

Temporal Databases for Mere Mortals

Dimitrios Souflis
dsouflis@acm.org



Pandamator (ancient Greek for *all-subduing*, an adjective commonly attached to time), is an open-source project aiming to enable handling of temporal databases using common RDBMSs lacking temporal support. It consists of:

- Predetermined decisions as to what kinds of temporal data can be accommodated and their representation in SQL
- Definitions of tables, views, functions and stored procedures that together provide a supporting infrastructure for defining and using temporal tables
- An SQL generator that generates triggers for maintaining temporal referential integrity and other procedural code
- Run-time support to create temporal UPDATE and DELETE scripts from within user programs
- A middleware that translates temporal SQL to regular SQL (unfinished)

Pandamator does not require a thorough understanding of temporal databases, but its benefits are not automatic either. Using a temporal database, even with the help of Pandamator, is an order of magnitude more difficult than using a non-temporal one. As such, it should be a conscious decision. However, the reason I created Pandamator, is that it is often a decision that has already been taken by necessity, since temporal data are almost ubiquitous. In that case, it is better to do it the correct way rather than improvise.

In its current incarnation, Pandamator works with Microsoft SQL Server.

Contents

Pandamator Cheat Sheet	9
Intended Audience	10
Temporal Notions	11
What are temporal data	11
Dimensions of time	11
Time-varying data	11
Timestamped data	12
User-defined time	13
Explicit and implicit data	13
Now and Forever	14
Snapshot at a specific instant	14
Query at each instant (sequenced query)	15
Query across time (non-sequenced query)	15
Coalescing	16
Integrity	16
Primary Key and Unique	16
Foreign Key	17
On Delete Cascade	17
On Delete Set Null	17
On Delete Restrict	18
Constant	18
Contiguous History	18
Specialized Temporal Relations	19
Temporal SQL (ATSQL)	20
Temporal Type of a Relation	20
Temporal Statement Modifiers	21
Inheritance of Temporal Statement Modifiers	22
Temporal DDL	24
Adding Validtime Support	24
Refreshing the Validtime Support	24
Declaring the Temporal Type of a Table	24
Undeclaring a Temporal Table	24
Declaring a Validtime Constraint	24
Undeclaring a Validtime Constraint	24

QueryExpress, the Pandamator IDE	25
Translating Temporal SQL	28
Translating Temporal Queries into SQL92.....	28
Implementing Coalescing in SQL92.....	29
Translating Sequenced Queries into SQL92.....	30
Translating Temporal Deletes into SQL92	31
Translating Temporal Updates into SQL92.....	33
Useful Non-Sequenced Idioms	37
Why Sequenced Queries Are Not Enough.....	37
Event Statemachines	37
Status Transitions.....	40
Fold Queries.....	40
Valid-time Partitioning	41
Event Succession Queries	41
Branching Time (not yet incorporated).....	43
Introduction.....	43
Timeline Segments.....	43
Timelines.....	43
Representation.....	44
Virtual Database per Timeline	45
SQL Infrastructure	46
Temporal Metadata	46
temporal_metadata.tables.....	46
temporal_metadata.table_constraints.....	46
temporal_metadata.constraint_columns.....	46
temporal_metadata.referential_constraints	46
Pseudo-DDL	47
TemporalDeclareTable	47
TemporalUndeclareTable	47
TemporalDeclarePrimaryKey	47
TemporalDeclareUnique.....	47
TemporalDeclareConstant	48
TemporalDeclareContiguousHistory	48
TemporalDeclareForeignKey	48
TemporalUndeclareConstraint.....	48

Scheduled entities	48
SQL Generation	51
Things to do beforehand	51
Primary key support	51
Coalescing	51
Contiguous History	52
Constant	52
Unique	52
Foreign Key	52
Checking the integrity	53
Delete in the presence of foreign keys	53
Update	53
Metadata management	53
Cleaning up	53
Run-time support	53
ReadMetadata	54
CreateSQLDeleteCascading	54
CreateSQLUpdateNonKey	55
CreateCoalescingCTEs	55
Using Temporal SQL (experimental)	56
Description of test data and test execution	57
Sample “A, B, C” data	57
Calling the deletion procedure inside a transaction	58
RESTRICT from D to C and from G to B	59
CASCADE to B, C and E	59
SET NULL on H and on F	59
References	61
Lorentzos	61
Jensen Thesis	61
Snodgrass: Developing Time-Oriented Database Applications in SQL	61
Andreas Steiner Thesis	61
TimeDB	61
Michael Boehlen Thesis	61
Consensus Glossary of Temporal Database Concepts – February 1998 Version	61
SIMILE Timeline	61

Teradata Temporal Option.....	61
Appendix A: Query Transformation Example (Not Coalesced).....	62
Appendix B: Query Transformation Example (Coalesced).....	63

Figures

Figure 1: Bitemporal Annotation Schematic	11
Figure 2: Snapshots s1, s2, s3 etc.	14
Figure 3: Primary Key Violation example.....	17
Figure 4: Foreign key violation example.....	17
Figure 5: On Delete Cascade	17
Figure 6: On Delete Set Null	18
Figure 7: On Delete Restrict	18
Figure 8: Contiguous History violation example.....	19
Figure 9 The QueryExpress connection dialog.....	25
Figure 10 The database selection control.....	25
Figure 11 An active "Add ValidTime" button.....	25
Figure 12 An inactive "Add ValidTime" button.....	26
Figure 13 The object browser	26
Figure 14 Command execution.....	26
Figure 15 Result tabs	26
Figure 16 Data grid	27
Figure 17 Timeline.....	27
Figure 9: Sample Query Tree for Translating Sequenced Queries	31
Figure 10: Query Subtrees for Translating Sequenced Queries.....	31
Figure 11: Timeline Segment Branching.....	43
Figure 12: Timelines	43
Figure 13: Entity scheduled using a Schedule	50
Figure 14: Non-temporal Entity scheduled using a Schedule.....	50
Figure 15: FK relationships of Sample "A, B, C" date.....	57
Figure 16: Chart of "A, B, C" Sample Data.....	58
Figure 17: Cascade in "A, B, C" Sample Data	59
Figure 18: Set Null in "A, B, C" Sample Data (states)	60
Figure 19: Set Null in "A, B, C" Sample Data (events).....	60

Tables

Table 1: Temporal Type of a Pairwise Join.....	20
Table 2: Temporal Statement Modifier effect on SELECT.....	21
Table 3: Temporal Statement Modifier effect on DELETE/UPDATE.....	21
Table 4: Temporal Statement Modifier effect on INSERT.....	22
Table 5: Inherited Temporal Statement Modifier	23

Data Tables

Data Table 1: Example of a state relation	12
Data Table 2: Example of an event relation	13
Data Table 3: Snapshot of the example state relation	14
Data Table 4: Example of a sequenced query	15
Data Table 5: Example of a sequenced subquery of a non-sequenced query.....	15
Data Table 6: Event Relation to apply statemachine to	39
Data Table 7: Event Statemachine results.....	39

Pandamator Cheat Sheet

Using Pandamator consists of the following steps:

1. Prepare the database

Run AddTemporalSupport.sql to define the metadata tables and other supporting SQL (cf. Adding Validtime Support).

2. Prepare your schema

Add the appropriate timestamp columns to your temporal-to-be tables (cf. Time-varying data, Timestamped data).

3. Declare temporal annotations on your schema

Declare the temporal type of tables and all necessary integrity constraints (cf. Declaring a Validtime Constraint).

4. Generate SQL for integrity constraints

Generate and run the necessary SQL to create views, procedures and triggers necessary to enforce the integrity constraints (cf. Refreshing the Validtime Support).

5. Create temporal-aware SQL in your code

Run-time support for temporal DML is provided in two ways: specialized calls for code generation of DELETE and UPDATE (cf. CreateSQLDeleteCascading, CreateSQLUpdateNonKey), and code transformation from temporal SQL (ATSQL) to SQL92 (cf. Using Temporal SQL (experimental)).

Intended Audience

This text was written to accompany Pandamator, an open-source project that helps programmers work with temporal databases in the absence of such support from the RDBMSs. Because working with temporal data is not usually taught and not usually practiced, this text presents all relevant notions without presupposing anything more than knowledge of SQL. It was written with the professional programmer in mind.

The field of temporal database research is vast and old, but has not found its way into the mainstream yet. This is unfortunate, as we are using temporal data here and now, and doing so in an ad-hoc and mostly wrong manner. You won't find raw science in this text, although I present links to free information you can start your own research from, and academic sources can provide you with many more. This text contains distilled knowledge only, and often I do not sidetrack in order to present alternative views or solutions, in the interest of parsimony. So, if you are a scientist in this domain, you will probably not find anything original in this text, but it might still be of interest as the practitioner's view on the subject.

Temporal Notions

What are temporal data

Temporal data are often characterized as data that vary with time (*state relations*, as you will see below), for example when we need to track all modifications of the attributes of an entity, or when we need to plan accommodations for hotel rooms in advance. I take a slightly broader view than that, encompassing also simple timestamped data (*event relations*, as you will see below), usually downplayed in the field of temporal databases (for example, SQL2 and ATSQL do not mention them at all). Temporal data exist, that are neither state nor event relations, but they can always be transformed to one or the other in order to become amenable to proper temporal treatment.

Plain relations, like the ones in conventional databases, are called *snapshot relations*.

Dimensions of time

There are at least two dimensions of time that are usually employed in temporal databases, usually called *valid time* and *transaction time*. The valid time of some information marks when that information is supposed to hold in real life. In that sense, valid time in the past signifies the recorded past and valid time in the future signifies information that does not actually exist yet, but we know it is scheduled to. The transaction time of some information marks when this information was included in the database. As such, it records the history of modifications of a database. Blending the two dimensions, we have *bitemporal databases* that record the history of modifications of a database that records time-varying information.

These two dimensions of time are regarded as orthogonal, which they are, but there is a subtle difference between them. A piece of valid-time temporal information can be regarded as the annotation of some piece of non-temporal data with the corresponding valid time. The valid time characterizes the information. A piece of transaction-time temporal information, on the other hand, is some piece of information (temporal or not), annotated with a transaction time. In the case of bi-temporal data, the transaction time annotates the valid time, not the other way around.

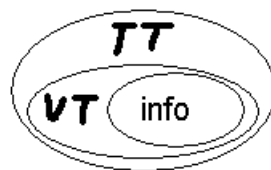


Figure 1: Bitemporal Annotation Schematic

Pandamator currently supports only the valid time dimension.

Time-varying data

Data varying with time will be described by rows annotated with a time period, represented in SQL by two datetime columns, namely *ValidFromDate* and *ValidToDate*. Together, they record a *Closed-Open* interval in the valid time dimension, meaning that the associated row is valid at any time between

ValidFromDate inclusive and ValidToDate exclusive. Closed-open intervals can be chained back-to-back to cover longer periods. To be frank, using closed-closed intervals would allow a uniform representation that could be used for both state and event data, but chaining closed-closed intervals would necessitate dealing with the granularity of time in an RDBMS and how to form such intervals. This is a topic best avoided since the established RDBMSs have wildly ad-hoc behavior in that domain¹.

Data sets of time-varying data will be called *state sets* or *state relations*, as they record distinct states of the world at different instants of time. For reasons of parsimony, state relations that are persisted in tables or returned to the user (as opposed to intermediate results of operations) will have the extra property of being *coalesced*, meaning that adjacent time spans are merged when they contain the same attribute values. Being coalesced or not does not alter the meaning of a state relation, though.

Individual rows in a state relation are called *temporal items*, and identifiers of the modeled objects are called *object surrogates*. The set of all temporal items describing the lifetime of a modeled object is called a *life-line* or a *time-sequence* (in which case a whole relation is called a *time-sequence collection*).

Let's look at a first minimal example of a state relation.

Data Table 1: Example of a state relation

id (PK)	val	ValidFromDate	ValidToDate
1	1	2008-01-01	2008-01-10
1	2	2008-01-10	2008-01-20
1	1	2008-02-01	2008-02-10
2	1	2008-01-15	2008-02-25

This relation describes the lifetime of two entities (as defined by the “id” primary key). The entity with id=1 had a “val” attribute of value 1 from January 1st to January 10th (exclusive) and from February 1st to February 10th (exclusive), and a “val” attribute of value 2 from January 10th to January 20th (exclusive). The entity with id=2 had a “val” attribute of value 1 from January 15th to February 25th (exclusive).

We are always operating under a “closed-world assumption”, so the entity with id=1 is supposed to not exist at all before January 1st, during January 20th and February 1st, and after February 10th. The entity with id=2 is supposed to not exist at all before January 15th and after February 25th.

Timestamped data

Timestamped data will be described by rows annotated with a timestamp, represented in SQL by a datetime column named ValidOnDate. Data sets of timestamped data will be called *event sets* or *event relations*, as they record distinct events in time.

¹ For example, Microsoft SQL Server offers a granularity of 1/300th of a second and the smallest increments / decrements it allows are milliseconds but rounded approximately to 3 ms (1/300th of a second). Oracle offers a configurable fractional second granularity of up to 1 nano-second and allows arithmetic using the INTERVAL datatype with fractional seconds.

Like I said, event relations are often not given any attention. I believe this is wrong, and these are my reasons. For one thing, such data are very common right now in all application domains and the need they fulfill will not go away after the adoption of temporal databases. They represent discrete events of the modeled reality, observations, readings from sensors and other similar data. In addition to their inherent value, they are useful even when the temporal data we intend to use are state relations, because many useful queries are naturally formulated in terms of event relations. For example, “At what instants did something happen?” (a transition between two states, the beginning or ending of something), “How many times did something happen?”.

Event relations can be used to represent transitions between states, and state relations can be used to represent what holds between events. In this manner, event relations and state relations are complementary views of reality and switching between these views can be very useful. I call the operation of producing event timestamps from a state relation, “demarcation”, and the converse operation of producing indivisible spans from an event relation, “spanning”.

Let’s look at a first minimal example of an event relation.

