Maintaining Time Span Information in a Relational Database

J. Trevathan and W. Read

Griffith University,
170 Kessels Road, Brisbane, Australia 4111

j.trevathan@griffith.edu.au

Abstract

Current database practitioners cannot easily store information about transient phenomena using existing commercial off-the-shelf tools, because vendors do not support temporal database characteristics. This paper describes a practical problem where a relational database must record a series of events that occur over a given time span, without any modification to the underlying database management system. The database must be able to perform queries on the temporal information of an event occurrence. However, the database must also manage overriding events which take precedence over previously scheduled events, while maintaining knowledge of the overridden event. A basic approach to the problem is discussed, which is conceptually simple but cannot efficiently record the required information, nor can it cope with complex overriding event scenarios. We propose a second approach that models the events using time spans. The time span approach is then refined to create a third scheme that can efficiently handle complex overriding event scenarios. The two time span approaches result in significant storage gains compared to the first approach. However, the storage gains come with a trade-off in efficiency for data manipulation queries. The proposed approach is implementable with current relational standards (and SQL), and does not have the overhead and complexity of a full-scale temporal database.

Keyword: Temporal databases, time spans, events, SQL.

I. Introduction

Temporal databases are an active area of database research that provides functionality to keep track of events or phenomena over a time range (see [1, 3, 5, 6, 7, 8, 16, 17, 19, 20, 24, 27, 32, 33]). While temporal databases are well suited to specific applications (i.e., those dealing with historical data), most existing databases still use the standard *relational* (or *object-relational*) approach (see [11]). Relational databases often need to track temporal phenomena, but do not need all of the features and sophistication of a temporal database. Furthermore, temporal databases require specific formatting of data and a temporal query language [8, 17], which is not supported by most database management systems. Temporal databases also lack ability for keeping track of temporal phenomena when knowledge must be retained about events that have been overridden by a newer event.

This paper describes a problem where a standard relational database must record a series of events that occurs over a given time span. A simple approach would be to use one record per time unit. For example, ('event A', Day 1), ('event A', Day 2), ('event A', Day 3), ('event B', Day 4), ('event B', Day 5), ('event C', Day 6), ('event C', Day 7), ('event C', Day 8), ('event C', Day 9). However, a major concern for this approach is regarding storage efficiency. Given n time units, this yields a space complexity of $O(n)$. Furthermore, the database has to be able to perform queries that indicate during which times the events occurred, and must be able to cope with overriding events which take precedence over previously scheduled events. In the more complicated scenario, given m amendments to the schedule results in a space complexity of $O(n+m)$, plus additional storage required for the extra database fields in a record.

This paper presents an approach to storing temporal events in a standard unmodified *Relational Database Management System* (RDBMS), which uses time spans rather than individual records for

each time unit. That is, each event has a time span associated with it that indicates which time units are applicable. For example, the above database would become ('event A', Days 1 - 3), ('event B', Days 4, 5), ('event C', Days 6 - 9). This results in storage savings, but comes with the trade-off of increased transaction costs. We then propose a more sophisticated scheme that keeps track of overridden events and is extendable to complex overriding event scenarios. This fulfils a function that temporal databases were not designed for. Furthermore, the temporal functionality is achieved using a standard unmodified RDBMS, rather than a temporal database. This approach has not been fully explored by the literature. Note that this problem has many practical applications. This paper presents two scenarios to illustrate the need to maintain time-based events and/or overriding events within a standard database environment. The first scenario describes a medical database that stores data on cancer patients taking part in clinical trials. The second scenario is a counter-terrorism information system that tracks the history of, and integrates intelligence on, 'people of interest' to predict and manage potential future threats (events).

The paper is organised as follows: Section 2 illustrates how the problem addressed in this paper is applicable to real-world database scenarios involving the capture and retention of temporal information. Section 3 provides background on temporal databases and related work. Section 4 introduces a mathematical model for our proposal, and describes complex scenarios that must be catered for. Section 5 presents a solution and implementation to the problem using a relational database. Section 6 proposes a comprehensive solution that keeps track of complex overriding and overridden events. Section 7 gives a complexity comparison for the proposed approaches presented in this paper. Section 8 provides some concluding remarks.

II. Practical Temporal Database Scenarios

This section presents two use-case scenarios where information is recorded on time-based events. In both cases, there is a requirement to maintain the data not only on the overriding events, but also on the original event that is being overridden.

