



UNIVERSITY OF CAPE TOWN



DEPARTMENT OF COMPUTER SCIENCE

# CS Honours Project Final Paper 2024

Title: Algorithmically Transforming TREND Temporal  
Conceptual Data Models Into SQL

Author: Stephan Maree

Project Abbreviation: TRENDy

Supervisor(s): Maria Keet

Category	Min	Max	Chosen
Requirement Analysis and Design	0	20	20
Theoretical Analysis	0	25	0
Experiment Design and Execution	0	20	0
System Development and Implementation	0	20	20
Results, Findings and Conclusions	10	20	10
Aim Formulation and Background Work	10	15	10
Quality of Paper Writing and Presentation	10		10
Quality of Deliverables	10		10
<u>Overall General Project Evaluation</u> ( <i>this section allowed only with motivation letter from supervisor</i> )	0	10	0
<b>Total marks</b>		<b>80</b>	<b>80</b>

# Algorithmically Transforming TREND Temporal Conceptual Data Models Into SQL

Stephan Maree  
University of Cape Town  
Cape Town, Western Cape, South Africa  
mrkste013@myuct.ac.za

## ABSTRACT

With the excess of storage available, temporal databases have gained increased viability and allow for richer data analysis. With the limited support for temporal data in current Database Management Systems, the temporal constraints represented in temporal logic needs to be manually implemented in the application code by developers. This project aimed to solve this problem by developing a software tool for automatically converting a TREND Temporal Conceptual Data Model into a database schema. The database schemas generated are fully functioning in a Database Management System and enforce TREND's temporal constraints in accordance to its specified formal semantics. The algorithm will scale log linearly with the input size and incorporate all the features of a standard Entity Relationship Diagram and its temporal extensions.

## CCS CONCEPTS

• **Information systems** → **Temporal data**; *Database utilities and tools*; **Entity relationship models**.

## 1 INTRODUCTION

Temporal databases expand typical databases by incorporating the time at which a particular entry is valid [14]. This differs from typical atemporal databases in which the validity of the entry is based on whether or not the entry exists in the database. These databases have become increasingly prevalent in both industry and academia as they allow for more expressive data and better analysis [7]. Examples of their use are in clinical data analysis [5] and video surveillance systems [19].

Like atemporal databases, temporal databases can be modelled using a Conceptual Data Modelling Language (CDML). An example of an atemporal CDML is the Entity-Relation Diagram (ER Diagram), which is the standard for modelling what data has to be stored in a database [10, 23]. When it comes to Temporal Conceptual Data Modelling Languages (TCDMLs), there are many proposed extensions of the ER Diagram, like *ER<sub>VT</sub>* [3], *TimeER<sub>plus</sub>* [11], or *ERT* [22]. Of particular note to this project is the TCDML TREND [6, 12].

TREND, which is short for Temporal information Representation in Entity-Relationship Diagrams, is a TCDML which has undergone multiple usability tests and shows promise in being accessible to beginner and experienced modellers [6]. It enables the user to model how entities change and evolve over time.

There exists no generic implementation methods for any TCDML. If a user wants to implement these temporal features, they would need to manually do it in the application logic, which costs development time and resources [14].

The project aims to solve this problem by algorithmically implementing the TREND TCDML by converting a model drawn by the user into a functional database schema, with full temporal logic enforcement. We do this by creating an algorithm that converts a serialised TREND model into a physical database schema, with full enforcement of temporal transition logic. Rigorous evaluation showed that the algorithm can generate databases for complex models and accurately implement TREND's temporal transitions.

This paper will first discuss some of the standards for temporal data implementations, and explain the temporal features of TREND in Section 2. Following this, in Section 3, the software development approaches and designs will be explained. This will be followed by an in depth explanation of the schema generation algorithm in Section 4, before exploring the implementation of the algorithm and SQL code generated by the software in Section 5. The process of testing the software is discussed in Section 6. Finally the quality of the software is discussed in Section 7, before concluding the paper.

## 2 BACKGROUND AND RELATED WORK

Currently, temporal data support in Database Management Systems (DBMSs) are limited. There are two temporal data standards that have received some attention, namely SQL:2011 [14] and TSQL2 [20].

SQL:2011 implements temporal data in two ways, application-time periods and system-time periods. Application-time periods refer to the period of time in which a fact is considered true in the environment being modelled, whereas system-time periods refer to the time in which a fact is represented in the database. In the literature, these are commonly referred to as valid time and transaction time respectively [7]. In SQL:2011, a period is represented as two columns in the table, namely the start time and the end time. It allows for referential integrity along periods, as well as updating and deleting on portions of a period. SQL:2011 has seen adoption in DBMSs such as MariaDB [18] and TDSQL [16].

TSQL2 is a standard that was proposed in 1994, but still sees some implementations in DBMSs such as Teradata [1]. There was an attempt to include it in the SQL standard, but after various criticisms it was not included [8, 14]. Unlike in SQL:2011, the periods in TSQL2 go into a single unnamed column. This is done using a new period datatype.

Neither of these standards natively implement any of TREND's temporal features. TREND extends ER Diagrams with the following features [6]:

- *Temporal elements*: Each entity, relationship, or attribute can have a clock icon appended to it. This signifies that the element is temporal and is only valid for a certain period.

An attribute can be pinned to indicate that its value cannot change once assigned.

- *Dynamic Extension and Evolution*: An element can transition in two ways. It can either change into another element and stop being the original (evolution), or it can go from only being a member of one element, to being a member of another element as well (extension). These are modelled with curved arrows, which are solid if the transition is mandatory, and dashed if it is optional. Extension is indicated with 'ext', and evolution is indicated with 'chg'. The text is capitalised if the transition occurs in the future, and lower case if it occurs in the past.
- *Quantitative Transition Constraints*: By adding a number to the dynamic extension or evolution, the transition becomes quantitative. This specifies the time at which the transition must occur relative to the original element.

A pin can be added to a temporal transition to enforce persistence, meaning that once the transition has occurred it the element needs to be in the final state for the rest of its existence in the database.

### 3 SYSTEM DESIGN

This section discusses the software development approaches and designs used while developing the system, as well as the data structures used in the object orientated system.

#### 3.1 Methodology

The project followed an iterative development style. This involved developing a subset of the final project, testing, and completing it before increasing the scope. The project started with developing the most basic features of the ER Diagram, these include entities, relationships, attributes, primary keys, and the multiplicity constraints for relationships.

After this, the more complex features of ER diagrams were implemented, these are features like weak entities, inheritance, and composite attributes.

Once all the atemporal features were present and functioning, the temporal tables were developed. Periods were added to temporal elements, and triggers for enforcing the referential integrity were developed.

The final features were the temporal transition constraints. Triggers were designed and written for each possible case, mandatory or optional, past or future, persistent or not, and quantitative or non-quantitative.

An iterative design process was chosen over an Agile process as Agile requires creating a minimal viable product early on in the development and refining it over a series of sprints. The size of the project was too large to be able to create a minimal viable product in a few sprints, and the majority of the project time allocation was needed to develop it.

#### 3.2 Requirements Gathering

The software needs to implement all of the standard Enhanced Entity Relationship Diagram (EER Diagram) features as well as each of TREND's temporal features. The implementation rules for the temporal transition constraints were gathered from the formalisation of TREND introduced in the paper [6].

#### 3.3 Architecture

The system was designed as a pipeline existing of three components: the SerialProcessor, the GraphProcessor, and the SQL\_Writer. The pipeline is managed by a Driver class, which instantiates each component of the pipeline, passes along the inputs and outputs, and provides a single point of contact for other systems implementing the software. The architecture is diagrammatically represented in Figure 1.

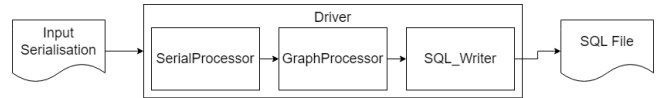


Figure 1: Architecture diagram for the software.

The SerialProcessor component is responsible for converting the input serialisation into a list of nodes and edges that match the expected serialisation for the program. If one wanted to modify this program to work with a different serialisation, this is what one would modify.

The nodes and edges from the SerialProcessor get passed as inputs into the GraphProcessor class. This component is responsible for creating and processing the objects used for writing the SQL. The implementation of this component is discussed in detail in the Algorithm section of this paper.

The SQL\_Writer component is responsible for taking in each of the object given by the GraphProcessor and converting it into SQL text. This SQL text is then downloaded to the user's device as a .sql file.

A pipeline architecture was chosen to make the code modular and maintainable. Each class is designed in a way that the coupling is low and the cohesion is high. They each have a clearly defined role and responsibility and do not have much dependency on each other. This allows developers to easily identify and isolate the parts of the code causing problems or needing changes. If a future developer wanted to rework how the serialisation is interpreted, they only need to understand and modify the SerialProcessor class.

A package diagram showing the interaction of all the classes in the software is included in the appendix in Figure 7.

#### 3.4 Objects and Data Structures

The system was designed to be Object Orientated, this means that the data is stored and acted upon as instances of classes. There are five classes that represent specific types of nodes: Attribute, Entity, Relation, ISA, and Trigger. Attribute, Entity, and Relation both implement the Element interface.

For a detailed view of the classes, the UML class diagram can be found in the Appendix in Figure 6.

The Attributes, Entity and Relation classes are used for storing all the features needed to write the SQL query for the table. Additionally, they all have their own 'writeSQL()' method, which processes their features and returns the required query as a string.

The ISA class exists for facilitating inheritance. It stores the ID of the parent node, and allows each of the children access to this value.

The Trigger class is used to write the triggers for implementing the temporal transition constraints, temporal referential integrity, and frozen attributes. It stores up to two Elements, and then the type of trigger. The SQL code for all the triggers are written in this class and the returned trigger is based on the 'type' parameter.

### 3.5 Input/Output

The input is a list of nodes and edges that are either manually written, or provided by a modelling tool. It combines all the possible characteristics of the TREND icons [6] into a single structure, where the combination of specified attributes are used to identify the TREND element. The translation between the TREND formalisation and the serialisation can be found in the appendix in Table 5. Given an entity named Person with a plain edge connecting to it, the serialisation in Listing 1 is produced.

```

1 {
2   Nodes: [
3     {
4       "id": "1",
5       "name": "Person",
6       "type": "entity",
7       "underlined": false,
8       "dash_underlined": false,
9       "temporal": false,
10      "pinned": false,
11      "double": false,
12      "optional": false,
13      "disjoint": false,
14      "complete": false,
15      "datatype": ""
16    },
17    ...
18  ],
19  Edges: [
20    {
21      "source": "1",
22      "destination": "2",
23      "double": false,
24      "dashed": false,
25      "source_arrow": false,
26      "dest_arrow": false,
27      "parent": false,
28      "curved": false,
29      "type": "",
30      "duration": 0,
31      "pinned": false,
32      "chronon": "YEAR",
33      "cardinality": "1..n"
34    },
35    ...
36  ]
37 }
```

**Listing 1: Example serialisation of a TREND model**

This serialisation was used instead of the TREND formalisation as it would allow for easier integration with modelling tools.

Each possible variation in the model was kept as a separate attribute, instead of having fewer, more detailed attributes like a type "optional quantitative change". This was done in order to reduce the dependence on a specific modelling tool's serialisation, as the given tool might not have such specific attributes.

Each node has an ID which is used for processing. These are used as the keys in their respective hash tables and are used to identify

the nodes in the edges. The Nodes have four possible types: 'entity', 'relation', 'attribute', and 'isa'. Here 'isa' refers to the inheritance node. The 'datatype' field can be any string recognised as a SQL data type, for example, 'INT' or 'VARCHAR(30)'. If the node is not an attribute and has no data type, the field is left as "". Various Boolean fields are used to describe the appearance and properties of the node. These are used to determine more complex properties such as if they are weak entities, or multi-valued attributes.

Each edge is defined by a pair of IDs indicating which nodes it connects. The "source\_arrow" and "dest\_arrow" fields indicate which end, if any, the arrow symbol is on. The temporal transitions in TREND are represented by setting "curved" to true and including both a duration and chronon for the transition. The duration can be any valid date and time data type recognised by the DBMS. Like the nodes, various Boolean fields exist to allow for processing interactions such as total participation in a relation and which node is the parent of an inheritance relationship.

## 4 ALGORITHM DESIGN

The serialisation is converted from a list of nodes and edges into data structures representing the final relational database using the algorithm discussed in this section. The algorithm is explained and an example is walked through to illustrate how it works. The theoretical complexity will also be discussed, as well as the considerations made to improve its efficiency.

### 4.1 The Algorithm

Once the user picks the option to generate a schema, the file will be read, and it will first loop through all the nodes. For each node it will determine the type and add it to the appropriate hash table. Hash tables are used as they can determine if a given item is contained within them in constant time, which is important for the efficiency and scalability of the program.

If the node is an attribute, an additional check is made to see if the node should get its own table in the database. This would be the case if it were either a temporal attribute or a multi-valued attribute. Should the attribute need its own table it gets added to another hash table. This process is modelled in Figure 2.

After the Nodes are all processed, the edges are processed. The edges are traversed twice; in the first loop it checks for edges that connect two attributes and combines them, and in the second loop it processes them normally. This is split into two separate loops to handle the case in which an attribute is added to an entity/relation before it is combined with its composite parts.

In the second loop, the edge is first checked to see if it is a temporal transition constraint. If it is, and the constraint is determined to be valid, a new Trigger object is generated and added to an array.

If the edge does not indicate a temporal transition constraint, the system checks if either end of the edge attaches to a relationship. In the event that it does, the other end is checked to see if it is an attribute or entity. Attributes are added to the relationship, and the edge is stored in the relationship object if there is an entity for later processing. If both ends of the edge are temporal, a trigger is created for enforcing temporal referential integrity.

The next case considered is if either end is an entity. Since the relationship edges would have been considered before, the only

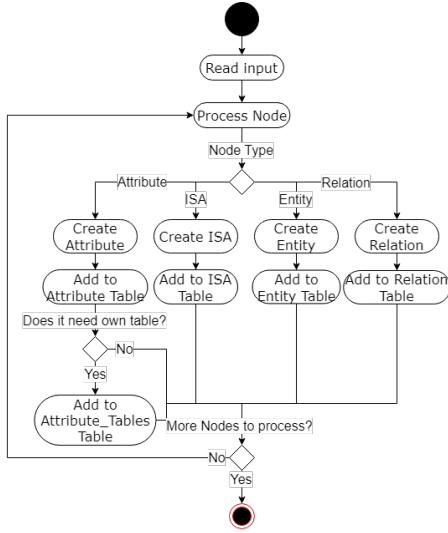


Figure 2: The initial processing of the nodes array.

options are an edge connecting an entity to an attribute, or an entity to an inheritance node. Attributes are added to the entity, assuming they do not have their own table. If the node is an inheritance node, the entity is added to the children or parents array based on if the edge is a parent edge or not. If the inheritance is disjoint and complete, the entity's table is disabled. The edge processing is modelled in Figure 3.

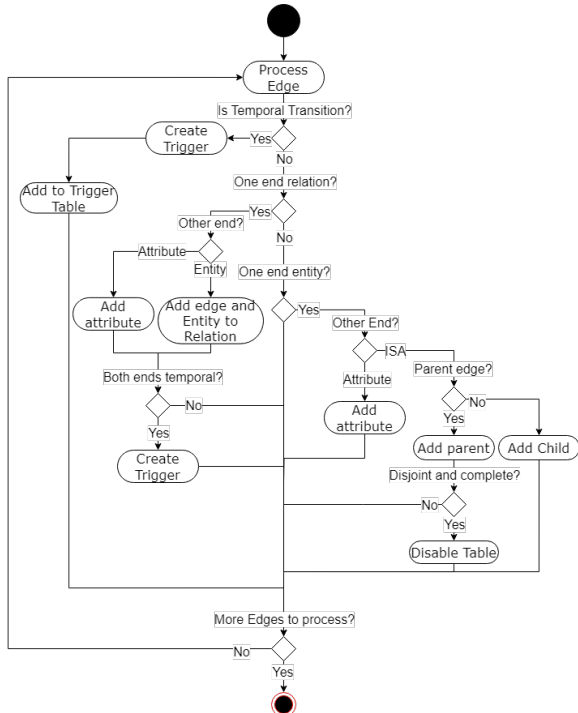


Figure 3: The initial processing of the edges array.

Once the edges are processed, the program works through everything in the 'Relations' list. The number of edges connecting to the relationship is checked, and if there are less than two, an error is thrown. Otherwise, if there are more than two entities connected, an N-ary relationship is present and a table is created for it. If there are exactly two entities connected to the relationship, then the cardinality is considered. The cardinality of both edges are compared and if the relationship is many-to-many, the relationship gets its own table. In the event that the relationship is one-to-one, the entity with total participation receives the identifier of the other entity as a foreign key. Total participation is randomly assigned if it is not specified in the model. If the relationship is one-to-many, the identifier of the entity on the 'one' side of the relationship is added as a foreign key to the entity on the 'many' side. It should be noted that values such as '0..1' are considered as 'one', and values such as '2..3' and '0..n' are considered as 'many' when it comes to structuring the tables. In both these cases with foreign keys, the attributes of the relationship are added to the foreign key holder. This implementation of relationship cardinality minimises unnecessary tables and improves the performance of data analysis [10, 15].