Data Table 2: Example of an event relation

id (PK)	val	ValidOnDate
1	1	2008-01-01
2	2	2008-01-10
3	2	2008-02-01
4	1	2008-02-15

This relation describes four events that happened on the dates January 1st, January 10th, February 1st and February 15th.

Although event relations are represented differently than state relations, Pandamator operates on them as if they were state relations with ValidFromDate=ValidOnDate and ValidToDate=ValidOnDate + dt, where dt is an arbitrarily short duration that never needs to be specified explicitly, as it is factored out of all computations. It does not correspond to the granularity of the database, it’s just supposed to be small enough so that for every time T that could appear in the data, there is no time T’ that could appear in the data, so that $T \leq T' \leq T+dt$. So, event relations are used as state relations that can only be valid at a single moment, of those moments that can be represented in our database.

User-defined time

Columns in a temporal table may also be of a temporal type. They are treated as any other value would be, without any special treatment. In order to make evident that these values do not interact with the concept of time dimensions, they are thought of as user-defined time.

Explicit and implicit data

I also make the distinction between explicit and implicit data. Explicit data are data actually stored in SQL tables containing the annotation columns just described. Implicit data, on the other hand, are data that are described indirectly but are,

nevertheless, most useful to us when transcribed to state or event relations. Such data in Pandamator are periodically scheduled data and data described by state transition rules on event relations, to be described later.

Now and Forever

The last row of a history needs to have some specific value for its ValidToDate, which we might intend to be 'now', 'forever', 'don't know', 'until changed' or NULL i.e. no value at all. In order not to digress in this discussion, the value I use is January 1st, 3000, which is sufficiently distant not to be mistaken for an actual input value and can be represented by virtually all RDBMSs (cf. Snodgrass: Developing Time-Oriented Database Applications in SQL). But practically any distant value will do, so long as the interval it forms with a real one will make sense.

Snapshot at a specific instant

A temporal database can be thought of as describing a multitude of non-temporal databases, one for each moment in time. Projecting a temporal database to the time dimension produces what is called a *snapshot*, which is the state of the described universe of discourse at the specific moment. The result is a non-temporal relation.

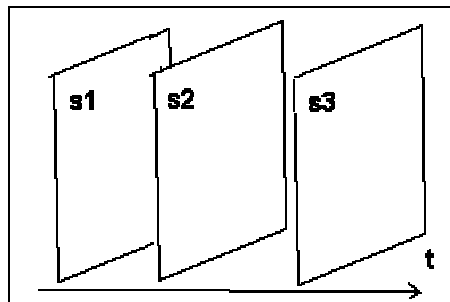


Figure 2: Snapshots s1, s2, s3 etc.

Producing a snapshot is very simple with the representation I have described, as it suffices to adorn every WHERE clause of every subquery with the condition $p.\text{ValidFromDate} \leq T$ and $T < p.\text{ValidToDate}$, where T is the moment in question, for every table p that appears in the subquery. The simplicity of this operation, sadly, is not shared with any other operation in the field of temporal support.

Given the state relation example from a previous section, its snapshot on February 5th is the following snapshot relation.

Data Table 3: Snapshot of the example state relation

id (PK)	val
1	1
2	1

Viewing a temporal database as shorthand for describing snapshot databases at various time instants allows one to structure temporal data correctly. For example, a relation with columns for a validity period and a column to hold the total energy consumption in kWh during that period does not make sense as a state relation, because projecting at any given instant does not produce anything meaningful. A state relation with a column to hold current energy consumption in kW, on the other hand, is correctly designed because its snapshot equivalent means something at any instant

in time. Such “semi-conformant” relations can be transformed into proper state relations in order to benefit from all the tools available for proper temporal data.

Query at each instant (sequenced query)

Generalizing a query to produce results at each instant in time is quite difficult, but it is central to using a temporal database. These kinds of queries are referred to as “sequenced queries” in most of the bibliography, and you’ll have to look it up if you want to know why on earth they are named that way. All it matters is that sequenced queries are state relations composed of the equivalent snapshots, which means that there is no interaction whatsoever of different instants among themselves. Sequenced queries do not and can not refer to the timestamping columns.

An example of a sequenced query is calculating COUNT(*) on the state relation from a previous section. The resulting state relation follows.

Data Table 4: Example of a sequenced query

COUNT(*)	ValidFromDate	ValidToDate
1	2008-01-01	2008-01-15
2	2008-01-15	2008-01-20
1	2008-01-20	2008-02-01
2	2008-02-01	2008-02-10
1	2008-02-10	2008-02-25

Query across time (non-sequenced query)

The third kind of queries in a temporal database is queries across all time. These queries do not enjoy any kind of support from Pandamator (or any other temporal SQL) but I have tried to tackle some common kinds of such queries in the sequel, and hopefully provide useful tips.

Many useful non-sequenced queries can be expressed on top of sequenced queries. For example, using the state relation from a previous section, finding the longest amount of time two or more entities had the same “val” attribute values. This can be done in two steps.

First, calculate `VAL GROUP BY VAL HAVING COUNT(*) > 1` as a sequenced query (rows eliminated by HAVING are shown for clarity, but struck through).

Data Table 5: Example of a sequenced subquery of a non-sequenced query

val	COUNT(*)	ValidFromDate	ValidToDate
1	1	2008-01-01	2008-01-10
2	1	2008-01-10	2008-01-20
1	1	2008-01-15	2008-02-01
1	2	2008-02-01	2008-02-10
1	1	2008-02-10	2008-02-25

The remaining task is easy to compute given the resulting state relation and making explicit use of the timestamping columns.

This partitioning of non-sequenced queries into sequenced subqueries and non-sequenced outer queries means that, even though there will be no inherent support for non-sequenced queries in Pandamator, its upcoming support of sequenced queries will benefit this area as well.

Coalescing

Coalescing, in the context of temporal data, means to merge adjacent timestamped rows having identical column values, in order to represent a temporal relation in the fewest number of timestamped rows possible. There is but a single such representation for each relation, so a coalesced representation can be considered a *normal form*.

Whether two values are identical or not, is a notion that is wider than the three-valued equality of SQL. In essence:

- When both values are non-null, they are identical if they are equal.
- When both values are null, they are identical, as well.
- When one is null and the other is not, they are not identical.

Integrity

One of the big hurdles when working with temporal databases is maintaining integrity. Integrity comes for free when using plain SQL in a plain RDBMS, but expressing it for temporal data is especially challenging. You will see that the amount and the complexity of the SQL code needed virtually ensures that, even when people annotate data with intervals and effortlessly use snapshot queries, do not go the extra step to add it. Combined with the difficulty to modify temporal data correctly, the result is damaged data.

The integrity constraints currently offered by Pandamator are explained subquently. Candidate constraints for inclusion in the future are:

- Foreign keys on Unique constraints, not just on the Primary Key
- Time-invariant constraints on specific columns

Primary Key and Unique

There cannot be any instant where two or more rows have the same values for the *primary key* columns. A primary key for event relations can be described in plain SQL, but state relations demand a complex trigger.

Pandamator also supports *unique* constraints, which have similar semantics but whose columns are nullable. Currently, foreign keys can only reference primary key columns, not columns of unique constraints.

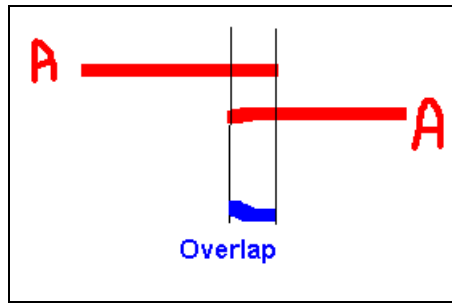


Figure 3: Primary Key Violation example

(In this and the following such diagrams, time increases to the right.)

There are two entities (named “A”) with the same values for their primary key, overlapping during the period shown in blue.

Foreign Key

There cannot be any instant where a row in the referencing table does not correspond to a row in the referenced table.

Pandamator does not support other *unique* constraints, so a foreign key can only refer to a primary key.

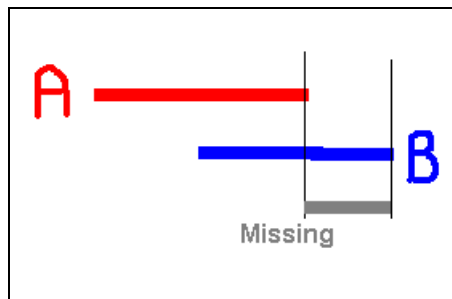


Figure 4: Foreign key violation example

Entity B has a foreign key towards entity A, but entity A does not exist during the period shown in grey.

Foreign Key is characterized by a *delete rule*, prescribing what happens during a DELETE. I don’t encourage updates to primary keys, consequently Pandamator does not support *update rules* on Foreign Keys.

On Delete Cascade

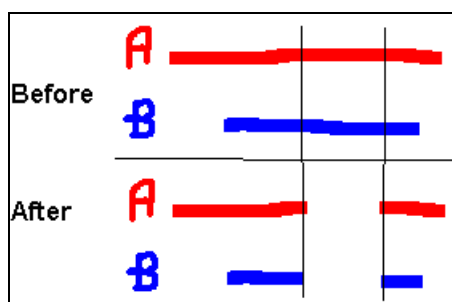


Figure 5: On Delete Cascade

Deletion of A cascades to B for the deletion period, breaking B up as needed.

On Delete Set Null

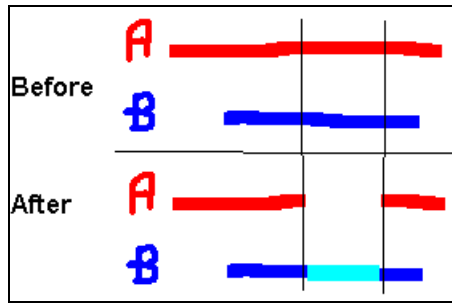


Figure 6: On Delete Set Null

Deletion of A sets the foreign key on B to NULL for the deletion period, breaking B up as needed. In the example, a period of B shown in blue, is broken up into three periods, the middle one shown in light blue having NULL as foreign key.

On Delete Restrict

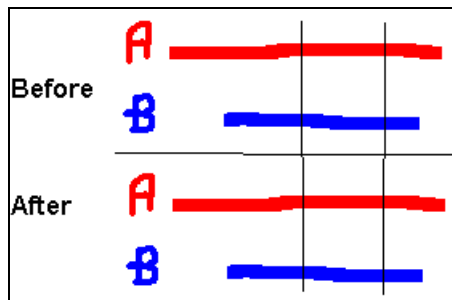


Figure 7: On Delete Restrict

Deletion of A is inhibited if a foreign key exists to it at any time during the deletion period.

Constant

Columns defined as constant are immutable over time. This constraint, like that of *contiguous history*, has no counterpart in plain SQL, and is a *non-sequenced constraint*.

Contiguous History

This is just a handy constraint to ensure that a state relation does not have gaps in its history. In other words, between two instants when an entity does exist with a specific primary key, there cannot be any instant when that entity does not exist. This constraint, like *constant*, has no counterpart in plain SQL, and is a *non-sequenced constraint*. A relation that has a contiguous history and whose columns are all constant is coalesced to a single database row.

The state relation used as an example in a previous section is not contiguous, as the entity with id=1 has a gap from January 20th to February 1st.



Entity A has a gap in its history, during the period shown in grey.

Figure 8: Contiguous History violation example

Specialized Temporal Relations

Although Pandamator does not handle transaction time, when a relation is used in such a manner that transaction time interacts in a specific manner with valid time, it is called a *specialized temporal relation*, and one can find a corresponding taxonomy in the Jensen Thesis. According to that taxonomy, a relation whose transaction time coincides with its valid time, it is called a *degenerate* relation. If one just adds valid time to an existing application, using snapshot operations at time ‘now’ throughout, the database will consist of degenerate relations. The valid time of such a relation, effectively stores the history of modifications.

Temporal SQL (ATSQL)

The dialect of temporal SQL understood by Pandamator is based on ATSQL. Differences from ATSQL stem mainly from the fact that ATSQL was supposed to be an extension of SQL3, while Pandamator ATSQL is based on SQL2 and makes no effort to simulate the existence of the PERIOD datatype. Another difference is that ATSQL did not support INSERT that mentions a whole query as the source, did not support UPDATE and DELETE that mention joined tables, nor did it support common table subexpressions. Also, ATSQL did not support event relations and did not offer a way to perform a time-slice operation on anything other than the current instant². However, for reasons of continuity I will refer to the temporal dialect of Pandamator as ATSQL, since someone familiar with the published version of ATSQL will find it instantly recognizable. Whenever there is a notable difference from ATSQL, I will clearly signal it.

In the current version (1.9.4), support of ATSQL is partial, has not been tested extensively but seems stable enough. In order to permit a seamless user experience, the user can delimit any SQL statements which should be passed uninterpreted to the RDBMS with BEGIN NONSEQUENCED and END NONSEQUENCED.

Temporal Type of a Relation

Temporal SQL statements mean different things depending on the temporal type of the relation they apply to and the specific SQL statement. I will call that relation, the “unmodified relation”, a term of my own invention. The term “temporal type”, also of my own invention, does not have a concise name in ATSQL and is roughly referred to as “whether the table has temporal support”, since ATSQL did not extend beyond state relations.

The following table can be used to determine the temporal type of a pairwise join. In order to determine the temporal type of a multiple join in a query, one has to process all joins pairwise. The type of join (cross, inner, etc) is irrelevant, and Pandamator does not even interpret the logic behind each type of join, since it transforms temporal SQL into regular SQL that uses the same types of joins. Note that the presence of an event relation forces a temporal type of ‘event’.

Table 1: Temporal Type of a Pairwise Join

	Snapshot (regular, non-temporal)	State	Event
Snapshot (regular)	Snapshot	State	Event
State		State	Event
Event			Event

² Teradata (<http://www.teradata.com/t/database/Teradata-Temporal/>) uses the syntax “AS OF timestamp” where Pandamator uses “ON timestamp” for such queries

In order to determine the temporal type of an unmodified relation, one should also combine the table source (table or join) in FROM with the temporal type of all directly referenced subqueries, using the same pairwise table. This is because subqueries are syntactic sugar and that they ultimately correspond to joins (‘hidden’ joins).