A. A Medical Database for Clinical Trials

Consider a medical database for clinical trials of cancer therapy. The information to track includes the methods of chemotherapy (e.g., intravenously, orally, dermal application, etc.) in conjunction with targeted therapies (e.g., inhibiting drugs, immunotherapy, etc.). A prescribed mix of chemotherapy type and targeted therapy are enforced over a period of days. The dosages are changed after a specified number of days until the end of a course. For ease of reference, we will denote this prescribed mix of medications as colours. For example, intravenous chemotherapy with high doses of enzyme inhibitor drugs are applied on Days 1, 2 and 3, and are represented with a red code. The dosages are decreased for Days 4 and 5 (represented by a green code), and again for Days 6, 7, 8 and 9 (represented by a blue code). We need to answer questions such as "on which days was a red treatment used?", or "how many days did was the blue treatment in use for?".

This problem can be generalised as, given a set of events or phenomena, we want to record what length of time those events/phenomena occurred, and be able to retrieve this information using a *standard relational query language*. The basic approach may be to use a single record to represent the event for each time unit. That is, a single record is used to store the fact that on Day 1, a red code was used. Likewise, another record is used to store that a red code was used on Day 2. The result would be the following:

```
Day 1, red
Day 2, red
Day 3, red
Day 4, green
Day 5, green
Day 6, blue
Day 7, blue
Day 8, blue
Day 9, blue
```

The database schema for such a table (referred to as "Record") is:

Record(RecordID, TimeUnit, Event)

where *RecordID* is the primary key, *TimeUnit* refers to the date, and *Event* is the phenomenon of interest that we are recording temporal information about.

By using this approach, performing queries is conceptually simple. For example, to work out which days a green code was used would involve using the following SQL query:

```
SELECT TimeUnit, Event
FROM Record
WHERE Event = 'green'
```

This query would return the following results:

```
Day 4, green
Day 5, green
```

This simple approach can also be extended to cope with scenarios that are more complex. For example, suppose that a red treatment code was scheduled to be used on Days 1, 2 and 3. However, after the second day, the patient developed a side effect caused by the targeted therapy rather than the chemotherapy, and the targeted drug prescription was discontinued. This forces a "ceased treatment-event" code (i.e., for the sake of this example is an orange code) to be used on Day 3, rather than a red code. The database can be easily altered to reflect this fact by updating Day 3 to indicate an orange code is in effect. Likewise, any other amendment to the treatment code schedule can be made in a similar manner.

However, this approach lacks power when more stringent rules must be applied. For example, a business rule that requires an audit of which treatments were scheduled regardless of the code actually used that day. So let us assume that a red treatment code caused a reaction at the end of Day 2, and as a result, an orange treatment code was used on Day 3. The database must retain knowledge that a red treatment code was scheduled for Day 3, but must also reflect that an orange code was actually used on Day 3. Each event would require additional information in the table schema on whether it was the originally scheduled event, or whether it has been overwritten by a newer event.

B. Counter Terrorism Information System

Consider a counter-terrorism information system that collects intelligence on 'people and organisations of interest' to predict potential future threats (events). This information system holds intelligence gathered via various surveillance methods on persons and organisations that have

affiliations with known terrorist activities. The information could be used to predict possible locations and times of attacks, and the suspects that may be involved. For example, a planned visit from a foreign dignitary coincides with an increase in suspicious meetings, movements and purchases. The predictions would be used for counter insurgency plans for surveillance on suspects, and the system would track the distribution of manpower and resources in various locations over a period of days. The dignitary's itinerary may initially be covered by taskforce X at location A on Days 1, 2 and 3, which is represented by the red code. Then taskforce Y covers location B for Days 4 and 5 (green code) and taskforce X covers location C for Days 6, 7, 8 and 9 (blue code).

An overriding event (represented by the orange code) may occur if there is an unexpected change in the dignitary's schedule, such as an unplanned visit to location D on day 3. The taskforce (X) covering location A would not move, but instead a backup taskforce (taskforce Z) is deployed to the new location (D). Records of both events must still be maintained in the database.

The underlying difference between this scenario and the medical database scenario is the events are predicted to occur, and contingencies are planned based on the predictions, as opposed to the orderly scheduling of drug treatments. However, both scenarios have the common requirement that information on events, which have been overridden by new events, must be preserved.

III. Background and Related Work

This section provides a brief background on temporal databases, the concepts and terminology used, alternate approaches to dealing with temporal phenomena, and the implications for the problem presented in this paper.

A. Temporal Databases and Problem Positioning

A temporal database has built-in time aspects, e.g., a temporal data model and a temporal version of SQL (see [8, 17, 18, 19, 24, 27, 32]).

The temporal aspects are as follows:

- *Valid time* denotes the time period during which a fact is true with respect to the real world.

- *Transaction time* is the time period during which a fact is stored in the database.
- *Bitemporal data* combines both valid and transaction time.

Note that these two time periods do not have to be the same for a single fact. Consider a temporal database that stores data about the 18th century. The valid time of these facts is somewhere between 1700 and 1799, whereas the transaction time starts when facts are inserted into the database, for example, January 21, 2013.

