Jasmin Ebner, 13-747-159 03. Dezember 2015

Integrity Constraints in Temporal Ground and Now-Relative Databases

Introduction

In contrast to non-temporal databases, temporal databases have the advantage that they can record time-varying data. Here one has to distinguish between valid time and transaction time. Valid time describes the period in which a tuple is valid in the real world, whereas transaction time refers to the time when a tuple is recorded in the database. But often one does not and cannot know how long in the future a fact will hold, so it's desired to be able to simply say that a fact is valid till now. In this case we talk about now-relative databases.

As in normal databases the two most important integrity constraints in temporal databases are primary keys and foreign keys. A primary key (PK) is a constraint that uniquely identifies each record in a database table and a foreign key (FK) points to such a primary key. These principles can be adapted to temporal databases, here the constraints must hold at every point in time.

In this report I will have a closer look at how these integrity constraints can be violated in temporal ground databases and how this principles can be adapted to now-relative databases. When I speak about *now*, this *now* does always refer to the current time. Furthermore, all time intervals that are used in this report are assumed to be left-closed and right-open. Besides, only the cases where *now* is used as upper bound for an interval are considered.

Integrity constraint violation in ground temporal databases

Primary Keys with valid time

First we'll start having a look at the primary keys. As mentioned above, a primary key must be unique at every point in time. This in general means that for all the tuples having the same PK, their valid time must not overlap.

So let's assume we have a relation with different tuples with a primary key that in the current situation do not violate the primary key constraint. Here it is possible to violate the PK-constraint by either inserting a new tuple or updating an already inserted tuple. It is not possible to violate the constraint by deleting an existing tuple. This is because by deleting tuples it can only happen that there exist new gaps between the valid times of the tuples, but no intersection can occur.

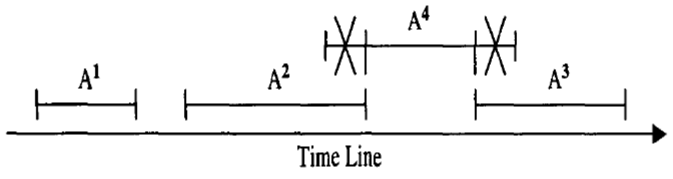

Figure 1 - Violation of primary key constraint

Inserting

Assume Attribute A is the primary key, in Figure 1 the tuple 4 cannot be inserted in the table because it intersects with tuple 2 and tuple 4. Figure 2 shows the four different cases in which the primary key constraint can be violated by inserting a new tuple. So before I can insert a new tuple, it must be checked that none of these cases occurs, otherwise the insertion must be rejected.

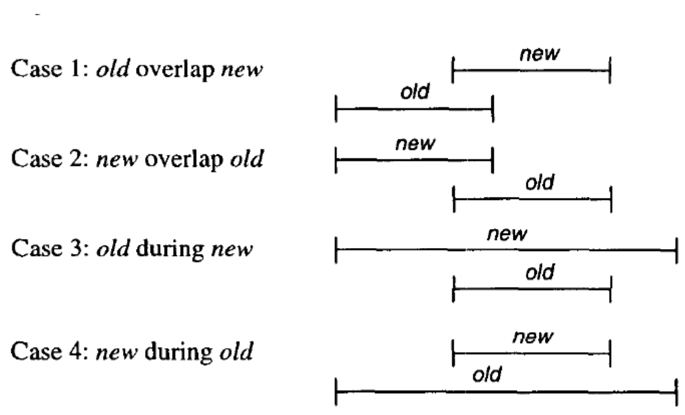
Case 1: *old* overlap *new*
new
old
Case 2: *new* overlap *old*
new
old
Case 3: *old* during *new*
new
old
Case 4: *new* during *old*
new
old

Figure 2 - Different cases of primary key violation

Updating

As mentioned above, deleting tuples will not violate the primary key constraint. So updating an existing tuple is actually the same as deleting the old tuple, checking if the new tuple does not violate any primary key constraints and then, if it does not violate any constraint, inserting it. But if the new tuple does violate a constraint, the insertion must be rejected and the old tuple that we have deleted must be reinserted.