EXCEPT, INTERSECT and UNION between relations of different temporal type is forbidden.

Temporal Statement Modifiers

ATSQL makes use of what are called “temporal statement modifiers”, which are short annotations placed before regular SQL to denote the appropriate temporal interpretation. This concept allows a programmer to transfer regular SQL skills to the temporal domain, and appropriate defaults allow programs with existing SQL code to switch to temporal SQL without any modification.

Temporal Statement Modifiers make use of the “reserved words” VALIDTIME and NONSEQUENCED. The following tables define the operation and the resulting temporal type of SQL statements.

Table 2: Temporal Statement Modifier effect on SELECT

Temporal Statement Modifier	Meaning on regular tables	Meaning on state or event tables
(nothing)	Regular SQL, returns snapshot relation	Snapshot at current instant, returns snapshot relation
VALIDTIME ON <x>	Not applicable	Snapshot at specified instant, returns snapshot relation
VALIDTIME	Not applicable	Sequenced semantics, returns state or event relation
VALIDTIME FROM <x> TO <y>	Not applicable	Sequenced semantics, limits operation to specified span, returns state or event relation
NONSEQUENCED VALIDTIME	Regular SQL, returns snapshot relation	Non-sequenced semantics, returns non-temporal relation
NONSEQUENCED VALIDTIME FROM <x> TO <y>	Regular SQL, returns state relation with specified span	Non-sequenced semantics, returns state relation with specified span
NONSEQUENCED VALIDTIME ON <x>	Returns event relation with specified timestamp	Non-sequenced semantics, returns event relation with specified timestamp

Table 3: Temporal Statement Modifier effect on DELETE/UPDATE

Temporal Statement Modifier	Meaning on regular tables	Meaning on state or event tables
(nothing)	Regular SQL	Operate from current instant to forever
VALIDTIME ON <x>	Not applicable	Operate from specified instant to forever
VALIDTIME	Not applicable	Sequenced semantics, operate at each instant
VALIDTIME FROM <x> TO <y>	Not applicable	Sequenced semantics, limit operation to specified span
NONSEQUENCED VALIDTIME	Regular SQL	Non-sequenced semantics
NONSEQUENCED VALIDTIME FROM <x> TO <y>	Not applicable	Not applicable
NONSEQUENCED VALIDTIME ON <x>	Not applicable	Not applicable

Table 4: Temporal Statement Modifier effect on INSERT

Temporal Statement Modifier	Meaning on regular tables	Meaning on state or event tables
(nothing)	Regular SQL	Insert from current instant to forever
VALIDTIME ON <x>	Not applicable	Insert from specified instant to forever
VALIDTIME	Not applicable	Sequenced semantics, insert at each instant
VALIDTIME FROM <x> TO <y>	Not applicable	Sequenced semantics, limit insertion to specified span
NONSEQUENCED VALIDTIME	Regular SQL	Non-sequenced semantics
NONSEQUENCED VALIDTIME FROM <x> TO <y>	Not applicable	Not applicable
NONSEQUENCED VALIDTIME ON <x>	Not applicable	Not applicable

Inheritance of Temporal Statement Modifiers

Temporal statement modifiers are inherited from the outer scope to inner scopes, unless overridden by explicit modifiers. This necessitated the introduction of an explicit modifier for snapshot operation, because in usual ATSQL, there was no way to specify snapshot semantics for an inner query whose absence of an explicit modifier would make it inherit an implicit one from its outer scope. The following table lists the inherited temporal statement modifiers.

Table 5: Inherited Temporal Statement Modifier

Outer	Inherited
VALIDTIME ON <x>	VALIDTIME ON <x>
VALIDTIME	VALIDTIME
VALIDTIME FROM <x> TO <y>	VALIDTIME
NONSEQUENCED VALIDTIME	NONSEQUENCED VALIDTIME
NONSEQUENCED VALIDTIME FROM <x> TO <y>	NONSEQUENCED VALIDTIME
NONSEQUENCED VALIDTIME ON <x>	NONSEQUENCED VALIDTIME

Temporal DDL

As of version 1.9.3, Pandamator supports temporal DDL.

Adding Validtime Support

```
CREATE VALIDTIME
```

Adds the infrastructure (cf. SQL Infrastructure). The file “AddTemporalSupport.sql” should be in the current directory.

Refreshing the Validtime Support

```
ALTER VALIDTIME
```

Recreates all artifacts based on the current temporal metadata (cf. SQL Generation). It should be called once after any of the following statements have been called, because this operation is quite expensive, as it can create a huge SQL script.

Declaring the Temporal Type of a Table

```
ALTER TABLE <table name> ADD VALIDTIME (STATE)
```

```
ALTER TABLE <table name> ADD VALIDTIME (EVENT)
```

Declares a table as a temporal table with the specified temporal type.

Undeclaring a Temporal Table

```
ALTER TABLE <table name> DROP VALIDTIME
```

Undeclares a table as a temporal table.

Declaring a Validtime Constraint

```
ALTER TABLE <table name> ADD VALIDTIME CONSTRAINT  
<constraint name> <constraint>
```

Declares a validtime constraint, where <constraint> is one of the following:

- CONTIGUOUS
- PRIMARY KEY (<columns>)
- UNIQUE (<columns>)
- CONSTANT (<columns>)
- FOREIGN KEY (<columns>) REFERENCES <primary table name> ON DELETE <delete rule>

Undeclaring a Validtime Constraint

```
ALTER TABLE <table name> DROP VALIDTIME CONSTRAINT  
<constraint name>
```

Undeclares a validtime constraint.

QueryExpress, the Pandamator IDE

Pandamator has extended an open-source IDE called QueryExpress by Joseph Albahari. Working with QueryExpress is no different than working with a usual IDE, so this chapter will mainly focus on the temporal additions.

QueryExpress can connect to SQL Server and Oracle databases, as well as databases accessible through OLE-DB. Be aware that only SQL Server is currently supported by Pandamator.



Figure 9 The QueryExpress connection dialog

There is a drop-down control to select the current database.

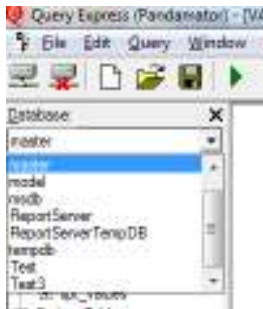


Figure 10 The database selection control

A database which does not currently have temporal support can be identified because a button with a cross and the marking “vt” will be active (“Add ValidTime”). Pushing that button will have the effect of executing “CREATE VALIDTIME” (cf. Adding Validtime Support).



Figure 11 An active "Add ValidTime" button

When the database already has temporal support, the button will be inactive.



Figure 12 An inactive "Add ValidTime" button

The object browser has been extended to show the temporal type and the validtime constraints of the tables and views.

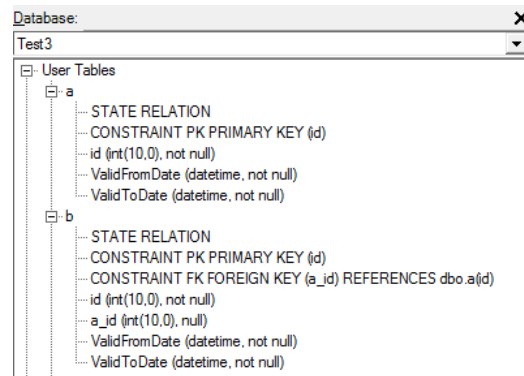


Figure 13 The object browser

One can use the command window to execute arbitrary ATSQL. The message pane underneath reflects the outcome.

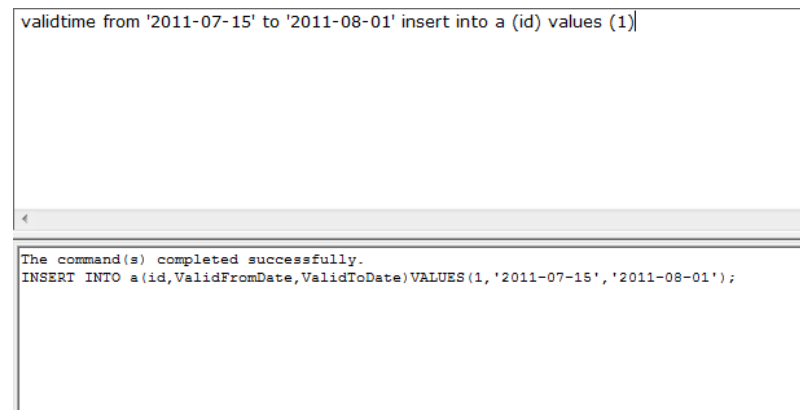


Figure 14 Command execution

Executing one or more queries presents the results in a series of tabs, which are either data grids or timelines.

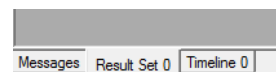


Figure 15 Result tabs

validtime select * from a|

	a_id	ValidFromDate	ValidToDate
▶	3	12/7/2011 12:00:00 ημ	14/7/2011 12:00:00 ημ
	1	15/7/2011 12:00:00 ημ	1/8/2011 12:00:00 ημ
	2	17/7/2011 12:00:00 ημ	5/8/2011 12:00:00 ημ

Figure 16 Data grid

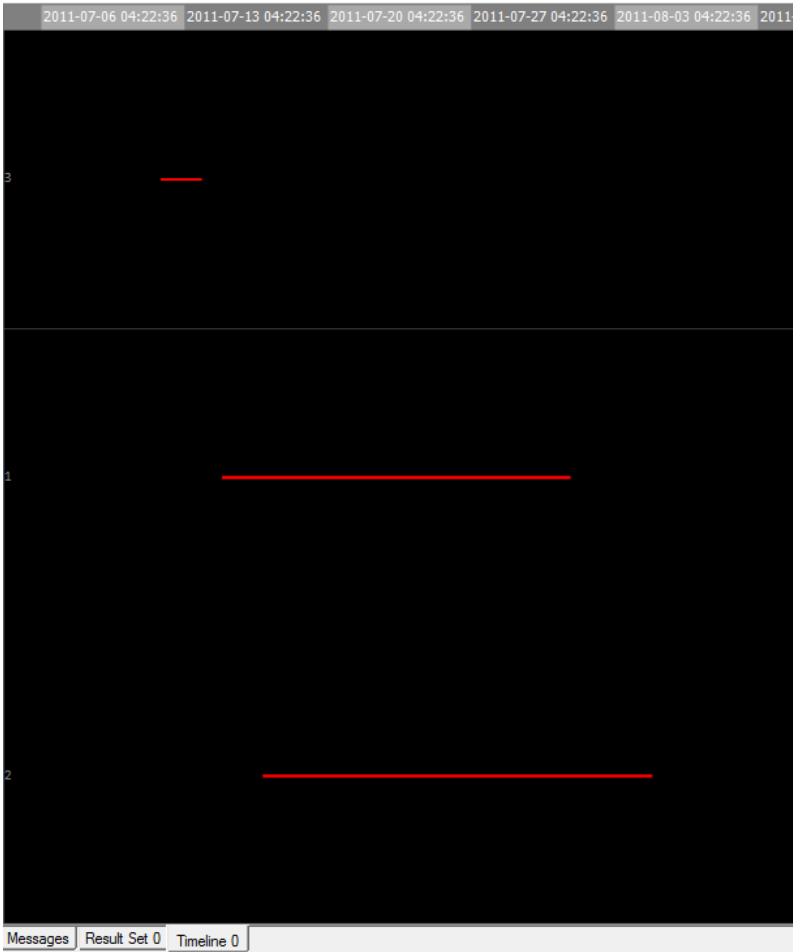


Figure 17Timeline

Translating Temporal SQL

Translating Temporal Queries into SQL92

The prototype implementation of ATSQL (TimeDB) is available for download on the Internet, and is by definition the standard to compare with. The translation Pandamator does is not based in rephrasing temporal SQL using step-by-step applications of temporal algebra operators, like TimeDB does. Instead, it fragments the time axis into all spans defined by all endpoints of rows in the relations involved, runs the SQL query in each one, and coalesces the resulting relation. This amounts to running a separate snapshot query at every instant in time. Instants within the same span cannot be distinguished, because nothing changes during it, so running the query on the span instead of at each individual instant is an optimization. As a matter of convention, I will refer to the relations containing all pertinent instants and the resulting spans as *SpanBoundaries* and *AllSpans* correspondingly.

For example, to treat a query that involves relation A and relation B (in the presence of a period of applicability defined by `@start_date` and `@end_date`, to make it a little harder), one would define the following CTEs.

```
with SpanBoundaries
as (
    select ValidFromDate thedate from A
    union
    select ValidToDate thedate from A
    union
    select ValidFromDate thedate from B
    union
    select ValidToDate thedate from B
    union
    select @start_date
    union
    select @end_date
),
AllSpans(ValidFromDate, ValidToDate)
as (
    select a.thedate,b.thedate from SpanBoundaries a,
    SpanBoundaries b
    where a.thedate < b.thedate
    and not exists (select * from SpanBoundaries c where a.thedate
    < c.thedate and c.thedate < b.thedate)
)
```

Expressing the original query on each span of AllSpans is an operation that is defined syntactically, and is independent of the complexity of the query. In short, AllSpans is added to the source tables and all joins are modified to join to it as well. Since a row of AllSpans is always contained within or coincides with any row in any table involved, this is done with the simple expression `p1.ValidFromDate <= span.ValidFromDate and span.ValidToDate <= p1.ValidToDate`.