Table 1. Example Temporal Database Table

EmpID	Name	Department	Salary	ValidStartTime	ValidEndTime
10	John	Electrical	$11,000	1985	1990
10	John	Hardware	$11,000	1990	1993
10	John	Hardware	$12,000	1993	INF
11	Graeme	Electrical	$15,000	1988	1995
12	Sally	Electrical	$30,500	2001	INF
13	Bob	Hardware	$35,500	2005	INF

Table 1 is an example of a temporal database valid time table that stores the history of the employees with respect to the real world. The attributes ValidTimeStart and ValidTimeEnd represent a time interval which is closed at its lower and open at its upper bound. From the table, we can work out that during the time period (1985 - 1990), employee John was working in the Electrical department, with a salary of $11,000. Then he changed to the Hardware department, still earning $11,000. In 1993, he got a salary raise to $12,000. The upper bound INF denotes that the tuple is valid until further notice. Note that it is now possible to store information about past states. We see that Graeme was employed from 1988 until 1995. In the corresponding non-temporal table, this information would be (physically) deleted when Graeme left the company.

A temporal database is insufficient for the examples depicted in this paper. Firstly, we are seeking to maintain information regarding overridden events. A temporal database simply overwrites this information by either deleting the record, or modifying a record's valid time and inserting the new record. Secondly, the problem tackled in this paper needs to be implemented using standard SQL (i.e., Transact SQL or PL/SQL), rather than a temporal version of SQL. That is, a regular RDBMS

is the preferred implementation because there is no requirement to hold bitemporal data or support bitemporal operations in the manner required by a temporal database.

An important concept in temporal database research is *current time* (i.e., now, the present) (see [10, 12]). All activity is trapped in current time. Current time separates the past from the future. The spatial equivalent is here. "Here" does not share the properties of "now". Time cannot be reused, whereas space can. That is, it is possible for two objects to occupy the same point in space at two different times, whereas it is not possible for two events to occupy the same point in time. The problem addressed in this paper pushes the properties of "now", in that two or more objects can occupy the same point in time. However, only one of the objects is valid.

B. Related Work and Problem Justification

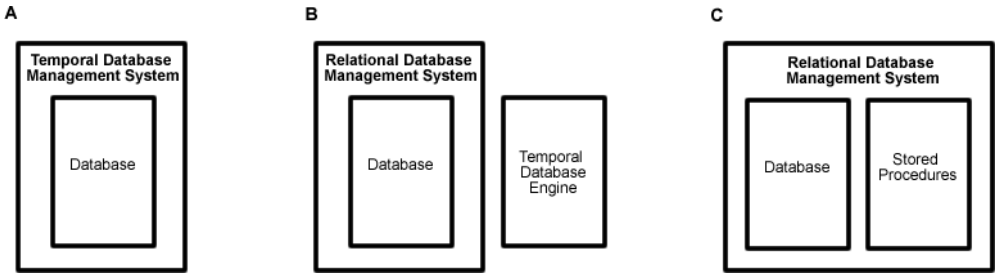


Figure 1. Various approaches to implementing temporal functionality using database management systems. (A) shows a complete temporal database. (B) is a relational database with an accompanying temporal database engine to provide temporal functionality. (C) is the approach explored by this paper that extends a relational database using SQL stored procedures.

There has been a significant amount of research conducted on temporal databases. This includes defining the characteristics/format of temporal data [1, 2, 4, 5, 7, 13, 20, 24, 26, 32], query and processing operations involving bitemporal data [3, 6, 8, 12, 14, 18, 28], and performance analyses of temporal database systems [4, 26]. However, despite the advanced state of the research, no current commercial database vendors support the entire range of temporal database functions (refer to Figure 1A). This limits practitioners to use existing tools to support temporal-style operations.

There are only three practical approaches available to database practitioners at present. The first is to not attempt to model time-span information in any sophisticated way, and therefore only record the most basic information using the Date data type. However, this is insufficient, lacks any sort of

temporal-querying power, and is ignorant of the work already established by temporal database researchers.

The second approach is investigated by [21, 23] who modify an existing commercial database engine to support temporal applications (refer to Figure 1B). They propose a system called *Immortal DB* which is built into Microsoft SQL Server, and integrates a temporal indexing technique referred to as the TSB-tree (see [22]). This system is notable in that it is a high-performance transaction time database system which was built into a RDBMS engine – which no one had achieved or documented before. However, the main concern with this approach is that it is tied specifically to a particular vendor, and any changes to the underlying engine would require substantial modification to the system. Furthermore, practitioners are required to install what are essentially two separate database systems to support temporal operations.

