pk_attribute_names* could reference *id*.

It is also necessary that there exists a column *timeframe* that is of the data type *daterange*. The timeframe is not supposed to be added to the *set_of_pk_attribute_names*. When trying to add timeframe to the list of *set_of_pk_attribute_names* an error is risen. The reason is, that *set_of_pk_attribute_names* contains those attributes that identify two tuples representing the same object. E.g. do the tuples t_1 and t_2 represent the same object, as they have the same id.

$$t_1 = Product(\underline{1}, "Product1", 4.5, [2010-01-01, 2011-01-01])$$

$$t_2 = Product(\underline{1}, "Product1", 5, [2011-01-01, 2012-01-01])$$

If someone would add the timeframe to the list of key attributes, the behaviour of the ongoing integrity constraints would completely change. Tuples t_3 and t_4 could be inserted into the same OPK relation and no violation would occur, although their timeframes do overlap in time. This is, because they would not be identified as the same product any more.

$$t_3 = Product(\underline{1}, "Product1", 4.5, \underline{[2010-01-01, 2012-01-01]})$$

$$t_4 = Product(\underline{1}, "Product1", 5, \underline{[2011-06-01, 2012-01-01]})$$

Summed up, a valid statement for creating a new table with this constraint is:

Listing 4.2: Example: Create Table with OPK constraint - example

```
CREATE TABLE product (id integer, name varchar,
premium decimal, timeframe daterange,
PRIMARY KEY ONGOING (id));
```

The syntax for creating a new table with OFK constraint is similar (cf. Listing 4.3), but some more conditions must be respected.

Listing 4.3: Create Table with OFK constraint

```
CREATE TABLE ofk_table_name (set_of_attributes,
timeframe daterange,
FOREIGN KEY ONGOING (set_of_fk_attribute_names)
REFERENCES opk_table_name (set_of_pk_attribute_names);
```

Anew, the column timeframe must exist and being of the data type *daterange*. Once again the *set_of_attributes* lists all attributes (except the timeframe) of the new relation and *set_of_fk_attribute_names* specifies, which attributes are the key attributes. Once again, these

set_of_fk_attribute_names do not include the timeframe column, as this would change the behaviour of the ongoing integrity constraints.

What's new is the last part of the query, namely "*REFERENCES opk_table_name (set_of_pk _attribute_names)*". Here *opk_table_name* is the name of the referenced relation. While creating, an assertion is made, that this table does actually exist and has an OPK constraint. Then *set_of_pk_attribute_names* refers to the list of attributes in the OPK relation. It must be ensured that the number of OPK attributes is equal to the number of OFK attributes and that these attributes are of compatible types. E.g. integer and varchar are not compatible because not all varchars can be casted to integers. As all these conditions are fulfilled, tables with OPK and OFK constraints can be created. The verifications of the fulfilment of the conditions are made in the file *tablecmds.c*.

To drop an ongoing constraint, a statement of the form seen in Listing 4.4 must be called.

Listing 4.4: Drop constraint

```
ALTER TABLE table_name DROP constraint_name;
```

The *constraint_name* is composed of: *table_name* + "_" + *name_of_first_key_attribute* + "_" + "og_pkey" respectively "og_fkey".

- Product(id, name, premium, timeframe)
- Customer(customerId, insuredSum, productId *references Product(id)*, timeframe)

For the relations *Product* and *Customer* from the application scenarios the OPK constraint name would be "product_id_og_pkey" and the OFK constraint name "customer_productId_og _fkey".

4.1.2 Ongoing Primary Key Constraint

As mentioned at the beginning of the chapter, for every violation case (cf. Table 3.3 in section 3.5) a trigger has been written in the PostgreSQL kernel. All of them are *constraint triggers*, i.e. they are always fired after a statement is completed. This makes the checking of the constraint especially in update cases easier, as the old values of the tuple are not present in the relation any more. Otherwise there would be a chance that when retrieving all tuples in the relation that have the same key attributes as the updated tuple and that overlap in time with the updated timeframe, also the tuple with the old values would be retrieved and that this would lead to a violation, although it is not.

Constraint triggers have a little disadvantage for the insert case. Because the trigger is called afterwards, the new tuple is already inserted when the constraint verification takes place. If for instance the OPK trigger does check, if there exists a tuple that has the same key attributes as the tuple to be inserted and whose timeframes do overlap in time, it must be ensured, that not the newly inserted tuple itself leads to a violation. In the insert case, it could therefore also be a good option to use normal triggers that are called before the statement is executed. But to keep the implementation clearly arranged and to be able to use the same logic for the

insert and update modifications, for every modification statement, the same kinds of triggers are used.

All triggers have been written into the new file *og_triggers.c*, where "og" stands for ongoing. The excerpts of the triggers can be found in the appendix in Listings 6.4 to 6.7. The implementation of the triggers uses the Server Programming Interface (short as SPI) that PostgreSQL provides. SPI enables to run SQL queries inside the C code. This is necessary to retrieve bits of information about relation schemas and the tuples that are present in those relations. The tuples themselves are necessary to verify that no ongoing integrity violation occurs. I am not aware of any other possibility to retrieve the tuples stored in a relation inside the PostgreSQL kernel, therefore the SPI interface is used.

Insert Trigger

Algorithm 1 How the SPI interface is used to create and execute a query

Require: Connection to SPI interface established
Require: *attnums* is an array, that contains all key column numbers
1: - - Get the number of the timeframe column - -
2: $timeframeColumnNumber = SPI_fnumber(TupleDescription,'timeframe')$
3: - - Get the value of the timeframe column of the newly inserted tuple - -
4: $timeframe_value =$
5: $\quad SPI_getvalue(NewRow, TupleDescription, timeframeColumnNumber)$
6: - - Create the query seen in Listing 4.5 - -
7: $query\ = "SELECT * FROM\ table_name\ WHERE"$
8: **for** $every\ i\ in\ attnums$ **do**
9: $\quad$ - - Get the name of the key attribute - -
10: $\quad name = SPI_fname(TupleDescription, attnums[i])$
11: $\quad$ - - Get the value of the key attribute of the inserted tuple - -
12: $\quad value = SPI_getvalue(NewRow, TupleDescription, timeframeColumnNumber)$
13: $\quad$ - - Expand the query - -
14: $\quad query += "\%s = \%s\ AND\ ",\ using\ name\ and\ value$
15: - - Add checking of overlapping to query - -
16: $query += "timeframe\&\&\%s",\ using\ timeframe_value$
17: - - Execute the query seen in Listing 4.5 - -
18: $SPI_execute(query, false, 2)$
19: - - Get the number of retrieved tuples - -
20: $processed = SPI_processed$
21: - - Reject the statement, if not only the inserted tuple itself was found - -
22: **if** $processed! = 1$ **then**
23: $\quad throw\ error("OPK\ violated")$

On every table with OPK constraint a trigger exists that is automatically fired, when an insert statement on this table is called. The trigger identifies the column number of the timeframe column with the aid of the function SPI_fnumber from the SPI interface (cf. Algorithm 1,

line 2). This number is needed for the SPI_getvalue function to retrieve the column's value (lines 4 + 5). Like that, the value of the newly inserted tuple's timeframe (*timeframe_value*) is available.

With the aid of a loop (line 8), all column names of the key attributes and the values of the keys are retrieved and the query in Listing 4.5 is created. The column names are retrieved with the aid of SPI_fname (line 10) and the values with SPI_getvalue (line 12).

Listing 4.5: SQL statement in the insert trigger

```
SELECT * FROM opk_table_name
WHERE keyAttributesMatch AND timeframe && timeframe_value;
```

After having created the query, it is performed with the aid of SPI_execute (cf. Algorithm 1, line 18). SPI_execute takes three arguments. The first argument is the query to be executed. The second argument indicates, if the query is a read-only statement and the third argument indicates how much rows will be retrieved at most [5]. If that value is reached, the execution stops. In this case, the execution stops, when two tuples were found. It is not necessary to retrieve more than two tuples, as two tuples are already enough to be sure, that a violation occurs.

Then SPI_processed returns the number of tuples found (line 20). If SPI_processed is equal to 1, then it means that the only tuple that was found is the new tuple itself (because the constraint trigger was called after the insertion) and therefore no violation occurs. The insertion is performed. But if SPI_processed is equal to 2 then it means, that there does exist at least one other tuple in the relation that has the same values in the key attributes and whose timeframe overlaps with the new tuple's timeframe (line 22). Therefore a violation occurs and the insertion is rejected (line 23).

Listing 4.5 shows the schema of the query that was created with the SPI interface. In the query, keyAttributesMatch means that all OPK attribute values are the same as the values from the newly inserted tuple. The symbol '&&' is the SQL symbol used for overlap. It returns true, if the ranges left and right of it overlap at least at one point in time. So for instance, somebody could insert the tuple (304, 'Limited', '[2017-01-01, 2018-01-01)') into the relation product defined before. Then the insert trigger would execute the query in Listing 4.6.

Listing 4.6: SQL statement on the product example

```
SELECT * FROM product
WHERE id = 304 AND timeframe && '[2017-01-01, 2018-01-01)';
```

Update Trigger

The implementation of the update trigger is in fact the same as the implementation seen in the insert trigger. This is possible because the trigger is called after the update is completed. That means, that the state that is present, when the trigger is called, would be the same, as when someone would delete the tuple and insert a new one, instead of updating it. As a deletion of an OPK tuple does not lead to any OPK violation, the insert trigger logic can be reused for the update case. Besides, it has also be seen in OPK-Violation 3 in Chapter 2 that the old values of the tuple are not needed to check if a violation occurs.

4.1.3 Ongoing Foreign Key Constraint

After the completion of the OPK triggers, in the next step the OFK triggers are expounded. Like the others, they use the SPI interface. What is different is, that some of these OFK triggers are recorded on the referencing table, where others are stored on the referenced table. Moreover, each of the triggers needs information from the other relation. As this is already the case for non-temporal foreign key triggers, this did not provide additional challenges in the implementation.

Insert Trigger (Referencing Table)

The insert trigger is called when trying to insert a tuple into a relation with OFK constraint. Therefore that trigger is added to the referencing table. The constraint logic will be explained with the aid of the pseudo code in Algorithm 2.

Algorithm 2 Check OFK constraint

Require: Query seen in Listing 4.7 has been created and saved into *query*

1: - - Retrieve referenced tuples - -
2: $SPI_execute(query, false, 2)$
3: - - Check if at least one referenced tuple exist - -
4: **if** $SPI_processed == 0$ **then**
5: $\quad throw\ error("OPK\ violated")$
6: - - Create a new list and initialize first element with timeframe of inserted value - -
7: $list \rightarrow first = OFKtimeframe$
8: **for** *every OPK tuple (t) retrieved* **do**
9: $\quad$ - - Get the timeframe value of the referenced tuple - -
10: $\quad referenced_timeframe_value = getTimeframeValue(t)$
11: $\quad$ - - Assert that the list contains at least one element - -
12: $\quad$ **if** *list is empty* **then**
13: $\quad\quad$ - - Accept Insertion - -
14: $\quad\quad accept\ and\ exit\ the\ trigger$
15: $\quad$ **while** *list is not empty* **do**
16: $\quad\quad$ - - Get the difference of the current referenced tuple's timeframe
17: $\quad\quad$ and the current element in the list - -
18: $\quad\quad difference = difference(list \rightarrow current, t \rightarrow timeframe)$
19: $\quad\quad$ **if** *difference is not empty* **then**
20: $\quad\quad\quad$ - - Replace current list element with difference - -
21: $\quad\quad\quad list \rightarrow current = difference$
22: $\quad\quad$ **else**
23: $\quad\quad\quad remove(list \rightarrow current)$
24: - - Reject the statement, because not whole timeframe is covered - -
25: $throw\ error("OPK\ violated")$

The assertion that no OFK violation occurs is verified over multiple steps. In the first step,

the trigger collects all tuples from the referenced relation that have the same key attributes as the tuples to be inserted and that overlap in time (line 2). This happens with the query:

Listing 4.7: SQL statement to retrieve referenced tuples

```
SELECT * FROM referenced_table_name
WHERE keyAttributesMatch AND timeframe && timeframe_value;
```

As the creation of this query and also the queries in the remaining triggers is very similar to the creation of the query in Listing 4.5 that is shown in Algorithm 1, the creation will not be explained here anew. If there are no tuples retrieved, the insertion is rejected immediately (line 4 + 5). Else the next step is to check, if the whole timeframe is covered. The idea is, to subtract the timeframes of the referenced tuples from the tuple to be inserted. If all timeframes were subtracted and there is no remaining part of the inserted timeframe, then it means that the timeframe was fully covered and no constraint violation occurs.

To this end, a new list is created. This list contains all parts of the timeframe of the tuple to be inserted that are not covered by referenced tuples. Initially, the list contains one timeframe that is the whole timeframe of the tuple planned to insert (line 7). The trigger does loop the referenced tuples (line 8). For every tuple, the timeframe value is retrieved (line 10). Then it is checked if the list is empty (line 12). If this is true, then this means that the whole timeframe of the inserted tuple is covered by referenced tuples, and therefore no violation occurs. The statement can be accepted.

Otherwise, the list is iterated (line 15). In the next step, the difference of the current list item and the current tuple's timeframe of the referenced tuples is calculated (line 18). This happens with an adapted version of the range function difference (provided by PostgreSQL). Normally this function does at most return one range as a result. But as a subtraction of one range from another can lead up to two tuples, this method has been adapted by Yvonne Mülle. The adapted, internal function range_difference_now returns a list that contains up to two range elements.

If the difference of the two timeframes returns one or two range elements (line 19), the current element in the list will be replaced with those elements (line 21). Otherwise if the result is an empty element (line 22), the element in the list will be removed (line 23).

If the loop finishes and there are still elements in the list, then this means that the timeframe is not fully covered and the statement needs to be rejected (line 25).

Summed up, the statement is rejected if either no referenced tuple at all is found (line 5) or if not the whole timeframe of the tuple to be inserted is covered by referenced tuples' timeframes (line 24). The statement is accepted, if the whole tuple's timeframe is covered by referenced tuples' timeframes (line 14).

Update Trigger (Referencing Table)

Again, the update trigger on the referenced table uses the same implementation as the insert trigger. This is possible, because the update can once more be regarded as a combination of a deletion and an insertion.