Foreign Keys with valid time

The more complicated of the two integrity constraints is the referential constraint. If you have a foreign key, it must be ensured that at every point in time of the validity period of a tuple in a referencing relation, there is a single corresponding tuple in the referenced relation with the required foreign key value. But in the overall valid time of the foreign key it can reference to multiple tuples.

There are two ways in which violation can occur, either by changing the referencing or the referenced relation.

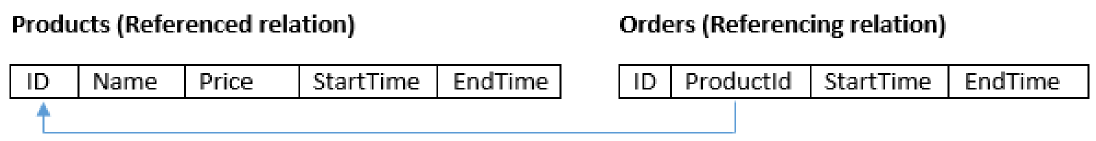

Figure 3 - Foreign Key example

It is assumed, that at the moment we have the two relations that are shown in Figure 3 that do not violate any constraint. That is, we have a relation *Orders* with a foreign key *ProductId* that is referencing the primary key *ID* in the *Products* relation. In the *Products* relation, the primary key constraint is not violated. More, for the whole valid time period of the tuples in *Orders*, there exist one or more tuples in *Products* with the right primary key. In the following section I will discuss the cases in which referential integrity constraints can be violated based on the example of Figure 3.

Changing the referencing relation

First we'll have a look at how violation can occur by changing the referencing relation *Orders*. This is quite similar to how violations to primary keys can occur. So first a violation can occur when a new tuple is inserted. Because it could be that not for every point in the valid time of the new tuple, there exists a corresponding primary key *ID* in the referenced relation *Products*. Be it because in the referenced relation the valid time starts too late, finishes too early or contains gaps.

Here, too, violation cannot occur when I delete a tuple in the referencing relation. So the other case when violation by changing a referencing relation can occur is when a tuple is updated. Once again this can be handled like above: I delete the old tuple and insert, if no violation occurs, the new tuple with the wished values into the relation. If a violation does occur the insertion must be rejected and the deletion must be undone.

Changing the referenced relation

The other situation in which a violation can occur, is when the referenced relation *Products* is changed. Here the cases in which a violation to the foreign key constraint can occur are deleting and updating.

Starting with deleting a tuple of *Products*, we do have to ensure that no tuples in *Orders* are referencing to the tuple we want to delete in its valid time period. Because otherwise these tuples in *Orders* will point to something that does not exist anymore. Therefore the deletion of such a tuple would have to be rejected.

Violating the FK constraint by updating can occur when I short the valid time. Because by shortening a tuple's valid time in *Products*, the same problems as for deleting can occur and therefor I virtually need to check the same as for deleting.

Integrity constraint violation in transaction time

So far, this have been the cases in which violation in valid-time tuples can occur. The next step is to discuss about how tuples with transaction time can violate the two integrity constraints. As already mentioned, transaction time refers to the time when a tuple is recorded in the database. The tables of these databases are called system-versioned tables. This is because the system, and not the user, sets the start and end times of the tuples and this times cannot be modified by the user.

To illustrate, a tuple with transaction time is handled as follows. When a new tuple is inserted into the database, its start-time is set to the current time – also called the transaction-timestamp - whereas its end-time is set to the highest possible date-value available in the database, this could be something like the year 9999. This is because we are talking about ground temporal databases and the value *now* does not exist there.

When that tuple is updated, it's handled like this: The old tuple is being copied and the copies end-date is set to the transaction timestamp. In the original tuple, the start-date is set to the new transaction timestamp.

Now there exist two different kind of tuples in the database: the historical rows (the ones that have been valid before) and the current system row, that is the row that contains the current time.

A deletion of a tuple does not really delete the tuple, it only sets the end-time of the current tuple to the transaction-timestamp. This means that the tuple existed until this timestamp but not anymore.