This introduces the problem of entities needing to be written in the correct order in the final SQL file. If a table references another table as a foreign key before that table is created, the SQL code will be unable to execute. To solve this, the entities need to be sorted in the correct order. After each relationship is processed, an 'Order Statement' is generated, this is a simple data structure which contains two IDs of entities. The first index indicates the entity which must occur before the second. These are used later in the program to sort the entities. The relationship processing is modelled in Figure 4.

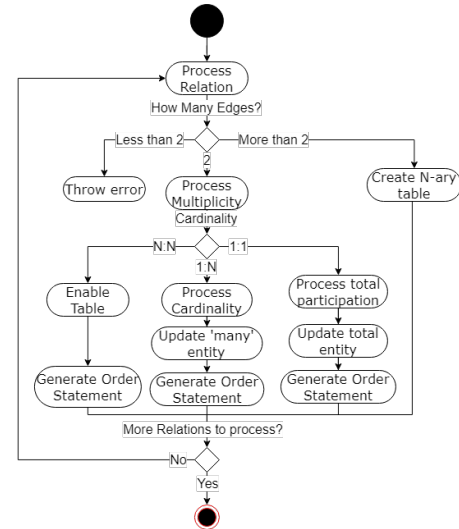


Figure 4: The processing of relationships.

The next thing processed is the inheritance constraints. This is done by looping through the entities table twice. The first loop checks if an entity stores the ID of an inheritance node, and if it is it adds the information of the parent entity. The second loop goes through all the entities again and adds the attributes and identifiers

of a parent to its children. This is split into two loops to handle the case when there is an inheritance chain, and the bottom of the chain needs the attributes from the top. If this were processed in one loop, the top of the chain might not have been processed, meaning the bottom would be unable to get the attributes from it.

It is at this point where the ordering of the entities are enforced. This is done using a priority queue approach. Each entity has a default priority of  $-1$ , indicating that its order is unimportant. The order statements created earlier are iterated through, and at each step, the priority of the “before” and “after” entities are updated as follows:

$$P_{\text{before}} = \max(0, P_{\text{before}})$$

$$P_{\text{after}} = \max(P_{\text{after}}, P_{\text{before}} + 1)$$

This ensures that the “after” entities always have a larger priority than the “before”. The order statements are iterated through twice, once forward and once in reverse. This ensures that larger, more complex models with multiple orderings for a single node are handled.

The entities are then added to an array which is sorted from lowest to highest priority. The sorted list, along with the relationships, attributes with their own tables, and the triggers are passed to the next component of the pipeline.

## 4.2 Example Walkthrough

Given the example model shown in Figure 5, the process of the algorithm to convert it into a SQL file will be discussed.

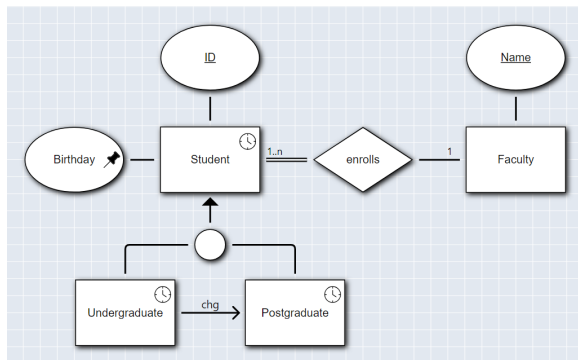


Figure 5: Example TREND model made with Richard Taylor’s modelling tool [21].

Before the input is given to the algorithm, the model shown converted into the serialisation discussed in section 3.5 using the SerialProcessor class. It is then passed to the GraphProcessor class as an array of nodes and an array of edges.

The array loops through all the nodes, creating the respective objects for the entities, attributes, relationships, and inheritance nodes. It then makes an initial sweep through all the edges, seeing if there are any edges connecting two attributes, indicating composite attributes. There are none in this example, so it proceeds to make another loop through the edges.

For every connection between an attribute and entity, the attributes are added to the entity. When adding the Birthday attribute to the Student entity, a Trigger object is created for enforcing the frozen attribute. If an attribute is underlined, it is added as a primary key of the entity. The temporal transition between Undergraduate and Postgraduate is found and a Trigger is created between the objects. The edge between the inheritance node and Student causes the id of Student node to be appended to the ISA object as it is the parent. The ID of the inheritance node is stored in Undergraduate and Postgraduate when the respective edges connecting them is found. When the edges connecting to the relationship are found, they are stored inside the relationship along with the entities they connect.

The algorithm goes through all the relationships, and in this example there is only one. Since there are two only two edges connected to the node, it compares the cardinality of the edges. In this example the relationship is one-to-many, so the primary key of Faculty is added as a foreign key in Student and an order statement is created that specifies that Faculty’s table must be written before Student’s. After this the entities are looped through to see if they are children connected to an inheritance node. Since Undergraduate and Postgraduate are, the parent node gets added to the them. It then loops through all the entities, adding the primary key of parents to all the entities. Since the inheritance is not disjoint and complete, Student gets its own table and its attributes are not passed down to its children.

The order statements are then processed. In this example, the only order statement is that Faculty comes before Student. Faculty is then assigned a priority of 0, and Student is assigned a priority of 1. The entities are then sorted in order of ascending priority.

The entities, relationships, attributes with tables, and triggers are then passed to the SQLWriter class. This loops through each of the lists provided, generating and appending the relevant SQL code to a file, and downloads it to the user’s device upon completion.

## 4.3 Complexity

Given  $n$  nodes and  $e$  edges, the algorithm runs asymptotically in  $O(n \log n)$ . The core part of the algorithm scales linearly in  $O(n + e)$ .

For the core algorithm (everything excluding the sorting), linear time is achieved by using a strategy of multiple linear iterations through the data, rather than nesting complex actions within a single loop. Inside each loop no action was permitted that does not run in constant time, such as nested loops or recursion. Hash tables were used to store elements as they allow you to see if an element exists in the table in constant time by calling the table with the element’s ID and seeing if the reference exists. Determining the length of arrays and hash tables is also done in constant time as each data structure contains a length variable that gets incremented on insertion and decremented on deletion. Arrays were combined by concatenation, rather than going through each element in the array and inserting it into another.

The bottleneck of the algorithm is the sorting of the entities. Since there are no known sorting algorithms that scale linearly with input size in the worst case, the algorithm is limited to scaling at  $O(n \log n)$ .

## 5 IMPLEMENTATION

This section focuses on the languages and tools used to develop the software, as well as the SQL implementations of the temporal transition constraints of TREND.

### 5.1 Languages

This program was implemented in JavaScript version 1.5 as to simplify the integration with web-based applications. The SQL code for the temporal features were written for MariaDB version 11.3.2 using application-time tables [17] and follow the SQL:2011 specifications [14]. All testing was done on the HeidiSQL 12.6 client [4].

JavaScript does not natively support interfaces, instead, it uses what is called Duck Typing (If it walks like a duck and quacks like a duck, it is a duck). This means that the classes need to implement certain methods, and have them return the right objects, but it will not be enforced by the language.

These informal interfaces are still necessary for the project as the code in the Trigger class for writing SQL queries needs to work for entities, attributes, and relationships. Ensuring the classes implement the correct method types is important for the modularity of the code, and needs to be enforced by the developers.

A consequence of using JavaScript is that a custom implementation of Merge sort needed to be used. A custom Merge sort is used as the default sorting algorithm of JavaScript is Quicksort, which has a worst case time complexity of  $O(n^2)$ , whereas Merge sort's is only  $O(n \log(n))$ .

### 5.2 Temporal Elements

The atemporal features of the models are implemented in the standard way and will not be discussed in this paper [10, 15]. When an entity is labelled as temporal, two columns are added at the end of the table. These are named *name\_start* and *name\_end*, where *name* is the name of the table. They use the DATE datatype, and are included as part of the primary key. The period between the start and the end is labelled as unique and overlaps are disabled.

Given a simple model consisting of only a temporal entity named 'Person' and an identifier named 'ID', with a single attribute 'Name', the SQL code in Listing 2 is generated. Note that the text in blue indicates the names taken from the input model.

```
1 CREATE OR REPLACE TABLE Person(
2   ID VARCHAR(30) NOT NULL,
3   Name VARCHAR(30) NOT NULL,
4   Person_start DATE,
5   Person_end DATE,
6   PERIOD FOR Person_period(Person_start, Person_end),
7   UNIQUE (ID, Person_period WITHOUT OVERLAPS),
8   PRIMARY KEY (ID, Person_start, Person_end)
9 );
```

**Listing 2: Temporal table example**

The SQL code for the tables are generated line-by-line based on various factors, such as whether or not they are temporal, how many attributes and primary keys they have, if they have foreign keys, or if they are weak entities. The SQL code in Listing 2 will generate the table shown in Table 1.

**Table 1: An example table with application-time periods**

ID	Name	Person_start	Person_end

If a relationship is marked as temporal, it will always have its own table with a start and end column, regardless of the multiplicity. This was done to avoid having multiple periods in one table. For similar reasons temporal attributes get their own table with the identifier of the entity or relationship they belong to, the value itself, and the period columns.

If we were to add a temporal attribute 'Home' to the previous model, Table 2 will be generated

**Table 2: Temporal attribute example**

ID	Name	Person_start	Person_end

ID	Home	Home_start	Home_end

In this case, the ID, Home, Home\_start and Home\_end tuple will be the primary key.

Here the Home column will store the value of the attribute. The decision was made to not include the periods from the Person table in the Home table as it would add needless complexity to the database. Instead, the Person's periods are implicitly present via triggers that enforce temporal referential integrity.

Whenever an edge is processed where both ends have temporal elements (e.g. a temporal entity and temporal relation), a trigger is created that ensures the periods of entries overlap. In the previous example used to illustrate temporal attributes, the SQL code in Listing 3 will be generated.

```
1 CREATE TRIGGER Home_Person_temporal_attribute_insert
2 BEFORE INSERT ON Home
3 FOR EACH ROW
4 BEGIN
5   DECLARE period_match INT;
6   SELECT COUNT(*)
7   INTO period_match
8   FROM Person
9   WHERE (ID = NEW.ID)
10  AND NEW.Home_start >= Person_start
11  AND NEW.Home_end <= Person_end;
12  IF period_match = 0 THEN
13    SIGNAL SQLSTATE '45000'
14    SET MESSAGE_TEXT = Period in Home does not exist
15    in Person.;
16  END IF;
17 END;
```

**Listing 3: Temporal referential integrity example**

Two of these are generated, one that runs on insert and another that runs on update. In the case of a temporal attribute the trigger will always be applied to the attribute table. If there is a temporal entity and temporal relationship edge, the trigger will be applied to the relationship table. This trigger will not be generated if the other element does not have its own table, e.g. if it connects to the parent of a disjoint and complete inheritance relationship.



### 5.3 Dynamic Temporal Transitions

Other triggers written are those of the temporal transitions implemented in TREND. The implementation of these can be divided into two groups, being the dynamic transition constraints and the quantitative transition constraints. The main difference between these implementations are the need to consider contiguous periods in the quantitative case.

Unlike the regular tables, the temporal transition constraints are generated using code templates. The names and identifiers of the elements are filled in when the code is generated. The template does change depending on how many primary keys each of the tables have, but the logic is the same for all temporal elements. A full list of the SQL code for all the temporal transition constraints can be found in section B of the appendix. The logic behind the code will be discussed here.

Consider the case with two temporal entities, Child and Adult. Both of these have an identifier ID. If these are connected with a solid, curved line from Child to Adult labelled with chg, it indicates a mandatory temporal evolution in the past. In layman's terms, it says that each Adult was a Child before, but is no longer a Child. There are two components in this statement that need to be enforced. The first one is that each Adult must have been a Child. The trigger to enforce this is shown in Listing 4.

```

1 CREATE TRIGGER Child_chg_to_Adult_1
2 BEFORE INSERT ON Adult
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exist INT;
6     DECLARE overlap_count INT;
7     SELECT COUNT(*)
8     INTO initial_exists
9     FROM Child
10    WHERE NEW.ID = ID
11          AND NEW.Adult_start >= Child_end;
12     SELECT COUNT(*)
13     INTO overlap_count
14     FROM Child
15    WHERE NEW.ID = ID
16          AND (
17              (NEW.Adult_end > Child_start AND NEW.
18                Adult_start <= Child_start)
19              OR
20              (NEW.Adult_start >= Child_start AND NEW.
21                Adult_end <= Child_end)
22              OR
23              (NEW.Adult_start >= Child_start AND NEW.
24                Adult_end <= Child_end)
25          );
26     IF overlap_count > 0 THEN
27         SIGNAL SQLSTATE '45000'
28         SET MESSAGE_TEXT = 'Adult cannot overlap with
29           Child.';
30     END IF;
31     IF initial_exists = 0 THEN
32         SIGNAL SQLSTATE '45000'
33         SET MESSAGE_TEXT = 'Adult must have been in Child
34           before.';
35     END IF;
36 END;
```

**Listing 4: First dynamic evolution trigger**

This compares the start date of the new entry in the Adult table and the end dates of all the entries in the Child table. If there are

no entries where the Child's end time is less than the new Adult's start time, then the temporal evolution constraint is not satisfied and the entry is not made. It also sees if there are any entries in the Child table that overlap with the new Adult entry. If this is the case then the new entry is also prevented.

The next component of the statement is that each Adult is no longer a member of the Child entity. The implementation of this component varies based on if the transition is persistent or not. If there is no persistence, you should be able to add the entity back in the Child table once its period in the Adult table is over. The SQL implementation for that case is shown in Listing 5.

```

1 CREATE TRIGGER Child_chg_to_Adult_2
2 BEFORE INSERT ON Child
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     SELECT COUNT(*)
7     FROM Adult
8     WHERE NEW.ID = ID
9           AND (
10              (NEW.Adult_end > Child_start AND NEW.
11                Adult_start <= Child_start)
12              OR
13              (NEW.Adult_start >= Child_start AND NEW.
14                Adult_end <= Child_end)
15              OR
16              (NEW.Adult_start >= Child_start AND NEW.
17                Adult_end <= Child_end)
18          );
19     IF initial_exists > 0 THEN
20         SIGNAL SQLSTATE '45000'
21         SET MESSAGE_TEXT = 'Child has already changed into
22           Adult.';
23     END IF;
24 END;
```

**Listing 5: Second dynamic evolution trigger**

This checks if there are any periods in Adult that overlap with the new period in Child and prevents the insertion if there are. In the case with persistence, the condition for the 'WHERE' clause in line 8-15 is replaced with

```

WHERE NEW.ID = ID
AND NEW.Child_end > Adult_start;
```

Rather than preventing just the overlaps, this prevents any insertions into Child with a period going past an entry in Adult. The other dynamic transition constraints function in a similar way, but with minor alterations. The optional dynamic evolution in the past works the same as its mandatory counterpart in terms of preventing overlaps, but it does not have the check for an entry in Child. The future dynamic transitions are implemented in the same way, since we cannot enforce a transition that might not have occurred yet.

If the model contained a mandatory dynamic extension in the past, when inserting in Adult, the new entry's start date would be compared to each corresponding entry in Child's start date. If there exists a Child with a start date before the new Adult's start date, then the entry can be made, otherwise it is prevented. Persistence adds a trigger that prevents anything from being inserted in Child that doesn't overlap with an entry in Adult if that object has a period in Adult with an earlier period. Optional dynamic extensions in the past do not produce any triggers, and neither do the future dynamic extensions.