Update Trigger (Referenced Table)

The next trigger that was implemented into the PostgreSQL kernel is the one on the referenced table that is called, when a tuple on the referenced relation is updated. Because this relation does already have an on update trigger (cf. section 4.1.3), in case of an update statement, both triggers will be called and so, both constraints will be checked.

This trigger uses an somewhat different approach. It does first retrieve the old values as well as the new values of the updated tuple. This does happen with the SPI_getvalue function, already seen before. It does then distinguish between three cases.

1. Neither the key attributes nor the timeframe attribute did change
2. The key attributes changed (independent of whether the timeframe attribute changed)
3. The timeframe attribute (but not the key attributes) changed

If case 1 takes place, then the update is not rejected by this trigger, because for sure, no OFK violation occurs.

Listing 4.8: Assert that the old key was not referenced

```
SELECT * FROM referencing_table_name
WHERE keyAttributesMatch
AND timeframe && old_timeframe_value;
```

If case 2 occurs, then it has to be ensured that the old tuple was not referenced by the OFK table (cf. Listing 4.8), otherwise the update is rejected.

In this case, *keyAttributesMatch* means that the origin values of the key attributes of the tuple to be updated are the same as the key attributes in the referencing relation. The *old_timeframe_value* stands for the timeframe value of the tuple before it has been updated.

Listing 4.9: Retrieve all intersection of timeframes

```
SELECT 'old_timeframe_value' * timeframe,
'new_timeframe_value' * timeframe,
FROM referencing_table_name WHERE keyAttributesMatch
AND timeframe && old_timeframe_value;
```

Also in case 3, the trigger has to make an additional verification. With the aid of the PostgreSQL range function intersection (represented as * in SQL), the trigger makes sure, that for all referencing tuples, the intersection of their timeframes with the old timeframe value is the same as the intersection with the new timeframe value. Listing 4.9 returns a table with all old and new intersections. If for at least one row in this table, the old intersection differs from the new one, the update is rejected. As already mentioned in chapter 3, this logic is based on the assumption that only one modification per transaction is allowed. It is therefore not possible, to insert another tuple into the referenced relation in the same transaction that would cover the missing part of the timeframe. Therefore, the intersection of the origin and the new tuple's timeframe must be the same.

Delete Trigger (Referenced Table)

The last trigger is recorded on the referenced table. It is executed, when a delete statement is called to make sure no tuple is deleted that is still being referenced. To find out if an OFK tuple depends on it, the query in Listing 4.10 is executed with the aid of SPI_execute. In the query, *keyAttributesMatch* means that the key attributes of the tuple to be deleted are the same, as the key attributes of the tuples in the referencing relation.

Listing 4.10: SQL statement to retrieve referenced tuples

```
SELECT * FROM referencing_table_name
WHERE keyAttributesMatch
AND timeframe && timeframe_value;
```

This query, which retrieves the referencing tuples, is only called until one match is found. This is handled with *SPI_execute*. If this is the case, the statement will be rejected. Otherwise the tuple will be deleted.

4.2 Implementation with User Defined Triggers

In this chapter, I will explain how the constraint triggers can be manually added to a relation by a user. I tried to keep the trigger logic of both approaches as similar as possible. In the single explanations of the user defined triggers, the main differences to the internal triggers will be mentioned.

All user defined triggers were adapted to the following relation schemas. That means, when someone wants to add those constraint triggers to his or her table, he or she is supposed to adapt the key attributes as well as the table names to the own relations.

- pknow(id integer, timeframe daterange)
- fknow(pkid integer *references pknow(id)*, timeframe daterange)

4.2.1 Creation of Ongoing Constraint Tables

As the user defined triggers have to be manually added to the relations, at the beginning two relations without any constraints have to be created. The statements are shown in Listings 4.11 and 4.12.

Listing 4.11: Example: Create Table with OPK constraint

```
CREATE TABLE pknow(id integer, timeframe daterange);
```

Listing 4.12: Example: Create Table with OPK constraint

```
CREATE TABLE fknow(pkid integer, name varchar);
```

After the creation of the tables, the user needs to add all six triggers (defined in the following), to the relations. The statement for adding a trigger to a relation is given in Listing 4.13. In the listing, *trigger_name* refers to the name of the trigger the user has created. Then *modification_type* is the event, after which the trigger should be executed. This can be *INSERT, UPDATE* or *DELETE*. *Relation_name* specifies, on which relation the statement is called. In this case, this is either the referencing relation *pknow* or the referenced relation *fknow*. Then the trigger calls the *trigger_function* defined in the trigger for each modified row.

Listing 4.13: Example: Create Table with OPK constraint

```
CREATE CONSTRAINT TRIGGER trigger_name
AFTER modification_Type
ON relation_name
FOR EACH ROW EXECUTE PROCEDURE trigger_function;
```

4.2.2 Ongoing Primary Key Constraint

Insert Trigger

The first trigger ensures that no OPK violation occurs when inserting a new tuple into the relation *pknow*. The complete trigger is shown in Listing 4.14. The variable *NEW* seen in the trigger contains the values of the row to be inserted [5].

Listing 4.14: Insert trigger on OPK relation that checks the OPK constraint

```
1  CREATE OR REPLACE FUNCTION check_pk_insert_update()
2  RETURNS trigger AS $check_pk_insupd$
3  DECLARE
4  r record;
5  cnt integer;
6  BEGIN
7  IF NOT (SELECT EXISTS (SELECT 1
8  FROM information_schema.columns WHERE
9  table_name=TG_TABLE_NAME AND column_name='timeframe'))
10 THEN
11     RAISE EXCEPTION 'timeframe required';
12 END IF;
13
14 cnt = 0;
15 FOR r in SELECT id, timeframe FROM pknow
16 WHERE id = NEW.id LOOP
17     IF r.timeframe != NEW.timeframe THEN
18         IF r.timeframe && NEW.timeframe THEN
19             RAISE EXCEPTION 'OPK violation';
20         END IF;
21     END IF;
```

```
22
23            IF r.timeframe = NEW.timeframe THEN
24                cnt = cnt + 1;
25                IF cnt = 2 THEN
26                    RAISE EXCEPTION 'OPK violation';
27                END IF;
28            END IF;
29        END LOOP;
30        RETURN NULL;
31        END;
32        $check_pk_insupd$ LANGUAGE plpgsql;
```

In lines 1 and 2, the name of the trigger function and the trigger itself is declared. They are used to add the trigger to the relation (cf. Listing 4.13). Lines 3 to 5 declare some variables used in the trigger. Then the next lines are for all triggers quite similar. In line 7 to 12 it is ensured, that the created relation does have a column timeframe. Otherwise it raises an exception, because the column timeframe is required in the trigger.

Then the trigger loops through all the tuples already present in the relation (cf. line 15) and checks if they do have the same key attributes (cd. line 16). For this relation the key attribute is *id*. For every tuple *r* with the same *id* found, the trigger checks if a tuple with overlapping timeframe does already exist in the relation (cf. line 17 to 28). This is comparable with Listing 4.7 in the internal approach.

If it finds a tuple that does indeed not have the same timeframe as the tuple to be inserted (cf. line 17), but whose timeframe does overlap with the new timeframe (cf. line 18), the insertion is rejected immediately. Because then it means, that a tuple with overlapping timeframe does exist in the relation, that is for sure, not the tuple to be inserted itself.

But since it could be that a tuple is inserted into the relation that has exactly the same values as a tuple already present in the relation, an additional check must be made (cf. line 23 to 28). First it is checked if a tuple exists with the same timeframe (cf. line 23). If this is the case, a counter is set to 1 (cf. line 24). Currently no violation occurs, as this could be the newly inserted tuple itself. But when a second tuple is found with the exactly same timeframe, the counter is set to two and then, a violation occurs and the statement is rejected. Because then, for sure at least one tuple with the same key attribute and the same timeframe exist in the relation that is not the tuple to be inserted itself.

If the loop ends and no error has been risen, the tuple is allowed to be inserted into the relation (cf. line 30). This happens, when the triggers returns null.

Update Trigger

Like already in the internal implementation, also here, the update trigger can use the same implementation as the insert trigger. The variable *NEW* that did stand for the tuple to be inserted in Listing 4.14, does in this case stand for the new values of the updated row [5]. As seen in case OPK-Violation 2 in section 3.3, the old values of the tuple are not needed to check if an OPK violation occurs.

4.2.3 Ongoing Foreign Key Constraint

Insert Trigger (Referencing Table)

As mentioned before, it was tried to keep the implementation of the internal triggers and the user defined ones as similar as possible. In this trigger, that is the insert trigger (that shares its code with the update trigger) on the referenced relation, there are some aspects that are different from the internal implementation. Those differences and the implementation of the user defined insert trigger will be shown with help of Listing 4.15.

Listing 4.15: Insert / Update trigger on OFK relation that checks the OFK constraint

```
1   CREATE OR REPLACE FUNCTION check_fk_insert_update()
2   RETURNS trigger AS $check_fk_insupd$
3   DECLARE
4   r record;
5   cnt integer;
6   notcovered daterange[];
7   tempdiff daterange[];
8   t daterange;
9   isToRemove boolean;
10  BEGIN
11  /* [...] check if column timeframe exist on
12  * referencing and referenced relations */
13  cnt := 0;
14  notcovered := array_append(notcovered, NEW.timeframe);
15  FOR r IN SELECT id, timeframe FROM pknow
16  WHERE id = NEW.pkid AND timeframe && NEW.timeframe
17  LOOP
18      cnt = cnt + 1;
19      isToRemove = false;
20      FOREACH t IN ARRAY notcovered
21      LOOP
22          IF r.timeframe && t THEN
23              isToRemove := true;
24          END IF;
25          IF NOT isempty(range_minus_first(t, r.timeframe))
26          THEN
27              tempdiff := array_append(tempdiff,
28              range_minus_first(t, r.timeframe));
29          END IF;
30          IF NOT isempty(range_minus_second(t, r.timeframe))
31          THEN
32              tempdiff := array_append(tempdiff,
33              range_minus_second(t, r.timeframe));
34          END IF;
```

```
35              IF isToRemove = true THEN
36                  notcovered := array_remove(notcovered, t);
37              END IF;
38          END LOOP;
39          IF array_length(tempdiff, 1) != 0 THEN
40              notcovered := array_cat(notcovered, tempdiff);
41              tempdiff := '{}';
42          END IF;
43      END LOOP;
44
45      IF cnt = 0 THEN
46          RAISE EXCEPTION 'OFK violation';
47      END IF;
48
49      IF array_length(notcovered, 1) != 0 THEN
50          RAISE EXCEPTION 'OFK violation';
51      END IF;
52      RETURN NULL;
53      END;
54      $check_fk_insupd$ LANGUAGE plpgsql;
```

The main logic of the trigger is the same as the one used for the internal trigger. When inserting a tuple into the referencing relation, first all tuples with the same *id* are retrieved. Then the timeframes of those tuples are subtracted from the newly inserted tuple. If all timeframes were subtracted and there is no remaining part of the inserted timeframe, then it means that the timeframe was fully covered and no constraint violation occurs. What is different are on the one hand the functions that are used to calculate the difference. The internal trigger uses the native function range_minus_all that calculates the difference of two ranges and returns a list of ranges. As SQL cannot handle the returned lists, this function had to be replaced in the user defined triggers by the two functions range_minus_first and range_minus_second (the functions were provided by my supervisor Yvonne Mülle). These function do either return a range or are empty. On the other hand, also another type is used to store the remaining timeframes. As it was a C list in the internal trigger, it is an array in the user defined trigger.

As the beginning of the trigger is very similar to the insert trigger on the referenced relation, lines 1 to 12 will not be explained any further. In line 13 a counter is initialized with 0. The counter counts the tuples found with the same *id*. Then in line 14, the array *notcovered* that contains all parts of the timeframe of the tuple to be inserted, that are not covered yet, is initialized with the value of the timeframe of the tuple to be inserted.

In the next step (line 15) all tuples with the same *id* present in the referenced relation are retrieved. For every retrieved tuple, the counter is increased by 1. If at the end of the loop the counter is still zero, then no tuples with the same *id* were found in the referenced relation, and therefore, the insertion is rejected (line 45). Otherwise for every timeframe *t* in the array *notcovered* (line 20), it is checked if it overlaps with the current tuple's (*t*) timeframe from the referenced relation (line 22). If so, it means that the difference will not be empty and

t has to be removed from *notcovered* and replaced by the calculated difference. Therefore *isToRemove* is set to true (cf. line 23). Then the difference is calculated. As mentioned above, the native PostgreSQL function range_minus_all that would return one or two ranges when the difference is not empty, cannot be used. So the difference is calculated in two steps using the functions range_minus_first and range_minus_second (cf. lines 30 - 37). For both functions, if the returned range is not empty, the range is added to a temporal array called *tempdiff* (cf. line 27 and 32). As already mentioned, the timeframe *t* is removed from *notcovered* when the variable *isToRemove* is true (cf. line 36).

After all differences of the current referenced tuple's timeframe and the timeframes in the array *notcovered* have been calculated, the timeframes in the array *tempdiff* are pushed to the *notcovered* array (cf. line 40) and the array *tempdiff* is cleared (cf. line 41). When the end of the loop is reached and the array *notcovered* is not empty, then a violation occurs and the statement is rejected (cf. line 50). Otherwise the tuple is inserted (cf. line 52).

Update Trigger (Referencing Table)

Again, the trigger for the update case is the same as the trigger for the insert case. The variable *NEW* that did stand for the tuple to be inserted in Listing 4.15 does in this case stand for the new values of the updated tuple [5]. As seen in case OFK-Violation 2 in section 3.4, the old values of the tuple are not needed to check if an OFK violation occurs in this case.

Update Trigger (Referenced Table)

In the next trigger, it is ensured that no violation occurs when an update on the referenced relation is performed. As already the internal trigger, the user defined trigger distinguishes between three cases.

1. Neither the key attributes nor the timeframe attribute did change
2. The key attributes changed (independent of whether the timeframe attribute changed)
3. The timeframe attribute (but not the key attributes) changed

If case 1 takes place, then for sure no violation occurs. In this case, the update will not be rejected (cf. Listing 4.16 line 10 - 12).