So actually, if the system set's the right times when inserting, updating and deleting tuples, there is no further need for integrity checking because this indicates that all historical rows do not violate any constraint. This does only hold if transaction time databases are considered isolated. When talking about bitemporal databases, that are databases with transaction time and valid time, also the cases discussed in the section "Primary Keys with valid time" and "Foreign Keys with valid time" have to be checked.

Adaption to now-relative databases

Integrity constraints in now-relative databases

As seen before, there a lot of scenarios in which integrity constraints can be violated in temporal databases, all of them can still occur in now-relative databases. In this section the additional scenarios I found for now-relative databases will be handled.

Primary Keys with now-relative databases

First, we'll handle the cases in which the primary key constraints with valid times can be violated, so whenever I talk about start and end times, valid times are meant.

In the previous chapter we've seen that a new inserted tuple with a time period can violate the primary key constraint if its time period overlaps with another time period of a tuple with the same Primary Key. Let's adapt this scenario to now-relative databases.

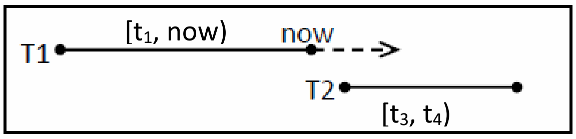

Figure 4 – Case 1 Primary Key Violation with determinate times

Description of the case 1

Figure 4 shows a tuple *T1* whose end time is *now*. This means it is currently valid. Suppose I insert a new tuple *T2* to the database whose start time is bigger than the current time. Presently when I insert the tuple into the database, no violation occurs, because no intersection between the time periods occurs. But when time moves on, the value of the variable *now* grows and therefore the periods of the two tuple are going to overlap. This is the time when a violation occurs.

Discussion of case 1

It is not that elementary to figure out how to handle this case. A possibility would be, that as soon as I insert the new tuple, the variable *now* in the old tuple's end time is set to the new tuple's start time. In our case, *now* in *T1* would be replaced at insertion time with *t3.* Like this no intersection occurs and the new tuple would from the given start time on replace the old tuple. Instead of replacing *now* at insertion time with *t3*, it could only be replaced at the time when the violation would occur.

But this does not have to be the wished behaviour. It could also be that a mistake occurred and somebody did not notice that there is a tuple with the variable *now* that might still be valid in the future. In this case, it would be more appropriate not to be able to insert the tuple at all, that means that the insertion will be rejected.

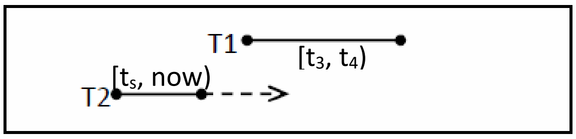

Figure 5 - Case 2 Primary Key violation with determinate times

Description of case 2

The reverse scenario might also be possible. Figure 5 shows a tuple *T1* with a ground start and a ground end time that already is in the database. A new tuple *T2* is inserted which ground start time and variable end time are at the moment both smaller than the old tuple's start time. Once again at this point in time, no violation occurs. But as time moves on, the new tuple's end time will catch up with the old tuple's start time and a violation occurs.

Possible solution to case 2

Here I show you one way how this case could be handled and what problems are associated with it. When someone adds a tuple with a valid time that is not ground, it most probably means that the tuple should still be valid in the future but it is not known how long. This means, that if the new tuple would only be valid until the old tuple becomes valid, one could have set the new tuple's end time to the old tuple's start time. So more probably the intention of inserting this tuple must have been that this tuple should be valid from a given time to a time in the future (expressed by *now*) but not in the period, where the old tuple is valid.

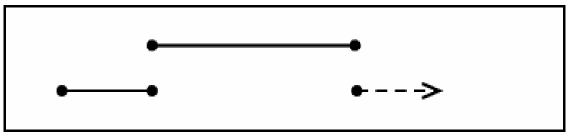

Figure 6 – Possible solution to case 2: insert two tuples instead of one

This could be solved like in Figure 6. That is, instead of inserting one tuple, two new tuples are inserted, a ground one that is valid until the old tuple becomes valid, and one that is valid from when the old tuple is not valid anymore.