For example, a query

```
validtime from @ValidFromDate to @ValidToDate
select p1.[id], null, p1.[val] from [dbo].[C] p1
inner join [dbo].[B] p2 on p1.[b_id] = p2.[id]
```

```
inner join [dbo].[A] p3 on p2.[a_id] = p3.[id]
and p3.val < @val
```

which joins tables A, B and C, will be transformed to

```
select p1.[id], null,
p1.[val], span.ValidFromDate, span.ValidToDate from AllSpans span
inner join [dbo].[C] p1 on p1.ValidFromDate <=
span.ValidFromDate and span.ValidToDate <= p1.ValidToDate
inner join [dbo].[B] p2 on p2.ValidFromDate <=
span.ValidFromDate and span.ValidToDate <= p2.ValidToDate and
p1.[b_id] = p2.[id]
inner join [dbo].[A] p3 on p3.ValidFromDate <=
span.ValidFromDate and span.ValidToDate <= p3.ValidToDate and
p2.[a_id] = p3.[id]
where @ValidFromDate <= span.ValidFromDate and span.ValidToDate
<= @ValidToDate
and p3.val < @val
```

Naturally, the presence of CTEs, subqueries and aggregates, demands a little more effort than what is described here, but the important thing to retain is the central idea behind the transformation. This approach extends to aggregates, for example, if one takes care to add span.ValidFromDate to the grouping columns.

Implementing Coalescing in SQL92

Coalescing is an important operation to ensure a succinct representation of temporal data.

Pandamator uses a recursive formulation of coalescing that has been found to scale well on more than 100,000 rows³, in contrast to the code found in Snodgrass. Column equality is implemented to cope with NULL values, for example `1. col1 = f. col1 AND (1. col2 = f. col2 or coalesce(1. col2, f. col2) is null)` where column *col1* is non-null whereas column *col2* is nullable.

```
with
Coal(columns, ValidFromDate, ValidToDate)
as (
    select columns, ValidFromDate, ValidToDate
    from R p2
    where not exists(select * from R p1
                     where p1.ValidToDate = p2.ValidFromDate
                     and p1 and p2 column equality
                    )
    union all
    select p1 columns, p1.ValidFromDate, p2.ValidToDate
    from Coal p1
    inner join R p2 on p1.ValidToDate = p2.ValidFromDate
                     and p1 and p2 column equality
```

³ A version that, curiously, performs slightly worse is:

```
Coalesced(columns, ValidFromDate, ValidToDate) as (
    select distinct p1 columns, p1.ValidFromDate, p1.ValidToDate from Coal p1
    where not exists(select * from R p2 where p1.ValidToDate = p2.ValidFromDate
                     and p1 and p2 column equality)
)
```

```

),
Coalesced(columns,ValidFromDate,ValidToDate)
as (
    select p1 columns, p1.ValidFromDate, max(p1.ValidToDate)
    from Coal p1
    group by p1 columns, p1.ValidFromDate
)
select * from Coalesced

```

Coalescing corresponds to replacing relation R with its coalesced equivalent. Pandamator does this in-place without making use of a temporary table, by means of the following mechanism. Rows that are first in a series of coalesced ones have their ValidToDate extended, and rows that were subsequent in the series end up being subsumed by the extended ones and are then deleted.

```

with
CoalescedSpans(columns,start_date,end_date)
as (
    coalescing code
)
update R
set ValidToDate = p2.ValidToDate
from R p1
inner join CoalescedSpans p2
    on p1 and p2 column equality
    and p1.ValidFromDate=p2.ValidFromDate and p1.ValidToDate <
p2.ValidToDate;

delete from R
from R p1
where exists(
    select * from R p2
    where p1 and p2 column equality
    and p2.ValidFromDate < p1.ValidFromDate and p1.ValidToDate <=
p2.ValidToDate
);

```

The meaning of a state relation is independent of whether it is coalesced or not. This notion is called “snapshot reducibility”. However, for reasons of parsimony and ease of use, Pandamator always coalesces state relations that are used either at top-level, or as subqueries in snapshot or non-sequenced queries. This enables one, for example, to write a query to find durations during which an attribute did not change might seem straightforward to formulate thus:

```

with r1 as (validtime select x from R)
nonsequenced validtime select x, ValidToDate-ValidFromDate from r1

```

Unfortunately, coalescing is a complex operation that, when applied at the end of a complex generated query can literally freeze the RDBMS. Pandamator enables conditional inhibition of coalescing with the FULL directive before SELECT (subject to change).

Translating Sequenced Queries into SQL92

Sequenced queries, whether they are the main query or a subquery, are treated in the same manner. I will use the following query tree for my example, where queries are presented as rectangles marked with “T”, if they result in state relations, or “S”, if they result in snapshot relations.

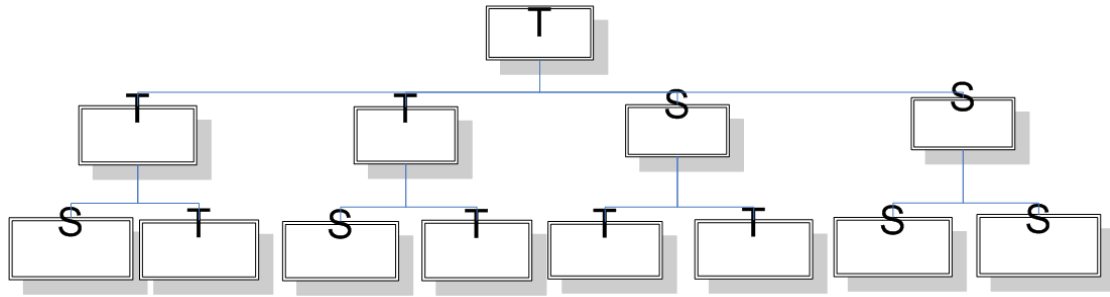


Figure 18: Sample Query Tree for Translating Sequenced Queries

First, the queries resulting in state relations are identified, along with their subqueries that also result in state relations, recursively. This identifies distinct query subtrees for further processing, like in the following sketch.

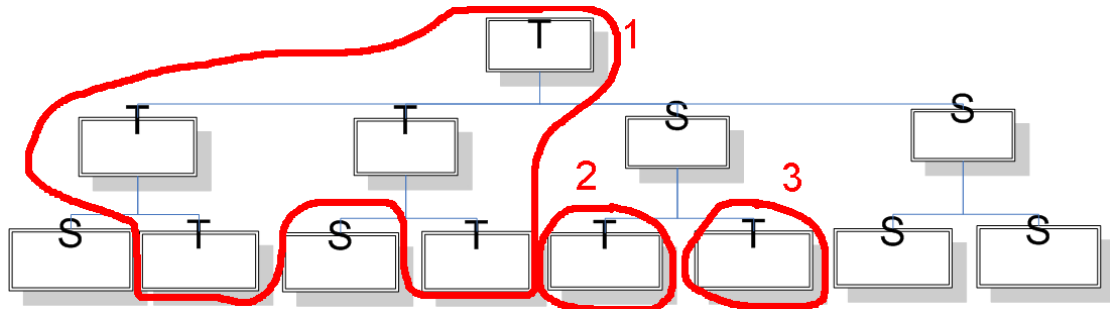


Figure 19: Query Subtrees for Translating Sequenced Queries

Three query subtrees, labeled 1, 2 and 3 are identified here.

For each subtree, Pandamator will create a CTE for the root query and also create CTEs for the coalesced version of the resulting relation. The original root queries are replaced, at the point of use, with `select * from Coalesced_T` for each query subtree headed by T. See Appendix A: Query Transformation Example (Not Coalesced), and Appendix B: Query Transformation Example (Coalesced).

Translating Temporal Deletes into SQL92

Snodgrass (Snodgrass: Developing Time-Oriented Database Applications in SQL) describes a method to execute updates and deletes based on single-table queries and suggests a “case analysis on the interaction” of periods from all tables involved when updates and deletes involve more tables. Unfortunately, this advice does not provide a concrete way to mechanize the process. I will describe, in the sequel, a way to describe temporal updates and deletes in a way that is independent of the number of tables involved in the associated query or its complexity. The underlying mechanism owes its inspiration to N. Lorentzos whose interval-extended SQL (IXSQL) was based semantically on operations that split and joined intervals at instants determined by the actual data in a table, to implement all necessary temporal algebra operators.

DELETE is defined at minimum using a WHERE clause on a table (which, however, could mention other tables using subqueries), or by utilizing a full FROM clause containing join expressions. Either way, it is defined on top of a query that describes the primary key and the spans of the entities that should be deleted. I will refer to this query as *DeletedSpans*. The complementary relation of DeletedSpans with respect to the whole relation R is *RetainedSpans*. In other words, (DeletedSpans union RetainedSpans) \equiv R. Deletion amounts to replacing R by RetainedSpans.

The canonical way of expressing DELETE on top of DeletedSpans (which will be chopped according to AllSpans, as shown in Translating Temporal Queries) would be as follows. Note that only primary key columns are actually used by the code, even though all columns are contained in the CTEs.

```

Definition of SpanBoundaries, AllSpans & DeletedSpans,
RetainedSpans(columns, start_date, end_date)
as (
    select p1 columns, span.start_date, span.end_date from AllSpans
    span
    inner join R p1 on p1.start_date <= span.start_date and
    span.end_date <= p1.end_date
    where not exists (
        select * from DeletedSpans p2
        where p1 pkey = p2 pkey and span.start_date=p2.start_date
        and span.end_date=p2.end_date
    )
),
CoalescedRetainedSpans(columns, start_date, end_date)
as (
    coalescing code
)
select columns, start_date, end_date
into #temp
from CoalescedRetainedSpans;

delete from R;

insert into R(columns, start_date, end_date)
select columns, start_date, end_date
from #temp;

drop table #temp;

```

Pandamator employs an optimized version that needs less data to be moved to and from the temporary table. This is done by saving only retained spans of R resulting from chopping off a row in R, and saving *markers* to rows of R that must be deleted as a whole. Markers are distinguished by setting ValidToDate = ValidFromDate, which is invalid in a state relation.

```

Definition of SpanBoundaries, AllSpans & DeletedSpans,
RetainedSpans(columns, [val], ValidFromDate, ValidToDate)
as (
    select p1 columns, span.ValidFromDate, span.ValidToDate from
    AllSpans span
    inner join R p1 on p1.ValidFromDate <= span.ValidFromDate and
    span.ValidToDate <= p1.ValidToDate
    where not exists ( -- there is no DeletedSpan coinciding with
    it
        select * from DeletedSpans p2
        where p1 pkey = p2 pkey and
        span.ValidFromDate=p2.ValidFromDate and
        span.ValidToDate=p2.ValidToDate
    ) and exists ( -- but there is a DeletedSpan overlapping the
    whole row
        select * from DeletedSpans p2
        where p1 pkey = p2 pkey and
        p1.ValidFromDate<p2.ValidToDate and p2.ValidFromDate<p1.ValidToDate
    )
)

```



```

),
CoalescedRetainedSpans ( columns, start_date, end_date)
as (
    coalescing code
) ,
SavedSpans ( columns, ValidFromDate, ValidToDate)
as (
    select p1 columns, p1.ValidFromDate, p1.ValidToDate from
CoalescedRetainedSpans p1 -- saved spans
    union
    select p1 columns, p1.ValidFromDate, p1.ValidFromDate from
DeletedSpans p1 -- markers
)
select columns, ValidFromDate, ValidToDate
into #temp
from CoalescedRetainedSpans;

-- Delete from R every row that overlaps with saved spans
delete from R
from R p1
where exists(
    select * from #temp p2
    where p1 pkey = p2 pkey and p1.ValidFromDate<p2.ValidToDate and
p2.ValidFromDate<p1.ValidToDate
);

--delete rows pointed to by #temp markers
delete from R
from R p1
where exists(
    select * from #temp p2
    where p1 pkey = p2 pkey and p1.ValidFromDate=p2.ValidFromDate
and p1.ValidFromDate=p2.ValidToDate
);

insert into columns, ValidFromDate, ValidToDate)
select columns, ValidFromDate, ValidToDate
from #temp
where ValidFromDate<>ValidToDate; --bypass markers

drop table #temp;

```

Translating Temporal Updates into SQL92

UPDATE is defined at minimum using a WHERE clause on a table (which, however, could mention other tables using subqueries), or by utilizing a full FROM clause containing join expressions. Either way, it is defined on top of a query that describes the primary key, the column values and the spans of the entities that should be the outcome. I will refer to this query as *AlteredSpans*. The complementary relation of *AlteredSpans* with respect to the whole relation *R* is *RetainedSpans*. In other words, (*AlteredSpans* union *RetainedSpans*) \equiv *R*. Update amounts to replacing *R* by *RetainedSpans*.