Case 2 is handled in lines 14 to 22. The variable *NEW* does again contain the updated tuple whereas *OLD* contains the origin values of the tuple. If the new *id* is not the same as the old *id* (cf. line 14) it must be ensured, that the old tuple was not referenced. This is done in lines 15 to 17. There, all rows are counted that were referenced in the referencing relation. If this count is greater than 0 (cf. line 18), then the tuple was referenced and the update is rejected (cf. line 19). Otherwise the tuple is updated (cf. line 20)

Case 3 is handled in lines 24 to 34. This section can only be reached if not the id but the timeframe did change, and therefore, no additional *if statement* is required. The trigger performs the same query (lines 24 - 28) that was already seen in the internal implementation. The query retrieves all tuples *r* in the referencing relation that were referencing the old tuple.

Then for each of these tuples it gets the intersection (represented as * in SQL) of the tuple's timeframe with the origin and the updated tuple's timeframe. If for any of these tuples, the intersections are not the same (cf. line 29) then the update is rejected (cf. line 30).

If line 33 is reached, then no violation occurs when updating the tuple. Therefore the modification is completed.

Listing 4.16: Update trigger on OPK relation that checks the OFK constraint

```
1  CREATE OR REPLACE FUNCTION check_pkfk_update()
2  RETURNS trigger AS $check_pkfk_upd$
3  DECLARE
4  r record;
5  cnt integer;
6  BEGIN
7  /* [...] check if column timeframe exist on
8  * referencing and referenced relations */
9
10  IF OLD.id = NEW.id AND OLD.timeframe = NEW.timeframe THEN
11      RETURN NULL;
12  END IF;
13
14  IF OLD.id != NEW.id THEN
15      SELECT COUNT(*) FROM fknow
16      WHERE pkid = OLD.id AND timeframe && OLD.timeframe
17      INTO cnt;
18      If cnt > 0 THEN
19          RAISE EXCEPTION 'OFK violation';
20      END IF;
21      RETURN NULL;
22  END IF;
23
24  FOR r IN
25  SELECT OLD.timeframe * timeframe as oldtimeframe,
26  NEW.timeframe * timeframe as newtimeframe FROM fknow
27  WHERE pkid = OLD.id AND timeframe && OLD.timeframe
28  LOOP
29      IF r.oldtimeframe != r.newtimeframe THEN
30          RAISE EXCEPTION 'timeframe required';
31      END IF;
32  END LOOP;
33  RETURN NULL;
34  END;
35  $check_pkfk_upd$ LANGUAGE plpgsql;
```

Delete Trigger (Referenced Table)

The last user defined trigger is seen in Listing 4.17. The idea is simple. The query in line 10 and 11 counts the number of tuples in the referencing relation that have the same *id* as the tuple to be deleted and whose timeframe do overlap with the timeframe of the tuple to be deleted. If this counter is greater than 0 (line 13) the deletion is rejected (line 14), as the tuple to be deleted is still referenced in the referencing relation. Otherwise, the deletion is performed (line 16).

Listing 4.17: Delete trigger on OPK relation that checks the OFK constraint

```
1  CREATE OR REPLACE FUNCTION check_pkfk_deletion()
2  RETURNS trigger AS $check_pkfk_del$
3  DECLARE
4  r pknow%rowtype;
5  cnt integer;
6  BEGIN
7  /* [...] check if column timeframe exist on
8  * referencing and referenced relations */
9
10 SELECT COUNT(*) FROM fknow
11 WHERE pkid = OLD.id AND timeframe && OLD.timeframe
12 INTO cnt;
13 If cnt > 0 THEN
14     RAISE EXCEPTION 'OFK violation';
15 END IF;
16 RETURN NULL;
17 END
18 $check_pkfk_del$ LANGUAGE plpgsql;
```

5 Experimental Evaluation

In this chapter the implementation will be experimentally evaluated. First, the setup of the experiment will be described. Then the results will be presented and discussed.

5.1 Experiment Setup

The experiments were carried out on a ubuntu virtual machine given 3 GB RAM. The windows machine the VM was running on has a 2.50 GHz Intel CPU. The PostgreSQL server ran inside of Eclipse (version 3.8.1). In the evaluation, different queries were called and their execution time was measured with the aid of the PostgreSQL internal function timing.

5.1.1 Relations

To get an idea of the performance the statements were always executed on two different relations. On the one hand the relations had the above explained OPK and OFK constraint with the internal constraint triggers (cf. section 4.1). On the other hand, tables with the user defined triggers were used (cf. section 4.2). To distinguish, the first will be referenced as the internal and the second as the user defined triggers.

5.1.2 Data Set

For both implementation approaches the same data set was used, this will be described in this section. The data set was created for a simple relation with the schema (id integer, timeframe daterange) with *id* being the key attribute of the relation. To make sure no OFK violation occurs while inserting the data set, the same data set was initially used for both (OPK as well as OFK) relation.

The dataset was created with a little Java program that takes random integers i between 0 and 100. For each of them, it generates $2 * i$ tuples matching the schema given above. Each of this tuple duo being created contains in overall approximately 18.75% of the cases ongoing timestamps, either in the lower bound or the upper bound. Finally, the generated data set contains 1,046 tuples.

5.1.3 Test Case Selection

Overall 94 statements were executed and their duration measured. The statements were chosen, so that every possible case of all triggers is reached at least once. The triggers were also called with tuples including ongoing as well as fixed timeframes. In total 710 measurements

were made, half of them on the internal, half on the user defined tables. Because it would be beyond the scope of this study to have a look at all 94 tests, this section will present a selection of the statements. It will be shown, what test cases will be evaluated in the next section and why they are important. All test cases have the common purpose, to evaluate if the internal implementation is significantly faster than the user defined approach. Besides, with the overall comparison of the test cases, it can be evaluated, how fast the single constraint triggers are in respect to the others. Each described case will be summed up in the terms *statement, description, triggers* and *expected result*.

The first two test cases (Test Case 1.1 and Test Case 1.2) import the data set mentioned in section 5.1.2 in the OPK relations respectively in the OFK relations. The statements were chosen because they copy about 1,000 tuples into the tables and for each of them a constraint trigger is called. Therefore, it can be analysed, how fast the implementation is for relatively big data.

Test Case 1.1	
Statement	COPY opk_table FROM 'output.txt' USING DELIMITERS ':'
Description	Imports the data set into the OPK relations.
Triggers	Insert trigger on the OPK relation
Expected result	No violation, 1,046 tuples imported.

Test Case 1.2	
Statement	COPY ofk_table FROM 'output.txt' USING DELIMITERS ':'
Description	Imports the data set into the OFK relations.
Triggers	Insert trigger on the OFK Relation
Expected result	No violation, 1,046 tuples imported

The next test case includes Test Cases 2.1 to 2.4. Test Case 2.1 inserts a tuple into the OPK relation, so that no violation occurs. The tuple is chosen, so that the *id* is the one, that the least number of tuples have in the data set. For this data set, the *id* is 7 and includes 14 tuples. The timeframe of the tuple to be inserted contains only fixed timestamps.

Test Case 2.2 inserts a tuple into the OPK relation so that no violation occurs. The tuple is in this case chosen, so that the *id* is the one, that the most number of tuples have in the data set. For this data set, the *id* is 87 and includes 174 tuples. The timeframe of the tuple to be inserted contains only fixed timestamps.

Test Case 2.3 is very similar to Test Case 2.1 and Test Case 2.4 is very similar to Test Case 2.2. The only difference is, that Test Cases 2.3 and 2.4 do insert tuples with ongoing timeframes.

When comparing Test Case 2.1 and 2.2 as well as Test Case 2.3 and 2.4 an important question can be answered: Does the insertion take longer, when more tuples with the same key attributes are already present in the relation? It is expected that this is indeed true, as in this case, multiply tuple's timeframes have to be compared.

When comparing Test Case 2.1 and 2.3 as well as Test Case 2.2 and 2.4 the question can be answered, if the insertion takes longer, when a tuple with an ongoing timestamp is inserted. It

is expected that this is true, as in this case, another function has to be called to check if two timeframes overlap.

Test Case 2 is also interesting because it is the *worst case* of the insertion into a relation with OPK constraint. This is because no violation occurs, and therefore every single tuple already present in the relation has to be checked.

Test Case 2.1

Statement	INSERT INTO opk_table VALUES (7, '[1970-01-01, 1971-01-01)')
Description	A tuple that does not lead to a violation is inserted into the OPK relation. here are 14 tuple's with the same *id* present in the relation. The tuple's timeframe does only have fixed timestamps.
Triggers	Insert trigger on the OPK Relation
Expected result	No violation (but the tuple has to be deleted after every insertion)

Test Case 2.2

Statement	INSERT INTO opk_table VALUES (87, '2050-01-01, 2051-01-01)')
Description	A tuple that does not lead to a violation is inserted into the OPK relation. There are 174 tuple's with the same *id* present in the relation. The tuple's timeframe does only have fixed timestamps.
Triggers	Insert trigger on the OPK Relation
Expected result	No violation (but the tuple has to be deleted after every insertion)

Test Case 2.3

Statement	INSERT INTO opk_table VALUES (7, '[1971-01-01, min1980-01-01NOW1972-01-01)')
Description	A tuple that does not lead to a violation is inserted into the OPK relation. There are 14 tuple's with the same *id* present in the relation. The tuple's timeframe does include an ongoing timestamp.
Triggers	Insert trigger on the OPK Relation
Expected result	No violation (but the tuple has to be deleted after every insertion)

Test Case 2.4

Statement	INSERT INTO opk_table VALUES (87, '2051-01-01, min2060-01-01NOW2052-01-01)')
Description	A tuple that does not lead to a violation is inserted into the OPK relation. There are 174 tuple's with the same *id* present in the relation. The tuple's timeframe does include an ongoing

	timestamp.
Triggers	Insert trigger on the OPK Relation
Expected result	No violation (but the tuple has to be deleted after every insertion)

Test Case 3 calls the same insert statement on the OFK relation that does not lead to a violation ten times. This is possible without a deletion of the tuple between the statements, because in the OFK relation, the same tuple can exist multiple times. The tuple to be inserted is chosen, so that as much tuples as possible with the same *id* exist in the referenced relation (the *id* is 87 and the number of referenced tuples is 174). Then the timeframe (which includes an ongoing timestamp) is so large that all 174 tuples are needed to cover it. This is the worst case scenario of the insertion of a tuple into a relation with an OFK constraint.

Test Case 3	
Statement	INSERT INTO ofk_table VALUES (87, '[1960-01-01, min2047-01-01NOW2037-01-01)')
Description	Insertion of a tuple into the OFK relation, whose timeframe is covered over 174 tuples
Triggers	Insert trigger on the OFK Relation
Expected result	No violation

Test Cases 4.1 and 4.2 are evaluated together, as both of them call both update triggers on the OPK relations. But Test Case 5 will lead to an OPK violation and Test Case 6 to an OFK violation. Like that it is possible to compare the influence of the violation type on the execution time.

Test Case 4.1	
Statement	update opk_table set timeframe = '[min1994-06-30NOW1994-01-01, 1995-01-01)' where timeframe = '[min1994-06-30NOW1994-02-01,1995-01-01)' and id = 37
Description	An update statement is called on the OPK relation.
Triggers	Both update triggers on the OPK relation (one that checks the OPK constraint and one that checks the OFK constraint)
Expected result	OPK violation

Test Case 4.2	
Statement	update opk_table set timeframe = '[1967-01-01, 1967-06-29)' where timeframe = '[1967-01-01,1967-06-30)' and id = 48
Description	An update statement is called on the OPK relation.
Triggers	Both update triggers on the OPK relation (one that checks the OPK constraint

	and one that checks the OFK constraint)
Expected result	OFK violation

In Test Case 5 the update trigger on the OFK relation will be covered. As the update trigger shares its code with the insert trigger that is already analysed in Test Case 3 and does not lead to a violation, this time a statement is chosen, that does actually cause an OFK violation, to also cover this case.

Test Case 5	
Statement	update ofk_table set timeframe = '[2007-06-30, NOW2008-06-30)' where pkid = 48 and timeframe = '[2007-06-30, NOW2008-01-01)
Description	An update statement is called on the OFK relation.
Triggers	Update trigger on the OFK relation
Expected result	OFK Violation

The last test case covers the remaining constraint trigger. This is the delete trigger on the OPK relation, which checks the OFK constraint.

Test Case 6	
Statement	delete from pk2 where id = 37 and timeframe = '[1983-01-01, 1983-02-01)'
Description	Tries to delete a referenced tuple.
Triggers	Delete trigger on the OPK relation
Expected result	OFK violation

5.2 Test Results and Evaluation

After having defined the test cases, this section will present the test results. For each case, average internal and user defined execution time, as well as their coefficient $(= \frac{\varnothing UserDefinedTime}{\varnothing InternalTime})$ will be listed. From the definition of the coefficient it can be concluded, that whenever the coefficient is greater than 1, the internal execution has been faster on average. The results will then be evaluated. At the end, also a comparison and evaluation of all measurements will be given.

Test Cases 1.1 and 1.2 are the first ones to be analyzed. For both relations (OPK and OFK) the copying into the table with the internal triggers is about two times faster than the copying into the tables with the user defined triggers. This was predictable because the internal triggers can directly use the native functions, whereas the user defined triggers need to work with the SQL functions. Also the fact, that a user using the internal implementation does not even have to wait a second until the insertion of 1,000 tuples in either constraint relation is completed, speaks in favour of the internal approach.

But when measuring the time that is needed to insert those 1,046 tuples into a relation with no constraint, the execution takes only about *10 ms*. 10 ms is approximately the time that is used to insert one single tuple into a relation with an OPK constraint (cf. Test Case 2). As the table without a constraint does not have to call a trigger for each row, I do not think it is surprising that the runtime for the insertion without a constraint is about 100 times faster. Because the tables with the constraints do have to call the triggers for each row and with each new row, the already inserted rows have to be considered too.

Comparing the insertion into the OPK relation with the one in the OFK relation, the first one is in the internal as well as in the user defined approach faster. This result is as expected, as the insert trigger in the OPK relation does only have to check if a tuple with matching key attributes and an overlapping timeframe exists. This check must also be made by the OFK trigger, but in addition the trigger must also ensure, that the whole timeframe of the inserted tuple is covered by the corresponding tuples. But although the execution time of the insertion into a relation with OPK constraint is faster, the difference is small. It is only about 10% faster. This is probably, because both relations use the same data set and therefore, the timeframe of a referencing tuple is always covered by only one single referenced tuple.