With this solution another problem may occur, because it could be that for some circumstances the new tuple stops being valid before the old one becomes valid. In this case it would be impractical to have insert a ground and a variable tuple because when the tuple stops being invalid, the ground tuple must be adapted and the variable tuple must be deleted.

This were the basic scenarios in which primary key violation can occur. In the next part will be discussed how these violation can occur when the valid time of tuples is indeterminate.

Primary keys with indeterminate times

Indeterminate valid time means, that the start and / or end time lies somewhere between two values, but it is not exactly sure where. In now-relative databases this values can also be a variable. When we have tuples with indeterminate valid times, we distinguish between possible and definite time.

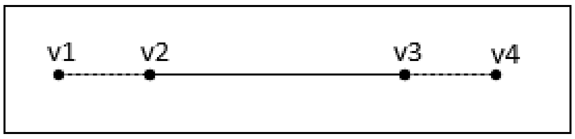

Figure 7 – Possible and definite time

Having a look at Figure 7 the possible timespan lies between *v1* and *v4* and the definite timespan lies between *v2* and *v3*.

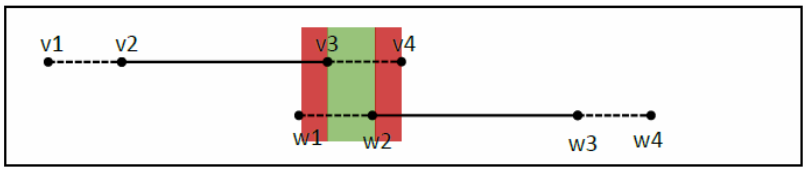

Figure 8 – Primary key violation with indeterminate times

In picture we see two tuples with indeterminate times. Where the line is dashed it means that the valid time is possible, otherwise it's definite. Now a primary key violation does not automatically occur when the times intersect, but I assume that a violation occurs when at least a part of one of the two tuples' valid times that overlap is definite. I make this assumption because as long as only possible times overlap, it only might be that there exist multiple tuples having the same primary key at the same time. But as soon as a part of the intersection is definite, it is clear that no other tuple with the same primary key can exist at that definite time. In Figure 8, a violation occurs where it is marked as red, no violation occurs where it is marked as green.

Now let adapt this scenario to now-relative databases. I will consider three different cases.

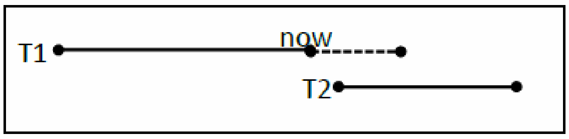

Figure 9 – Case 1 Primary Key violation with indeterminate times

Discussion of Case 1 (Primary Key violation with indeterminate times)

Case 1 is shown in Figure 9. In this case I have a tuple *T1* whose end time lies somewhere between *now* and a ground variable. If a new tuple *T2* is inserted whose definite time overlaps somewhere with the valid time (be it possible or definite) a primary key violation occurs, no matter what value there is for *now*. This is because the new tuple does only consist of definite time, and therefore at every point in the valid time period of the tuple *T2,* no other tuple with the same primary key can exist. So this tuple *T2* cannot be inserted and is rejected.

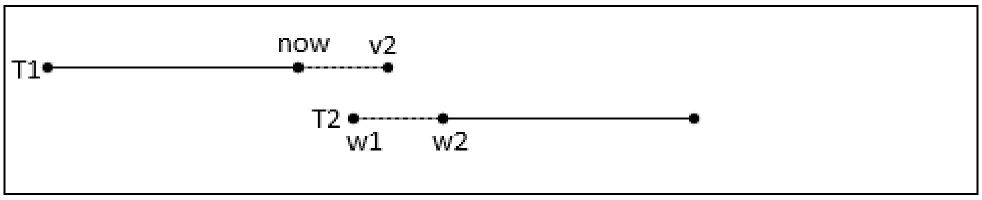

Figure 10 – Case 2 Primary Key violation with indeterminate times

Discussion of Case 2 (Primary Key violation with indeterminate times)