The canonical way of expressing UPDATE on top of *AlteredSpans* is as follows.

```

Definition of SpanBoundaries, AllSpans & AlteredSpans,
RetainedSpans ( columns, ValidFromDate, ValidToDate)
as (

```

```

        select p1 columns, span.ValidFromDate, span.ValidToDate from
AllSpans span
        inner join R p1 on p1.ValidFromDate <= span.ValidFromDate and
span.ValidToDate <= p1.ValidToDate
        where not exists ( -- there is no AlteredSpan coinciding with
it
            select * from AlteredSpans p2
            where p1 pkey = p2 pkey and
span.ValidFromDate=p2.ValidFromDate and
span.ValidToDate=p2.ValidToDate
        )
    ),
FinalSpans
as (
    select * from AlteredSpans
    union all
    select * from RetainedSpans
),
CoalescedFinalSpans(columns, ValidFromDate, ValidToDate)
as (
    coalescing code
)
select columns, start_date, end_date
into #temp
from CoalescedFinalSpans;

delete from R;

insert into R(columns, start_date, end_date)
select columns, start_date, end_date
from #temp;

drop table #temp;

```

Pandamator employs an optimized version that needs less data to be moved to and from the temporary table. This is done by saving only altered spans of R resulting from updating a portion of a row in R and retained spans that are the remaining portions. Consequently, only those rows in R that are affected need be deleted before the contents of the temporary table are inserted back into R.

```

Definition of SpanBoundaries, AllSpans & AlteredSpans,
RetainedSpans(columns, ValidFromDate, ValidToDate)
as (
    select p1 columns, span.ValidFromDate, span.ValidToDate from
AllSpans span
    inner join R p1 on p1.ValidFromDate <= span.ValidFromDate and
span.ValidToDate <= p1.ValidToDate
    where not exists ( -- there is no AlteredSpan coinciding with
it
        select * from AlteredSpans p2
        where p1 pkey = p2 pkey and span.ValidFromDate =
p2.ValidFromDate and span.ValidToDate = p2.ValidToDate
    ) and exists ( -- but there is an AlteredSpan overlapping the
whole row
        select * from AlteredSpans p2
        where p1 pkey = p2 pkey and p1.ValidFromDate <
p2.ValidToDate and p2.ValidFromDate < p1.ValidToDate
    )
),

```

```

FinalSpans
as (
    select * from AlteredSpans
    union all
    select * from RetainedSpans
),
CoalescedFinalSpans(columns, ValidFromDate, ValidToDate)
as (
    coalescing code
)
select columns, ValidFromDate, ValidToDate
into #temp
from CoalescedFinalSpans;

-- Delete every row that overlaps with saved spans
delete from R
from R p1
where exists(
    select * from #temp p2
    where p1 pkey = p2 pkey and p1.ValidFromDate < p2.ValidToDate
and p2.ValidFromDate < p1.ValidToDate
);

insert into R(columns, ValidFromDate, ValidToDate)
select columns, ValidFromDate, ValidToDate
from #temp;

drop table #temp;

```

When UPDATE refers to a single table, Pandamator uses simpler code taken from Snodgrass.

```

INSERT INTO R (columns, ValidFromDate, ValidToDate)
SELECT columns, ValidFromDate, @ValidFromDate
FROM R
WHERE condition on table
AND ValidFromDate < fromdate
AND fromdate < ValidToDate;

```

```

INSERT INTO R( columns, ValidFromDate, ValidToDate)
SELECT columns, @ValidToDate, ValidToDate
FROM R
WHERE condition on table
AND ValidFromDate < todate
AND todate < ValidToDate;

```

```

UPDATE R
SET values
WHERE condition on table
AND ValidFromDate < todate
AND fromdate < ValidToDate;

```

```

UPDATE R
SET ValidFromDate = fromdate
WHERE condition on table
AND ValidFromDate < fromdate
AND fromdate < ValidToDate;

```

```

UPDATE R

```

```
SET ValidToDate = @ValidToDate  
WHERE condition on table  
AND ValidFromDate < todate  
AND todate < ValidToDate;
```

Call to coalescing code

Useful Non-Sequenced Idioms

Why Sequenced Queries Are Not Enough

Truth be told, ATSQL (and SQL2, its older sibling), provide a solution for just a single real problem in using temporal databases, which is sequenced operations. Unfortunately, the majority of useful queries one might think of after adopting the temporal point of view, are non-sequenced, which amounts to saying that no help is available for implementing them.

I have collected here a number of useful non-sequenced idioms that I hope will provide some comfort in that area. Some of them might end up as language extensions, although I am not very eager to add isolated solutions to isolated problems.

Event Statemachines

Timestamped data that represent observations, gauge readings, invoices etc are very common in non-temporal databases in every application area. Sometimes a stream of such events is thought of as implying an associated succession of states, which is expressed as a state relation in a temporal database. A statemachine, the way it will be described here, is applicable only when the underlying events do not coincide. If they do, the logic to “serialize” the events and feed them, one-by-one, to the statemachine, will not work and will need to be extended. This property of the event relation is called being *per-partition sequential*, a *partition* in that context being the item collection corresponding to the same object surrogate (see terminology in Time-varying data).

Formulating the query that produces this state relation is cumbersome, but it looks something like the following.

```
with
AllInvoiceSpans(id,invoicetype,val,ValidFromDate,ValidToDate)
as (
    select a.id,a.invoicetype,a.val,a.ValidOnDate,b.ValidOnDate
from InvoiceEvents a,InvoiceEvents b
    where a.ValidOnDate < b.ValidOnDate
    and a.id = b.id
    and not exists (select * from InvoiceEvents c where a.id = c.id
and a.ValidOnDate < c.ValidOnDate and c.ValidOnDate < b.ValidOnDate)

    union all

    select a.id,a.invoicetype,a.val,a.ValidOnDate,'3000-01-01' from
InvoiceEvents a
    where not exists (select * from InvoiceEvents b where
a.ValidOnDate < b.ValidOnDate and a.id = b.id)
),
States(id,state,val,ValidFromDate,ValidToDate)
as (
    select id, 0, val, ValidFromDate, ValidToDate from
AllInvoiceSpans a
    where invoicetype = 100
```

```

        and not exists (select * from AllInvoiceSpans b where a.id =
b.id and b.ValidToDate = a.ValidFromDate)

        union all

        select id, 1, val, ValidFromDate, ValidToDate from
AllInvoiceSpans a
        where invoicetype = 110
        and not exists (select * from AllInvoiceSpans b where a.id =
b.id and b.ValidToDate = a.ValidFromDate)

        union all

        select a.id, 10, a.val+b.val, b.ValidFromDate, b.ValidToDate
from States a, AllInvoiceSpans b
        where a.id = b.id and a.ValidToDate = b.ValidFromDate and
a.state = 0 and b.invoicetype = 120

        union all

        select a.id, 11, a.val+b.val, b.ValidFromDate, b.ValidToDate
from States a, AllInvoiceSpans b
        where a.id = b.id and a.ValidToDate = b.ValidFromDate and
a.state = 1 and b.invoicetype = 120

        union all

        select a.id, 14, a.val+b.val, b.ValidFromDate, b.ValidToDate
from States a, AllInvoiceSpans b
        where a.id = b.id and a.ValidToDate = b.ValidFromDate and
a.state = 10 and b.invoicetype = 140

        union all

        select a.id, 13, a.val-b.val, b.ValidFromDate, b.ValidToDate
from States a, AllInvoiceSpans b
        where a.id = b.id and a.ValidToDate = b.ValidFromDate and
a.state = 10 and b.invoicetype = 130

        union all

        select a.id, 24, a.val+b.val, b.ValidFromDate, b.ValidToDate
from States a, AllInvoiceSpans b
        where a.id = b.id and a.ValidToDate = b.ValidFromDate and
a.state = 11 and b.invoicetype = 140

        union all

        select a.id, 23, a.val-b.val, b.ValidFromDate, b.ValidToDate
from States a, AllInvoiceSpans b
        where a.id = b.id and a.ValidToDate = b.ValidFromDate and
a.state = 11 and b.invoicetype = 130
    )
select * from States

```

It could be facilitated by introducing some appropriate syntax, like

```

select validtime set from <someeventset> where <maincond> as (
    goto <s1> select <values...> where <cond>
    goto <s2> select <values...> where <cond>
from <s1>

```

```

    goto <s3> select <values...> where <cond>
    goto <s4> select <values...> where <cond>
from any
    goto <s5> select <values...> where <cond>
    goto <s6> select <values...> where <cond>
    ...
)

```

The point would be to produce SQL code like the own shown previously, given something like

```

select validtime set from InvoiceEvents as (
    goto 0 select id, val where invoicetype = 100
    goto 1 select id, val where invoicetype = 110
from 0
    goto 10 select id, old.val+val where invoicetype = 120
from 1
    goto 11 select id, old.val+val where invoicetype = 120
from 10
    goto 14 select id, old.val+val where invoicetype = 140
    goto 13 select id, old.val-val where invoicetype = 130
from 11
    goto 24 select id, old.val+val where invoicetype = 140
    goto 23 select id, old.val-val where invoicetype = 130
)

```

Such helper syntax, is under consideration for inclusion in Pandamator.

As an example of a statemachine in action, consider the following event relation.

Data Table 6: Event Relation to apply statemachine to

id	invoicetype	val	ValidOnDate
1	100	3	2008-01-01
1	120	0	2008-01-10
1	140	3	2008-01-12
2	110	4	2008-01-05
2	120	0	2008-01-06
2	130	4	2008-01-10

Running the statemachine above produces the following state relation.

Data Table 7: Event Statemachine results

id	state	Val	ValidFromDate	ValidToDate
1	0	1	2008-01-01	2008-01-10
1	10	2	2008-01-10	2008-01-12
1	14	2	2008-01-12	3000-01-01
2	1	2	2008-01-05	2008-01-06
2	11	1	2008-01-06	2008-01-10
2	23	1	2008-01-10	3000-01-01

Status Transitions

When using a state relation that implements a statemachine (for example, having a “status” column), either created out of an event relation, like before, or available directly, there is a feature of Pandamator that can help enforce the correct state transitions declaratively.

One can create a view implementing an event relation that records the transitions (“status” values before and after), having a foreign key to a table holding the allowed such pairs. Referential integrity constraints will ensure that attempting to make a disallowed transition will be rejected.

Fold Queries

Fold queries are a class of non-sequenced queries that are formulated in a well-structured manner and can formulated many problems that involve accumulation or integration over time.

They are based on the *fold* higher-order functions commonly used in functional programming. An example can help illustrate the operation of *fold*.

Given a sequence of values $S=[s_0, s_1, s_2, \dots s_n]$, a function of two arguments f and a starting value a , we can define the following functions.

$$\text{fold_left}(f, a, S) = f(\dots f(f(a, s_0), s_1), \dots s_n)$$

$$\text{fold1_left}(f, S) = f(\dots f(f(s_0, s_1), s_2) \dots s_n)$$

$$\text{fold_right}(f, a, S) = f(s_0, \dots f(s_{n-1}, f(s_n, a)) \dots)$$

$$\text{fold1_right}(f, S) = f(s_0, \dots f(s_{n-2}, f(s_{n-1}, s_n)) \dots)$$

The difference between the two variants is whether we supply an initial value or not (in the latter case the sequence cannot have less than two elements).

Translating the same concept to a temporal relation, a fold query is a pairwise application of a function to successive states to perform some kind of accumulation. In order to account for gaps in the history, first we define the *temporal complement* of a state relation (a term I coined myself). A temporal complement is a relation that fills the gaps inside a non-contiguous history within a relation with null values for columns other than the primary key ones. Its definition is

```

select p1 key columns, p1 null non-key columns, p1.ValidToDate
as ValidFromDate, p2.ValidFromDate as ValidToDate
from R p1
inner join R p2 on p1 pkey = p2 pkey
where p1.ValidToDate < p2.ValidFromDate
and not exists(
    select * from R p3
    where p1 pkey = p3 pkey
    and p1.ValidToDate < p3.ValidFromDate
    and p3.ValidFromDate < p2.ValidFromDate
)

```

Fold queries can be expressed easily as recursive queries over the union of a relation and its temporal complement.

An example of a *foldl_left* query is the following, that accumulates column *val* using addition, starting with value -2 and treating NULL as -3.

```
with
-- Temporal complement
H_TC (id, b_id, val, ValidFromDate, ValidToDate)
as (
    select p1.id as id, null as b_id, null as val, p1.ValidToDate
as ValidFromDate, p2.ValidFromDate as ValidToDate
    from H p1
    inner join H p2 on p1.id=p2.id
    where p1.ValidToDate < p2.ValidFromDate
    and not exists(select * from H p3 where p3.id=p1.id and
p1.ValidToDate < p3.ValidFromDate and p3.ValidFromDate <
p2.ValidFromDate)
),
H_Full (id, b_id, val, ValidFromDate, ValidToDate)
as (
    select id, b_id, val, ValidFromDate, ValidToDate from H
    union all
    select id, b_id, val, ValidFromDate, ValidToDate from H_TC
),
SampleFold (id, ret, ValidOnDate)
as (
    select id, -2, ValidFromDate from H_Full p2
    where not exists(select * from H_Full p1 where p1.id=p2.id and
p1.ValidToDate <= p2.ValidFromDate)

    union all

    select p1.id, p1.ret+(CASE WHEN p2.val is null THEN -3 ELSE
p2.val END), p2.ValidToDate
    from SampleFold p1
    inner join H_Full p2 on p1.id=p2.id and p1.ValidOnDate =
p2.ValidFromDate
)
select id, ret, ValidOnDate from SampleFold order by id, ValidOnDate;
```

This example uses an operation which is concerned purely with the succession of states and not their duration, but it is possible to use an operation which takes into account durations as well. In this manner we could implement *integration*, as well as any other non-standard aggregate operator.

Valid-time Partitioning

Valid-time partitioning is the partitioning of valid-time in spans over which we can compute useful aggregates (this is called Valid-time Cumulative Aggregation). When this is done using spans defined by a calendar, it is called Static Valid-time Partitioning.

Joining with a schedule of the appropriate definition accomplishes this easily (cf. Scheduled entities). The partitioning spans could be overlapping, to calculate something based on a moving window.