Test Case 1.1	
Statement	COPY opk_table FROM 'output.txt' USING DELIMITERS ':'
Triggers	Insert trigger on the OPK relation
Result	No violation occurs, 1046 tuples imported.
∅ **Internal time [ms]**	825.268
∅ **User defined time [ms]**	1,668.609
Coefficient	2.02

Test Case 1.2	
Statement	COPY ofk_table FROM 'output.txt' USING DELIMITERS ':'
Triggers	Insert trigger on the OFK relation
Result	No violation, 1046 tuples imported
∅ **Internal time [ms]**	908.05
∅ **User defined time [ms]**	2,052.121
Coefficient	2.25

The next test cases are the insertions of different tuples into the OPK relations. On the one hand, tuples were inserted into the relation, so that only a few tuples with the same key attribute were already present in the relation (Test Case 2.1 and 2.3). On the other hand, tuples were inserted into the relation so that many tuples with the same key attribute were present (Test Case 2.2 and Test Case 2.4).

For both cases, again two different types of tuples were inserted. While two of the tuples' timeframes (Test Case 2.1 and 2.2) do have fixed timestamps, the other two tuples' timeframes (Test Case 2.3 and 2.4) include ongoing timestamps. Table 5.1 sums up the measured values of the four different test cases, each for the internal and the user defined implementation.

Table 5.1: Overview Test Case 2

	internal only fixed ts	**internal ongoing ts**	**user defined only fixed ts**	**user defined ongoing ts**
only a few tuples with same id present	7.423 ms	9.0 ms	8.696 ms	9.23 ms
many tuples with same id present	9.51 ms	9.66 ms	9.83 ms	11.19 ms

The question if the internal triggers are faster than the user defined triggers for the test cases 2.1 to 2.4 can be answered with yes, as all coefficients are greater than 1. This can also be seen in Table 5.1. The values in the columns *user defined - only fixed ts* and *user defined - ongoing ts* are always greater than the values in the columns *internal - only fixed ts* and *internal - ongoing ts*.

An other question that was asked in Test Case 2 is: "Does the insertion take longer, when more tuples with the same key attributes are already present in the relation?" It was expected, that this is true, as in this case, more tuples have to be checked. The questions can be answered comparing the rows *only a few tuples with same id present* and *many tuples with same id present*. Table 5.2 extends Table 5.1 with the coefficients $(= \frac{\varnothing manytuples}{\varnothing fewtuples})$. As all of them are greater than 1, the question can be answered with yes.

Table 5.2: Overview - a few tuples to check vs. many tuples to check

	internal only fixed ts	**internal ongoing ts**	**user defined only fixed ts**	**user defined ongoing ts**
only a few tuples with same id present	7.423 ms	9.0 ms	8.696 ms	9.23 ms
many tuples with same id present	9.51 ms	9.66 ms	9.83 ms	11.19 ms
$\frac{\varnothing manytuples}{\varnothing fewtuples}$	1.28	1.07	1.13	1.21

A last question that was asked in Test Case 2 is: "Does the insertion take longer when a tuple with ongoing timestamps is inserted, than when a tuple with only fixed timestamps is inserted?" Is was expected that this is true, as an additional function has to be called to check if two timeframes with ongoing timestamps overlap. Table 5.3 extends Table 5.1 with the coefficients $(= \frac{\varnothing ongoing}{\varnothing fixed})$. As all of them are greater than 1, the question can be answered with yes.

As mentioned in section 5.1.3 Test Case 2 includes the worst case scenarios for the insertion into a relation with an OPK constraint, as no violation occurs and therefore, every tuple already present in the relation has to be checked. As all statements on the relations with the internal triggers (and in 3 out of 4 cases on the user defined triggers) do take less than 10 ms, this is a good result for a worst case scenario.

To be able to better evaluate the result, I did also insert the tuples from Test Cases 2.1 to 2.4 ten times each into a relation without any constraint at all. Table 5.4 sums up the measured values for the insertion into a relation without any constraint, into a relation with the internal implementation of the constraints and into a relation with the user defined triggers. Especially

Table 5.3: Overview - fixed timestamps vs. ongoing timestamps

	internal only fixed ts	**internal ongoing ts**	$\frac{\varnothing ongoing}{\varnothing fixed}$	**user defined only fixed ts**	**user defined ongoing ts**	$\frac{\varnothing ongoing}{\varnothing fixed}$
only a few tuples with same id present	7.423 ms	9.0 ms	1.21	8.696 ms	9.23 ms	1.06
many tuples with same id present	9.51 ms	9.66 ms	1.02	9.83 ms	11.19 ms	1.14

Table 5.4: Overview Test Case 2

	Average time (no constraint)	**Average time (internal)**	$\frac{\varnothing internal}{\varnothing noconstraint}$	**Average time (user defined)**	$\frac{\varnothing userdefined}{\varnothing noconstraint}$
Test Case 2.1	7.41 ms	7.42 ms	1.0	8.7 ms	1.17
Test Case 2.2	9.0 ms	9.51 ms	1.06	9.83 ms	1.09
Test Case 2.3	7.8 ms	9.0 ms	1.15	9.23 ms	1.83
Test Case 2.4	8.33 ms	9.66 ms	1.16	11.19 ms	1.34

in the Test Case 2.1 and 2.2 where there were only tuples with fixed timestamps present in the relation, the execution time of an insertion into a relation without any constraint or a relation with the internal implementation of the OPK constraint, are very similar. But also in cases 2.3 and 2.4 where there are tuples with ongoing timestamps present, the insertion into a relation without a constraint is not even 20% faster. Therefore it can be said that the execution times for the worst case scenarios are an acceptable result.

Test Case 2.1

Statement	INSERT INTO opk_table VALUES (7, ’[1970-01-01, 1971-01-01)’)
Triggers	Insert trigger on the OPK Relation
Result	No violation (but the tuple has to be deleted after every insertion)
∅ Internal time [ms]	7.423
∅ User defined time [ms]	8.696
Coefficient	1.17

Test Case 2.2

Statement	INSERT INTO opk_table VALUES (87, ’2050-01-01, 2051-01-01)’)
Triggers	Insert trigger on the OPK Relation
Result	No violation (but the tuple has to be deleted after every insertion)
∅ Internal time [ms]	9.51
∅ User defined time [ms]	9.83

Coefficient	1.03

Test Case 2.3

Statement	INSERT INTO opk_table VALUES (7, '[1971-01-01, min1980-01-01NOW1972-01-01)')
Triggers	Insert trigger on the OPK Relation
Result	No violation (but the tuple has to be deleted after every insertion)
∅ Internal time [ms]	9.0
∅ User defined time [ms]	9.23
Coefficient	1.03

Test Case 2.4

Statement	INSERT INTO opk_table VALUES (87, '2051-01-01, min2060-01-01NOW2052-01-01)')
Triggers	Insert trigger on the OPK Relation
Result	No violation (but the tuple has to be deleted after every insertion)
∅ Internal time [ms]	9.66
∅ User defined time [ms]	11.19
Coefficient	1.15

In Test Case 3 the same tuple was inserted ten times. Due to that, it was surprising, that the measured values are subject to a large variability (cf. Figure 5.1). I assume that is, because the logic of the insert trigger on a relation with OFK constraint is relatively complex.

Test Case 3

Statement	INSERT INTO ofk_table VALUES (87, '[1960-01-01, min2047-01-01NOW2037-01-01)')
Triggers	Insert trigger on OFK relation
Result	No violation
∅ Internal time [ms]	10.9
∅ User defined time [ms]	16.98
Coefficient	1.56

The measurements also show, that in Test Case 3 the average internal execution time is about 1.5 times times faster than the user defined execution time. The main reason for that is probably that the user defined implementation of the insert trigger on the OFK relations are quite different as explained in section 4.2.3. The internal implementation can use the

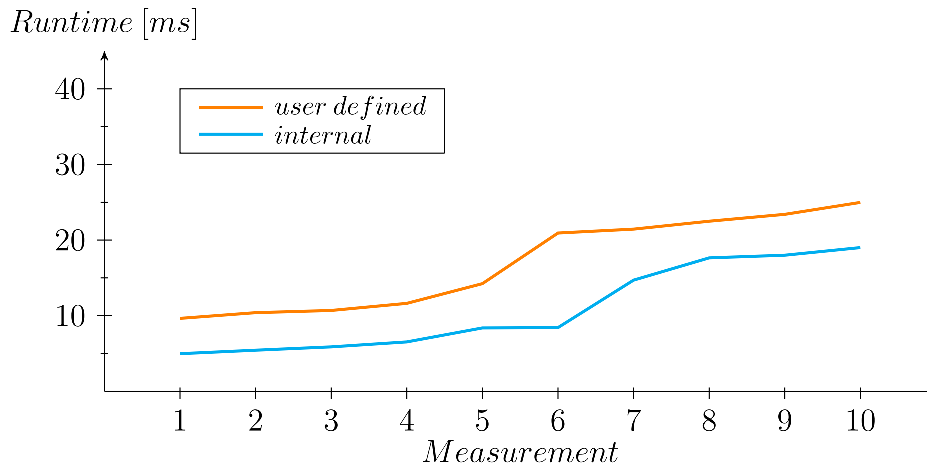


Figure 5.1: Measurements Test Case 3

Table 5.5: Overview Test Case 2.4 and Test Case 3

	Average time (internal)	**Average time (user defined)**
Test Case 2.4	9.66 ms	11.19 ms
Test Case 3	10.9 ms	16.98 ms
$\frac{\varnothing TestCase3}{\varnothing TestCase2}$	1.23	1.52

PostgreSQL internal function range_minus_all, whereas the user defined trigger needs to work with the slower SQL functions range_minus_first and range_minus_second.

Table 5.5 compares the execution times of Test Case 2.4 and Test Case 3. Both of them are worst case scenarios, one for the insertion of a tuple into a relation with OPK constraint and one for a relation with OFK constraint. The scenario of the insertion into the relation with OPK constraint is in the internal and user defined approach faster than into the relation with the OFK constraint. This result is as expected, as the insert trigger in the OPK relation does only have to check if for all the retrieved tuples with the same *id*, the timeframes overlap with the the timeframe of the tuple to be inserted. This check must also be made by the OFK trigger, but in addition, all the timeframes of the retrieved tuples (this are 174 in Test Case 3) must then be subtracted from the timeframe of the tuple to be inserted, to make sure, the whole timeframe is covered.

Test Case 4.1

Statement	update opk_table set timeframe = '[min1994-06-30NOW1994-01-01, 1995-01-01)' where timeframe = '[min1994-06-30NOW1994-02-01,1995-01-01)' and id = 37
Triggers	Both update triggers on the OPK relation (one that checks the OPK constraint and one that checks the OFK constraint)

Result	OPK violation
∅ **Internal time [ms]**	12.05
∅ **User defined time [ms]**	2.144
Coefficient	0.18

Test Case 4.2	
Statement	update opk_table set timeframe = '[1967-01-01, 1967-06-29)' where timeframe = '[1967-01-01,1967-06-30)' and id = 48
Triggers	Both update triggers on the OPK relation (one that checks the OPK constraint and one that checks the OFK constraint)
Result	OPK violation
∅ **Internal time [ms]**	3.796
∅ **User defined time [ms]**	23.625
Coefficient	6.22

Test Cases 4.1 and 4.2 are evaluated together, as both of them call the same triggers, namely the two update triggers on the referenced relation. The difference is, that Test Case 4.1 leads to an OPK violation whereas Test Case 4.2 leads to an OFK relation.

Inspecting the two coefficients and the graph shown in Figures 5.2 and 5.3, it is striking that on the one hand in case 4.1 where the query leads to an OPK relation, the user defined implementation is more than five times faster. But in return, when it leads to an OFK relation, the internal implementation is more than six times faster. After a while of debugging, it became clear that in the user defined variant, the trigger checking the OPK constraint was always called before the other one. On the internal implementation, it was exactly the contrary. This explains the results seen in Test Cases 4.1 and 4.2.

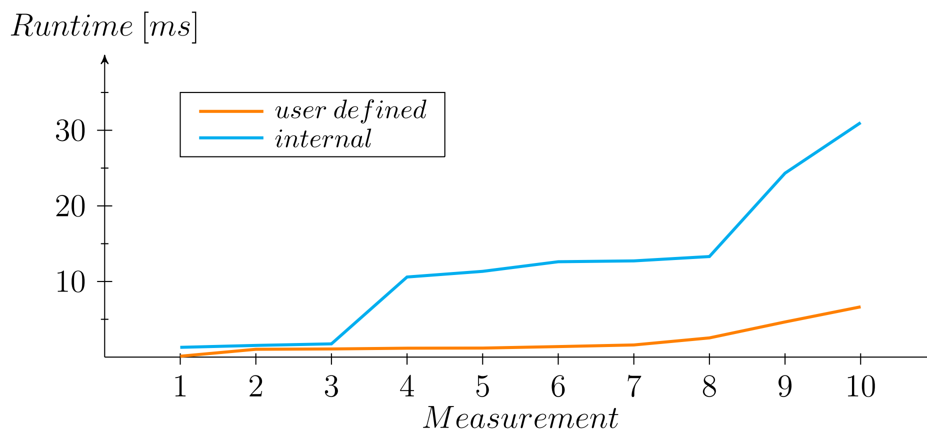


Figure 5.2: Measurements Test Case 4.1

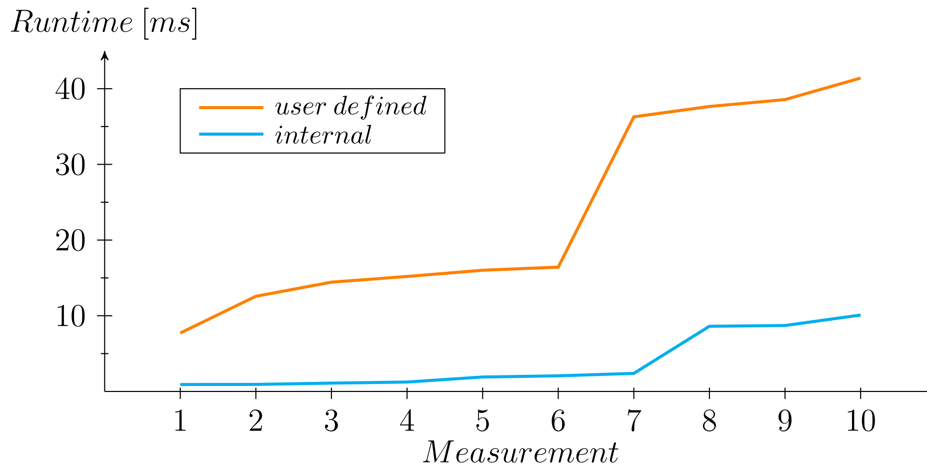


Figure 5.3: Measurements Test Case 4.2

To be more precise, in case 4.1 the user defined implementation does first call the OPK trigger - that is per se already faster than the OFK trigger - as this leads to a violation, the second update trigger is not called at all. The internal implementation does first go through the complete logic of the update trigger that checks the OFK constraint. But the OFK constraint is not violated, therefore the second trigger is also called. In contrast, in case 4.2, the internal implementation does this time only need to call one trigger, namely the update trigger checking the OFK constraint. But here, the user defined variant must call both triggers.