## 5.4 Quantitative Temporal Transitions

Consider the previous example of the Child and Adult. This time, there is a solid, curved arrow labelled with ‘CHG 18’ going from Child to Adult, meaning every entity in Child must become a member of Adult after 18 units, ceasing to be a Child. In this example the units will be years. In order to ensure that the transition occurs 18 years after the Child was created, we need to prevent any Adult from being added when it has not been in Child for 18 years, prevent any Child for existing for a period longer than 18 years, and ensure no overlap occurs between them. We can ensure no Child can exceed 18 years using the trigger in Listing 6.

```

1 CREATE TRIGGER Child_CHG_to_Adult_2
2 BEFORE INSERT ON Adult
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_overlap INT DEFAULT 0;
6     DECLARE total_time INT DEFAULT 0;
7     DECLARE prev_end DATE DEFAULT NULL;
8     DECLARE cur_start DATE;
9     DECLARE cur_end DATE;
10    DECLARE done INT DEFAULT 0;
11    DECLARE cursor_a CURSOR FOR
12        SELECT Child_start, Child_end
13        FROM Child
14        WHERE NEW.ID = ID
15        AND Child_end <= NEW.Adult_start
16        ORDER BY Child_start;
17    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
18
19    SET total_time = TIMESTAMPDIFF(YEAR, NEW.Child_start,
20        NEW.Child_end);
21    IF total_time > 18 THEN
22        SIGNAL SQLSTATE '45000'
23        SET MESSAGE_TEXT = 'No period in Child can result
24        in more than 18 continuous units.';
25    END IF;
26    OPEN cursor_a;
27
28    read_loop: LOOP
29        FETCH cursor_a INTO cur_start, cur_end;
30        IF done THEN
31            LEAVE read_loop;
32        END IF;
33
34        IF prev_end IS NULL OR prev_end = cur_start THEN
35            SET total_time = total_time + TIMESTAMPDIFF(
36            YEAR, cur_start, cur_end);
37            SET prev_end = cur_end;
38        ELSE
39            SET total_time = TIMESTAMPDIFF(YEAR, cur_start
40            , cur_end);
41            SET prev_end = cur_end;
42        END IF;
43        IF total_time > 18 THEN
44            SIGNAL SQLSTATE '45000'
45            SET MESSAGE_TEXT = 'No period in Child can
46            result in more than 18 continuous units.';
47        END IF;
48    END LOOP;
49
50    CLOSE cursor_a;
51 END;
```

**Listing 6: Quantitative temporal transition example**

This creates a temporary table of all the entries with matching IDs and orders them with respect to the start date. After checking that

the new entry does not exceed 18 years on its own, it then loops through each of the periods in the table, taking note of the duration of each period. If the periods are contiguous, the lengths are added to the total, otherwise the total gets reset. Once a contiguous period exceeds 18 years it gives an error.

Preventing an Adult from being inserted without first having been a child for 18 years is enforced in a similar way. It loops through each period in descending order, starting from the entry in Child with an end that matches the start of the new adult. This time, instead of resetting the counter if the periods do not match, it leaves the loop as the periods are no longer contiguous with the new Adult. The overlapping constraints are enforced in the same way as the dynamic temporal transitions, both when persistent and not.

The quantitative evolutions in the past remove the check for values exceeding the specified duration in Child, as the statement “Every Adult must have been in Child for 18 years” does not imply that every child needs to become an adult after 18 years. When the transition is optional, it needs to happen after “at least” 18 years. The only difference from the mandatory case is that an error is thrown if Child has existed for less than 18 years, as opposed to being thrown when the period is not exactly 18 years.

Quantitative extension is implemented similarly to Quantitative evolution, except that when inserting into Child, it only throws an error if you exceed 18 units and have not entered anything in Adult.

## 5.5 Additional Features

Beyond the primary features discussed so far, there are some implementations worth mentioning. Another feature of TREND is frozen attributes. These are different from regular attributes in that once they are set, their value cannot change. Given an entity named ‘Person’, with an identifier called ‘ID’, and a frozen attribute ‘birthday’, the trigger from Listing 7 gets generated.

```

1 CREATE TRIGGER Person_Birthday_pin
2 BEFORE UPDATE ON Person
3 FOR EACH ROW
4 BEGIN
5     IF NEW.Birthday <> OLD.Birthday THEN
6         SIGNAL SQLSTATE '45000'
7         SET MESSAGE_TEXT = 'Birthday has been pinned and
8         cannot be changed.';
9     END IF;
10 END;
```

**Listing 7: Frozen attribute example**

This implementation is basic and states that whenever a value is updated, if the original value of the attribute is not the same as the new one, the update is blocked.

Another case worth discussing is that of transitions between temporal relationships. This raises the problem of needing to compare two primary keys to match cases. Since the order in which primary keys are written in relationship tables is not set, it is possible that the two tables have the primary keys written in different orders. Comparing the primary keys together in two different ways handles this case. Assume relationships with two identifiers, ‘AID’ and ‘BID’. Entries are considered to refer to the same relationship when the following holds:

```
(NEW.AID = AID AND NEW.BID = BID)
OR
(NEW.AID = BID AND NEW.BID = AID)
```

By accounting for both possible permutations, it ensures that the entries in the relationship tables are correctly matched.

Recursive relationships also need special treatment. If a relationship has the same entity on both sides it always gets its own table. This is because if a table needed an existing entry in itself before being able to insert, then nothing will be able to be added to table.

In a standard relationship between two different entities the name of the entity is appended to the start of its foreign key. This avoids problems that could result from the entities using the same name for their identifiers. This solution would not work if an entity has a relationship with itself as the entity names are the same as well. Instead, we modify the names in the table to add a number to the entity names. Given an entity named 'Person' with an identifier 'ID' and a recursive relationship labelled 'Knows', the primary keys of the relationship table will be named 'Person\_1\_ID' and 'Person\_2\_ID'.

## 6 TESTING AND EVALUATION

Testing of the software was divided into two parts, the algorithm and the SQL implementation of TREND transition constraints. The algorithm requires testing with various input models, these could be small to test individual features or large to test the interactions between different features. The testing models were created using Richard Taylor's TREND modelling tool [21].

Test models were created that covered all of TREND's features. These tests showed that the models correctly implemented the features in isolation. Afterwards, models were made that incorporated multiple features interacting with each other, like weak entities connecting to parents of inheritance relationships. These interactions were all implemented correctly. Many logically invalid models were also tested to see how the software would respond. Errors are typically thrown, or the invalid part of the model is simply ignored. Details of these tests can be seen in Table 3 in the appendix. Correct databases are generated for various scenarios and the interactions between these, proving the reliability of the software.

The SQL code for the temporal transitions were validated by comparing them to the formal semantics of TREND. The temporal logic implementation is shared by entities, attributes, and relationships, meaning that they could be tested at the same time. Basic models were created for each temporal transition, and various entries were made to test the constraint implementations. The testing showed that entries that violated the constraint semantics were not allowed, while those that didn't were allowed to be inserted. The tests done to validate the implementation of TREND's semantics can be found in Table 4 of the appendix.

## 7 DISCUSSION

The software produced is able to efficiently and correctly generate SQL code for databases from a TREND model. The testing of the schema generation algorithm shows great promise in terms of robustness, and the comparisons with the TREND semantics indicate that the temporal transition constraints are correctly implemented. The project followed modern software development practices and

produced a fully functional object orientated tool. These factors indicate that the software successfully meets all the project requirements.

There are some limitations with the software, as it assumes the models provided are semantically correct. Incorrect models have the potential to cause unintended behaviour, or to not produce a database at all. Fixing incorrect models, however, falls outside of the scope of the project and thus this does not negatively reflect on the software produced. The other problem is with semantically correct, but logically invalid statements. These would require an automated reasoner to identify, which is outside of the project's scope. There are also the inherent limitations of automatically generated code, as a developer might prefer to add some personal touches or modifications to the generated SQL code. To assist with this, the generated code follows general conventions for the layout of SQL, including indents and line breaks. This makes the SQL code more readable and understandable, should one desire to go over it.

Since the algorithm scales in log linear time, it scales efficiently with the size of the model. In most cases, the model generates a file in the order of magnitude of a millisecond, increasing by an average of 5 milliseconds if you include the time of downloading the file to the user's device. This demonstrates the scalability of the code. The pipeline architecture of the software is designed to make the software portable and modular. Future developers can easily modify and add sections of code thanks to the low coupling and high coherence. The modifications needed to be made to implement the software to an existing tool have been isolated to a single class, allowing for quick integration. This demonstrates the cohesion and portability of the code.

In the future, the software can be expanded integrating an automated reasoner to ensure complete correctness of models. This could perhaps be used to implement specific solutions to these cases and allow for more expressive use of the TREND TCDML. The software could also be converted into extensions of popular modelling tools, allowing for more widespread use. It could also be extended to implement temporal graph databases [2, 9, 24], or ontology based data models such as [13].

## 8 CONCLUSIONS

Using a planned and iterative development approach, this software project produced a tool that can generate complex temporal databases from a TREND model. This was done by developing an algorithm that converts serialised diagrams into objects which represent the final database, and generating the SQL code from these. The modularity and portability of the code means that it can be easily integrated with modelling tools to effectively and efficiently generate SQL code. This removes the need for developers to implement advanced temporal features in the application logic, saving people and organisations development costs and time. Thorough testing and comparisons with the semantics of TREND have shown that the software is robust and can effectively implement the user's models.

Possible expansions are integrating automated reasoners to ensure the models contain no logical errors, modifying it as an extension to popular modelling tools, and extending it to work with temporal graph databases.

The software is expected to reduce the barrier to entry of temporal databases and helping to realise TREND as the preferred TCDML in research and industry.

## REFERENCES

- [1] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. 2013. Temporal query processing in Teradata. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) (EDBT '13). Association for Computing Machinery, New York, NY, USA, 573–578. <https://doi.org/10.1145/2452376.2452443>
- [2] Landy Andriamampianina, Franck Ravat, Jiefu Song, and Nathalie Vallès-Parlangeau. 2022. Graph data temporal evolutions: From conceptual modelling to implementation. *Data & Knowledge Engineering* 139 (2022), 102017. <https://doi.org/10.1016/j.datak.2022.102017>
- [3] Alessandro Artale, Christine Parent, and Stefano Spaccapietra. 2007. Evolving objects in temporal information systems. *Annals of Mathematics and Artificial Intelligence* 50, 1 (2007), 5–38.
- [4] Ansgar Becker. 2002. *HeidiSQL - MariaDB/MySQL, Microsoft SQL, PostgreSQL, SQLite, Interbase and Firebird*. <https://www.heidisql.com/>
- [5] Andreas Behrend, Philip Schmiegelt, Jingquan Xie, Ronny Fehling, Adel Ghoneimy, Zhen Liu, Eric Chan, and Dieter Gawlick. 2015. Temporal State Management for Supporting the Real-Time Analysis of Clinical Data. *Advances in Intelligent Systems and Computing* 312 (01 2015), 159–170. [https://doi.org/10.1007/978-3-319-10518-5\\_13](https://doi.org/10.1007/978-3-319-10518-5_13)
- [6] Sonia Berman, C. Maria Keet, and Tamindran Shunmugam. 2024. The temporal conceptual data modelling language TREND. arXiv:2408.09427 [cs.DB] <https://arxiv.org/abs/2408.09427>
- [7] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. 2018. Temporal Data Management – An Overview. In *Business Intelligence and Big Data*, Esteban Zimányi (Ed.). Springer International Publishing, Cham, 51–83.
- [8] Hugh Darwen and C.J. Date. 2006. *An Overview and Analysis of Proposals Based on the TSQL2 Approach*. Apress. <http://www.dcs.warwick.ac.uk/~hugh/TTM/OnTSQL2.pdf>
- [9] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. 2021. A model and query language for temporal graph databases. *The VLDB Journal* 30, 5 (2021), 825–858.
- [10] R. Elmasri and S.B. Navathe. 2004. *Fundamentals of Database Systems*. Addison-Wesley.
- [11] Heidi Gregersen. 2005. Timeer plus: A temporal eer model supporting schema changes. In *British National Conference on Databases*. Springer, Springer, Berlin, Heidelberg, 41–59.
- [12] C Maria Keet and Sonia Berman. 2017. Determining the preferred representation of temporal constraints in conceptual models. In *Conceptual Modeling: 36th International Conference, ER 2017, Valencia, Spain, November 6–9, 2017, Proceedings 36 (Lecture Notes in Computer Science, Vol. 10650)*. Springer, Cham, 437–450.
- [13] Christina Khnaisser, Vincent Looten, Luc Lavoie, Anita Burgun, and Jean-François Ethier. 2024. Building ontology-based temporal databases for data reuse: An applied example on hospital organizational structures. *Health Informatics Journal* 30, 2 (2024), 14604582241259336. <https://doi.org/10.1177/14604582241259336> arXiv:https://doi.org/10.1177/14604582241259336 PMID: 38848696.
- [14] Krishna Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *ACM Sigmod Record* 41, 3 (2012), 34–43.
- [15] Vincent S Lai, Jean-Pierre Kuilboer, and Jan L Guynes. 1994. Temporal databases: model design and commercialization prospects. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems* 25, 3 (1994), 6–18.
- [16] Wei Lu, Zhanhao Zhao, Xiaoyu Wang, Haixiang Li, Zhenmiao Zhang, Zhiyu Shui, Sheng Ye, Anqun Pan, and Xiaoyong Du. 2019. A lightweight and efficient temporal database management system in TDSQL. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2035–2046.
- [17] MariaDB. 2024. *Application-Time Periods*. <https://mariadb.com/kb/en/application-time-periods/>
- [18] MariaDB. 2024. *MariaDB Server: the innovative open source database*. <https://mariadb.org/>
- [19] Fabio Persia, Fabio Bettini, and Sven Helmer. 2017. An Interactive Framework for Video Surveillance Event Detection and Modeling. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (Singapore, Singapore) (CIKM '17). Association for Computing Machinery, New York, NY, USA, 2515–2518. <https://doi.org/10.1145/3132847.3133164>
- [20] R Snodgrass. 1995. *The TSQL2 Temporal Query Language*. Springer, New York, NY.
- [21] R Taylor. 2024. CS Honours Project: Designing a graphical modelling tool for the TREND conceptual data modelling language. Honours Project for CSC4002W. University of Cape Town, South Africa. 10 pages.
- [22] C Theodoulidis, Pericles Loucopoulos, and Benkt Wangler. 1991. A conceptual modelling formalism for temporal database applications. *Information Systems* 16, 4 (1991), 401–416.
- [23] Jeffrey D. Ullman. 1988. *Principles Of Database And Knowledge-Base Systems*. Computer Science Press.
- [24] Fu Zhang, Zhiyin Li, Dunhong Peng, and Jingwei Cheng. 2021. RDF for temporal data management—a survey. *Earth science informatics* 14 (2021), 563–599.