Event Succession Queries

At times we need to query about patterns in an event stream. I call these queries event succession queries. The following is an example that queries how many times any of the following patterns appeared in a fictitious stream of invoices:

- 100 120 (meaning 100 followed by 120)
- 110 . 130 (meaning 110 followed by anything followed by 130)

```

SuccessionSpans(id,combinedvalue,thedata)
as (
    select a.id,a.[value]+b.[value],b.ValidOnDate from
InvoiceEvents a, InvoiceEvents b
    where a. ValidOnDate < b. ValidOnDate
    and a.id = b.id
    and a.invoicetype = 100 and b.invoicetype = 120
    and not exists (select * from InvoiceEvents c where a.id = c.id
and a. ValidOnDate < c. ValidOnDate and c. ValidOnDate < b.
ValidOnDate)

    union all

    select a.id, a.[value]+b.[value]+c.[value], c. ValidOnDate from
InvoiceEvents a,InvoiceEvents b,InvoiceEvents c
    where a. ValidOnDate < b. ValidOnDate and b. ValidOnDate < c.
ValidOnDate
    and a.id = b.id and b.id = c.id
    and a.invoicetype = 110 and c.invoicetype = 130
    and not exists (select * from InvoiceEvents c1 where a.id =
c1.id and a. ValidOnDate < c1. ValidOnDate and c1. ValidOnDate < b.
ValidOnDate)
    and not exists (select * from InvoiceEvents c2 where b.id =
c2.id and b. ValidOnDate < c2. ValidOnDate and c2. ValidOnDate < c.
ValidOnDate)

)

select count(*) cnt from SuccessionSpans

```

Branching Time (not yet incorporated)

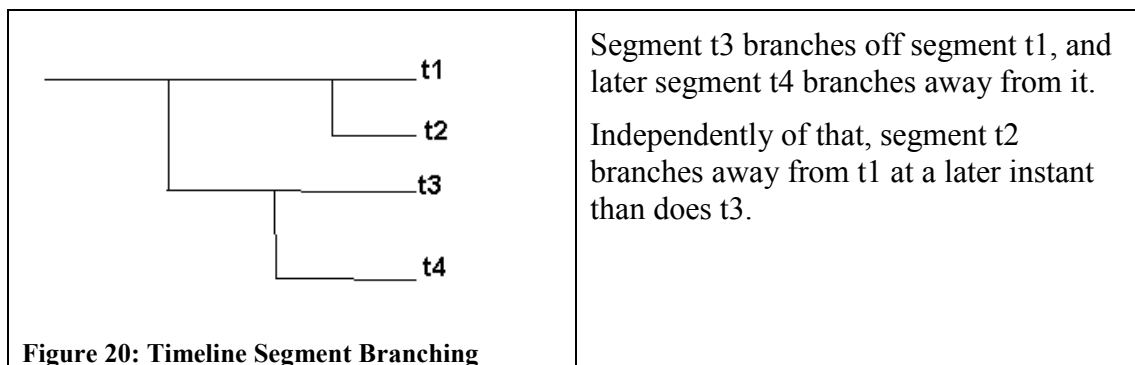
Introduction

Branching time is an extension of valid time (which will be referred to as *linear time* in the sequel) that can be used to represent alternative time axes (*timelines*), *branching* freely off one another to create alternative represented realities. Although it sounds arcane, such a concept can be used whenever one needs to create alternative future data that nevertheless share a common past, like in a Web Content Management System where designers create alternative future versions of a page in advance.

I could not find any treatment of branching time in the (limited) bibliography I studied, consequently I consider support of branching time in SQL to be a novel contribution of Pandamator, albeit experimental.

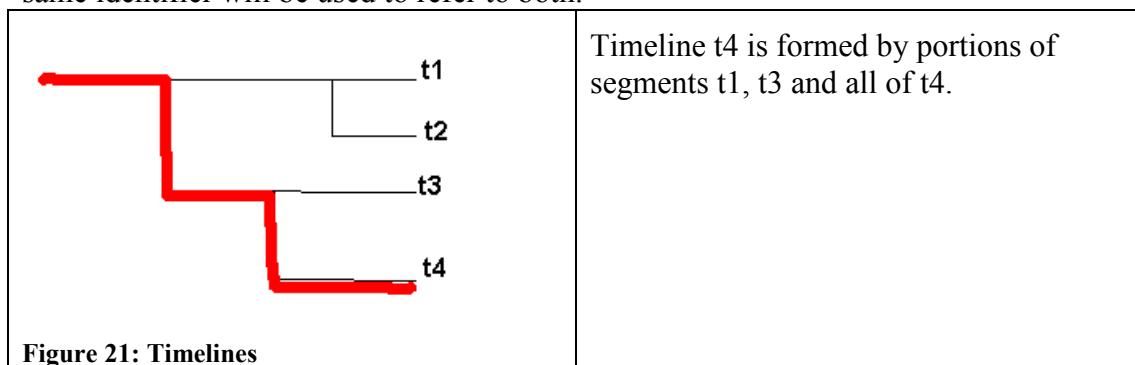
Timeline Segments

The basic elements for the representation of branching time are *timeline segments*. Timeline segments are time periods that optionally have a *parent segment* they branch away from, at a specific *branching instant*.



Timelines

The branching of timeline segments forms a proper tree, and the paths from the root to the leaves are *timelines*. As timelines are uniquely identified by the leaf segment, the same identifier will be used to refer to both.



Timeline segments would normally be valid forever, however one could specify a ValidToDate to denote that we need not represent any temporal entity on that timeline past that date.

It is possible to form forests, rather than trees, if needs dictate so. It is also possible to form a forest of degenerate trees, each consisting of a single timeline, in which case the database will be simply partitioned in separate “realities”, each identified by a separate timeline identifier. In any of the aforementioned cases, the same logic holds and we need not distinguish special cases in our treatment.

This representation of timelines is asymmetrical by design. I first experimented with abutting segments, a design that made for simpler representation and treatment. However, the operation of creating a new branch was a lot costlier in such a design, resulting in massive updates to existing data. I concluded that the guiding principle should be that existing data should be unperturbed by new data, particularly when new data are supposed to lie in alternative “realities”.

Representation

The only modification to a temporal table, in order to support branching time, is to add column ValidSegment, which holds a timeline segment id. Obviously a temporal constraint is needed to ensure that the period falls within the validity of the timeline.

The referenced timeline segment actually annotates ValidToDate. The segment ValidFromDate belongs to can be computed if needed but it is irrelevant to queries and SQL generation as we focus on whole timelines.

Timeline segments are held in the following table.

```
create table timeline_segments (
  segment_id int not null primary key,
  parent_segment_id int,
  ValidFromDate datetime not null,
  ValidToDate datetime not null
);
```

Timelines are defined by the following view.

```
create view [dbo].[timelines] as
with timelines (timeline_segment_id, segment_id, ValidFromDate,
ValidToDate)
as (
  -- All segments belong to a timeline of their own
  select segment_id, segment_id, ValidFromDate, ValidToDate
  from timeline_segments p1

  union all

  -- All segments belong to the timelines of their branching
  segments, up to the branching instant
  select p2.segment_id, p1.segment_id, p1.ValidFromDate,
p2.ValidFromDate
  from timeline_segments p1, timeline_segments p2
  where p1.segment_id = p2.parent_segment_id

  union all

  -- All segments belong to the timelines of all segments
  branching off their immediate branching segments, recursively
```

```

        select p2.segment_id, p1.segment_id, p1.ValidFromDate,
p1.ValidToDate
        from timelines p1, timeline_segments p2
        where p1.timeline_segment_id = p2.parent_segment_id and
p1.segment_id <> p2.parent_segment_id
    )

select * from timelines

```

Queries in branching time are slightly more complicated than queries in linear time, because all interacting periods must be constrained to belong to the same timeline. The following is a simple query on a table, in branching time.

```

select t1.timeline_segment_id timeline, p1.*
from tab p1
inner join timelines t1
on p1.ValidSegment=t1.segment_id and t1.ValidFromDate <=
p1.ValidToDate and p1.ValidToDate <= t1.ValidToDate

```

And the following is a query computing primary key violations in branching time.

```

SELECT distinct t3.timeline_segment_id, I1.id
id, dbo.LAST_INSTANT(I1.ValidFromDate, I2.ValidFromDate)
ValidFromDate, dbo.FIRST_INSTANT(I1.ValidToDate, I2.ValidToDate)
ValidToDate
        FROM [dbo].[BA] I1
        inner join [dbo].[BA] AS I2
        ON I1.[id] = I2.[id]
        AND (I1.ValidFromDate <> I2.ValidFromDate OR
I1.ValidToDate <> I2.ValidToDate)
        AND I1.ValidFromDate < I2.ValidToDate
        AND I2.ValidFromDate < I1.ValidToDate
        inner join timelines t3
        on i1.ValidSegment=t3.segment_id and
t3.ValidFromDate <= i1.ValidToDate and i1.ValidToDate <=
t3.ValidToDate
        inner join timelines t4
        on i2.ValidSegment=t4.segment_id and
t4.ValidFromDate <= i2.ValidToDate and i2.ValidToDate <=
t4.ValidToDate
        and t4.timeline_segment_id = t3.timeline_segment_id

```

The modifications are limited to one extra join with view “timelines” per table, and the constraint that all timelines should coincide.

Virtual Database per Timeline

Like the representation of data in linear time can be viewed as a succinct way to model separate database snapshots, one per time instant, the representation of data in branching time can be viewed as a succinct way to model separate temporal databases, one per timeline. Tables in branching time can be thought of as specifications, and joining with view “timelines” actually produces the multitude of distinct versions of the data on the appropriate timelines.

SQL Infrastructure

A single SQL file, named *AddTemporalSupport.sql*, is meant to be run against an SQL Server Database to prepare for temporal support. Its contents are the following.

Temporal Metadata

A schema is created, called *temporal_metadata*. Inside it are created the four tables which are explained in the sequel.

temporal_metadata.tables

- table_schema
- table_name
- table_type (one of “EVENT”, “STATE”)

temporal_metadata.table_constraints

- table_schema
- table_name
- constraint_name
- constraint_type (one of “PRIMARY KEY”, “FOREIGN KEY”, “CONTIGUOUS”, “UNIQUE”, “CONSTANT”)

temporal_metadata.constraint_columns

- table_schema
- table_name
- column_name
- constraint_name
- ordinal_position

temporal_metadata.referential_constraints

- foreign_table_schema
- foreign_table_name
- foreign_constraint_name
- primary_table_schema
- primary_table_name
- primary_constraint_name
- delete_rule (one of “CASCADE”, “RESTRICT”, “SET NULL”)

Pseudo-DDL

Temporal metadata are populated by the following pseudo-DDL procedures. Contrary to plain SQL, views are allowed as well as base tables.

TemporalDeclareTable

Declares the table as one needing temporal support. Annotation columns must already exist in the table.

Called as: `exec TemporalDeclareTable <schema>, <table>,
<table type>`

For example: `exec TemporalDeclareTable 'dbo', 'a', 'STATE'`

TemporalUndeclareTable

Undeclares the table from the temporal metadata.

Called as: `exec TemporalUndeclareTable <schema>, <table>`

For example: `exec TemporalUndeclareTable 'dbo', 'a'`

TemporalDeclarePrimaryKey

Declares a temporal primary key. An index on the columns will not be created automatically. Also, declaring a temporal primary key on an event table does not result in any constraint-checking code being generated. The programmer should declare a normal SQL PRIMARY KEY constraint on the same columns, as well.

Called as: `exec TemporalDeclarePrimaryKey <constraint name>,
<schema>, <table>, <columns>`

Columns are written in a single string, delimited by '[' and ']'.

For example: `exec TemporalDeclarePrimaryKey 'PK', 'dbo', 'a',
'[col1][col2]'`

Because columns defined as the result of functions are always nullable, there is a way to instruct `TemporalDeclarePrimaryKey` to not perform the check for nullable columns, by prefixing the column string with a 'U', as in `'U[col1][col2]'`. The responsibility for ensuring that the columns will never contain nulls, lies with the programmer. Should the column contain any nulls, the result of all generated SQL is unpredictable.

TemporalDeclareUnique

Declares a temporal UNIQUE constraint. An index on the columns will not be created automatically.

Called as: `exec TemporalDeclareUnique <constraint name>,
<schema>, <table>, <columns>`

Columns are written in a single string, delimited by '[' and ']'.

For example: `exec TemporalDeclareUnique 'a_unq', 'dbo', 'a',
'[col1][col2]'`

TemporalDeclareConstant

Declares a temporal CONSTANT constraint. An index on the columns will not be created automatically.

Called as: `exec TemporalDeclareConstant <constraint name>, <schema>, <table>, <columns>`

Columns are written in a single string, delimited by '[' and ']'.

For example: `exec TemporalDeclareConstant 'a_cst', 'dbo', 'a', '[col1][col2]'`

TemporalDeclareContiguousHistory

Declares the table as having contiguous history.

Called as: `exec TemporalDeclareContiguousHistory <constraint name>, <schema>, <table>`

For example: `exec TemporalDeclareContiguousHistory 'CH', 'dbo', 'a'`

TemporalDeclareForeignKey

Declares a foreign key. An index on the columns will not be created automatically.

Called as: `exec TemporalDeclareForeignKey <constraint name>, <foreign schema>, <foreign table>, <foreign columns>, <primary schema>, <primary table>, <delete rule>`

Columns are written in a single string, delimited by '[' and ']'.

For example: `exec TemporalDeclareForeignKey 'FK', 'dbo', 'a', '[col1]', 'dbo', 'b', 'CASCADE'`

TemporalUndeclareConstraint

Undeclares a constraint by name.

Called as: `exec TemporalUndeclareConstraint <constraint name>, <schema>, <table>`

For example: `exec TemporalUndeclareConstraint 'PK', 'dbo', 'a'`

Scheduled entities

Support for scheduled entities makes use of a temporal table named *schedule_spec*, where a periodic schedule is defined in a manner reminiscent of *cron*, using intervals in datetime attributes.

Columns:

- `scheduleid`
- `scheduledweekday_from`, `scheduledweekday_to`
- `scheduleddate_from`, `scheduleddate_to`

- `scheduledmonth_from`, `scheduledmonth_to`
- `scheduledyear_from`, `scheduledyear_to`
- `scheduledhour_from`, `scheduledhour_to`
- `scheduledminute_from`, `scheduledminute_to`
- `duration_seconds`

Schedules defined in table *schedule_spec* are expressed as event relations by the view *scheduled_event*.

Scheduled_event makes use of *calendar* and *time-of-day* tables that are also defined in the file.

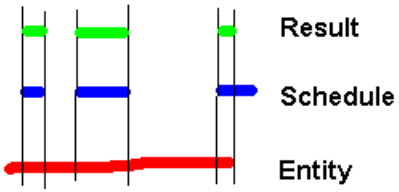
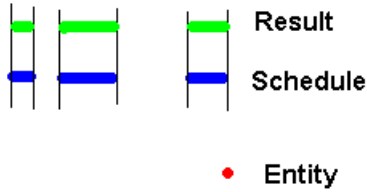
It adds the user-defined time columns `OnDate` of same value as `ValidOnDate`. It can be used as a tag when joining with state relations, in order to retain the scheduled event in the joined relation. Also, `OnDate` participates in the primary key of *scheduled_event* as a way to identify a specific event (e.g. the 8'o clock call).

Inline table-valued function *schedule_events_at* performs the same job but without a need for the schedule to be stored in *schedule_spec*.