To explain the order of the execution of the triggers: If multiply triggers are defined for the same event on the same relation, PostgreSQL will fire the triggers in alphabetical order by their name [5]. As I did not know this when creating the triggers, I named the triggers randomly. In the internal implementation the triggers are named *OG_PK_ConstraintTrigger_upd* (the OPK constraint trigger) and *OG_FK_ConstraintTrigger_upd_pk* (the OFK constraint). But in the user defined approach the triggers are called *check_pk_upd* (the OPK constraint trigger) and *check_pkfk_upd* (the OPK constraint trigger).

Test Case 5	
Statement	update ofk_table set timeframe = '[2007-06-30, NOW2008-06-30)' where timeframe = '[2007-06-30, NOW2008-01-01)' and id = 48
Triggers	Update trigger on the OFK relation
Result	OPK violation
∅ Internal time [ms]	4.341
∅ User defined time [ms]	4.835
Coefficient	1.11

Test Case 5 calls a statement that executes the update trigger on the OFK relation. As mentioned in section 4.1.3, the implementation is the same as the one for the insert trigger.

But when comparing the results of Test Case 3, where the insert trigger is called, with the results of this test case, it is striking that in this case the average user defined and internal time are closer together. The second thing that stands out is, that the execution times of the single measurements are quite widely distributed, what can also be seen in Figure 5.4. Here, it is difficult to say what exactly is the reason for this varying result. But a point that could be important is that the PostgreSQL function that measures the execution time, does measure the time of the whole statement and not only the time of the trigger itself. So it could be that update statements in general are executing some background work and that therefore, the trigger does not have a big influence on the execution time.

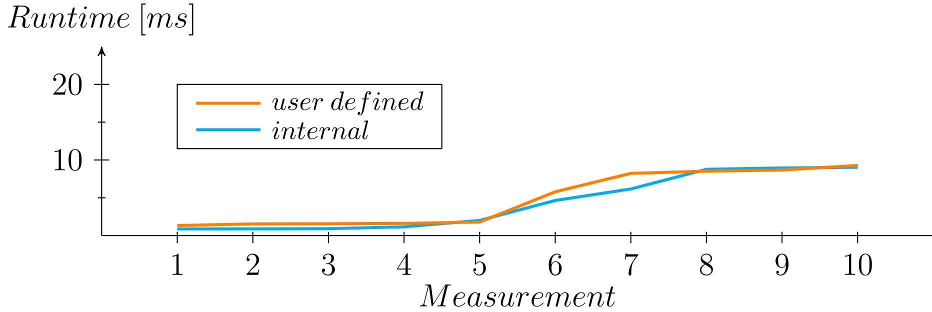


Figure 5.4: Measurements Test Case 5

The last test case to be evaluated is presented in Test Case 6 and Figure 5.5. Although in this case the average execution time of the internal implementation is almost 1.9 times as fast as the user defined approach, it is striking here, that like in Test Case 3, the measurements are widely distributed (in both cases the longest execution time is more than eight times bigger than the shortest one). It does again seem like the delete statement executes background work that is not influenced by the triggers. But still, a duration of less than ten milliseconds for a deletion is a good execution time. Due to the irregularities in both approaches and the fast execution times, it is not sure that the internal implementation is always faster for deletions. In the measurements I made, there have also been some cases (similar to Test Case 6), where the user defined approach has been faster. I assume this is due to the irregularities of the execution times.

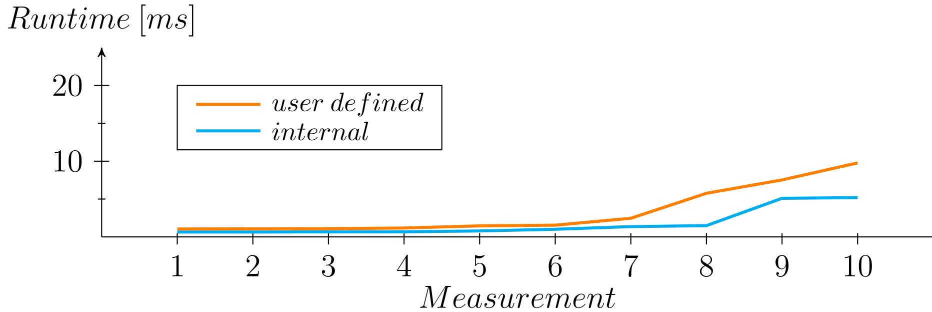


Figure 5.5: Measurements Test Case 6

Test Case 6

Statement	delete from pk2 where id = 37 and timeframe = '[1983-01-01, 1983-02-01)'
Triggers	Delete trigger on the OPK relation
Result	No violation
∅ **Internal time [ms]**	1,756
∅ **User defined time [ms]**	3,291
Coefficient	1,87

After having evaluated some special cases, in the next step a general evaluation is given. Of all 710 measurements made, the total average time of the internal and user defined trigger was calculated. Thereby it can be seen, that overall the internal approach is nearly twice as fast. This is a clear sign that the ongoing integrity checking is faster when using the internal implementation.

∅ **Internal time [ms]**	12.083
∅ **User defined time [ms]**	23.741
Coefficient	1.96

When removing the import statements, which have a big influence on the average time, from the total time, both average times fall by more than a third. Nonetheless, the coefficient does not even fall by 10%.

∅ **Internal time [ms]**	7.204
∅ **User defined time [ms]**	13.220
Coefficient	1.83

Inspecting those times, it is unambiguous, that the internal implementation is faster than the user defined approach. Summing up the results from the test cases, it is especially beneficial to use the internal implementation instead of the user defined one, when a lot of insert statements (on OPK or OFK) tables are made. But also the other triggers are in most cases still faster. An exception has to be made in the case of the update triggers on the OPK relation. With the current naming, the user defined triggers will most probably be faster when an OPK violation occurs, because then, the user defined update trigger on the OPK relation that checks the OFK constraint is not called - presupposed there is a relation with an OFK constraint referencing that relation. But in the internal implementation, the update trigger that checks the OFK constraint is called first, and therefore, the execution time in OPK violation cases is slower. But in general, it can still be said, that in aspect of the execution time, it is worth to use the internal implementation.

Another advantage of the internal implementation is that it is quite simple to create a table with an OPK respectively OFK constraint. If it would for some reason not be possible to add

a constraint on that table, it may be because the column timeframe is missing, PostgreSQL will immediately tell and cancel the create statement. But when using user defined triggers, adding the ongoing integrity constraints becomes more complicated.

First of all the triggers must be manually added to the table. But when adding the triggers, it is not yet verified, if all columns defined in the trigger (inclusively the timeframe column) do really exist on the table. It is also not ensured, that when adding an OFK trigger to the table, the relation that is referenced in the trigger as the OPK relation, does really exist and has the necessary OPK triggers implemented. All this does only happen when the triggers are executed. Of course one can check what condition was not fulfilled, delete the relation and create a new one, fulfilling that condition. But it is still more convenient if the table is not created at all. Besides, the condition checking must be added to every constraint trigger and cannot simply be added at one central place like in the internal approach.

Moreover, it is not so simple to write dynamic triggers in plain SQL that can be reused for other relations. As I tried to write a reusable OPK trigger for the insert case that is independent of the relation name and that takes a list of column names as parameter to be used as key attributes, I was not able to finish the trigger. This means, the triggers currently written (they can be found in the appendix), would need to be adapted to every new relation and its key attributes.

As a result of the evaluation it can be said that the implementation of the constraints in the PostgreSQL kernel does not only have advantages in respect to the execution time, but also in respect to convenience and simplicity.

6 Conclusions and Future Work

The target of this thesis was to define the terms *ongoing primary key* and *ongoing foreign key* and to integrate the constraints into the PostgreSQL kernel. The performance of the implementation was also empirically evaluated. To this end, the implementation was compared with a solution that realizes the constraints using user defined triggers.

First, in section 3, it was defined that the *Time Domain* consists of fixed as well as ongoing timestamps. Also the terms *Ongoing Primary Key* and *Ongoing Foreign Key* were defined:

The *ongoing primary key* on a relation R ensures that there do not exist two tuples with the same set of key attributes A and whose timeframes do overlap.

The *ongoing foreign key* on a relation R ensures that for every tuple r in R a set of tuples in S exists, where for every tuple in the set, the ongoing primary key attributes are the same as the ongoing foreign key attributes in r and when subtracting all timeframes of the set from the timeframe form r the difference is empty.

I identified the cases in which OPK and OFK violations can occur. An OPK violation can occur when a tuple is inserted and updated and can never occur, when a tuple is deleted.

An OFK violation can occur, either when modifications on the referencing or the referenced relation are made. When modifications on the referencing relation are made, an OFK violation can occur when a tuple is inserted or updated. When modifications on the referenced relation are made, an OFK violation can occur when a tuple is updated or deleted.

These constraints were then integrated into the PostgreSQL kernel, so that the database system is able to provide native support for them. The thesis includes a description of the implementation approach in section 4.1. Overall it was very exciting but also difficult to familiarize with the PostgreSQL source code, as the system is enormous and written in C, a language that I did not use often before. But as I started to understand the implementation of the non-temporal foreign keys which do also define internal triggers, I was able to implement a native support for the ongoing integrity constraints.

The implementation supports the functionality of creating a new table with an OPK respectively OFK constraint. The user does also have the possibility to delete the constraints or relations when they are not used any more. Besides, the implementation includes an automatic checking of the ongoing integrity constraints, whenever a statement that can lead to a violation is executed.

To be able to create a new table with an OPK respectively OFK constraint, the parser of PostgreSQL had to be adapted. When a new relation with any ongoing integrity constraint should be created, some conditions have to be fulfilled.

It must be assured that all attributes that are named in the create statement exist in the newly created relation and that the relation contains an attribute called timeframe.

For a relation with an OFK constraint some additional checks must be made. First, the relation must reference some attributes in another relation, which has an OPK constraint. The

number of the OPK attributes and OFK attributes must be the same and those attributes must be of compatible types.

As this conditions are fulfilled, the relation is created and for every of the modifications that can lead to a violation a trigger is added to the relation. They check if the modification leads to a violation at any point in time and, if so, rejects the statement. All of them are constraint trigger that are fired after a statement is completed.

To evaluate the implementation, the internal triggers were additionally written as user defined ones. All information can be found in section 4.2.

In the evaluation, the running time for both kind of triggers were measured and compared. The evaluation led to partially surprisingly results. In most of the test cases, the internal approach was clearly faster. But the running times for the execution of the triggers written for the deletion of an element on the OPK table or the update of an element on the OFK table were irregular, what made them difficult to evaluate. Through the evaluation it became also evident, that in case of an update on the OPK relation, the internal triggers were not called in the same order as the user defined ones. To change the names of the triggers, so that in the internal implementation, the trigger that checks the OPK constraint is called first, could definitely be a correctional facility for the future. But in total, it can be said that in aspect of the running time, it is worth to use the internal implementation.

Benefits can not only be derived from the running time, but also from simplicity. When using the internal triggers, all the user has to do, is, to create a table with the desired constraints. This is much more convenient than to adapt all triggers to the own relation schemas and to add them manually.

Apart from the changing of the order of the update triggers on the OPK relation, there are some other issues that might be addressed in future work. First, with the current implementation it is only possible to initially add an ongoing integrity constraint to a table. A later adding with *alter table add constraint* is at the moment not possible but could be desirable. For instance, when a user has forgotten to add the constraint initially to the relation, but realizes at a later point in time that it would in fact have been necessary. Adding the constraint with an alter statement provides the simpler solution than creating a new table with the desired constraint and copying all tuples from the relation without the constraint. But these extension does not only raise question in terms of the implementation, but also theoretical ones. It is for instance not clear what should happen, when a relation does not have a column called "timeframe". The alter statement could in this case be rejected or the column could be manually added to the relation. This would lead to further questions, as for example, how the values in the timeframe columns should be initialized.

A second open issue is the supplying and handling of different cases for the OFK constraint. Non-temporal foreign key constraints provide the possibility, that the user can define what should happen, when a tuple in the referenced relation is tried to be deleted or updated and an OFK violation would occur [5]. At the moment, the statements will always be rejected. But it would be beneficial, to give the user the possibility to add different behaviour. A possible option would be *cascading delete*, that deletes all referencing tuples as well. But also this extension raises theoretical questions.

Imagine the relations *Product* and *Customer* from the application scenario are given, including the tuples shown in Table 6.1 and 6.2. The product coloured in blue should be deleted

id	name	premium	timeframe
300	Standard	4.5	$[2001-01-01, 2002-01-01)$
300	Standard	5	$[2002-01-01, NOW2002-01-01)$

Table 6.1: Products in referenced relation - the blue one should be deleted

from the system. When using *cascading deletion* all tuples in *Customer* that are referencing this tuple must also be deleted. But the product to be deleted is only valid for one year, whereas the customer tuples are valid for a longer time. When all tuples in customer that are referencing the blue tuple should be deleted, the customer relation would be empty afterwards and this although, for the main part of the valid time (that is from 2002 on), a tuple in the referenced relation would still exist.

Therefore an other possibility would be not to delete the tuple in the customer relation, but to adapt the tuples' timeframes. This means, for implementing *cascading delete* and also *cascading update* theoretical questions must be answered first. It has to be clarified, if in case of ongoing integrity constraints, the aspect of those cascading modifications does make sense at all and should be implemented.

customerId	productId	insuredSum	timeframe
C-767	300	1,000	$[2001-01-01, 2002-01-01$
C-769	300	2,000	$[2001-01-01, NOW2002-01-01)$
C-843	300	2,000	$[2001-01-01, NOW2002-01-01)$
C-900	300	4,000	$[2001-01-01, min2017-01-01NOW2002-01-01)$

Table 6.2: Customer referencing a product to be deleted

Bibliography

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

[2] K. Kulkarni and J.-E. Michels. Temporal Features in SQL:2011. *SIGMOD Record*, 41(3):34-43, 2012.