## A SUPPLEMENTARY MATERIAL

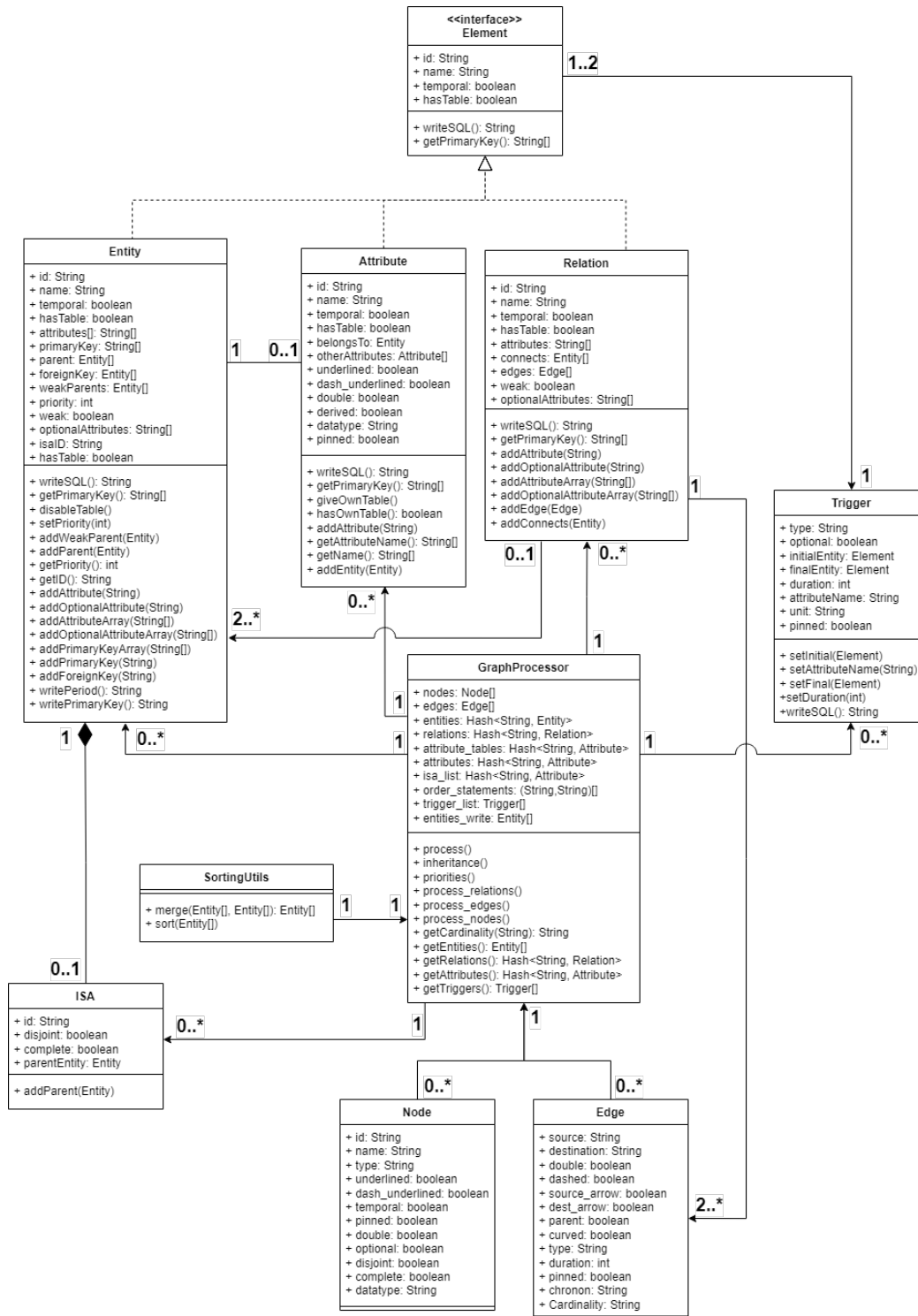


Figure 6: UML class diagram for data structures

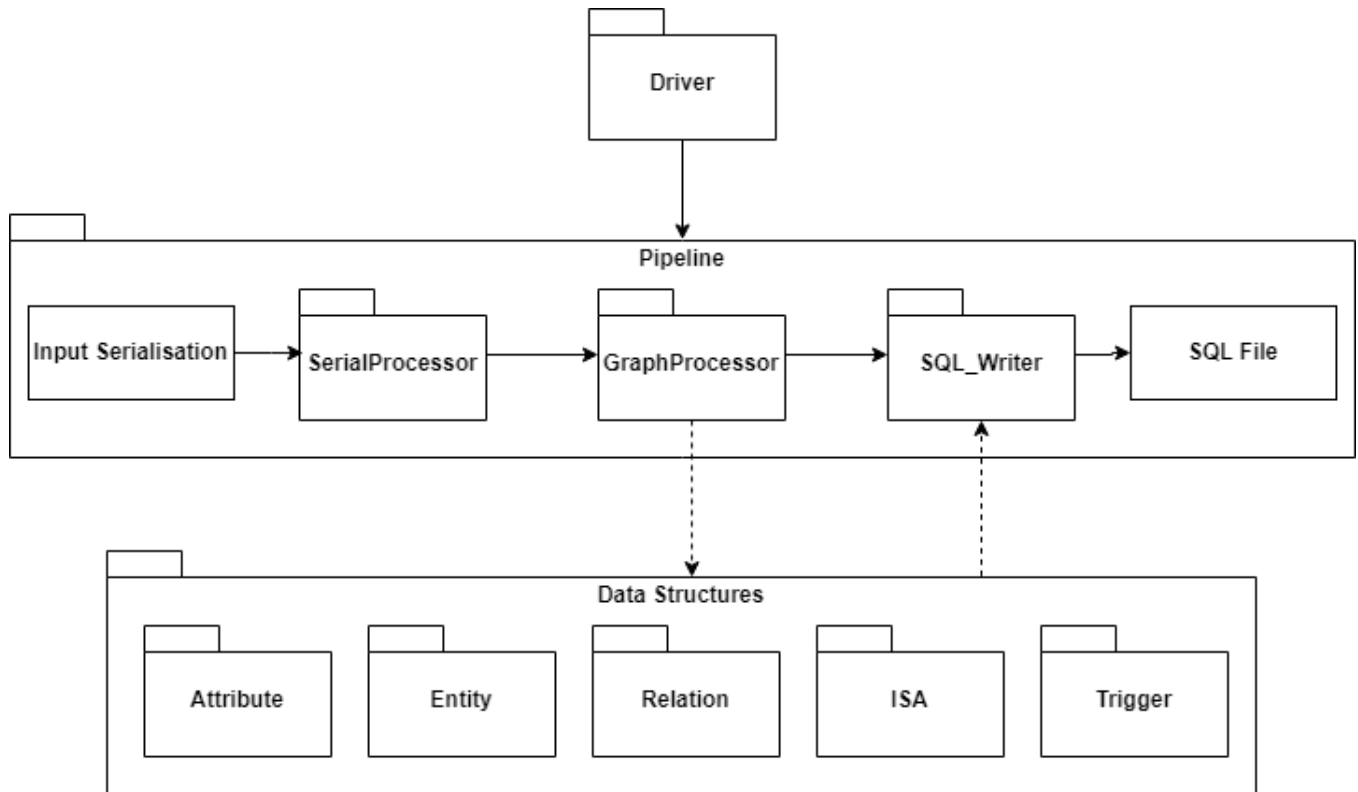
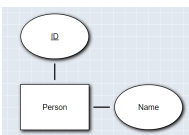
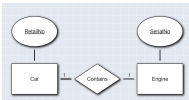
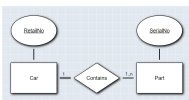
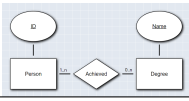
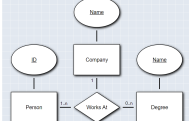
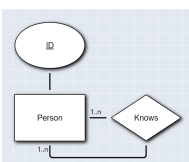
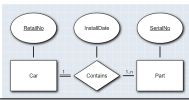
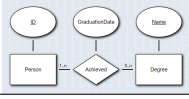
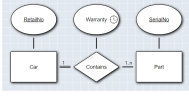
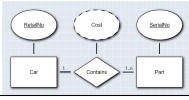
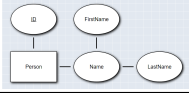
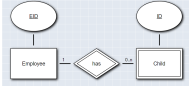
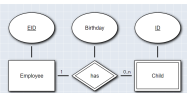
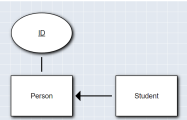
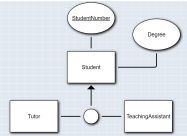
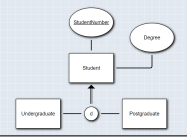
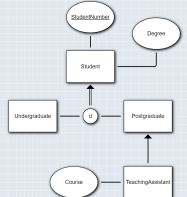
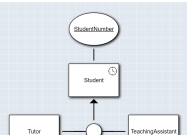
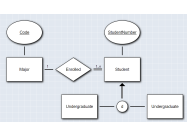
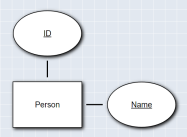
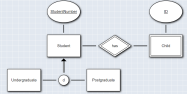
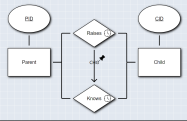
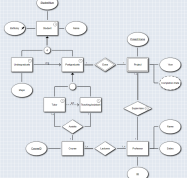


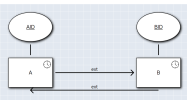
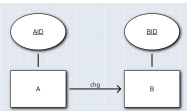
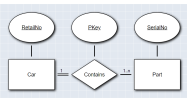
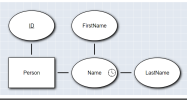
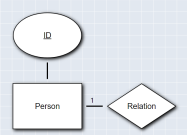
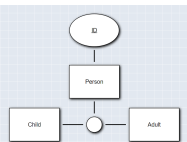
Figure 7: UML package diagram for software

**Table 3: Testing of model conversion**

Description	Input	Expected Outcome	Output	Result
One entity		Singular Table	Singular Table with correct primary key and attributes	PASS
One-to-one relationship		Two tables, total participant gets foreign key	Two tables, correct entity gets foreign key	PASS
One-to-many relationship		Two tables, table on 'one' side gets foreign key	Two tables, correct entity gets foreign key	PASS
Many-to-many relationship		New table created for relationship	Three tables, relationship table correct	PASS
Relationship between three entities		New table created for relationship	Four tables in total, relationship table correct	PASS
Relationship with itself		New table created for relationship, numbers added to names	New table created for relationship, duplicate names handled correctly	PASS
One-to-many relationship with attributes		Attributes should be added to foreign key holder	Attributes added to correct entity	PASS
Many-to-many relationship with attribute		Attributes should be added to table	Attributes added to relationship table	PASS
One-to-many relationship with temporal attribute		Temporal attribute should get own table, nothing should be added to foreign key holder	Temporal attribute gets own table and references both entities	PASS
One-to-many relationship with derived attribute		Attribute should not be added to foreign key holder	Same output as if derived attribute weren't there	PASS
Composite attributes		Base attribute should not be added, only the attributes connected to it	Only has the two attributes and primary key	PASS
Weak Relationship		Weak entity should have the both the identifier of parent and itself as primary key	Weak entity has both primary keys	PASS

Weak Relationship with attributes		Attributes should be added to table	Attributes added to weak entity just like a normal relationship	PASS
Inheritance with one entity		Child should get primary key of parent	Both tables have same primary key	PASS
Inheritance with two entities		Three tables, all with same primary key and children do not inherit attributes	Children inherit primary key but not attributes	PASS
Disjoint and Complete inheritance		Parent does not get table, children inherit primary key and attributes	Parent table disabled and children inherit both the primary key and attributes	PASS
Chain of inheritance relationships		Entity at the bottom of the chain should have the primary key of the entity at the top	Bottom entity has same primary key as top	PASS
Atemporal entities inheriting temporal entity		Child entities should become temporal	Children entities both became temporal	PASS
One-to-many relationship with disjoint and complete parent		Relationship should get own table	Relationship has own table and foreign keys correct	PASS
Multiple identifiers		Entity should have multiple primary keys	Both attributes are primary keys	PASS
Weak relationship with disjoint and complete parent		Relationship should get own table	Relationship has own table and correct primary keys	PASS
Temporal transition between relationships		Should compare both primary keys with each other	Triggers correctly made and compares primary keys correctly	PASS
Large Model, containing multiple interactions		Large model, all interactions should work together	Tables and Triggers correctly made	PASS



Two entities extending each other		Logically invalid. Should produce syntactically correct but unusable table	Tables created and loaded into DBMS, but unable to insert into either table	PASS
Temporal transition between atemporal entities		Logically invalid. Should not be created	Error given, and schema generated as if the transition did not exist	PASS
Identifier given to relationship		Logically invalid. No additional primary key should be created	Identifier treated as a regular attribute	PASS
Temporal attribute composed out of atemporal attributes		Logically invalid. Should not compose attributes	Composition attributes ignored and treated as if they are not there	PASS
Relationship attached to only one entity.		Logically invalid. Should give error	Error produced and schema generated as if relationship did not exist	PASS
Inheritance without subsumption edge		Logically invalid. Should give error	Error given and no table generated	PASS

**Table 4: Comparison between SQL implementation and TREND semantics. Assuming temporal transition from A to B, test cases listed in order of insertion.**

Transition	Semantic	Test Case	Output	Result
EXT, Optional extension in the future	$o \in \text{EXT}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t)} \wedge o \in C_2^{I(t+1)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
MEXT, Mandatory extension in the future	$o \in \text{MEXT}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists t' > t. o \in \text{EXT}_{C_1, C_2}^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
ext, Optional extension in the past	$o \in \text{ext}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t-1)} \wedge o \in C_2^{I(t)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
mext, Mandatory extension in the past	$o \in \text{mext}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists t' < t. o \in \text{EXT}_{C_1, C_2}^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: B must extend from A.	PASS
CHG, Optional change in future	$o \in \text{CHG}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t)} \wedge o \in C_2^{I(t+1)} \wedge o \notin C_1^{I(t+1)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A	PASS
MCHG, Mandatory change in future	$o \in \text{MCHG}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists t' > t. o \in \text{CHG}_{C_1, C_2}^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A	PASS
chg, Optional change in past	$o \in \text{chg}_{C_1, C_2}^{I(t)} \rightarrow (o \notin C_1^{I(t)} \wedge o \in C_2^{I(t)} \wedge o \notin C_2^{I(t-1)} \wedge o \in C_1^{I(t-1)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A	PASS

mchg, Mandatory change in past	$o \in \text{mchg}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists t' < t. o \in \text{CHG}_{C_1, C_2}^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must have been in A before.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A	PASS
PEXT, Persistent optional extension in the future	$o \in \text{PEXT}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t)} \wedge \forall t' > t. o \in C_2^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: A must always be in B after pinned transition.	PASS
PMEXT, Persistent mandatory extension in the future	$o \in \text{PMEXT}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists t' > t. o \in \text{PEXT}_{C_1, C_2}^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: A must always be in B after pinned transition.	PASS
pext, Persistent optional extension in the past	$o \in \text{pext}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t-1)} \wedge \forall t' \geq t. o \in C_2^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: Error: A must always be in B after pinned transition.	PASS
pmext, Persistent mandatory extension in the past	$o \in \text{pmext}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists t' < t. o \in \text{PEXT}_{C_1, C_2}^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: B must always extend from A.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2016⟩	Error: B must always extend from A.	PASS
PCHG, Persistent optional change in the future	$o \in \text{PCHG}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t)} \wedge \forall t' > t. (o \in C_2^{I(t')} \wedge o \notin C_1^{I(t')}))$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: A has already changed into B.	PASS

PMCHG, Persistent mandatory change in the future	$o \in \text{PMCHG}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists t' > t. o \in \text{PCHG}_{C_1, C_2}^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: A has already changed into B.	PASS
pchg, Persistent optional change in the past	$o \in \text{pchg}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t-1)} \wedge o \notin C_2^{I(t-1)} \wedge \forall t' \geq t. (o \in C_2^{I(t')} \wedge o \notin C_1^{I(t')}))$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: A has already changed into B.	PASS
pmchg, Persistent mandatory change in the past	$o \in \text{pmchg}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists t' < t. o \in \text{PCHG}_{C_1, C_2}^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must have been in A before.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: A has already changed into B.	PASS
For the following, the tests were done using time quanta of 2 years.				
QEXT, Optional quantitative extension in the future	$o \in \text{QEXT}_{C_1, C_2}^{I(t)} \rightarrow \exists (t+n) > t. (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t)} \wedge o \in C_2^{I(t+n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2011⟩ B:⟨a,2011,2014⟩	Error: B must have been in A for at least 2 units.	PASS
-	-	A:⟨a,2010,2013⟩ B:⟨a,2013,2014⟩	Insertion successful	PASS

-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must extend from A.	PASS
MQEXT, Mandatory quantitative extension in the future	$o \in \text{MQEXT}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists(t+n) > t. o \in \text{QEXT}_{C_1, C_2}^{I(t+n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2011⟩ B:⟨a,2011,2014⟩	Error: B must have been in A for at 2 units.	PASS
-	-	A:⟨a,2010,2013⟩ B:⟨a,2013,2014⟩	Error: No period in A can result in more than 2 continuous units without extending.	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must extend from A.	PASS
qext, Optional quantitative extension in the past	$o \in \text{qext}_{C_1, C_2}^{I(t)} \rightarrow \exists(t-n) < t. (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t-n)} \wedge o \in C_2^{I(t)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2011⟩ B:⟨a,2011,2014⟩	Error: B must have been in A for at least 2 units.	PASS
-	-	A:⟨a,2010,2013⟩ B:⟨a,2013,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must extend from A.	PASS
mqext, Mandatory quantitative extension in the past	$o \in \text{mqext}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists(t-n) < t. o \in \text{QEXT}_{C_1, C_2}^{I(t-n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2011⟩ B:⟨a,2011,2014⟩	Error: B must have been in A for 2 units.	PASS
-	-	A:⟨a,2010,2013⟩ B:⟨a,2013,2014⟩	Error: B must have been in A for 2 units.	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must extend from A.	PASS
QCHG, Optional quantitative change in future	$o \in \text{QCHG}_{C_1, C_2}^{I(t)} \rightarrow \exists(t+n) > t. (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t)} \wedge o \in C_2^{I(t+n)} \wedge o \notin C_1^{I(t+n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS


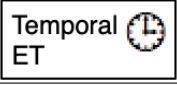



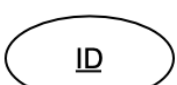
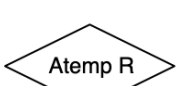
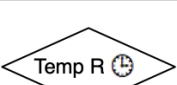




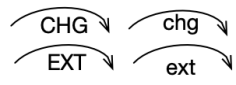
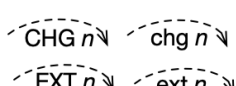
-	-	A:⟨a,2010,2011⟩ B:⟨a,2011,2014⟩	Error: B must have been in A for at least 2 units.	PASS
-	-	A:⟨a,2010,2013⟩ B:⟨a,2013,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must have been in A for at least 2 units.	PASS
MQCHG, Mandatory quantitative change in future	$o \in \text{MQCHG}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists(t+n) > t. o \in \text{QCHG}_{C_1, C_2}^{I(t+n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: No period in A can result in more than 2 continuous units.	PASS
-	-	A:⟨a,2010,2011⟩ B:⟨a,2011,2014⟩	Error: B must have been in A for 2 units.	PASS
-	-	A:⟨a,2010,2013⟩ B:⟨a,2013,2014⟩	Error: No period in A can result in more than 2 continuous units	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must have been in A for 2 units.	PASS
qchg, Optional quantitative change in past	$o \in \text{qchg}_{C_1, C_2}^{I(t)} \rightarrow \exists(t-n) < t. (o \notin C_1^{I(t)} \wedge o \in C_2^{I(t)} \wedge o \notin C_2^{I(t-n)} \wedge o \in C_1^{I(t-n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2010,2011⟩ B:⟨a,2011,2014⟩	Error: B must have been in A for at least 2 units.	PASS
-	-	A:⟨a,2010,2013⟩ B:⟨a,2013,2014⟩	Insertion successful	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must have been in A for at least 2 units.	PASS
mqchg, Mandatory quantitative change in past	$o \in \text{mqchg}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists(t-n) < t. o \in \text{QCHG}_{C_1, C_2}^{I(t-n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2014⟩	Error: A has already changed into B.	PASS
-	-	A:⟨a,2010,2011⟩ B:⟨a,2011,2014⟩	Error: B must have been in A for 2 units.	PASS
-	-	A:⟨a,2010,2013⟩ B:⟨a,2013,2014⟩	Error: B must have been in A for 2 units.	PASS
-	-	B:⟨a,2010,2012⟩ A:⟨a,2012,2014⟩	Error: B must have been in A for 2 units.	PASS