Case 2 is shown in Figure 10. Here we have the case that the end time of *T1* lies somewhere between *now* and a ground value. I can insert a new tuple as long as *w1* is bigger than *now* and *w2* is bigger than *v2*. But when time moves on, *now* grows and at some point the definite time of the first tuple will overlap with the possible time of the second tuple and a violation occurs. So a tuple *T2* must be rejected, if the end time of *T1* and the start time of *T2* overlap in possible valid time and the lower bound of *T1*'s end time is *now.*

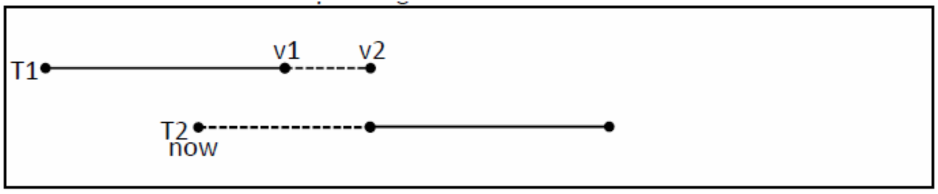

Figure 11 – Case Primary Key violation with indeterminate times

Discussion of case 3 (Primary Key violation with indeterminate times)

Case 3 is shown in Figure 11. This case is really interesting, because as time moves on, the violation disappears. We have a tuple *T1* with a determinate start time and an end time that lies between *v1* and *v2* (*v1* and *v2* are both ground values and *v1* is bigger than *now*). If someone wants to insert a new tuple with a start time that lies between *now* and a ground value, this is not possible at the moment because there is an intersection with the definite time of the first tuple. But as time moves on *now* will grow and this intersection will disappear. But of course it should not be allowed to insert such a tuple when *now* is smaller than *v1*

Foreign Keys with now-relative databases

In this section the scenarios for referential integrity constraints from ground temporal databases will be adapted to now-relative databases.

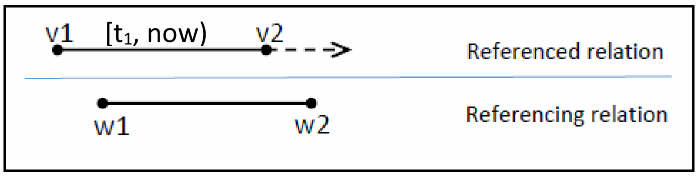

Figure 12 – Case 1 Foreign key violation

Discussion of case 1 (foreign key violation)

So assume we have a database with a tuple in the relation being referenced whose start time is a ground value *v1* and its end time is a variable value *v2* (like in Figure 12). If someone wants to insert a tuple in the referencing relation having a value *w1* between *v1* and *v2*, and *w2* bigger than *v2* the problem occurs, that we do not know if there exists a referenced tuple in the span from *v2* until *w2*. Because the only thing we know is that the tuple in the referenced relation is valid until *now* and if nobody changes this tuple it will also be valid in the future, but we are not allowed to assume that it will be valid in the future while inserting the referencing tuple. Therefore the referencing tuple must be rejected in this case.

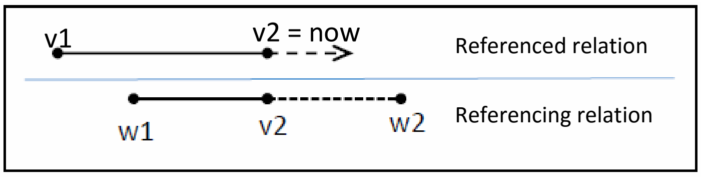

Figure 13 – Possible workaround for case 1

Possible workaround for case 1

A workaround I see, is that instead of inserting the tuple from above, a similar tuple could be inserted (like in Figure 13). The difference is, that the new tuple's end time is not the ground value w2 but a possible timespan that lies between *v2* and *w2*, where *v2* has always the same value as *v2* in the referenced relation. Like this, no violation occurs, because it is only possible but not definite that the tuple between *v2* and *w2* is valid. But of course this might not be the right solution for every case, because the tuple in the referencing relation does not have exactly the same meaning anymore.