[3] C.S. Jensen and R.T. Snodgrass. Temporal Data Management. *IEEE Transactions on Knowledge and Data Engineering*

[4] Nicola, Matthias and Sommerlandt, Martin. Managing time in DB2 with temporal consistency - Enfore time-based referential integrity. *IBM Developers Works*, 2011

[5] PostgreSQL 9.4.7 Documentation. *The PostgreSQL Global Development Group*

[6] Wei Li and Snodgrass, R.T. and Shiyan Deng and Fattu, V.K. and Kasthurirangan, A. Efficient sequenced temporal integrity checking. In *Proceedings. 17th International Conference on Data Engineering.*, 2011

Appendix

Abbreviations

PK	Primary Key
FK	Foreign Key
OPK	Ongoing Primary Key
OFK	Ongoing Foreign Key
TPK	Temporal Primary Key
TFK	Temporal Foreign Key
DBMS	Database Management System

Content of the CD

- Zusfsg.txt Contains the German version of the abstract
- Abstract.txt Contains the English abstract
- Bachelorarbeit.odf Contains the complete thesis
- postgresql-9.4.0.zip Contains the complete source code. The guideline for the installation of PostgreSQL using the source code distribution is provided in the file named *INSTALL*.

Excerpts of most important changes

In this section, relevant code snippets of changed files can be found. Due to the big size of some files, only excerpts are provided here. Listing 6.1 shows the changes made to the parser.

Listing 6.1: gram.y

```
ConstraintElem:
    ...
    PRIMARY KEY ONGOING '(' columnList ')'
    ConstraintAttributeSpec {
        Constraint *n = makeNode(Constraint);
        n->contype = CONSTR_ONGOING_PRIMARY;
        n->location = @1;
```

```
            n->og_keys   = $5;
            n->options = NULL;
            n->indexname = NULL;
            n->indexspace = NULL;
            processCASbits($7, @7, "PRIMARY KEY ONGOING",
            &n->deferrable, &n->initdeferred,
            n->skip_validation,
            NULL, yyscanner);
            n->initially_valid = !n->skip_validation;
            $$ = (Node *)n;
        } |
        FOREIGN KEY ONGOING '(' columnList ')'
        REFERENCES qualified_name
        opt_column_list ConstraintAttributeSpec
            {
                Constraint *n    = makeNode(Constraint);
                n->contype       = CONSTR_ONGOING_FOREIGN;
                n->location      = @1;
                n->ogpktable     = $8;
                n->og_fk_attrs   = $5;
                n->og_pk_attrs   = $9;
                processCASbits($10, @10,
                "ONGOING FOREIGN KEY",
                &n->deferrable, &n->initdeferred,
                &n->skip_validation, NULL,
                yyscanner);
                n->initially_valid = !n->skip_validation;
                $$ = (Node *)n;
            }
        ...
```

Listing 6.2 and Listing 6.3 show how the constraints are added to a table, when creating a new table. For both constraints it is necessary to check, if the timeframe column exists. For the OFK constraint, the additional assertions are necessary, to check that the referenced relation is a table with an OPK constraint and that the key attributes of the two relations are compatible.

Listing 6.2: tablecmds.c - Add an OPK constraint to a table

```
/*
* Add an ongoing-primary-key constraint to a single table
*/
static void
ATAddOngoingPrimaryKeyConstraint(AlteredTableInfo *tab, Relation rel,
Constraint *ogpkconstraint, LOCKMODE lockmode) {
    // [...] declarations of variables

    /*
    * Look up the referencing attributes to make sure they exist,
    * and record their attnums and type OIDs.
    */
    MemSet(attnum, 0, sizeof(attnum));
    MemSet(typoid, 0, sizeof(typoid));
    MemSet(opclasses, 0, sizeof(opclasses));
```

```
    MemSet(pfeqoperators, 0, sizeof(pfeqoperators));
    MemSet(ppeqoperators, 0, sizeof(ppeqoperators));
    MemSet(ffeqoperators, 0, sizeof(ffeqoperators));

    numpks = transformColumnNameList(RelationGetRelid(rel),
    ogpkconstraint->og_keys,
    attnum, typoid);

    /*
    * Connect to the SPI interface
    * and make sure, timeframe column exists
    */
    if (SPI_connect() != SPI_OK_CONNECT)
    elog(ERROR, "SPI_connect failed");

    timeframecolnum = SPI_fnumber(rel->rd_att, "timeframe");

    if (timeframecolnum == SPI_ERROR_NOATTRIBUTE) {
    elog(ERROR, "Column timeframe does not exist on pk relation," +
        "but is required");
    }


    if (SPI_finish() != SPI_OK_FINISH)
    elog(ERROR, "SPI_finish failed");


    /*
    * Record the OG PK constraint in pg_constraint.
    */
    constrOid = CreateConstraintEntry(ogpkconstraint->conname,
        RelationGetNamespace(rel), CONSTRAINT_ONGOING_PRIMARY,
        ogpkconstraint->deferrable, ogpkconstraint->initdeferred,
        ogpkconstraint->initially_valid, RelationGetRelid(rel),
        attnum, numpks, InvalidOid, InvalidOid,
        InvalidOid, NULL, NULL, NULL, NULL, 0, ' ', ' ',
        ' ', NULL, NULL, NULL, NULL, true, 0, true, false);

    /*
    * Create the triggers that will enforce the constraint.
    */
    createOngoingPrimaryKeyTriggers(rel, ogpkconstraint, constrOid);
}

/*
* Create the triggers that implement an ongoing PK constraint. The
* implemention of the triggers themselves, are found in og_triggers.c
*/
static void
createOngoingPrimaryKeyTriggers(Relation rel,
    Constraint *ogpkconstraint, Oid constraintOid) {
    // [...] declarations of variables
```

```
myRelOid = RelationGetRelid(rel);

/* Make changes-so-far visible */
CommandCounterIncrement();

/*
* Build and execute a CREATE CONSTRAINT TRIGGER statement for
* the ON INSERT action
*/
ogpk_trigger_insert = makeNode(CreateTrigStmt);
ogpk_trigger_insert->trigname = "OG_PK_ConstraintTrigger_ins";
ogpk_trigger_insert->relation = NULL;
ogpk_trigger_insert->row = true;
ogpk_trigger_insert->timing = TRIGGER_TYPE_AFTER;
ogpk_trigger_insert->events = TRIGGER_TYPE_INSERT;
ogpk_trigger_insert->columns = NIL;
ogpk_trigger_insert->whenClause = NULL;
ogpk_trigger_insert->isconstraint = true;
ogpk_trigger_insert->constrrel = NULL;
ogpk_trigger_insert->deferrable = ogpkconstraint->deferrable;
ogpk_trigger_insert->initdeferred = ogpkconstraint->initdeferred;
ogpk_trigger_insert->funcname = SystemFuncName("OG_PK_insert");

ogpk_trigger_insert->args = NIL;

(void) CreateTrigger(ogpk_trigger_insert, NULL,
    myRelOid, InvalidOid, constraintOid, InvalidOid, true);

/* Make changes-so-far visible */
CommandCounterIncrement();

/*
* Build and execute a CREATE CONSTRAINT TRIGGER statement for
* the ON UPDATE action
*/
ogpk_trigger_update = makeNode(CreateTrigStmt);
ogpk_trigger_update->trigname = "OG_PK_ConstraintTrigger_upd";
ogpk_trigger_update->relation = NULL;
ogpk_trigger_update->row = true;
ogpk_trigger_update->timing = TRIGGER_TYPE_AFTER;
ogpk_trigger_update->events = TRIGGER_TYPE_UPDATE;
ogpk_trigger_update->columns = NIL;
ogpk_trigger_update->whenClause = NULL;
ogpk_trigger_update->isconstraint = true;
ogpk_trigger_update->constrrel = NULL;
ogpk_trigger_update->deferrable = ogpkconstraint->deferrable;
ogpk_trigger_update->initdeferred = ogpkconstraint->initdeferred;
ogpk_trigger_update->funcname = SystemFuncName("OG_PK_update");

ogpk_trigger_update->args = NIL;
```

```
    (void) CreateTrigger(ogpk_trigger_update, NULL, myRelOid,
        InvalidOid, constraintOid, InvalidOid, true);

    /* Make changes-so-far visible */
    CommandCounterIncrement();

}
```