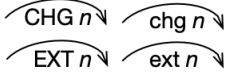

PQEXT, Persistent optional quantitative extension in the future	$o \in \text{PQEXT}_{C_1, C_2}^{I(t)} \rightarrow \exists(t+n) > t. (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t)} \wedge \forall t' > (t+n). o \in C_2^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B must have been in A for at least 2 units.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2016⟩	Error: A must always be in B after pinned transition.	PASS
PMQEXT, Persistent mandatory quantitative extension in the future	$o \in \text{PMQEXT}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists(t+n) > t. o \in \text{PQEXT}_{C_1, C_2}^{I(t+n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: Cannot insert into B as it has not been in A for 2 continuous units.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2016⟩	Error: No period in A can result in more than 2 continuous units without extending.	PASS
pqext, Persistent optional quantitative extension in the past	$o \in \text{pqext}_{C_1, C_2}^{I(t)} \rightarrow \exists(t-n) < t. (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t-n)} \wedge \forall t' \geq (t-n). o \in C_2^{I(t')})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion Successful	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	B must have been in A for at least 2 units.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2016⟩	Error: A must always be in B after pinned transition.	PASS
pmqext, Persistent mandatory quantitative extension in the past	$o \in \text{pmqext}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists(t-n) < t. o \in \text{PQEXT}_{C_1, C_2}^{I(t-n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion Successful.	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B must have been in A for 2 units.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2012,2016⟩	Error: A must always be in B after pinned transition.	PASS
PQCHG, Persistent optional quantitative change in the future	$o \in \text{PQCHG}_{C_1, C_2}^{I(t)} \rightarrow \exists(t+n) > t. (o \in C_1^{I(t)} \wedge o \notin C_2^{I(t)} \wedge \forall t' > (t+n). (o \in C_2^{I(t')} \wedge o \notin C_1^{I(t')}))$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: A has already changed into B.	PASS



PMQCHG, Persistent mandatory quantitative change in the future	$o \in \text{PMQCHG}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists(t+n) > t. o \in \text{PQCHG}_{C_1, C_2}^{I(t+n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: No period in A can result in more than 2 continuous units.	PASS
pqchg, Persistent optional quantitative change in the past	$o \in \text{pqchg}_{C_1, C_2}^{I(t)} \rightarrow \exists(t-n) < t. (o \in C_1^{I(t-n)} \wedge o \notin C_2^{I(t-n)} \wedge \forall t' \geq (t-n). (o \in C_2^{I(t')} \wedge o \notin C_1^{I(t')}))$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: A has already changed into B.	PASS
pmqchg, Persistent mandatory quantitative change in the past	$o \in \text{pmqchg}_{C_1, C_2}^{I(t)} \rightarrow (o \in C_1^{I(t)} \rightarrow \exists(t-n) < t. o \in \text{PQCHG}_{C_1, C_2}^{I(t-n)})$	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩	Insertion successful	PASS
-	-	A:⟨a,2012,2014⟩ B:⟨a,2012,2014⟩	Error: B cannot overlap with A.	PASS
-	-	A:⟨a,2010,2012⟩ B:⟨a,2012,2014⟩ A:⟨a,2014,2016⟩	Error: A has already changed into B.	PASS

**Table 5: Comparison of input serialisation and TREND's syntax**

Description	Element Icon	Textual Syntax	Input parameters
Atemporal entity		$c$	Node, type: "entity", all Booleans false
Temporal entity		$c^T$	Node, type: "entity", temporal: true, all other Booleans false
Atemporal attribute		$ATT$	Node, type: "attribute", all Booleans false
Temporal attribute		$ATT^T$	Node, type: "attribute", temporal: true, all other Booleans false
Frozen attribute		$FRZ$	Node, type: "attribute", pinned: true, all other Booleans false
Identifier attribute		$ID$	Node, type: "attribute", underlined: true, all other Booleans false
Atemporal relationship		$R$	Node, type: "relation", all Booleans false
Temporal relationship		$R^T$	Node, type: "relation", temporal: true, all other Booleans false
Subsumption		$ISAC, ISAR, ISAU$	Edge, parent: true, all other Booleans false
Cardinality	$0..n$ $1..n$ $n..m$	$CARD_C, CARD_R$	Edge, cardinality: "0..n", "1..n", "2..4", "1", etc.
Disjointness		$DISJ_C, DISJ_R$	Node, type: "isa", disjoint: true, all other Booleans false
Cover		$COVER$	Edge, double: true, all other Booleans false
Optional change or extension		$CHG, chg, EXT, ext, CHGR, chgR, EXTR, extR, CHGA$	Edge, curved: true, dashed: true, duration: 0, type: "chg", "CHG", "ext", or "EXT"
Mandatory change or extension		$MCHG, mchg, MEXT, mext, MCHGR, mchgR, MEXTR, mextR$	Edge, curved: true, dashed: false, duration: 0, type: "chg", "CHG", "ext", or "EXT"
Quantitative optional change or extension		$QCHG, qchg, QEXT, qext, QCHGR, qchgR, QEXTR, qextR, QCHGA$	Edge, curved: true, dashed: true, duration: n, type: "chg", "CHG", "ext", or "EXT"

Quantitative mandatory change or extension		MQCHG, <i>mqchg</i> , MQEXT, <i>mqext</i> , MQCHGR, <i>mqchgR</i> , MQEXTR, <i>mqextR</i> , MQCHGA	Edge, curved: true, dashed: false, duration: n, type: "chg", "CHG", "ext", or "EXT"
Persistent transitions		PCHG, PEXT, PCHGR, PEXTR	Edge, like previous examples but with pinned: true

## B TEMPORAL TRANSITIONS SQL CODE

The following Listings show the SQL code for the temporal transitions. Each transition goes from temporal element A with identifier AID to temporal element B with identifier BID. The quantitative temporal transitions were all made with using a value of 2 days. The blue text indicates where the variables from the model have been filled in. Note that the delimiter for triggers is //.

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5   DECLARE initial_exists INT;
6   SELECT COUNT(*)
7   INTO initial_exists
8   FROM A
9   WHERE NEW.BID = AID
10  AND (
11    (NEW.B_end > A_start AND NEW.B_start <= A_start)
12    OR
13    (NEW.B_end >= A_end AND NEW.B_start < A_end)
14    OR
15    (NEW.B_start >= A_start AND NEW.B_end <= A_end)
16  );
17  IF initial_exists > 0 THEN
18    SIGNAL SQLSTATE '45000'
19    SET MESSAGE_TEXT = 'B cannot overlap with A.';
20  END IF;
21 END;

```

Listing 8: chg

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5   DECLARE initial_exists INT;
6   DECLARE overlap_count INT;
7   SELECT COUNT(*)
8   INTO initial_exists
9   FROM A
10  WHERE NEW.BID = AID
11  AND NEW.B_start >= A_end;
12  SELECT COUNT(*)
13  INTO overlap_count
14  FROM A
15  WHERE NEW.BID = AID
16  AND (
17    (NEW.B_end > A_start AND NEW.B_start <= A_start)
18    OR
19    (NEW.B_end >= A_end AND NEW.B_start < A_end)
20    OR
21    (NEW.B_start >= A_start AND NEW.B_end <= A_end)
22  );
23  IF overlap_count > 0 THEN
24    SIGNAL SQLSTATE '45000'
25    SET MESSAGE_TEXT = 'B cannot overlap with A.';
26  END IF;
27  IF initial_exists = 0 THEN
28    SIGNAL SQLSTATE '45000'
29    SET MESSAGE_TEXT = 'B must have been in A before.';
30  END IF;
31 END;
32 //
33
34 CREATE TRIGGER A_chg_to_B_2
35 BEFORE INSERT ON A
36 FOR EACH ROW
37 BEGIN
38   DECLARE initial_exists INT;
39   SELECT COUNT(*)
40   INTO initial_exists

```

```

41 FROM B
42 WHERE NEW.AID = BID
43 AND (
44     (NEW.A_end > B_start AND NEW.A_start <= B_start)
45     OR
46     (NEW.A_end >= B_end AND NEW.A_start < B_end)
47     OR
48     (NEW.A_start >= B_start AND NEW.A_end <= B_end)
49 );
50 IF initial_exists > 0 THEN
51     SIGNAL SQLSTATE '45000'
52     SET MESSAGE_TEXT = 'A has already changed into B.';
53 END IF;
54 END;

```

**Listing 9: mchg**

```

1 CREATE TRIGGER A_CHG_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     SELECT COUNT(*)
7     INTO initial_exists
8     FROM A
9     WHERE NEW.BID = AID
10    AND (
11        (NEW.B_end > A_start AND NEW.B_start <= A_start)
12        OR
13        (NEW.B_end >= A_end AND NEW.B_start < A_end)
14        OR
15        (NEW.B_start >= A_start AND NEW.B_end <= A_end)
16    );
17 IF initial_exists > 0 THEN
18     SIGNAL SQLSTATE '45000'
19     SET MESSAGE_TEXT = 'B cannot overlap with A.';
20 END IF;
21 END;
22 //
23
24 CREATE TRIGGER A_CHG_to_B_2
25 BEFORE INSERT ON A
26 FOR EACH ROW
27 BEGIN
28     DECLARE initial_exists INT;
29     SELECT COUNT(*)
30     INTO initial_exists
31     FROM B
32     WHERE NEW.AID = BID
33    AND (
34        (NEW.A_end > B_start AND NEW.A_start <= B_start)
35        OR
36        (NEW.A_end >= B_end AND NEW.A_start < B_end)
37        OR
38        (NEW.A_start >= B_start AND NEW.A_end <= B_end)
39    );
40 IF initial_exists > 0 THEN
41     SIGNAL SQLSTATE '45000'
42     SET MESSAGE_TEXT = 'A has already changed into B.';
43 END IF;
44 END;

```

**Listing 10: chg**

```

1 CREATE TRIGGER A_CHG_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     SELECT COUNT(*)

```

```

7  INTO initial_exists
8  FROM A
9  WHERE NEW.BID = AID
10     AND (
11         (NEW.B_end > A_start AND NEW.B_start <= A_start)
12         OR
13         (NEW.B_end >= A_end AND NEW.B_start < A_end)
14         OR
15         (NEW.B_start >= A_start AND NEW.B_end <= A_end)
16     );
17  IF initial_exists > 0 THEN
18      SIGNAL SQLSTATE '45000'
19      SET MESSAGE_TEXT = 'B cannot overlap with A.';
20  END IF;
21 END;
22 //
23
24 CREATE TRIGGER A_CHG_to_B_2
25 BEFORE INSERT ON A
26 FOR EACH ROW
27 BEGIN
28     DECLARE initial_exists INT;
29     SELECT COUNT(*)
30     INTO initial_exists
31     FROM B
32     WHERE NEW.AID = BID
33         AND (
34             (NEW.A_end > B_start AND NEW.A_start <= B_start)
35             OR
36             (NEW.A_end >= B_end AND NEW.A_start < B_end)
37             OR
38             (NEW.A_start >= B_start AND NEW.A_end <= B_end)
39         );
40     IF initial_exists > 0 THEN
41         SIGNAL SQLSTATE '45000'
42         SET MESSAGE_TEXT = 'A has already changed into B.';
43     END IF;
44 END;

```

Listing 11: MCHG

```

1  CREATE TRIGGER A_ext_to_B
2  BEFORE INSERT ON B
3  FOR EACH ROW
4  BEGIN
5      DECLARE initial_exists INT;
6      SELECT COUNT(*)
7      INTO initial_exists
8      FROM A
9      WHERE NEW.BID = AID
10         AND A_start <= NEW.B_start;
11     IF initial_exists = 0 THEN
12         SIGNAL SQLSTATE '45000'
13         SET MESSAGE_TEXT = 'B must extend from A.';
14     END IF;
15 END;

```

Listing 12: mext

```

1  CREATE TRIGGER A_EXT_to_B_1
2  BEFORE INSERT ON B
3  FOR EACH ROW
4  BEGIN
5      DECLARE initial_exists INT;
6      DECLARE total_overlap INT DEFAULT 0;
7      DECLARE total_time INT DEFAULT 0;
8      DECLARE prev_end DATE DEFAULT NULL;
9      DECLARE cur_start DATE;
10     DECLARE cur_end DATE;
11     DECLARE done INT DEFAULT 0;

```

```

12 DECLARE cursor_a CURSOR FOR
13     SELECT A_start, A_end
14     FROM A
15     WHERE NEW.BID = AID
16     AND (A_end >= NEW.B_start)
17     ORDER BY A_start;
18 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
19
20 OPEN cursor_a;
21
22 read_loop: LOOP
23     FETCH cursor_a INTO cur_start, cur_end;
24     IF done THEN
25         LEAVE read_loop;
26     END IF;
27
28     IF prev_end IS NULL OR prev_end >= cur_start THEN
29         IF prev_end IS NOT NULL AND prev_end > cur_start THEN
30             SET cur_end = prev_end;
31         END IF;
32         IF cur_end >= NEW.B_start THEN
33             SET cur_end = NEW.B_start;
34         END IF;
35         SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
36         SET prev_end = cur_end;
37     ELSE
38         SET total_time = 0;
39         LEAVE read_loop;
40     END IF;
41 END LOOP;
42
43 CLOSE cursor_a;
44
45 SELECT COUNT(*)
46 INTO initial_exists
47 FROM A
48 WHERE NEW.BID = AID
49     AND (A_start <= NEW.B_start AND A_end >= NEW.B_end);
50 IF initial_exists = 0 THEN
51     SIGNAL SQLSTATE '45000'
52     SET MESSAGE_TEXT = 'B must always extend from A.';
53 END IF;
54 IF total_time < 2 THEN
55     SIGNAL SQLSTATE '45000'
56     SET MESSAGE_TEXT = 'B must have been in A for at least 2 units.';
57 END IF;
58 END;

```

Listing 13: QEXT

```

1 CREATE TRIGGER A_EXT_to_B_1
2 BEFORE INSERT ON A
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_match INT DEFAULT 0;
6     DECLARE total_time INT DEFAULT 0;
7     DECLARE prev_end DATE DEFAULT NULL;
8     DECLARE cur_start DATE;
9     DECLARE cur_end DATE;
10    DECLARE done INT DEFAULT 0;
11    DECLARE cursor_a CURSOR FOR
12        SELECT A_start, A_end
13        FROM A
14        WHERE NEW.BID = AID
15        ORDER BY A_start;
16    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
17
18    SET total_time = TIMESTAMPDIF(DAY, NEW.A_start, NEW.A_end);
19    IF total_time > 2 THEN

```



```

20     SIGNAL SQLSTATE '45000'
21     SET MESSAGE_TEXT = 'No period in A can result in more than 2 continuous units without extending.';
22 END IF;
23 SET total_time = 0;
24 OPEN cursor_a;
25
26 read_loop: LOOP
27     FETCH cursor_a INTO cur_start, cur_end;
28     IF done THEN
29         LEAVE read_loop;
30     END IF;
31
32     IF prev_end IS NULL OR prev_end = cur_start THEN
33         SET total_time = total_time + TIMESTAMPDIFF(DAY, cur_start, cur_end);
34         SET prev_end = cur_end;
35     ELSE
36         SET total_time = TIMESTAMPDIFF(DAY, cur_start, cur_end);
37         SET prev_end = cur_end;
38     END IF;
39     SELECT COUNT(*)
40     INTO total_match
41     FROM B
42     WHERE NEW.AID = BID
43     AND B_start = cur_end;
44     IF total_time >= 2 AND total_match != 0 THEN
45         LEAVE read_loop;
46     END IF;
47 END LOOP;
48
49 CLOSE cursor_a;
50 IF (total_time > 2 AND total_match = 0) THEN
51     SIGNAL SQLSTATE '45000'
52     SET MESSAGE_TEXT = 'No period in A can result in more than 2 continuous units without extending to B.';
53 END IF;
54 END;
55 //
56
57 CREATE TRIGGER A_EXT_to_B_2
58 BEFORE INSERT ON B
59 FOR EACH ROW
60 BEGIN
61     DECLARE total_other INT DEFAULT 0;
62     DECLARE total_time INT DEFAULT 0;
63     DECLARE prev_end DATE DEFAULT NULL;
64     DECLARE cur_start DATE;
65     DECLARE cur_end DATE;
66     DECLARE done INT DEFAULT 0;
67     DECLARE cursor_a CURSOR FOR
68         SELECT A_start, A_end
69         FROM A
70         WHERE NEW.BID = AID
71         ORDER BY A_start;
72     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
73
74     OPEN cursor_a;
75
76     read_loop: LOOP
77         FETCH cursor_a INTO cur_start, cur_end;
78         IF done THEN
79             LEAVE read_loop;
80         END IF;
81
82         IF prev_end IS NULL OR prev_end = cur_start THEN
83             SET total_time = total_time + TIMESTAMPDIFF(DAY, cur_start, cur_end);
84             SET prev_end = cur_end;
85         ELSE
86             SET total_time = TIMESTAMPDIFF(DAY, cur_start, cur_end);
87             SET prev_end = cur_end;
88         END IF;