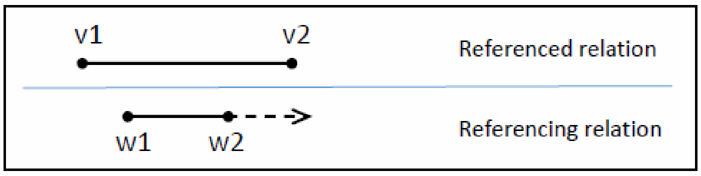

Figure 14 – Case 2 Foreign key violation

Description of case 2

The next case we'll have a look at is shown in Figure 14. We have a tuple in the referenced relation with the ground values *v1* and *v2* and a tuple in the referencing relation with *w1* being between *v1* and *v2* and *w2* being a variable value that at the moment also lies between *v1* and *v2*. Of course this tuple would not occur any violation at the moment, but as time moves on and no additional tuples have been inserted in the referenced relation in the meantime, *w2* grows and when *w2* becomes bigger than *v2*, the foreign constraint will not hold anymore, because not for every point in time there exists a referenced tuple anymore. There are two possible solutions I see.

Possible solutions to case 2

One is that as soon as time *v2* arrives, it must be checked if still no tuples for the future valid time of the referencing relation exists. If no, *w2* must be set to the current time. If there exist an accurate tuple (let's call it T3) that is valid from *v2* on in the referenced relation *w2* can keep its value *now.* But as soon as the valid time of T3 ends, the same inspection must be redone.

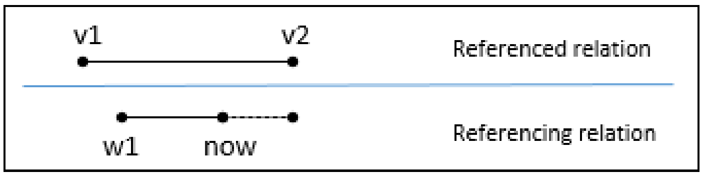

Figure 15 – Possible solution to case 2

The other solution is similar to the one in the previous example, that is that the violation is already checked when the tuple is inserted. So when someone wants to insert a tuple in the referencing relation, its end time cannot be depending on *now*. But it could once again be possible to insert an indeterminate tuple (like in Figure 15), whose end time lies between *now* and the ground value *v2*. Like this, it is still possible to say that the end time depends on *now* but only until this variable value has the same or a bigger value than *v2*. So the foreign constraint is not violated.

Transaction time with now-relative databases

Transaction time tuples in now-relative databased do not a lot differ from transaction time tuples in ground temporal databases. The difference that exists is, that if a new tuple is inserted to the database, the transaction end time is not set to the biggest possible value but to *now*. But the rest stays the same, when a tuple is updated, the old tuple is being copied, and in one tuple the transaction end time and in the other tuple the transaction start time is set to the current timestamp. When a tuple is being deleted, the transaction end time is set to the current time.

Therefore also in now-relative databases there exist historical system rows and current system rows that are generated automatically from the system. And therefore it must once again only be ensured that the system sets the right times when inserting, deleting and updating tuples and that there is no further need for integrity checking because this automatically indicates that all historical rows do not violate any constraint. Once again this does only hold if transaction time databases are considered isolated. When talking about bitemporal databases the primary key constraints and foreign key constraints with valid time have to be checked.

List of figures

If no source is mentioned, the figure was made by myself.

Bibliography

This report was written with help of these papers:

Wei Li, Richard T. Snodgrass, Shiyan Deng, Vineel K. Gattu, Aravindan Kasthurirangan. *Efficient Sequenced Temporal Integrity Checking*

Krishna Kulkarni, Jan-Eike Michels. *Temporal Features in SQL:2011*

James Clifford, Curtis Dyreson, Tomas Isakowitz, Christian S. Jensen, Richard T. Snodgrass. *On the Semantics of „Now" in Databases*

Matthias Nicola, Martin Sommerlandt. *Managing time in DB2 with temporal consistency – Enfore time-based referential integrity*