Listing 6.3: tablecmds.c - Add an OFK constraint to a table

```
/*
* Add an ongoing foreign key constraint to a single table
*/
static void
ATAddOngoingForeignKeyConstraint(AlteredTableInfo *tab, Relation rel,
Constraint *ogfkconstraint, LOCKMODE lockmode) {
    Relation    pkrel;
    // [...] more declarations of variables

    /*
    * Grab an exclusive lock on the opk table, so that someone doesn't
    * delete rows out from under us.
    */
    if (OidIsValid(ogfkconstraint->old_pktable_oid)) {
        pkrel = heap_open(ogfkconstraint->old_pktable_oid,
            AccessExclusiveLock);
    } else {
        pkrel = heap_openrv(ogfkconstraint->ogpktable,
            AccessExclusiveLock);
    }

    /*
    * [...] Validity checks:
    * some checks to ensure, that the referenced relation is a table,
    * that the referenced relation is not a temp relation,
    * that a permanent relation does not reference one that is not,
    * that an unlogged relation does only reference permanent
    * or unlogged tables
    */

    ...

    /*
    * Look up the referencing attributes to make sure they exist
    * and record their attnums and type OIDs.
    */
    MemSet(pkattnum, 0, sizeof(pkattnum));
    MemSet(fkattnum, 0, sizeof(fkattnum));
    MemSet(pktypoid, 0, sizeof(pktypoid));
    MemSet(fktypoid, 0, sizeof(fktypoid));
    MemSet(opclasses, 0, sizeof(opclasses));
    MemSet(pfeqoperators, 0, sizeof(pfeqoperators));
    MemSet(ppeqoperators, 0, sizeof(ppeqoperators));
```

```
MemSet(ffeqoperators, 0, sizeof(ffeqoperators));

/*
* Connect to the SPI interface
*/
if (SPI_connect() != SPI_OK_CONNECT)
elog(ERROR, "SPI_connect failed");

/*
* Make sure, column timeframe exists
*/
timeframecolnum = SPI_fnumber(rel->rd_att, "timeframe");

if (timeframecolnum == SPI_ERROR_NOATTRIBUTE) {
elog(ERROR, "Column timeframe does not exist on fk relation," +
    " but is required");
}

/*
* Check if referenced relation has an OPK constraint
*/
initStringInfo(&constraintnamebuf);
appendStringInfo(&constraintnamebuf, "%s\\_\%\_og_pkey",
    RelationGetRelationName(pkrel));

initStringInfo(&querybuf);
appendStringInfo(&querybuf,
    "SELECT conname FROM pg_constraint WHERE conname LIKE '%s'",
    constraintnamebuf.data);

SPI_execute(querybuf.data, false, 0);

proc = SPI_processed;

if (proc == 0) {
elog(ERROR, "Table %s has no ongoing primary constraint",
    RelationGetRelationName(pkrel));
}

pkconname = SPI_getvalue(SPI_tuptable->vals[0],
    SPI_tuptable->tupdesc, 1);

/*
* Get constraint of refernced table
*/
pkConstrOid = get_relation_constraint_oid(RelationGetRelid(pkrel),
    pkconname, false);

tp = SearchSysCache1(CONSTROID, ObjectIdGetDatum(pkConstrOid));

adatum = SysCacheGetAttr(CONSTROID, tp,
Anum_pg_constraint_conkey, &adatumisnull);
```

```
if (adatumisnull)
    elog(ERROR, "null conkey for constraint %u", pkConstrOid);
arr = DatumGetArrayTypeP(adatum); /* ensure not toasted */
if (ARR_NDIM(arr) != 1 || ARR_HASNULL(arr) ||
ARR_ELEMTYPE(arr) != INT2OID)
    elog(ERROR, "conkey is not a 1-D smallint array");
pknumkeys = ARR_DIMS(arr)[0];

memcpy(pktableattnum, ARR_DATA_PTR(arr),
    pknumkeys * sizeof(int16));
if ((Pointer) arr != DatumGetPointer(adatum))
    pfree(arr);

ReleaseSysCache(tp);

numfks = transformColumnNameList(RelationGetRelid(rel),
    ogfkconstraint->og_fk_attrs, fkattnum, fktypoid);

numpks = transformColumnNameList(RelationGetRelid(pkrel),
    ogfkconstraint->og_pk_attrs, pkattnum, pktypoid);

for (i = 0; i < pknumkeys; i++) {
    iscontained = false;
    for (j = 0; j < numpks; j++) {
        if (pktableattnum[i] == pkattnum[j]) {
            iscontained = true;
        }
    }

    if (!iscontained) {
        elog(ERROR, "there is no ongoing primary key matching " +
        "given keys for referenced table");
    }
}


if (SPI_finish() != SPI_OK_FINISH)
elog(ERROR, "SPI_finish failed");

/*
* Check if number of key attributes match
*/
if (numfks != numpks)
ereport(ERROR,
(errcode(ERRCODE_INVALID_FOREIGN_KEY),
errmsg("number of referencing and referenced columns " +
    "for ongoing foreign key disagree")));

/*
* Check if OPK and OFK attributes are of the same
* or of compatibel types
*/
```

```
for (i = 0; i < numpks; i++) {
// [...] declarations of variables

    if (fktype != pktype) {
        fktyped = getBaseType(fktype);

        // [...] declarations of variables

        input_typeids[0] = pktype;
        input_typeids[1] = fktype;
        target_typeids[0] = opcintype;
        target_typeids[1] = opcintype;
        if (!can_coerce_type(2, input_typeids, target_typeids,
            COERCION_IMPLICIT)) {
            ereport(ERROR,
                (errcode(ERRCODE_DATATYPE_MISMATCH),
                errmsg("ongoing foreign key constraint \"%s\" "
                "cannot be implemented",
                ogfkconstraint->conname),
                errdetail("Key columns \"%s\" and \"%s\" "
                "are of incompatible types: %s and %s.",
                strVal(list_nth(ogfkconstraint->og_fk_attrs, i)),
                strVal(list_nth(ogfkconstraint->og_pk_attrs, i)),
                format_type_be(fktype),
                format_type_be(pktype))));
        }
    }
}

/*
* Record the OG PK constraint in pg_constraint.
*/

constrOid = CreateConstraintEntry(ogfkconstraint->conname,
RelationGetNamespace(rel), CONSTRAINT_ONGOING_FOREIGN,
ogfkconstraint->deferrable, ogfkconstraint->initdeferred,
ogfkconstraint->initially_valid, RelationGetRelid(rel),
fkattnum, numfks, InvalidOid, InvalidOid,
RelationGetRelid(pkrel), pkattnum, NULL, NULL,
NULL, numpks, ' ', ' ', ' ', NULL, NULL,
NULL, NULL, true, 0, true, false);

/*
* Create the triggers that will enforce the constraint.
*/
createOngoingForeignKeyTriggers(rel, ogfkconstraint,
    constrOid, RelationGetRelid(pkrel));

heap_close(pkrel, NoLock);

/*
* Create the triggers that implement an ongoing FK constraint.
```

```
*/
static void
createOngoingForeignKeyTriggers(Relation rel,
Constraint *ogfkconstraint, Oid constraintOid, Oid refRelOid) {
    // [...] declarations of variables

    myRelOid = RelationGetRelid(rel);

    /* Make changes-so-far visible */
    CommandCounterIncrement();

    /*
    * Build and execute a CREATE CONSTRAINT TRIGGER statement for
    * the ON INSERT action
    */
    ogfk_trigger_insert = makeNode(CreateTrigStmt);
    ogfk_trigger_insert->trigname = "OG_FK_ConstraintTrigger_ins";
    ogfk_trigger_insert->relation = NULL;
    ogfk_trigger_insert->row = true;
    ogfk_trigger_insert->timing = TRIGGER_TYPE_AFTER;
    ogfk_trigger_insert->events = TRIGGER_TYPE_INSERT;
    ogfk_trigger_insert->columns = NIL;
    ogfk_trigger_insert->whenClause = NULL;
    ogfk_trigger_insert->isconstraint = true;
    ogfk_trigger_insert->constrrel = NULL;
    ogfk_trigger_insert->deferrable = ogfkconstraint->deferrable;
    ogfk_trigger_insert->initdeferred =
        ogfkconstraint->initdeferred;
    ogfk_trigger_insert->funcname = SystemFuncName("OG_FK_insert");

    ogfk_trigger_insert->args = NIL;

    (void) CreateTrigger(ogfk_trigger_insert, NULL, myRelOid,
    InvalidOid, constraintOid, InvalidOid, true);

    /* Make changes-so-far visible */
    CommandCounterIncrement();

    /*
    * Build and execute a CREATE CONSTRAINT TRIGGER statement for
    * the ON INSERT action
    */
    ogfk_trigger_update = makeNode(CreateTrigStmt);
    ogfk_trigger_update->trigname = "OG_FK_ConstraintTrigger_upd";
    ogfk_trigger_update->relation = NULL;
    ogfk_trigger_update->row = true;
    ogfk_trigger_update->timing = TRIGGER_TYPE_AFTER;
    ogfk_trigger_update->events = TRIGGER_TYPE_UPDATE;
    ogfk_trigger_update->columns = NIL;
    ogfk_trigger_update->whenClause = NULL;
    ogfk_trigger_update->isconstraint = true;
    ogfk_trigger_update->constrrel = NULL;
```

```
ogfk_trigger_update->deferrable = ogfkconstraint->deferrable;
ogfk_trigger_update->initdeferred =
    ogfkconstraint->initdeferred;
ogfk_trigger_update->funcname = SystemFuncName("OG_FK_update");

ogfk_trigger_update->args = NIL;

(void) CreateTrigger(ogfk_trigger_update, NULL, myRelOid,
InvalidOid, constraintOid, InvalidOid, true);

/* Make changes-so-far visible */
CommandCounterIncrement();

/*
* Build and execute a CREATE CONSTRAINT TRIGGER statement for
* the ON DELETE action on the pk relation
*/
ogfk_trigger_delete_pk = makeNode(CreateTrigStmt);
ogfk_trigger_delete_pk->trigname
    = "OG_FK_ConstraintTrigger_del_pk";
ogfk_trigger_delete_pk->relation = NULL;
ogfk_trigger_delete_pk->row = true;
ogfk_trigger_delete_pk->timing = TRIGGER_TYPE_AFTER;
ogfk_trigger_delete_pk->events = TRIGGER_TYPE_DELETE;
ogfk_trigger_delete_pk->columns = NIL;
ogfk_trigger_delete_pk->whenClause = NULL;
ogfk_trigger_delete_pk->isconstraint = true;
ogfk_trigger_delete_pk->constrrel = NULL;
ogfk_trigger_delete_pk->deferrable =
    ogfkconstraint->deferrable;
ogfk_trigger_delete_pk->initdeferred =
    ogfkconstraint->initdeferred;
ogfk_trigger_delete_pk->funcname =
    SystemFuncName("OG_FK_delete_in_pk");

ogfk_trigger_delete_pk->args = NIL;

(void) CreateTrigger(ogfk_trigger_delete_pk, NULL, refRelOid,
    myRelOid, constraintOid, InvalidOid, true);

/* Make changes-so-far visible */
CommandCounterIncrement();

/*
* Build and execute a CREATE CONSTRAINT TRIGGER statement for
* the ON UPDATE action on the pk relation
*/
ogfk_trigger_update_pk = makeNode(CreateTrigStmt);
ogfk_trigger_update_pk->trigname =
    "OG_FK_ConstraintTrigger_upd_pk";
ogfk_trigger_update_pk->relation = NULL;
ogfk_trigger_update_pk->row = true;
```

```
        ogfk_trigger_update_pk->timing = TRIGGER_TYPE_AFTER;
        ogfk_trigger_update_pk->events = TRIGGER_TYPE_UPDATE;
        ogfk_trigger_update_pk->columns = NIL;
        ogfk_trigger_update_pk->whenClause = NULL;
        ogfk_trigger_update_pk->isconstraint = true;
        ogfk_trigger_update_pk->constrrel = NULL;
        ogfk_trigger_update_pk->deferrable =
            ogfkconstraint->deferrable;
        ogfk_trigger_update_pk->initdeferred =
            ogfkconstraint->initdeferred;
        ogfk_trigger_update_pk->funcname =
            SystemFuncName("OG_FK_update_in_pk");

        ogfk_trigger_update_pk->args = NIL;

        (void) CreateTrigger(ogfk_trigger_update_pk, NULL, refRelOid,
             myRelOid, constraintOid, InvalidOid, true);

        /* Make changes-so-far visible */
        CommandCounterIncrement();

    }
}
```

The following listings show the implementation of all constraint triggers.

Listing 6.4: og_triggers.c - OPK trigger on insert and on update

```
/*
* checks OPK violation on update and on insert
*/
static Datum check_og_pk_constraint(TriggerData *trigdata) {
    // [...] declaration of variables

    // ensure that trigger was called on insert or on update
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        new_row = trigdata->tg_newtuple;
    else if (TRIGGER_FIRED_BY_INSERT(trigdata->tg_event))
        new_row = trigdata->tg_trigtuple;
    else
        elog(ERROR, "OG PK Trigger can only be fired" +
        " on update or on insert");

    rel = trigdata->tg_relation;

    constraintOid = trigdata->tg_trigger->tgconstraint;

    tp = SearchSysCache1(CONSTROID, ObjectIdGetDatum(constraintOid));

    /*
    * Get information from OPK table, based on the constraint oid
    */
    adatum = SysCacheGetAttr(CONSTROID, tp,
    Anum_pg_constraint_conkey, &adatumisnull);
```

```
if (adatumisnull)
    elog(ERROR, "null conkey for constraint %u", constraintOid);
arr = DatumGetArrayTypeP(adatum); /* ensure not toasted */
if (ARR_NDIM(arr) != 1 || ARR_HASNULL(arr) ||
ARR_ELEMTYPE(arr) != INT2OID)
    elog(ERROR, "conkey is not a 1-D smallint array");
numkeys = ARR_DIMS(arr)[0];
if (numkeys <= 0 || numkeys > OG_MAX_NUMKEYS)
    elog(ERROR, "OPK constraint cannot have %d columns",
    numkeys);
memcpy(attnums, ARR_DATA_PTR(arr), numkeys * sizeof(int16));
if ((Pointer) arr != DatumGetPointer(adatum))
    pfree(arr);

ReleaseSysCache(tp);

if (SPI_connect() != SPI_OK_CONNECT)
    elog(ERROR, "SPI_connect failed");

ogQuoteRelationName(relname, rel);

// get number and value of timeframe column
timeframecolnum = SPI_fnumber(rel->rd_att, "timeframe");

if (timeframecolnum == SPI_ERROR_NOATTRIBUTE) {
    elog(ERROR, "Column timeframe does not exist," +
    " but is required");
}
timeframe = SPI_getvalue(new_row, rel->rd_att, timeframecolnum);

/*
* Create query:
* SELECT * FROM table_name
* WHERE keyAttributesMatch and timeframe && timeframe_value
*/
initStringInfo(&querybuf);
appendStringInfo(&querybuf, "SELECT * FROM %s WHERE ", relname);
for (i = 0; i < numkeys; i++) {
    appendStringInfo(&querybuf, "%s = '%s' AND ",
        SPI_fname(rel->rd_att, attnums[i]),
        SPI_getvalue(new_row, rel->rd_att, attnums[i]));
}

appendStringInfo(&querybuf, "timeframe && '%s'", timeframe);
SPI_execute(querybuf.data, false, 2);

proc = SPI_processed;

// raise error, if more than one tuples was found
if (proc != 1) {
    elog(ERROR, "Ongoing primary key constraint violated,
    proc: %i", proc);
```

```
    }

    if (SPI_finish() != SPI_OK_FINISH)
        elog(ERROR, "SPI_finish failed");

    return PointerGetDatum(NULL);
}
```

Listing 6.5: og_triggers.c - OFK trigger on insert and on update

```
/*
* checks OFK violation on update and on insert
*/
static Datum check_og_fk_constraint(TriggerData *trigdata) {
    // [...] declarations of variables

    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        new_row = trigdata->tg_newtuple;
    else if (TRIGGER_FIRED_BY_INSERT(trigdata->tg_event))
        new_row = trigdata->tg_trigtuple;
    else
        elog(ERROR, "OG FK Triggers can only be fired" +
        " on update or on insert");

    // Get information of both tables
    fkrel = trigdata->tg_relation;
    constraintOid = trigdata->tg_trigger->tgconstraint;
    oginfo =
        (OG_ConstraintInfo *) og_LoadConstraintInfo(constraintOid);
    pkrel = heap_open(oginfo->pk_relid, RowShareLock);

    if (SPI_connect() != SPI_OK_CONNECT)
        elog(ERROR, "SPI_connect failed");

    ogQuoteRelationName(pkrelname, pkrel);
    ogQuoteRelationName(fkrelname, fkrel);

    // get timeframe columns and values
    pktimeframecolnum = SPI_fnumber(pkrel->rd_att, "timeframe");
    fktimeframecolnum = SPI_fnumber(fkrel->rd_att, "timeframe");
    if (pktimeframecolnum == SPI_ERROR_NOATTRIBUTE)
        elog(ERROR, "Column timeframe does not exist on opk" +
        " relation, but is required");
    if (fktimeframecolnum == SPI_ERROR_NOATTRIBUTE)
        elog(ERROR, "Column timeframe does not exist on ofk," +
        " relation, but is required");

    fktimeframeinternal = SPI_getbinval(new_row, fkrel->rd_att,
        fktimeframecolnum, &isnull);
    rangetypeid = SPI_gettypeid(fkrel->rd_att, fktimeframecolnum);

    fktimeframevalue = SPI_getvalue(new_row, fkrel->rd_att,
        fktimeframecolnum);
```

```c
typcache = lookup_type_cache(rangetypeid, TYPECACHE_RANGE_INFO);

/*
* Create query:
* SELECT * FROM table_name WHERE keyAttributesMatch
*/
initStringInfo(&querybuf);
appendStringInfo(&querybuf, "SELECT * FROM %s WHERE ", pkrelname);
for (i = 0; i < oginfo->numkeys; i++) {
    appendStringInfo(&querybuf, "%s = '%s' and ",
    SPI_fname(pkrel->rd_att, oginfo->pk_attnums[i]),
    SPI_getvalue(new_row, fkrel->rd_att, oginfo->fk_attnums[i]));
}

appendStringInfo(&querybuf, "timeframe && '%s'", fktimeframevalue);

SPI_execute(querybuf.data, false, 0);

proc = SPI_processed;

if (proc == 0) {
    elog(ERROR, "There are no matching tuples in the pk relation");
}

/*
* Initialize RangeTypeList with value of OFK timeframe
* Calculate difference from OFK timeframes and
* OPK timeframes.
* Store remaining timeframe parts in list.
* List should be empty at the end, otherwise raise error
*/
first = palloc(sizeof(RangeTypeList));
last = palloc(sizeof(RangeTypeList));
temp = palloc(sizeof(RangeTypeList));
first->range = DatumGetRangeType(fktimeframeinternal);
first->rangeEmpty = false;
first->next = (RangeTypeList*) 0;

for (i = 0; i < proc; i++) {
    pktimeframeinternal = SPI_getbinval(SPI_tuptable->vals[i],
        SPI_tuptable->tupdesc, pktimeframecolnum, &isnull);
    if (first == NULL) {
        if (SPI_finish() != SPI_OK_FINISH)
            elog(ERROR, "SPI_finish failed");
        heap_close(pkrel, RowShareLock);
        return PointerGetDatum(NULL);
    }
    last = (RangeTypeList*)0;
    current = first;

    while (current != NULL) {
```

```
        range_deserialize(typcache, current->range, &lower1,
            &upper1, &empty);

        range_deserialize(typcache,
            DatumGetRangeType(pktimeframeinternal),
            &lower2, &upper2, &empty);

        newList = range_difference_now(typcache, &lower1,
            &upper1, &lower2, &upper2);

        temp = current;
        if (!newList->rangeEmpty) {
            if(newList->next == NULL) {
                newList->next = current->next;
            } else {
                newList->next->next = current->next;
            }
            current = current->next;
            if (last != NULL) {
                last->next = newList;
            } else {
                first = newList;
            }
        } else {
            if (last != NULL) {
                current = current->next;
                last->next = current;
            } else {
                current = current->next;
                first = current;
            }
        }
        last = temp;
        }
    }

    if (SPI_finish() != SPI_OK_FINISH)
        elog(ERROR, "SPI_finish failed");

    if (first != 0) {
        elog(ERROR, "Foreign key can not be inserted." +
        " FK timeframe is not fully covered by pk timeframe");
    }

    heap_close(pkrel, RowShareLock);

    return PointerGetDatum(NULL);

}
```