```

```

89     IF total_time >= 2 AND prev_end = NEW.B_start THEN
90         LEAVE read_loop;
91     END IF;
92 END LOOP;
93
94 CLOSE cursor_a;
95 SELECT COUNT(*)
96 INTO total_other
97 FROM B
98 WHERE NEW.B_start = B_end;
99 IF (total_time < 2 OR prev_end != NEW.B_start) AND total_other = 0 THEN
100     SIGNAL SQLSTATE '45000'
101     SET MESSAGE_TEXT = 'Cannot insert into B as it has not been in A for 2 continuous units.';
102 END IF;
103 END;

```

Listing 14: MQEXT

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     DECLARE total_overlap INT DEFAULT 0;
7     DECLARE total_time INT DEFAULT 0;
8     DECLARE prev_end DATE DEFAULT NULL;
9     DECLARE cur_start DATE;
10    DECLARE cur_end DATE;
11    DECLARE done INT DEFAULT 0;
12    DECLARE cursor_a CURSOR FOR
13        SELECT A_start, A_end
14        FROM A
15        WHERE NEW.BID = AID
16        AND (A_end >= NEW.B_start)
17        ORDER BY A_start;
18    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
19
20    OPEN cursor_a;
21
22    read_loop: LOOP
23        FETCH cursor_a INTO cur_start, cur_end;
24        IF done THEN
25            LEAVE read_loop;
26        END IF;
27
28        IF prev_end IS NULL OR prev_end >= cur_start THEN
29            IF prev_end IS NOT NULL AND prev_end > cur_start THEN
30                SET cur_end = prev_end;
31            END IF;
32            IF cur_end >= NEW.B_start THEN
33                SET cur_end = NEW.B_start;
34            END IF;
35            SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
36            SET prev_end = cur_end;
37        ELSE
38            SET total_time = 0;
39            LEAVE read_loop;
40        END IF;
41    END LOOP;
42
43    CLOSE cursor_a;
44
45    SELECT COUNT(*)
46    INTO initial_exists
47    FROM A
48    WHERE NEW.BID = AID
49        AND (A_start <= NEW.B_start AND A_end >= NEW.B_end);
50    IF initial_exists = 0 THEN
51        SIGNAL SQLSTATE '45000'

```

```

52     SET MESSAGE_TEXT = 'B must always extend from A.';
53 END IF;
54 IF total_time < 2 THEN
55     SIGNAL SQLSTATE '45000'
56     SET MESSAGE_TEXT = 'B must have been in A for at least 2 units.';
57 END IF;
58 END;

```

### Listing 15: qext

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     DECLARE total_overlap INT DEFAULT 0;
7     DECLARE total_time INT DEFAULT 0;
8     DECLARE prev_end DATE DEFAULT NULL;
9     DECLARE cur_start DATE;
10    DECLARE cur_end DATE;
11    DECLARE done INT DEFAULT 0;
12    DECLARE cursor_a CURSOR FOR
13        SELECT A_start, A_end
14        FROM A
15        WHERE NEW.BID = AID
16        AND (A_end >= NEW.B_start)
17        ORDER BY A_start;
18    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
19
20    OPEN cursor_a;
21
22    read_loop: LOOP
23        FETCH cursor_a INTO cur_start, cur_end;
24        IF done THEN
25            LEAVE read_loop;
26        END IF;
27
28        IF prev_end IS NULL OR prev_end >= cur_start THEN
29            IF prev_end IS NOT NULL AND prev_end > cur_start THEN
30                SET cur_end = prev_end;
31            END IF;
32            IF cur_end >= NEW.B_start THEN
33                SET cur_end = NEW.B_start;
34            END IF;
35            SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
36            SET prev_end = cur_end;
37        ELSE
38            SET total_time = 0;
39            LEAVE read_loop;
40        END IF;
41    END LOOP;
42
43    CLOSE cursor_a;
44
45    SELECT COUNT(*)
46    INTO initial_exists
47    FROM A
48    WHERE NEW.BID = AID
49    AND (A_start <= NEW.B_start AND A_end >= NEW.B_end);
50    IF initial_exists = 0 THEN
51        SIGNAL SQLSTATE '45000'
52        SET MESSAGE_TEXT = 'B must always extend from A.';
53    END IF;
54    IF total_time <> 2 THEN
55        SIGNAL SQLSTATE '45000'
56        SET MESSAGE_TEXT = 'B must have been in A for 2 units.';
57    END IF;
58 END;

```

### Listing 16: mqext

```

1 CREATE TRIGGER A_CHG_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_overlap INT DEFAULT 0;
6     DECLARE total_time INT DEFAULT 0;
7     DECLARE prev_end DATE DEFAULT NULL;
8     DECLARE cur_start DATE;
9     DECLARE cur_end DATE;
10    DECLARE done INT DEFAULT 0;
11    DECLARE cursor_a CURSOR FOR
12        SELECT A_start, A_end
13        FROM A
14        WHERE NEW.BID = AID
15        AND A_end <= NEW.B_start
16        ORDER BY A_start DESC;
17    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
18
19    OPEN cursor_a;
20
21    read_loop: LOOP
22        FETCH cursor_a INTO cur_start, cur_end;
23        IF done THEN
24            LEAVE read_loop;
25        END IF;
26
27        IF cur_end = NEW.B_start OR prev_end = cur_end THEN
28            SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
29            SET prev_end = cur_start;
30        ELSE
31            LEAVE read_loop;
32        END IF;
33    END LOOP;
34
35    CLOSE cursor_a;
36
37    SELECT COUNT(*)
38    INTO total_overlap
39    FROM A WHERE NEW.BID = AID
40        AND (
41            (NEW.B_end > A_start AND NEW.B_start <= A_start)
42            OR
43            (NEW.B_end >= A_end AND NEW.B_start < A_end)
44            OR
45            (NEW.B_start >= A_start AND NEW.B_end <= A_end)
46        );
47    IF total_overlap > 0 THEN
48        SIGNAL SQLSTATE '45000'
49        SET MESSAGE_TEXT = 'B cannot overlap with A.';
50    END IF;
51    IF total_time < 2 THEN
52        SIGNAL SQLSTATE '45000'
53        SET MESSAGE_TEXT = 'B must have been in A for at least 2 units.';
54    END IF;
55 END;
56 //
57
58 CREATE TRIGGER A_CHG_to_B_3
59 BEFORE INSERT ON A
60 FOR EACH ROW
61 BEGIN
62     DECLARE initial_exists INT;
63     SELECT COUNT(*)
64     INTO initial_exists
65     FROM B
66     WHERE NEW.AID = BID
67     AND (
68         (NEW.A_end > B_start AND NEW.A_start <= B_start)
69         OR

```

```

70      (NEW.A_end >= B_end AND NEW.A_start < B_end)
71      OR
72      (NEW.A_start >= B_start AND NEW.A_end <= B_end)
73  );
74  IF initial_exists > 0 THEN
75      SIGNAL SQLSTATE '45000'
76      SET MESSAGE_TEXT = 'A has already changed into B.';
77  END IF;
78 END;

```

### Listing 17: QCHG

```

1 CREATE TRIGGER A_CHG_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_overlap INT DEFAULT 0;
6     DECLARE total_time INT DEFAULT 0;
7     DECLARE prev_end DATE DEFAULT NULL;
8     DECLARE cur_start DATE;
9     DECLARE cur_end DATE;
10    DECLARE done INT DEFAULT 0;
11    DECLARE cursor_a CURSOR FOR
12        SELECT A_start, A_end
13        FROM A
14        WHERE NEW.BID = AID
15        AND A_end <= NEW.B_start
16        ORDER BY A_start DESC;
17    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
18
19    OPEN cursor_a;
20
21    read_loop: LOOP
22        FETCH cursor_a INTO cur_start, cur_end;
23        IF done THEN
24            LEAVE read_loop;
25        END IF;
26
27        IF cur_end = NEW.B_start OR prev_end = cur_end THEN
28            SET total_time = total_time + TIMEDIFF(DAY, cur_start, cur_end);
29            SET prev_end = cur_start;
30        ELSE
31            LEAVE read_loop;
32        END IF;
33    END LOOP;
34
35    CLOSE cursor_a;
36
37    SELECT COUNT(*)
38    INTO total_overlap
39    FROM A WHERE NEW.BID = AID
40    AND (
41        (NEW.B_end > A_start AND NEW.B_start <= A_start)
42        OR
43        (NEW.B_end >= A_end AND NEW.B_start < A_end)
44        OR
45        (NEW.B_start >= A_start AND NEW.B_end <= A_end)
46    );
47    IF total_overlap > 0 THEN
48        SIGNAL SQLSTATE '45000'
49        SET MESSAGE_TEXT = 'B cannot overlap with A.';
50    END IF;
51    IF total_time <> 2 THEN
52        SIGNAL SQLSTATE '45000'
53        SET MESSAGE_TEXT = 'B must have been in A for 2 units.';
54    END IF;
55 END;
56 //
57

```

```

58 CREATE TRIGGER A_CHG_to_B_2
59 BEFORE INSERT ON A
60 FOR EACH ROW
61 BEGIN
62     DECLARE total_overlap INT DEFAULT 0;
63     DECLARE total_time INT DEFAULT 0;
64     DECLARE prev_end DATE DEFAULT NULL;
65     DECLARE cur_start DATE;
66     DECLARE cur_end DATE;
67     DECLARE done INT DEFAULT 0;
68     DECLARE cursor_a CURSOR FOR
69         SELECT A_start, A_end
70         FROM A
71         WHERE NEW.AID = AID
72         ORDER BY A_start;
73     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
74
75     SET total_time = TIMESTAMPDIFF(DAY, NEW.A_start, NEW.A_end);
76     IF total_time > 2 THEN
77         SIGNAL SQLSTATE '45000'
78         SET MESSAGE_TEXT = 'No period in A can result in more than 2 continuous units.';
79     END IF;
80     OPEN cursor_a;
81
82     SET total_time = 0;
83
84     read_loop: LOOP
85         FETCH cursor_a INTO cur_start, cur_end;
86         IF done THEN
87             LEAVE read_loop;
88         END IF;
89
90         IF prev_end IS NULL OR prev_end = cur_start THEN
91             SET total_time = total_time + TIMESTAMPDIFF(DAY, cur_start, cur_end);
92             SET prev_end = cur_end;
93         ELSE
94             SET total_time = TIMESTAMPDIFF(DAY, cur_start, cur_end);
95             SET prev_end = cur_end;
96         END IF;
97         IF cur_end = NEW.A_start THEN
98             SET total_time = total_time + TIMESTAMPDIFF(DAY, NEW.A_start, NEW.A_end);
99         END IF;
100        IF total_time > 2 THEN
101            SIGNAL SQLSTATE '45000'
102            SET MESSAGE_TEXT = 'No period in A can result in more than 2 continuous units.';
103        END IF;
104    END LOOP;
105
106    CLOSE cursor_a;
107END;
108//
109
110CREATE TRIGGER A_CHG_to_B_3
111BEFORE INSERT ON A
112FOR EACH ROW
113BEGIN
114    DECLARE initial_exists INT;
115    SELECT COUNT(*)
116    INTO initial_exists
117    FROM B
118    WHERE NEW.AID = BID
119        AND (
120            (NEW.A_end > B_start AND NEW.A_start <= B_start)
121            OR
122            (NEW.A_end >= B_end AND NEW.A_start < B_end)
123            OR
124            (NEW.A_start >= B_start AND NEW.A_end <= B_end)
125        );
126    IF initial_exists > 0 THEN

```

```

127     SIGNAL SQLSTATE '45000'
128     SET MESSAGE_TEXT = 'A has already changed into B.';
129 END IF;
130 END;

```

### Listing 18: MQCHG

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_overlap INT DEFAULT 0;
6     DECLARE total_time INT DEFAULT 0;
7     DECLARE prev_end DATE DEFAULT NULL;
8     DECLARE cur_start DATE;
9     DECLARE cur_end DATE;
10    DECLARE done INT DEFAULT 0;
11    DECLARE cursor_a CURSOR FOR
12        SELECT A_start, A_end
13        FROM A
14        WHERE NEW.BID = AID
15        AND A_end <= NEW.B_start
16        ORDER BY A_start DESC;
17    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
18
19    OPEN cursor_a;
20
21    read_loop: LOOP
22        FETCH cursor_a INTO cur_start, cur_end;
23        IF done THEN
24            LEAVE read_loop;
25        END IF;
26
27        IF cur_end = NEW.B_start OR prev_end = cur_end THEN
28            SET total_time = total_time + TIMEDIFF(DAY, cur_start, cur_end);
29            SET prev_end = cur_start;
30        ELSE
31            LEAVE read_loop;
32        END IF;
33    END LOOP;
34
35    CLOSE cursor_a;
36
37    SELECT COUNT(*)
38    INTO total_overlap
39    FROM A WHERE NEW.BID = AID
40        AND (
41            (NEW.B_end > A_start AND NEW.B_start <= A_start)
42            OR
43            (NEW.B_end >= A_end AND NEW.B_start < A_end)
44            OR
45            (NEW.B_start >= A_start AND NEW.B_end <= A_end)
46        );
47    IF total_overlap > 0 THEN
48        SIGNAL SQLSTATE '45000'
49        SET MESSAGE_TEXT = 'B cannot overlap with A.';
50    END IF;
51    IF total_time < 2 THEN
52        SIGNAL SQLSTATE '45000'
53        SET MESSAGE_TEXT = 'B must have been in A for at least 2 units.';
54    END IF;
55 END;
56 //
57
58 CREATE TRIGGER A_chg_to_B_2
59 BEFORE INSERT ON A
60 FOR EACH ROW
61 BEGIN
62     DECLARE initial_exists INT;

```



```

63  SELECT COUNT(*)
64  INTO initial_exists
65  FROM B
66  WHERE NEW.AID = BID
67  AND (
68    (NEW.A_end > B_start AND NEW.A_start <= B_start)
69    OR
70    (NEW.A_end >= B_end AND NEW.A_start < B_end)
71    OR
72    (NEW.A_start >= B_start AND NEW.A_end <= B_end)
73  );
74  IF initial_exists > 0 THEN
75    SIGNAL SQLSTATE '45000'
76    SET MESSAGE_TEXT = 'A has already changed into B.';
77  END IF;
78  END;

```

Listing 19: qchg

```

1  CREATE TRIGGER A_chg_to_B_1
2  BEFORE INSERT ON B
3  FOR EACH ROW
4  BEGIN
5    DECLARE total_overlap INT DEFAULT 0;
6    DECLARE total_time INT DEFAULT 0;
7    DECLARE prev_end DATE DEFAULT NULL;
8    DECLARE cur_start DATE;
9    DECLARE cur_end DATE;
10   DECLARE done INT DEFAULT 0;
11   DECLARE cursor_a CURSOR FOR
12     SELECT A_start, A_end
13     FROM A
14     WHERE NEW.BID = AID
15     AND A_end <= NEW.B_start
16     ORDER BY A_start DESC;
17   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
18
19   OPEN cursor_a;
20
21   read_loop: LOOP
22     FETCH cursor_a INTO cur_start, cur_end;
23     IF done THEN
24       LEAVE read_loop;
25     END IF;
26
27     IF cur_end = NEW.B_start OR prev_end = cur_end THEN
28       SET total_time = total_time + TIMEDIFF(DAY, cur_start, cur_end);
29       SET prev_end = cur_start;
30     ELSE
31       LEAVE read_loop;
32     END IF;
33   END LOOP;
34
35   CLOSE cursor_a;
36
37   SELECT COUNT(*)
38   INTO total_overlap
39   FROM A WHERE NEW.BID = AID
40   AND (
41     (NEW.B_end > A_start AND NEW.B_start <= A_start)
42     OR
43     (NEW.B_end >= A_end AND NEW.B_start < A_end)
44     OR
45     (NEW.B_start >= A_start AND NEW.B_end <= A_end)
46   );
47   IF total_overlap > 0 THEN
48     SIGNAL SQLSTATE '45000'
49     SET MESSAGE_TEXT = 'B cannot overlap with A.';
50   END IF;

```