For example, the following retrieves all Fridays that fall on the 13th during 2010.

```
SELECT ValidOnDate, datepart (weekday, ValidOnDate) FROM
dbo.schedule_events_at (
    6
    , 6
    , 13
    , 13
    , null
    , null
    , null
    , null
    , 0
    , 12
    , null
    , null
    , '2010-01-01'
    , '2011-01-01' );
```

Schedules are expressed as state relations by the view *scheduled_span*. Scheduled entities can be formed by joining any non-temporal entity with *scheduled_span*, or by temporal-joining a temporal entity with it.

 <p>Result Schedule Entity</p> <p>Figure 22: Entity scheduled using a Schedule</p>	<p>In the picture, an Entity is scheduled using a Schedule, resulting in Result, a scheduled entity defined by the sequenced join of both</p>
 <p>Result Schedule Entity</p> <p>Figure 23: Non-temporal Entity scheduled using a Schedule</p>	<p>In the picture, a non-temporal Entity is scheduled using a Schedule, resulting in Result, a scheduled entity defined entirely by the Schedule</p>

Scheduled_span makes use of *calendar* and *time-of-day* tables that are also defined in the file.

It adds user-defined time columns *SpanningFromDate* and *SpanningToDate*, of same value as *ValidFromDate* and *ValidToDate*. They can be used together as tags when joining with state or event relations, in order to retain the scheduled span in the joined relation. Also, *SpanningFromDate* participates in the primary key of *scheduled_span* as a way to identify a specific span (e.g. the 8'o clock shift).

When they form consecutive spans, *SpanningFromDate* and *SpanningToDate* can be used as grouping columns in Valid-time Cumulative Aggregation (see "Valid-time Partitioning").

Inline table-valued function *partition_at* is used to easily form consecutive spans based on a schedule. For example, the following returns spans between all Fridays of 2010.

```
SELECT ValidFromDate,ValidToDate FROM dbo.partition_at (
    6
    ,6
    ,null
    ,null
    ,null
    ,null
    ,null
    ,null
    ,0
    ,0
    ,0
    ,0
    , '2010-01-01'
```

```
, '2011-01-01' );
```

SQL Generation

A separate program, *GenerationTemplates*, interprets temporal metadata that are declared in the corresponding schema of your database, and produces an SQL script to run against the database.

Current version needs SQL Server 2005 and runs as: `GenerationTemplates <dbserver> <user> <password>.`

Beware that the generated script can be quite huge (tens of thousands of lines), depending on how long-winded are the dependence chains of the foreign key relationships.

The same procedure should be followed every time there is any change in the schema or the referential constraints, after cleaning up the temporal metadata (cf. Metadata management

Procedures `dbo.DeclareTemporalMetadata` and

`dbo.UndeclareTemporalMetadata` do exactly what their names imply.

Cleaning up).

Things to do beforehand

Please note that all modifications to the schema, that are simple to do in SQL92, are not included in the generated SQL, namely:

- Declaration of the special columns `ValidFromDate`, `ValidToDate` and `ValidOnDate`, of type 'datetime'
- CHECK constraint which ensures that `ValidFromDate < ValidToDate`
- PRIMARY KEY for event tables, which is a plain SQL92 PRIMARY KEY

Primary key support

For each primary key defined in state table `<Schema>.<Table>`, the following artifacts are produced:

- View `<Schema>.<Table>_PKViol`, which returns primary key violations.
- Procedure `<Schema>.ChkPK_<Table>`, which checks for primary key violations. In this, and every other checking procedure, care has been taken to produce meaningful error messages that report details about the primary key of the entity producing the error, as well as the offending period.
- Trigger `TR_PK_<Table>`, which calls the procedure on every modification to the table. In the case of a view, a separate one is defined for each base table.

Coalescing

Coalescing refers to normalizing the data to use as few rows as is possible, by way of merging rows together.

For every state table `<Schema>.<Table>` the following artifacts are produced:

- View `<Schema>.Coal_<Table>`, which returns a coalesced state relation.
- Procedure `<Schema>.Coalesce_<Table>` which uses the view to update the table contents with coalesced ones. Not produced for a view.

Contiguous History

For each table `<Schema>.<Table>` which has a Contiguous History constraint, the following artifacts are produced:

- View `<Schema>.<Table>_CHViol`, which returns contiguous history violations.
- Procedure `<Schema>.ChkCH_<Table>`, which checks for contiguous history violations.
- Trigger `TR_CH_<Table>`, which calls the procedure on every modification to the table. In the case of a view, a separate one is defined for each base table.

Constant

For each table `<Schema>.<Table>` which has a Constant constraint, the following artifacts are produced:

- View `<Schema>.<Table>_<ConName>Viol`, which returns Constant violations.
- Procedure `<Schema>.Chk<ConName>_<Table>`, which checks for Constant violations.
- Trigger `TR_<ConName>_<Table>`, which calls the procedure on every modification to the table. In the case of a view, a separate one is defined for each base table.

Unique

For each table `<Schema>.<Table>` which has a Unique constraint, the following artifacts are produced:

- View `<Schema>.<Table>_<ConName>Viol`, which returns Unique violations.
- Procedure `<Schema>.Chk<ConName>_<Table>`, which checks for Unique violations.
- Trigger `TR_<ConName>_<Table>`, which calls the procedure on every modification to the table. In the case of a view, a separate one is defined for each base table.

Foreign Key

For every foreign key defined from table `<FSchema>.<FTable>` to table `<PSchema>.<PTable>`, the following artifacts are produced:

View `<FSchema>.<FTable>_To_<PTable>_<FKName>`, which reports foreign key violations (`<FTable> where Not Exists <PTable>`).

Procedure `<FSchema>.Chk<FKName>_<FTable>_To_<PTable>` which checks for foreign key violations.

Trigger `TR_<FKName>_<FTable>_To_<PTable>` which calls the procedure on every modification to the foreign table. In the case of a view, a separate one of each is defined for each base table.

Trigger `FK_<FSchema>_To_<PSchema>_R` which calls the procedure on every modification to the primary table. In the case of a view, a separate one of each is defined for each base table.

Checking the integrity

For every table `<Schema>.<Table>`, a procedure `<Schema>.Chk_<Table>` is defined that calls all integrity-checking procedures on that table.

Also, a procedure called `dbo.CheckAll` does the same database-wide.

Delete in the presence of foreign keys

For each table `<Schema>.<Table>`, a procedure `<Schema>.DelFrom_<Table>` is defined, which deletes from a table honoring foreign key constraints recursively outwards from the given table. Current code makes use of a primary key value to identify what to delete from the table, but the code can be adapted easily to other kinds of conditions, even mentioning other tables. If the table is a non-updatable view, running it will raise an exception.

When not run inside a transaction, it creates one and rolls it back in case of an exception.

Update

For each table `<Schema>.<Table>`, a procedure `<Schema>.Upd_<Table>` is defined, which deletes from a table based on a primary key value. When the table has a contiguous history, simpler SQL is used. If the table is a non-updatable view, running it will raise an exception.

When not run inside a transaction, it creates one and rolls it back in case of an exception.

Metadata management

Procedures `dbo.DeclareTemporalMetadata` and `dbo.UndeclareTemporalMetadata` do exactly what their names imply.

Cleaning up

Procedure `dbo.CleanAll` removes all database entities created by the generated script, except itself.

Run-time support

The run-time support is included in the same executable, *Pandamator.dll*, which must be referenced by the .Net code wishing to use it (use “Add Reference” in Visual Studio).

ReadMetadata

In order to read the metadata from the database, one must create a connection string and call ReadMetadata:

```
Metadata.TemporalMetadata metadata =
Metadata.ReadMetadata(connString);
```

The returned value (here assigned to variable `tables_list`) holds the metadata and will be passed to all the other functions.

CreateSQLDeleteCascading

Function `CreateSQLDeleteCascading` creates a statement block that will act on a table, specified by a schema name and a table name, and delete from it based on a condition, specified in SQL. The condition must not reference any other temporal table and must use the marker `{0}` as a table alias. This marker will be substituted as needed in the generated text.

To execute this statement block, one must supply values for the parameters `@ValidFromDate` and `@ValidToDate`, which together form the deletion interval, and also for all parameters mentioned in the condition string.

The statement block takes care of all delete rules invoked by the deletion, like the generated procedure `DelFrom_<table>`, which is in fact created using the same SQL generation mechanism.

Example code in C# follows.

```
string deleteSql = Templates.CreateSQLDeleteCascading(metadata,
"dbo", "A", "{0}.val < @val");

SqlTransaction transaction = connection.BeginTransaction();

SqlCommand command = new SqlCommand(deleteSql, connection);

command.Parameters.Add(new SqlParameter("@ValidFromDate", new
DateTime(2008, 03, 10)));
command.Parameters.Add(new SqlParameter("@ValidToDate", new
DateTime(2008, 03, 20)));
command.Parameters.Add(new SqlParameter("@val", 100));
command.Transaction = transaction;

try
{
    command.ExecuteNonQuery();
    transaction.Commit();
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
    transaction.Rollback();
}
```

CreateSQLUpdateNonKey

Function `CreateSQLUpdateNonKey` creates a statement block that will act on a table, specified by a schema name and a table name, and update it based on a condition, specified in SQL, and an *update map*. The update map is a Dictionary having key-value pairs for all table attributes to update and associated expressions, specified as strings.

The condition and the update expressions must not reference any other temporal table and must use the marker `{0}` as a table alias. This marker will be substituted as needed in the generated text.

To execute this statement block, one must supply values for the parameters `@ValidFromDate` and `@ValidToDate`, which together form the deletion interval, and also for all parameters mentioned in the condition string and the update expressions.

The statement block works like the generated procedure `Upd_<table>`, which is in fact created using the same SQL generation mechanism.

Example code in C# follows.

```
Dictionary<string, string> strings = new Dictionary<string,
string>();

strings.Add("val", "{0}.val + 5");

string updateSql = Templates.CreateSQLUpdateNonKey(metadata, "dbo",
"A", strings, "{0}.val > 100");

SqlTransaction transaction = connection.BeginTransaction();

SqlCommand command = new SqlCommand(updateSql, connection);

command.Parameters.Add(new SqlParameter("@ValidFromDate", new
DateTime(2008, 03, 10)));
command.Parameters.Add(new SqlParameter("@ValidToDate", new
DateTime(2008, 03, 11)));
command.Transaction = transaction;

try
{
    command.ExecuteNonQuery();
    transaction.Commit();
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
    transaction.Rollback();
}
```

CreateCoalescingCTEs

Function `CreateCoalescingCTEs` creates a statement fragment with CTEs that define a coalesced relation, whose name is provided by the programmer, on a subset of a table's columns, using the recursive coalescing code described earlier.

Optionally, one can provide a filtering condition. This fragment can then be

prepended to a statement using the coalesced relation, like `select * from R`, or `select a, b, min(ValidToDate-ValidFromDate) from R into #temp group by a, b`.

Using Temporal SQL (experimental)

Function `make_sql92_from_temporal` creates SQL2 given a statement in ATSQL as defined in this text. As of this time, only SELECT statements are supported, and the support is experimental. Various restrictions apply, such as lack of support for sequenced subqueries inside non-sequenced ones.

Example code in C# follows.

```
var sql92_query = Transform.make_sql92_from_temporal(metadata,
atsqlcode);
```


Description of test data and test execution

Test data consist of a number of state and event tables linked by Foreign Key relationships with various ON DELETE rules. An auxiliary program, *ABCTimelineVizualizer*, uses a Simile Timeline widget inside an HTML page to help visualize the data. Although I have tried to manipulate the results in order to be presented in the most useful manner, you will notice that the results are sometimes bizarre-looking.

Sample "A, B, C" data

The following graph illustrates these relationships. Tables "D", "E" and "F" are event tables.

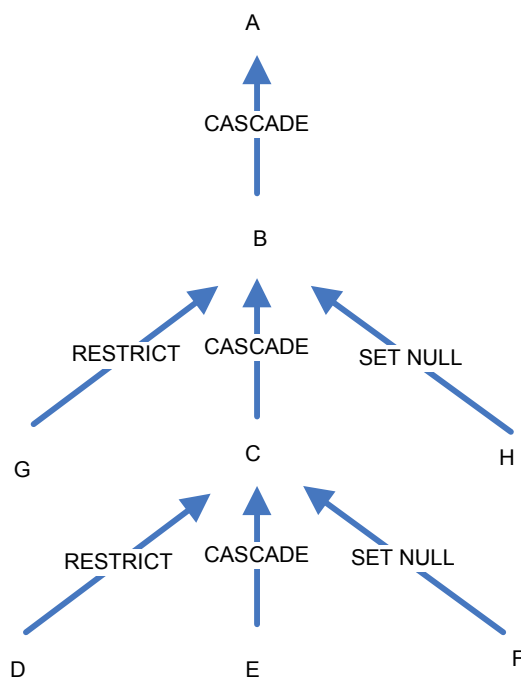


Figure 24: FK relationships of Sample "A, B, C" data

The following chart illustrates the sample data.

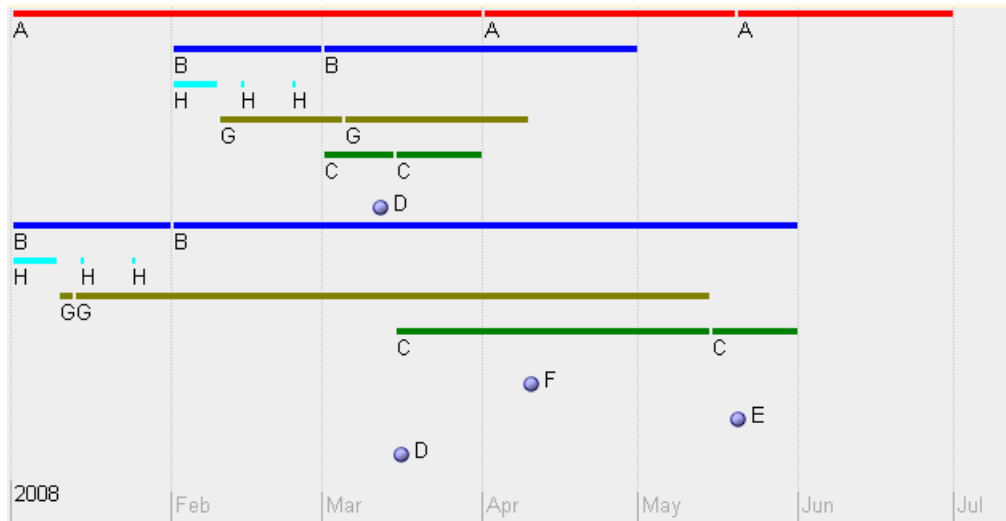


Figure 25: Chart of "A, B, C" Sample Data

The schema is created by *Sample-ABC-schema.sql* and the data insertion/re-insertion script is *Sample-ABC-data.sql*.