Listing 6.6: og_triggers.c - OFK trigger on udpate on referenced table

```
/*
```

```
* Trigger that checks ongoing foreign key constraint
* on update on the pk relation
*/
Datum OG_FK_update_in_pk(PG_FUNCTION_ARGS) {
// [...] declarations of variables

new_row = trigdata->tg_newtuple;
old_row = trigdata->tg_trigtuple;

// get information of both tables
pkrel = trigdata->tg_relation;
constraintOid = trigdata->tg_trigger->tgconstraint;
oginfo = (OG_ConstraintInfo *) og_LoadConstraintInfo(constraintOid);
fkrel = heap_open(oginfo->fk_relid, RowShareLock);

if (SPI_connect() != SPI_OK_CONNECT)
    elog(ERROR, "SPI_connect failed");

ogQuoteRelationName(pkrelname, pkrel);
ogQuoteRelationName(fkrelname, fkrel);
// get timeframe columns and values
pktimeframecolnum = SPI_fnumber(pkrel->rd_att, "timeframe");
fktimeframecolnum = SPI_fnumber(fkrel->rd_att, "timeframe");

if (pktimeframecolnum == SPI_ERROR_NOATTRIBUTE)
    elog(ERROR, "Column timeframe does not exist on opk " +
    " relation, but is required");
if (fktimeframecolnum == SPI_ERROR_NOATTRIBUTE)
    elog(ERROR, "Column timeframe does not exist on fk " +
    "relation, but is required");

//Check if key columns or timeframe changed
onlyNonRelevantColumnsChanged = true;
keyChanged = false;

for (i = 0; i < oginfo->numkeys; i++) {
    old = SPI_getvalue(old_row, pkrel->rd_att, oginfo->pk_attnums[i]);
    new = SPI_getvalue(new_row, pkrel->rd_att, oginfo->pk_attnums[i]);

    if (strcmp(old, new) != 0) {
        onlyNonRelevantColumnsChanged = false;
        keyChanged = true;
    }
}

if (strcmp(SPI_getvalue(old_row, pkrel->rd_att, pktimeframecolnum),
    SPI_getvalue(new_row, pkrel->rd_att, pktimeframecolnum)) != 0) {
    onlyNonRelevantColumnsChanged = false;
}

// Accept update if neither timeframe nor the key attributes changed
if (onlyNonRelevantColumnsChanged) {
```

```
        if (SPI_finish() != SPI_OK_FINISH)
            elog(ERROR, "SPI_finish failed");
        heap_close(fkrel, RowShareLock);
        return PointerGetDatum(NULL);
    }

    oldpktimeframe = SPI_getvalue(old_row, pkrel->rd_att,
        pktimeframecolnum);
    newpktimeframe = SPI_getvalue(new_row, pkrel->rd_att,
        pktimeframecolnum);

    // Case key changed. Check if old column was referenced
    if (keyChanged) {
        initStringInfo(&querybuf);
        appendStringInfo(&querybuf, "SELECT * FROM %s WHERE ", fkrelname);
        for (i = 0; i < oginfo->numkeys; i++) {
            appendStringInfo(&querybuf, "%s = '%s' and ",
                SPI_fname(fkrel->rd_att, oginfo->fk_attnums[i]),
                SPI_getvalue(old_row, pkrel->rd_att,
                oginfo->pk_attnums[i]));
        }

        appendStringInfo(&querybuf, "timeframe && '%s'", oldpktimeframe);

        SPI_execute(querybuf.data, false, 1);

        proc = SPI_processed;

        if (proc > 0) {
            elog(ERROR, "Update not possible, the old row was" +
            " referenced by the foreign key table");
        }
    // Case timeframe changed, check if intersection stays the same
    } else {
        initStringInfo(&querybuf);
        appendStringInfo(&querybuf, "SELECT '%s' * timeframe, " +
            "'%s' * timeframe FROM %s WHERE ",
            oldpktimeframe, newpktimeframe, fkrelname);
        for (i = 0; i < oginfo->numkeys; i++) {
            appendStringInfo(&querybuf, "%s = '%s' and ",
            SPI_fname(fkrel->rd_att, oginfo->fk_attnums[i]),
            SPI_getvalue(old_row, pkrel->rd_att, oginfo->pk_attnums[i]));
        }

        appendStringInfo(&querybuf, "timeframe && '%s'", oldpktimeframe);
        SPI_execute(querybuf.data, false, 0);
        proc = SPI_processed;

        if (proc == 0) {
            if (SPI_finish() != SPI_OK_FINISH)
            elog(ERROR, "SPI_finish failed");
            heap_close(fkrel, RowShareLock);
```

```
            return PointerGetDatum(NULL);
        } else {
            for (i = 0; i < proc; i++) {
                oldintersection = SPI_getvalue(SPI_tuptable->vals[i],
                    SPI_tuptable->tupdesc, 1);
                newintersection = SPI_getvalue(SPI_tuptable->vals[i],
                    SPI_tuptable->tupdesc, 2);
                if (strcmp(oldintersection, newintersection) != 0)
                    elog(ERROR, "Update rejected.");
            }
        }

    }

    if (SPI_finish() != SPI_OK_FINISH)
        elog(ERROR, "SPI_finish failed");

    heap_close(fkrel, RowShareLock);
    return PointerGetDatum(NULL);
    }
```

Listing 6.7: og_triggers.c - OFK trigger on delete on referenced table

```
/*
* Trigger that checks ongoing foreign key constraint
* on delete on the pk relation
*/
Datum OG_FK_delete_in_pk(PG_FUNCTION_ARGS) {
    // [...] declarations of variables;
    TriggerData *trigdata = (TriggerData *) fcinfo->context;

    // get information of both tables
    old_row = trigdata->tg_trigtuple;
    pkrel = trigdata->tg_relation;
    constraintOid = trigdata->tg_trigger->tgconstraint;
    oginfo =
        (OG_ConstraintInfo *) og_LoadConstraintInfo(constraintOid);

    fkrel = heap_open(oginfo->fk_relid, RowShareLock);
    if (SPI_connect() != SPI_OK_CONNECT)
        elog(ERROR, "SPI_connect failed");

    ogQuoteRelationName(pkrelname, pkrel);
    ogQuoteRelationName(fkrelname, fkrel);

    // get timeframe column and values
    pktimeframecolnum = SPI_fnumber(pkrel->rd_att, "timeframe");
    fktimeframecolnum = SPI_fnumber(fkrel->rd_att, "timeframe");

    if (pktimeframecolnum == SPI_ERROR_NOATTRIBUTE) {
        elog(ERROR, "Column timeframe does not exist on " +
        " opk relation, but is required");
    }
```

```
    if (fktimeframecolnum == SPI_ERROR_NOATTRIBUTE) {
        elog(ERROR, "Column timeframe does not exist on " +
        "ofk relation, but is required");
    }

    pktimeframe = SPI_getvalue(old_row,
        pkrel->rd_att, pktimeframecolnum);
    /*
    * Call query:
    * SELECT * FROM ofk_table WHERE keyAttributesMatch
    * and timeframe && opk_timeframe_value
    */
    initStringInfo(&querybuf);
    appendStringInfo(&querybuf, "SELECT * FROM %s WHERE ", fkrelname);
    for (i = 0; i < oginfo->numkeys; i++) {
        appendStringInfo(&querybuf, "%s = '%s' and ",
        SPI_fname(fkrel->rd_att, oginfo->fk_attnums[i]),
        SPI_getvalue(old_row, pkrel->rd_att, oginfo->pk_attnums[i]));
    }

    appendStringInfo(&querybuf, "timeframe && '%s'", pktimeframe);

    SPI_execute(querybuf.data, false, 1);
    proc = SPI_processed;

    // reject, if column was referenced
    if (proc > 0) {
        elog(ERROR, "Deletion not possible, " +
        "because row is referenced in foreign key table");
    }

    if (SPI_finish() != SPI_OK_FINISH)
    elog(ERROR, "SPI_finish failed");

    heap_close(fkrel, RowShareLock);

    return PointerGetDatum(NULL);
}
```

User Defined Triggers

Listing 6.8: Insert / Update trigger on OPK relation that checks OPK constraint

```
CREATE OR REPLACE FUNCTION check_pk_insert_update()
RETURNS trigger AS $check_pk_insupd$
DECLARE
r record;
cnt integer;
BEGIN
```

```
IF NOT (SELECT EXISTS (SELECT 1
FROM information_schema.columns WHERE
table_name=TG_TABLE_NAME AND column_name='timeframe'))
THEN
RAISE EXCEPTION 'timeframe required';
END IF;

cnt = 0;
FOR r in SELECT id, timeframe FROM pknow
WHERE id = NEW.id LOOP
IF r.timeframe != NEW.timeframe THEN
IF r.timeframe && NEW.timeframe THEN
RAISE EXCEPTION 'OPK violation';
END IF;
END IF;

IF r.timeframe = NEW.timeframe THEN
cnt = cnt + 1;
IF cnt = 2 THEN
RAISE EXCEPTION 'OPK violation';
END IF;
END IF;
END LOOP;
RETURN NULL;
END;
$check_pk_insupd$ LANGUAGE plpgsql;
```

Listing 6.9: Insert / Update trigger on OFK relation that checks OFK constraint

```
CREATE OR REPLACE FUNCTION check_fk_insert_update()
RETURNS trigger AS $check_fk_insupd$
DECLARE
r record;
cnt integer;

notcovered daterange[];
tempdiff daterange[];
t daterange;
isToRemove boolean;
BEGIN
IF NOT (SELECT EXISTS (SELECT 1
FROM information_schema.columns
WHERE table_name=TG_TABLE_NAME
AND column_name='timeframe'))
```

```
THEN
RAISE EXCEPTION 'timeframe required';
END IF;

IF NOT (SELECT EXISTS (SELECT 1
FROM information_schema.columns WHERE
table_name='fknow' AND column_name='timeframe')) THEN
RAISE EXCEPTION 'timeframe required';
END IF;

cnt := 0;
notcovered := array_append(notcovered, NEW.timeframe);
FOR r IN SELECT id, timeframe FROM pknow
WHERE id = NEW.pkid AND timeframe && NEW.timeframe
LOOP
cnt = cnt + 1;
isToRemove = false;
FOREACH t IN ARRAY notcovered
LOOP
IF r.timeframe && t THEN
isToRemove := true;
END IF;
IF NOT isempty(range_minus_first(t, r.timeframe)) THEN
tempdiff := array_append(tempdiff,
range_minus_first(t, r.timeframe));
END IF;
IF NOT isempty(range_minus_second(t, r.timeframe))
THEN
tempdiff := array_append(tempdiff,
range_minus_second(t, r.timeframe));
END IF;
IF isToRemove = true THEN
notcovered := array_remove(notcovered, t);
END IF;
END LOOP;
IF array_length(tempdiff, 1) != 0 THEN
notcovered := array_cat(notcovered, tempdiff);
tempdiff := '{}';
END IF;
END LOOP;

IF cnt = 0 THEN
RAISE EXCEPTION 'OFK violation';
END IF;
```

```
IF array_length(notcovered, 1) != 0 THEN
RAISE EXCEPTION 'OFK violation';
END IF;
RETURN NULL;
END;
$check_fk_insupd$ LANGUAGE plpgsql;
```

Listing 6.10: Update trigger on OPK relation that checks OFK constraint

```
CREATE OR REPLACE FUNCTION check_pkfk_update()
RETURNS trigger AS $check_pkfk_upd$
DECLARE
r record;
cnt integer;
BEGIN
IF NOT (SELECT EXISTS (SELECT 1
FROM information_schema.columns WHERE
table_name=TG_TABLE_NAME AND column_name='timeframe'))
THEN
RAISE EXCEPTION 'timeframe required';
END IF;

IF NOT (SELECT EXISTS (SELECT 1
FROM information_schema.columns
WHERE table_name='fknow' AND column_name='timeframe'))
THEN
RAISE EXCEPTION 'timeframe required';
END IF;

IF OLD.id = NEW.id AND OLD.timeframe = NEW.timeframe THEN
RETURN NULL;
END IF;

IF OLD.id != NEW.id THEN
SELECT COUNT(*) FROM fknow
WHERE pkid = OLD.id AND timeframe && OLD.timeframe
INTO cnt;
If cnt > 0 THEN
RAISE EXCEPTION 'OFK violation';
END IF;
RETURN NULL;
END IF;
```

```
FOR r IN
SELECT OLD.timeframe * timeframe as oldtimeframe,
NEW.timeframe * timeframe as newtimeframe FROM fknow
WHERE pkid = OLD.id AND timeframe && OLD.timeframe
LOOP
IF r.oldtimeframe != r.newtimeframe THEN
RAISE EXCEPTION 'timeframe required';
END IF;
END LOOP;
RETURN NULL;
END;
$check_pkfk_upd$ LANGUAGE plpgsql;
```

Listing 6.11: Delete trigger on OPk relation that checks OFK constraint

```
CREATE OR REPLACE FUNCTION check_pkfk_deletion()
RETURNS trigger AS $check_pkfk_del$
DECLARE
r pknow%rowtype;
cnt integer;
BEGIN
IF NOT (SELECT EXISTS (SELECT 1
FROM information_schema.columns WHERE
table_name=TG_TABLE_NAME AND column_name='timeframe'))
THEN
RAISE EXCEPTION 'timeframe required';
END IF;

IF NOT (SELECT EXISTS (SELECT 1
FROM information_schema.columns
WHERE table_name='fknow' AND column_name='timeframe'))
THEN
RAISE EXCEPTION 'timeframe required';
END IF;

SELECT COUNT(*) FROM fknow
WHERE pkid = OLD.id AND timeframe && OLD.timeframe
INTO cnt;
If cnt > 0 THEN
RAISE EXCEPTION 'OFK violation';
END IF;
RETURN NULL;
END
$check_pkfk_del$ LANGUAGE plpgsql;
```