```

51 IF total_time <> 2 THEN
52     SIGNAL SQLSTATE '45000'
53     SET MESSAGE_TEXT = 'B must have been in A for 2 units.';
54 END IF;
55 END;
56 //
57
58 CREATE TRIGGER A_chg_to_B_2
59 BEFORE INSERT ON A
60 FOR EACH ROW
61 BEGIN
62     DECLARE initial_exists INT;
63     SELECT COUNT(*)
64     INTO initial_exists
65     FROM B
66     WHERE NEW.AID = BID
67     AND (
68         (NEW.A_end > B_start AND NEW.A_start <= B_start)
69         OR
70         (NEW.A_end >= B_end AND NEW.A_start < B_end)
71         OR
72         (NEW.A_start >= B_start AND NEW.A_end <= B_end)
73     );
74 IF initial_exists > 0 THEN
75     SIGNAL SQLSTATE '45000'
76     SET MESSAGE_TEXT = 'A has already changed into B.';
77 END IF;
78 END;

```

Listing 20: mqchg

```

1 CREATE TRIGGER A_PEXT_to_B
2 BEFORE INSERT ON A
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     DECLARE existing INT;
7     SELECT COUNT(*)
8     INTO existing
9     FROM B
10    WHERE NEW.AID = BID
11    AND B_start <= NEW.A_start;
12    SELECT COUNT(*)
13    INTO initial_exists
14    FROM B
15    WHERE NEW.AID = BID
16    AND (B_start <= NEW.A_start AND B_end >= NEW.A_end);
17 IF initial_exists = 0 AND existing <> 0 THEN
18     SIGNAL SQLSTATE '45000'
19     SET MESSAGE_TEXT = 'A must always be in B after pinned transition.';
20 END IF;
21 END;

```

Listing 21: PEXT

```

1 CREATE TRIGGER A_PEXT_to_B
2 BEFORE INSERT ON A
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     DECLARE existing INT;
7     SELECT COUNT(*)
8     INTO existing
9     FROM B
10    WHERE NEW.AID = BID
11    AND B_start <= NEW.A_start;
12    SELECT COUNT(*)
13    INTO initial_exists
14    FROM B
15    WHERE NEW.AID = BID

```

```

16     AND (B_start <= NEW.A_start AND B_end >= NEW.A_end);
17 IF initial_exists = 0 AND existing <> 0 THEN
18     SIGNAL SQLSTATE '45000'
19     SET MESSAGE_TEXT = 'A must always be in B after pinned transition.';
20 END IF;
21 END;

```

**Listing 22: PMEXT**

```

1 CREATE TRIGGER A_pext_to_B
2 BEFORE INSERT ON A
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     DECLARE existing INT;
7     SELECT COUNT(*)
8     INTO existing
9     FROM B
10    WHERE NEW.AID = BID
11    AND B_start <= NEW.A_start;
12     SELECT COUNT(*)
13     INTO initial_exists
14     FROM B
15    WHERE NEW.AID = BID
16    AND (B_start <= NEW.A_start AND B_end >= NEW.A_end);
17 IF initial_exists = 0 AND existing <> 0 THEN
18     SIGNAL SQLSTATE '45000'
19     SET MESSAGE_TEXT = 'A must always be in B after pinned transition.';
20 END IF;
21 END;

```

**Listing 23: pext**

```

1 CREATE TRIGGER A_ext_to_B
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     SELECT COUNT(*)
7     INTO initial_exists
8     FROM A
9    WHERE NEW.BID = AID
10    AND A_start <= NEW.B_start;
11 IF initial_exists = 0 THEN
12     SIGNAL SQLSTATE '45000'
13     SET MESSAGE_TEXT = 'B must extend from A.';
14 END IF;
15 END;
16 //
17
18 CREATE TRIGGER A_pext_to_B
19 BEFORE INSERT ON A
20 FOR EACH ROW
21 BEGIN
22     DECLARE initial_exists INT;
23     DECLARE existing INT;
24     SELECT COUNT(*)
25     INTO existing
26     FROM B
27    WHERE NEW.AID = BID
28    AND B_start <= NEW.A_start;
29     SELECT COUNT(*)
30     INTO initial_exists
31     FROM B
32    WHERE NEW.AID = BID
33    AND (B_start <= NEW.A_start AND B_end >= NEW.A_end);
34 IF initial_exists = 0 AND existing <> 0 THEN
35     SIGNAL SQLSTATE '45000'
36     SET MESSAGE_TEXT = 'A must always be in B after pinned transition.';
37 END IF;

```

```
38 END;
```

#### Listing 24: pmext

```

1 CREATE TRIGGER A_CHG_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     SELECT COUNT(*)
7     INTO initial_exists
8     FROM A
9     WHERE NEW.BID = AID
10    AND (
11        (NEW.B_end > A_start AND NEW.B_start <= A_start)
12        OR
13        (NEW.B_end >= A_end AND NEW.B_start < A_end)
14        OR
15        (NEW.B_start >= A_start AND NEW.B_end <= A_end)
16    );
17    IF initial_exists > 0 THEN
18        SIGNAL SQLSTATE '45000'
19        SET MESSAGE_TEXT = 'B cannot overlap with A.';
20    END IF;
21 END;
22 //
23
24 CREATE TRIGGER A_CHG_to_B_2
25 BEFORE INSERT ON A
26 FOR EACH ROW
27 BEGIN
28     DECLARE initial_exists INT;
29     SELECT COUNT(*)
30     INTO initial_exists
31     FROM B
32     WHERE NEW.AID = BID
33     AND NEW.A_end > B_start;
34    IF initial_exists > 0 THEN
35        SIGNAL SQLSTATE '45000'
36        SET MESSAGE_TEXT = 'A has already changed into B.';
37    END IF;
38 END;
```

#### Listing 25: PCHG

```

1 CREATE TRIGGER A_CHG_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     SELECT COUNT(*)
7     INTO initial_exists
8     FROM A
9     WHERE NEW.BID = AID
10    AND (
11        (NEW.B_end > A_start AND NEW.B_start <= A_start)
12        OR
13        (NEW.B_end >= A_end AND NEW.B_start < A_end)
14        OR
15        (NEW.B_start >= A_start AND NEW.B_end <= A_end)
16    );
17    IF initial_exists > 0 THEN
18        SIGNAL SQLSTATE '45000'
19        SET MESSAGE_TEXT = 'B cannot overlap with A.';
20    END IF;
21 END;
22 //
23
24 CREATE TRIGGER A_CHG_to_B_2
25 BEFORE INSERT ON A
```

```

26 FOR EACH ROW
27 BEGIN
28     DECLARE initial_exists INT;
29     SELECT COUNT(*)
30     INTO initial_exists
31     FROM B
32     WHERE NEW.AID = BID
33           AND NEW.A_end > B_start;
34     IF initial_exists > 0 THEN
35         SIGNAL SQLSTATE '45000'
36         SET MESSAGE_TEXT = 'A has already changed into B.';
37     END IF;
38 END;

```

Listing 26: PMchg

```

1 CREATE TRIGGER A_pchg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     SELECT COUNT(*)
7     INTO initial_exists
8     FROM A
9     WHERE NEW.BID = AID
10           AND NEW.B_start < A_end;
11     IF initial_exists > 0 THEN
12         SIGNAL SQLSTATE '45000'
13         SET MESSAGE_TEXT = 'B cannot overlap with A.';
14     END IF;
15 END;
16 //
17
18 CREATE TRIGGER A_pchg_to_B_2
19 BEFORE INSERT ON A
20 FOR EACH ROW
21 BEGIN
22     DECLARE initial_exists INT;
23     SELECT COUNT(*)
24     INTO initial_exists
25     FROM B
26     WHERE NEW.AID = BID
27           AND NEW.A_end > B_start;
28     IF initial_exists > 0 THEN
29         SIGNAL SQLSTATE '45000'
30         SET MESSAGE_TEXT = 'A has already changed into B.';
31     END IF;
32 END;

```

Listing 27: pchg

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     DECLARE overlap_count INT;
7     SELECT COUNT(*)
8     INTO initial_exists
9     FROM A
10    WHERE NEW.BID = AID
11           AND NEW.B_start >= A_end;
12     SELECT COUNT(*)
13     INTO overlap_count
14     FROM A
15    WHERE NEW.BID = AID
16           AND (
17         (NEW.B_end > A_start AND NEW.B_start <= A_start)
18         OR
19         (NEW.B_end >= A_end AND NEW.B_start < A_end)

```

```

20      OR
21      (NEW.B_start >= A_start AND NEW.B_end <= A_end)
22  );
23  IF overlap_count > 0 THEN
24      SIGNAL SQLSTATE '45000'
25      SET MESSAGE_TEXT = 'B cannot overlap with A.';
26  END IF;
27  IF initial_exists = 0 THEN
28      SIGNAL SQLSTATE '45000'
29      SET MESSAGE_TEXT = 'B must have been in A before.';
30  END IF;
31  END;
32  //
33
34  CREATE TRIGGER A_chg_to_B_2
35  BEFORE INSERT ON A
36  FOR EACH ROW
37  BEGIN
38      DECLARE initial_exists INT;
39      SELECT COUNT(*)
40      INTO initial_exists
41      FROM B
42      WHERE NEW.AID = BID
43             AND NEW.A_end > B_start;
44      IF initial_exists > 0 THEN
45          SIGNAL SQLSTATE '45000'
46          SET MESSAGE_TEXT = 'A has already changed into B.';
47      END IF;
48  END;

```

Listing 28: pmchg

```

1  CREATE TRIGGER A_EXT_to_B_1
2  BEFORE INSERT ON B
3  FOR EACH ROW
4  BEGIN
5      DECLARE initial_exists INT;
6      DECLARE total_overlap INT DEFAULT 0;
7      DECLARE total_time INT DEFAULT 0;
8      DECLARE prev_end DATE DEFAULT NULL;
9      DECLARE cur_start DATE;
10     DECLARE cur_end DATE;
11     DECLARE done INT DEFAULT 0;
12     DECLARE cursor_a CURSOR FOR
13         SELECT A_start, A_end
14         FROM A
15         WHERE NEW.BID = AID
16             AND (A_end >= NEW.B_start)
17             ORDER BY A_start;
18     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
19
20     OPEN cursor_a;
21
22     read_loop: LOOP
23         FETCH cursor_a INTO cur_start, cur_end;
24         IF done THEN
25             LEAVE read_loop;
26         END IF;
27
28         IF prev_end IS NULL OR prev_end >= cur_start THEN
29             IF prev_end IS NOT NULL AND prev_end > cur_start THEN
30                 SET cur_end = prev_end;
31             END IF;
32             IF cur_end >= NEW.B_start THEN
33                 SET cur_end = NEW.B_start;
34             END IF;
35             SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
36             SET prev_end = cur_end;
37         ELSE

```

```

38     SET total_time = 0;
39     LEAVE read_loop;
40 END IF;
41 END LOOP;
42
43 CLOSE cursor_a;
44
45 SELECT COUNT(*)
46 INTO initial_exists
47 FROM A
48 WHERE NEW.BID = AID
49       AND A_start <= NEW.B_start;
50 IF initial_exists = 0 THEN
51     SIGNAL SQLSTATE '45000'
52     SET MESSAGE_TEXT = 'B must extend from A.';
53 END IF;
54 IF total_time < 2 THEN
55     SIGNAL SQLSTATE '45000'
56     SET MESSAGE_TEXT = 'B must have been in A for at least 2 units.';
57 END IF;
58 END;
59 //
60
61 CREATE TRIGGER A_PEXT_to_B
62 BEFORE INSERT ON A
63 FOR EACH ROW
64 BEGIN
65     DECLARE initial_exists INT;
66     DECLARE existing INT;
67     SELECT COUNT(*)
68     INTO existing
69     FROM B
70     WHERE NEW.AID = BID
71           AND B_start <= NEW.A_start;
72     SELECT COUNT(*)
73     INTO initial_exists
74     FROM B
75     WHERE NEW.AID = BID
76           AND (B_start <= NEW.A_start AND B_end >= NEW.A_end);
77 IF initial_exists = 0 AND existing <> 0 THEN
78     SIGNAL SQLSTATE '45000'
79     SET MESSAGE_TEXT = 'A must always be in B after pinned transition.';
80 END IF;
81 END;
82 //

```

### Listing 29: PQEXT

```

1 CREATE TRIGGER A_EXT_to_B_1
2 BEFORE INSERT ON A
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_match INT DEFAULT 0;
6     DECLARE total_time INT DEFAULT 0;
7     DECLARE prev_end DATE DEFAULT NULL;
8     DECLARE cur_start DATE;
9     DECLARE cur_end DATE;
10    DECLARE done INT DEFAULT 0;
11    DECLARE cursor_a CURSOR FOR
12        SELECT A_start, A_end
13        FROM A
14        WHERE NEW.AID = AID
15        ORDER BY A_start;
16    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
17
18    SET total_time = TIMESTAMPDIF(DAY, NEW.A_start, NEW.A_end);
19    IF total_time > 2 THEN
20        SIGNAL SQLSTATE '45000'
21        SET MESSAGE_TEXT = 'No period in A can result in more than 2 continuous units without extending.';

```

```

22  END IF;
23  SET total_time = 0;
24  OPEN cursor_a;
25
26  read_loop: LOOP
27    FETCH cursor_a INTO cur_start, cur_end;
28    IF done THEN
29      LEAVE read_loop;
30    END IF;
31
32    IF prev_end IS NULL OR prev_end = cur_start THEN
33      SET total_time = total_time + TIMESTAMPDIFF(DAY, cur_start, cur_end);
34      SET prev_end = cur_end;
35    ELSE
36      SET total_time = TIMESTAMPDIFF(DAY, cur_start, cur_end);
37      SET prev_end = cur_end;
38    END IF;
39    SELECT COUNT(*)
40    INTO total_match
41    FROM B
42    WHERE NEW.AID = BID
43    AND B_start = cur_end;
44    IF total_time >= 2 AND total_match != 0 THEN
45      LEAVE read_loop;
46    END IF;
47  END LOOP;
48
49  CLOSE cursor_a;
50  IF (total_time > 2 AND total_match = 0) THEN
51    SIGNAL SQLSTATE '45000'
52    SET MESSAGE_TEXT = 'No period in A can result in more than 2 continuous units without extending to B.';
53  END IF;
54 END;
55 //
56
57 CREATE TRIGGER A_EXT_to_B_2
58 BEFORE INSERT ON B
59 FOR EACH ROW
60 BEGIN
61   DECLARE total_other INT DEFAULT 0;
62   DECLARE total_time INT DEFAULT 0;
63   DECLARE prev_end DATE DEFAULT NULL;
64   DECLARE cur_start DATE;
65   DECLARE cur_end DATE;
66   DECLARE done INT DEFAULT 0;
67   DECLARE cursor_a CURSOR FOR
68     SELECT A_start, A_end
69     FROM A
70     WHERE NEW.BID = AID
71     ORDER BY A_start;
72   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
73
74   OPEN cursor_a;
75
76   read_loop: LOOP
77     FETCH cursor_a INTO cur_start, cur_end;
78     IF done THEN
79       LEAVE read_loop;
80     END IF;
81
82     IF prev_end IS NULL OR prev_end = cur_start THEN
83       SET total_time = total_time + TIMESTAMPDIFF(DAY, cur_start, cur_end);
84       SET prev_end = cur_end;
85     ELSE
86       SET total_time = TIMESTAMPDIFF(DAY, cur_start, cur_end);
87       SET prev_end = cur_end;
88     END IF;
89     IF total_time >= 2 AND prev_end = NEW.B_start THEN
90       LEAVE read_loop;

```



```

91     END IF;
92 END LOOP;
93
94 CLOSE cursor_a;
95 SELECT COUNT(*)
96 INTO total_other
97 FROM B
98 WHERE NEW.B_start = B_end;
99 IF (total_time < 2 OR prev_end != NEW.B_start) AND total_other = 0 THEN
100     SIGNAL SQLSTATE '45000'
101     SET MESSAGE_TEXT = 'Cannot insert into B as it has not been in A for 2 continuous units.';
102 END IF;
103 END;
104 //
105
106 CREATE TRIGGER A_PEXT_to_B
107 BEFORE INSERT ON A
108 FOR EACH ROW
109 BEGIN
110     DECLARE initial_exists INT;
111     DECLARE existing INT;
112     SELECT COUNT(*)
113     INTO existing
114     FROM B
115     WHERE NEW.AID = BID
116           AND B_start <= NEW.A_start;
117     SELECT COUNT(*)
118     INTO initial_exists
119     FROM B
120     WHERE NEW.AID = BID
121           AND (B_start <= NEW.A_start AND B_end >= NEW.A_end);
122     IF initial_exists = 0 AND existing <> 0 THEN
123         SIGNAL SQLSTATE '45000'
124         SET MESSAGE_TEXT = 'A must always be in B after pinned transition.';
125     END IF;
126 END;