Calling the deletion procedure inside a transaction

We will call `DelFrom_A` with various arguments in order to attempt to delete A and verify the delete rules of the foreign keys that will be invoked.

```
DECLARE @RC int;
DECLARE @ValidFromDate datetime;
DECLARE @ValidToDate datetime;
DECLARE @id int;

set @ValidFromDate='2008-01-13';
set @ValidToDate='2008-03-10';
set @id=1;
set @RC = 0

begin transaction;

begin try;
EXECUTE [Temporal2].[dbo].[DelFrom_A]
    @ValidFromDate
    ,@ValidToDate
    ,@id;
end try;

begin catch;
set @RC = -1;
end catch;

if @RC = 0
commit
else
rollback;
```

In order to recover from failures, the conditional statement ensures the rollback of the transaction.

RESTRICT from D to C and from G to B

Trying to delete A from Jan 13th to March 20th, the RESTRICT delete rule from D to C takes over and prohibits this update.

Msg 50000, Level 16, State 2, Procedure DelFrom_A, Line 66
Transaction violates RESTRICT delete rule for foreign key from dbo.D to C while deleting from dbo.A, e.g. id=1 ValidOnDate="Mar 12 2008 12:00AM"

The presence of a D event on March 12th rolled back the whole operation. One can verify that no data have been modified.

Trying to delete up to March 10th avoids D, but stumbles on G from February 10th to March 1st.

Msg 50000, Level 16, State 2, Procedure DelFrom_A, Line 325
Transaction violates RESTRICT delete rule for foreign key from dbo.G to B while deleting from dbo.A, e.g. id=1 ValidFromDate="Feb 10 2008 12:00AM" ValidToDate="Mar 1 2008 12:00AM"

The reason that D is tested before G, is that the tree traversal of the foreign key relationships happens in the particular order.

CASCADE to B, C and E

Deleting from May 16th to July 20th avoids the objects that restrict deletion.

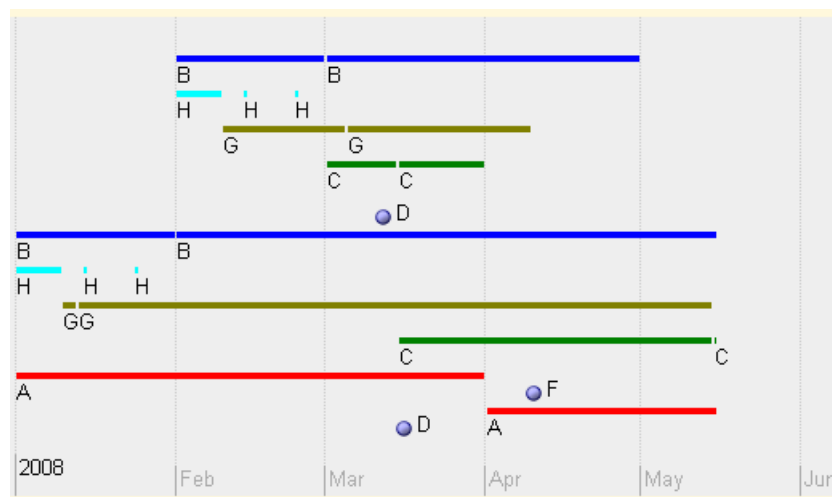


Figure 26: Cascade in "A, B, C" Sample Data

As you can see, B, C and E are deleted according to the CASCADE delete rule.

SET NULL on H and on F

Deleting A from January 4th to January 8th, cascades to B and SETs NULL on H.

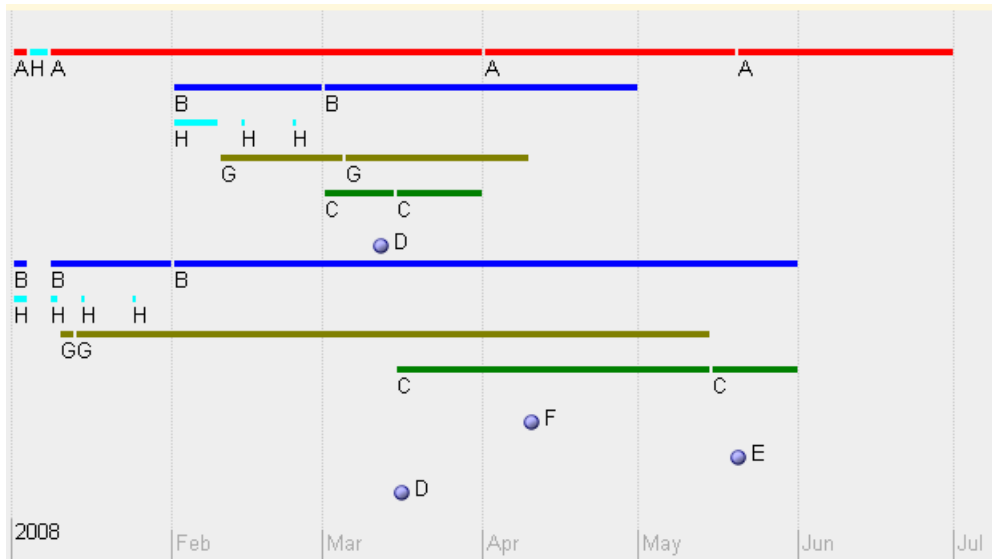


Figure 27: Set Null in "A, B, C" Sample Data (states)

The H that is shown out of place has a null foreign key to B.

Finally, in order to test SET NULL on F events, we will call `DelFrom_C` from April 5th to April 15th.

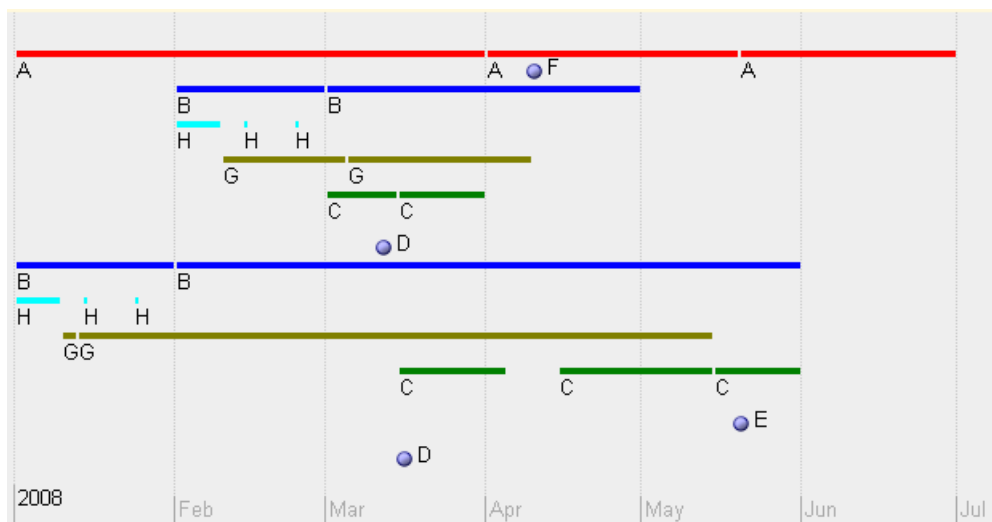


Figure 28: Set Null in "A, B, C" Sample Data (events)

The F that is shown out of place has a null foreign key to C.

References

Lorentzos

<http://portal.acm.org/citation.cfm?id=213476>

Jensen Thesis

<http://www.cs.aau.dk/~csj/Thesis/>

Snodgrass: Developing Time-Oriented Database Applications in SQL

<http://www.cs.arizona.edu/~rts/publications.html>

Andreas Steiner Thesis

<http://www.globis.ethz.ch/people/former/thesisSteiner.pdf>

TimeDB

<http://www.timeconsult.com/Software/Software.html>

Michael Boehlen Thesis

<http://www.sigmod.org/databaseSoftware/chronolog.txt>

(Download ChronoLog 3.0, the thesis is included)

Consensus Glossary of Temporal Database Concepts – February 1998 Version

<http://infolab.usc.edu/csci599/Fall2001/paper/glossary.pdf>

SIMILE Timeline

<http://www.simile-widgets.org/timeline/>

Teradata Temporal Option

<http://www.teradata.com/t/database/Teradata-Temporal/>

Appendix A: Query Transformation Example (Not Coalesced)

The query that is transformed, is the following, where “a”, “b” and “c” are all state relations. Notice the “keyword” FULL that instrau

```
FULL VALIDTIME
SELECT b.id,a.value
FROM b
INNER JOIN a ON b.a_id=a.id
WHERE not exists(SELECT * FROM c WHERE c.value<a.value)
```

The SQL92 query, that results from the transformation, is the following. Note that legible indentation and bracketing are not yet very high in the priority list...

```
WITH spanboundaries(TheDate) AS (
  (SELECT ValidFromDate AS TheDate
  FROM b)
  union
  ((SELECT ValidToDate AS TheDate
  FROM b)
  union
  ((SELECT ValidFromDate AS TheDate
  FROM a)
  union
  ((SELECT ValidToDate AS TheDate
  FROM a)
  union
  ((SELECT ValidFromDate AS TheDate
  FROM c)
  union
  (SELECT ValidToDate AS TheDate
  FROM c))))))
),
spans(ValidFromDate,ValidToDate) AS (
  SELECT a.TheDate AS ValidFromDate,b.TheDate AS ValidToDate
  FROM spanboundaries AS a
  INNER JOIN spanboundaries AS b ON a.TheDate<b.TheDate and not
  exists(SELECT *
  FROM spanboundaries AS c
  WHERE a.TheDate<c.TheDate and c.TheDate<b.TheDate)
)
SELECT b.id AS t0,a.value AS t1
FROM spans
INNER JOIN b ON b.ValidFromDate<=spans.ValidFromDate and
spans.ValidToDate<=b.ValidToDate
INNER JOIN a ON (b.a_id=a.id)and(a.ValidFromDate<=spans.ValidFromDate
and spans.ValidToDate<=a.ValidToDate)
WHERE not exists(SELECT *
FROM c
WHERE (c.value<a.value)and(c.ValidFromDate<=spans.ValidFromDate and
spans.ValidToDate<=c.ValidToDate))
ORDER BY ValidFromDate,ValidToDate;
```

Appendix B: Query Transformation Example (Coalesced)

The query that is transformed, is the following, where “a”, “b” and “c” are all state relations.

```
VALIDTIME
SELECT b.id,a.value
FROM b
INNER JOIN a ON b.a_id=a.id
WHERE not exists(SELECT * FROM c WHERE c.value<a.value)
```

The SQL92 query, that results from the transformation, is the following.

```
WITH spanboundaries(TheDate) AS (
  (SELECT ValidFromDate AS TheDate
   FROM b)
 union
  ((SELECT ValidToDate AS TheDate
   FROM b)
 union
  ((SELECT ValidFromDate AS TheDate
   FROM a)
 union
  ((SELECT ValidToDate AS TheDate
   FROM a)
 union
  ((SELECT ValidFromDate AS TheDate
   FROM c)
 union
  (SELECT ValidToDate AS TheDate
   FROM c))))
),
spans(ValidFromDate,ValidToDate) AS (
  SELECT a.TheDate AS ValidFromDate,b.TheDate AS ValidToDate
  FROM spanboundaries AS a
  INNER JOIN spanboundaries AS b ON a.TheDate<b.TheDate and not
  exists(SELECT *
  FROM spanboundaries AS c
  WHERE a.TheDate<c.TheDate and c.TheDate<b.TheDate)
),
R(t0,t1,ValidFromDate,ValidToDate) AS (
  SELECT b.id AS t0,a.value AS t1
  FROM spans
  INNER JOIN b ON b.ValidFromDate<=spans.ValidFromDate and
  spans.ValidToDate<=b.ValidToDate
  INNER JOIN a ON(b.a_id=a.id)and(a.ValidFromDate<=spans.ValidFromDate
  and spans.ValidToDate<=a.ValidToDate)
  WHERE not exists(SELECT *
  FROM c
  WHERE(c.value<a.value)and(c.ValidFromDate<=spans.ValidFromDate and
  spans.ValidToDate<=c.ValidToDate))
),
Coal(t0,t1,ValidFromDate,ValidToDate) AS (
  (SELECT p2.t0 AS t0,p2.t1 AS t1
  FROM Coal AS p1
  WHERE not exists(SELECT R.*
```

```

FROM R AS p1
WHERE (p1.t0=p2.t0 or coalesce(p1.t0,p2.t0) is null)and(p1.t1=p2.t1 or
coalesce(p1.t1,p2.t1) is null)and p1.ValidToDate = p2.ValidFromDate
GROUP BY t0,t1,ValidFromDate)
GROUP BY t0,t1,ValidFromDate)
union
(SELECT p2.t0 AS t0,p2.t1 AS t1
FROM R AS p1
INNER JOIN R AS p2 ON p1.ValidToDate = p2.ValidFromDate
and(p1.t0=p2.t0 or coalesce(p1.t0,p2.t0) is null)and(p1.t1=p2.t1 or
coalesce(p1.t1,p2.t1) is null))
),
Coalesced(t0,t1,ValidFromDate,ValidToDate) AS (
SELECT p1.t0 AS t0,p1.t1 AS t1
FROM R AS Coal
GROUP BY t0,t1,ValidFromDate
)
SELECT *
FROM Coalesced;

```