```

Listing 30: PMQEXT

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     DECLARE total_overlap INT DEFAULT 0;
7     DECLARE total_time INT DEFAULT 0;
8     DECLARE prev_end DATE DEFAULT NULL;
9     DECLARE cur_start DATE;
10    DECLARE cur_end DATE;
11    DECLARE done INT DEFAULT 0;
12    DECLARE cursor_a CURSOR FOR
13        SELECT A_start, A_end
14        FROM A
15        WHERE NEW.BID = AID
16              AND (A_end >= NEW.B_start)
17              ORDER BY A_start;
18    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
19
20    OPEN cursor_a;
21
22    read_loop: LOOP
23        FETCH cursor_a INTO cur_start, cur_end;
24        IF done THEN
25            LEAVE read_loop;
26        END IF;
27
28        IF prev_end IS NULL OR prev_end >= cur_start THEN
29            IF prev_end IS NOT NULL AND prev_end > cur_start THEN
30                SET cur_end = prev_end;

```

```

31     END IF;
32     IF cur_end >= NEW.B_start THEN
33         SET cur_end = NEW.B_start;
34     END IF;
35     SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
36     SET prev_end = cur_end;
37 ELSE
38     SET total_time = 0;
39     LEAVE read_loop;
40 END IF;
41 END LOOP;
42
43 CLOSE cursor_a;
44
45 SELECT COUNT(*)
46 INTO initial_exists
47 FROM A
48 WHERE NEW.BID = AID
49 AND A_start <= NEW.B_start;
50 IF initial_exists = 0 THEN
51     SIGNAL SQLSTATE '45000'
52     SET MESSAGE_TEXT = 'B must extend from A.';
53 END IF;
54 IF total_time < 2 THEN
55     SIGNAL SQLSTATE '45000'
56     SET MESSAGE_TEXT = 'B must have been in A for at least 2 units.';
57 END IF;
58 END;
59 //
60
61 CREATE TRIGGER A_pext_to_B
62 BEFORE INSERT ON A
63 FOR EACH ROW
64 BEGIN
65     DECLARE initial_exists INT;
66     DECLARE existing INT;
67     SELECT COUNT(*)
68     INTO existing
69     FROM B
70     WHERE NEW.AID = BID
71     AND B_start <= NEW.A_start;
72     SELECT COUNT(*)
73     INTO initial_exists
74     FROM B
75     WHERE NEW.AID = BID
76     AND (B_start <= NEW.A_start AND B_end >= NEW.A_end);
77     IF initial_exists = 0 AND existing <> 0 THEN
78         SIGNAL SQLSTATE '45000'
79         SET MESSAGE_TEXT = 'A must always be in B after pinned transition.';
80     END IF;
81 END;

```

Listing 31: pqext

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE initial_exists INT;
6     DECLARE total_overlap INT DEFAULT 0;
7     DECLARE total_time INT DEFAULT 0;
8     DECLARE prev_end DATE DEFAULT NULL;
9     DECLARE cur_start DATE;
10    DECLARE cur_end DATE;
11    DECLARE done INT DEFAULT 0;
12    DECLARE cursor_a CURSOR FOR
13        SELECT A_start, A_end
14        FROM A
15        WHERE NEW.BID = AID

```

```

16     AND (A_end >= NEW.B_start)
17     ORDER BY A_start;
18 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
19
20 OPEN cursor_a;
21
22 read_loop: LOOP
23     FETCH cursor_a INTO cur_start, cur_end;
24     IF done THEN
25         LEAVE read_loop;
26     END IF;
27
28     IF prev_end IS NULL OR prev_end >= cur_start THEN
29         IF prev_end IS NOT NULL AND prev_end > cur_start THEN
30             SET cur_end = prev_end;
31         END IF;
32         IF cur_end >= NEW.B_start THEN
33             SET cur_end = NEW.B_start;
34         END IF;
35         SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
36         SET prev_end = cur_end;
37     ELSE
38         SET total_time = 0;
39         LEAVE read_loop;
40     END IF;
41 END LOOP;
42
43 CLOSE cursor_a;
44
45 SELECT COUNT(*)
46 INTO initial_exists
47 FROM A
48 WHERE NEW.BID = AID
49     AND A_start <= NEW.B_start;
50 IF initial_exists = 0 THEN
51     SIGNAL SQLSTATE '45000'
52     SET MESSAGE_TEXT = 'B must extend from A.';
53 END IF;
54 IF total_time <> 2 THEN
55     SIGNAL SQLSTATE '45000'
56     SET MESSAGE_TEXT = 'B must have been in A for 2 units.';
57 END IF;
58 END;
59 //
60
61 CREATE TRIGGER A_pext_to_B
62 BEFORE INSERT ON A
63 FOR EACH ROW
64 BEGIN
65     DECLARE initial_exists INT;
66     DECLARE existing INT;
67     SELECT COUNT(*)
68     INTO existing
69     FROM B
70     WHERE NEW.AID = BID
71         AND B_start <= NEW.A_start;
72     SELECT COUNT(*)
73     INTO initial_exists
74     FROM B
75     WHERE NEW.AID = BID
76         AND (B_start <= NEW.A_start AND B_end >= NEW.A_end);
77     IF initial_exists = 0 AND existing <> 0 THEN
78         SIGNAL SQLSTATE '45000'
79         SET MESSAGE_TEXT = 'A must always be in B after pinned transition.';
80     END IF;
81 END;

```

Listing 32: pmqext

```

1 CREATE TRIGGER A_CHG_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_overlap INT DEFAULT 0;
6     DECLARE total_time INT DEFAULT 0;
7     DECLARE prev_end DATE DEFAULT NULL;
8     DECLARE cur_start DATE;
9     DECLARE cur_end DATE;
10    DECLARE done INT DEFAULT 0;
11    DECLARE cursor_a CURSOR FOR
12        SELECT A_start, A_end
13        FROM A
14        WHERE NEW.BID = AID
15        AND A_end <= NEW.B_start
16        ORDER BY A_start DESC;
17    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
18
19    OPEN cursor_a;
20
21    read_loop: LOOP
22        FETCH cursor_a INTO cur_start, cur_end;
23        IF done THEN
24            LEAVE read_loop;
25        END IF;
26
27        IF cur_end = NEW.B_start OR prev_end = cur_end THEN
28            SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
29            SET prev_end = cur_start;
30        ELSE
31            LEAVE read_loop;
32        END IF;
33    END LOOP;
34
35    CLOSE cursor_a;
36
37    SELECT COUNT(*)
38    INTO total_overlap
39    FROM A WHERE NEW.BID = AID
40    AND (
41        (NEW.B_end > A_start AND NEW.B_start <= A_start)
42        OR
43        (NEW.B_end >= A_end AND NEW.B_start < A_end)
44        OR
45        (NEW.B_start >= A_start AND NEW.B_end <= A_end)
46    );
47    IF total_overlap > 0 THEN
48        SIGNAL SQLSTATE '45000'
49        SET MESSAGE_TEXT = 'B cannot overlap with A.';
50    END IF;
51    IF total_time < 2 THEN
52        SIGNAL SQLSTATE '45000'
53        SET MESSAGE_TEXT = 'B must have been in A for at least 2 units.';
54    END IF;
55 END;
56 //
57
58 CREATE TRIGGER A_CHG_to_B_3
59 BEFORE INSERT ON A
60 FOR EACH ROW
61 BEGIN
62     DECLARE initial_exists INT;
63     SELECT COUNT(*)
64     INTO initial_exists
65     FROM B
66     WHERE NEW.AID = BID
67     AND NEW.A_end > B_start;
68     IF initial_exists > 0 THEN
69         SIGNAL SQLSTATE '45000'

```

```

70     SET MESSAGE_TEXT = 'A has already changed into B.';
71 END IF;
72 END;

```

## Listing 33: PQCHG

```

1 CREATE TRIGGER A_CHG_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_overlap INT DEFAULT 0;
6     DECLARE total_time INT DEFAULT 0;
7     DECLARE prev_end DATE DEFAULT NULL;
8     DECLARE cur_start DATE;
9     DECLARE cur_end DATE;
10    DECLARE done INT DEFAULT 0;
11    DECLARE cursor_a CURSOR FOR
12        SELECT A_start, A_end
13        FROM A
14        WHERE NEW.BID = AID
15        AND A_end <= NEW.B_start
16        ORDER BY A_start DESC;
17    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
18
19    OPEN cursor_a;
20
21    read_loop: LOOP
22        FETCH cursor_a INTO cur_start, cur_end;
23        IF done THEN
24            LEAVE read_loop;
25        END IF;
26
27        IF cur_end = NEW.B_start OR prev_end = cur_end THEN
28            SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
29            SET prev_end = cur_start;
30        ELSE
31            LEAVE read_loop;
32        END IF;
33    END LOOP;
34
35    CLOSE cursor_a;
36
37    SELECT COUNT(*)
38    INTO total_overlap
39    FROM A WHERE NEW.BID = AID
40        AND (
41            (NEW.B_end > A_start AND NEW.B_start <= A_start)
42            OR
43            (NEW.B_end >= A_end AND NEW.B_start < A_end)
44            OR
45            (NEW.B_start >= A_start AND NEW.B_end <= A_end)
46        );
47    IF total_overlap > 0 THEN
48        SIGNAL SQLSTATE '45000'
49        SET MESSAGE_TEXT = 'B cannot overlap with A.';
50    END IF;
51    IF total_time <> 2 THEN
52        SIGNAL SQLSTATE '45000'
53        SET MESSAGE_TEXT = 'B must have been in A for 2 units.';
54    END IF;
55 END;
56 //
57
58 CREATE TRIGGER A_CHG_to_B_2
59 BEFORE INSERT ON A
60 FOR EACH ROW
61 BEGIN
62     DECLARE total_overlap INT DEFAULT 0;
63     DECLARE total_time INT DEFAULT 0;

```

```

64 DECLARE prev_end DATE DEFAULT NULL;
65 DECLARE cur_start DATE;
66 DECLARE cur_end DATE;
67 DECLARE done INT DEFAULT 0;
68 DECLARE cursor_a CURSOR FOR
69     SELECT A_start, A_end
70     FROM A
71     WHERE NEW.AID = AID
72     ORDER BY A_start;
73 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
74
75 SET total_time = TIMESTAMPDIFF(DAY, NEW.A_start, NEW.A_end);
76 IF total_time > 2 THEN
77     SIGNAL SQLSTATE '45000'
78     SET MESSAGE_TEXT = 'No period in A can result in more than 2 continuous units.';
79 END IF;
80 OPEN cursor_a;
81
82     SET total_time = 0;
83
84 read_loop: LOOP
85     FETCH cursor_a INTO cur_start, cur_end;
86     IF done THEN
87         LEAVE read_loop;
88     END IF;
89
90     IF prev_end IS NULL OR prev_end = cur_start THEN
91         SET total_time = total_time + TIMESTAMPDIFF(DAY, cur_start, cur_end);
92         SET prev_end = cur_end;
93     ELSE
94         SET total_time = TIMESTAMPDIFF(DAY, cur_start, cur_end);
95         SET prev_end = cur_end;
96     END IF;
97     IF cur_end = NEW.A_start THEN
98         SET total_time = total_time + TIMESTAMPDIFF(DAY, NEW.A_start, NEW.A_end);
99     END IF;
100 IF total_time > 2 THEN
101     SIGNAL SQLSTATE '45000'
102     SET MESSAGE_TEXT = 'No period in A can result in more than 2 continuous units.';
103 END IF;
104 END LOOP;
105
106 CLOSE cursor_a;
107 END;
108 //
109
110 CREATE TRIGGER A_CHG_to_B_3
111 BEFORE INSERT ON A
112 FOR EACH ROW
113 BEGIN
114     DECLARE initial_exists INT;
115     SELECT COUNT(*)
116     INTO initial_exists
117     FROM B
118     WHERE NEW.AID = BID
119           AND NEW.A_end > B_start;
120 IF initial_exists > 0 THEN
121     SIGNAL SQLSTATE '45000'
122     SET MESSAGE_TEXT = 'A has already changed into B.';
123 END IF;
124 END;

```

Listing 34: PMQCHG

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_overlap INT DEFAULT 0;

```

```

6  DECLARE total_time INT DEFAULT 0;
7  DECLARE prev_end DATE DEFAULT NULL;
8  DECLARE cur_start DATE;
9  DECLARE cur_end DATE;
10 DECLARE done INT DEFAULT 0;
11 DECLARE cursor_a CURSOR FOR
12     SELECT A_start, A_end
13     FROM A
14     WHERE NEW.BID = AID
15     AND A_end <= NEW.B_start
16     ORDER BY A_start DESC;
17 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
18
19 OPEN cursor_a;
20
21 read_loop: LOOP
22     FETCH cursor_a INTO cur_start, cur_end;
23     IF done THEN
24         LEAVE read_loop;
25     END IF;
26
27     IF cur_end = NEW.B_start OR prev_end = cur_end THEN
28         SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
29         SET prev_end = cur_start;
30     ELSE
31         LEAVE read_loop;
32     END IF;
33 END LOOP;
34
35 CLOSE cursor_a;
36
37 SELECT COUNT(*)
38 INTO total_overlap
39 FROM A WHERE NEW.BID = AID
40 AND (
41     (NEW.B_end > A_start AND NEW.B_start <= A_start)
42     OR
43     (NEW.B_end >= A_end AND NEW.B_start < A_end)
44     OR
45     (NEW.B_start >= A_start AND NEW.B_end <= A_end)
46 );
47 IF total_overlap > 0 THEN
48     SIGNAL SQLSTATE '45000'
49     SET MESSAGE_TEXT = 'B cannot overlap with A.';
50 END IF;
51 IF total_time < 2 THEN
52     SIGNAL SQLSTATE '45000'
53     SET MESSAGE_TEXT = 'B must have been in A for at least 2 units.';
54 END IF;
55 END;
56 //
57
58 CREATE TRIGGER A_chg_to_B_2
59 BEFORE INSERT ON A
60 FOR EACH ROW
61 BEGIN
62     DECLARE initial_exists INT;
63     SELECT COUNT(*)
64     INTO initial_exists
65     FROM B
66     WHERE NEW.AID = BID
67     AND NEW.A_end > B_start;
68     IF initial_exists > 0 THEN
69         SIGNAL SQLSTATE '45000'
70         SET MESSAGE_TEXT = 'A has already changed into B.';
71     END IF;
72 END;

```

Listing 35: pqchg

```

1 CREATE TRIGGER A_chg_to_B_1
2 BEFORE INSERT ON B
3 FOR EACH ROW
4 BEGIN
5     DECLARE total_overlap INT DEFAULT 0;
6     DECLARE total_time INT DEFAULT 0;
7     DECLARE prev_end DATE DEFAULT NULL;
8     DECLARE cur_start DATE;
9     DECLARE cur_end DATE;
10    DECLARE done INT DEFAULT 0;
11    DECLARE cursor_a CURSOR FOR
12        SELECT A_start, A_end
13        FROM A
14        WHERE NEW.BID = AID
15        AND A_end <= NEW.B_start
16        ORDER BY A_start DESC;
17    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
18
19    OPEN cursor_a;
20
21    read_loop: LOOP
22        FETCH cursor_a INTO cur_start, cur_end;
23        IF done THEN
24            LEAVE read_loop;
25        END IF;
26
27        IF cur_end = NEW.B_start OR prev_end = cur_end THEN
28            SET total_time = total_time + TIMESTAMPDIF(DAY, cur_start, cur_end);
29            SET prev_end = cur_start;
30        ELSE
31            LEAVE read_loop;
32        END IF;
33    END LOOP;
34
35    CLOSE cursor_a;
36
37    SELECT COUNT(*)
38    INTO total_overlap
39    FROM A WHERE NEW.BID = AID
40        AND (
41            (NEW.B_end > A_start AND NEW.B_start <= A_start)
42            OR
43            (NEW.B_end >= A_end AND NEW.B_start < A_end)
44            OR
45            (NEW.B_start >= A_start AND NEW.B_end <= A_end)
46        );
47    IF total_overlap > 0 THEN
48        SIGNAL SQLSTATE '45000'
49        SET MESSAGE_TEXT = 'B cannot overlap with A.';
50    END IF;
51    IF total_time <> 2 THEN
52        SIGNAL SQLSTATE '45000'
53        SET MESSAGE_TEXT = 'B must have been in A for 2 units.';
54    END IF;
55 END;
56 //
57
58 CREATE TRIGGER A_chg_to_B_2
59 BEFORE INSERT ON A
60 FOR EACH ROW
61 BEGIN
62     DECLARE initial_exists INT;
63     SELECT COUNT(*)
64     INTO initial_exists
65     FROM B
66     WHERE NEW.AID = BID
67         AND NEW.A_end > B_start;
68     IF initial_exists > 0 THEN
69         SIGNAL SQLSTATE '45000'

```



```
70      SET MESSAGE_TEXT = 'A has already changed into B.';  
71  END IF;  
72 END;
```

**Listing 36: pmqchg**