C. Our Contribution

This paper investigates the practicality of adopting a third approach of using software in conjunction with an existing database. The underlying database engine does not require any modification, and the solution is transportable between different systems. The practitioner only requires his/her database engine and SQL version of choice to implement temporal features (refer to Figure 1C). While this approach has been criticised as being cumbersome and impractical, it has never been fully explored and no rigorous analysis performed on its merits. This paper's novelty is in investigating such an approach.

The closest known proposals for this approach are by [30, 31, 15]. [30] define a temporal data model based on generalising a non-temporal data model into a temporal one. Using generalisation means that all constructs of the underlying non-temporal data model - its data structures, operations and integrity constraints are enhanced to support the management of time-varying data. Stiener investigates three approaches to managing temporal data and provides corresponding prototype implementations.

[31] describes the concept of *temporal processing middleware*. They developed a bitemporal extended software platform upon a non-temporal RDBM. This is based on an extension to TimeDB1 and allows translation from the temporal query language ATSQL2 [9] to standard SQL. [15] also present a similar system using Java RMI (Remote Method Invocation) technology. However, these proposals are just preliminary and further work is required on refining the system.

The proposal in this paper differs from previous related work in that it is not attempting to recreate a full temporal database and its associated functionality. This paper aims at only providing a subset of temporal operations without the need to interact with any complete bitemporal data structures and temporal versions of SQL. Furthermore, this is achieved by using a layer of software that can be considered as being between middleware and the database engine. The proposal uses existing RDBMS data structures and the operations can be compiled as a set of stored procedures that are part of the database (Figure 1C). Finally, the proposal concentrates on the efficiency of storing/manipulating time spans and the problem of maintaining knowledge of overridden and overriding events.

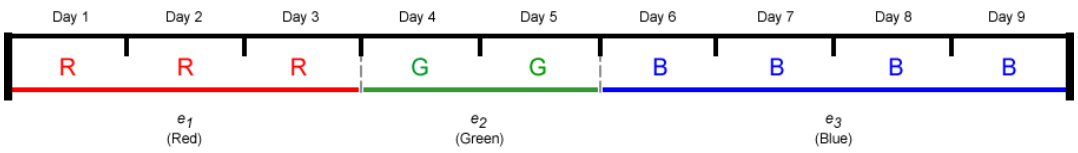
IV. Using Time Spans

This section introduces a mathematical model upon which the proposed time span approach is based. The database schema is described and the complex scenarios the approach must cope with are discussed.

A. Basic Mathematical Model

The following notation is used throughout this paper.

Time is an ordered set $T = \{t_1, t_2, ..., t_\infty\}$ of time units t_i where $1 \leq i \leq \infty$.



[1] www.timeconsult.com/

Figure 2. Treatment code example using the time span approach.

$E = \{e_1, e_2, ..., e_n\}$ is a set of events. T is split up into disjoint subsets of events where $T = e_1 \cup e_2 \cup ... \cup e_n$. For each $e \in E$ there is a time span (t_i, t_j) where $i \leq j$. Figure 2 graphically illustrates the ongoing treatment code example where e_1 - (t_1, t_3), e_2 - (t_4, t_5) and e_3 - (t_6, t_9) correspond to red, green and blue treatments respectively. Note that an event for a given time unit is atomic. That is, a red code cannot be in effect for half a day and a green code used for the other half.

For completeness, we introduce the concept of a null event. A null event (denoted as e_ϕ) is a time span that has no event associated with it. For example, a chemotherapy treatment might only have a five-day course that can only be administered during week days. In this case a treatment is not required on weekends, therefore Saturday and Sunday is the null event. A null event is not stored in the database. The absence of an entry for a time span indicates the presence of a null event.

Throughout T there may be numerous null events. For example, if an entire year of treatment codes are scheduled, there will be *52* null events, one for each weekend (assuming the hospital does not close for holidays, which in themselves would result in further null events). Let $E^\phi = \{e^\phi_1, e^\phi_2, ..., e^\phi_m\}$ be the set of null events. We can now say that $E \cup E^\phi$ forms a partition on T. That is, $T = e_1 \cup e_2 \cup ... \cup e_n \cup e^\phi_1 \cup e^\phi_2 \cup ... \cup e^\phi_m$ and $e_i \cap e_j = \phi$, $i \neq j$.

B. Database Schema

The database schema for a table using the time span approach is

Record(RecordID, Event, StartDate, EndDate, DateTimeStamp)

where *RecordID* is the primary key, Event is the phenomena of interest that we are recording temporal information about, *StartDate* and *EndDate* represent the time span for the event (i.e., valid time), and *DateTimeStamp* is the time the entry was inserted into the database (i.e., transaction time).

C. Basic Event Scenarios

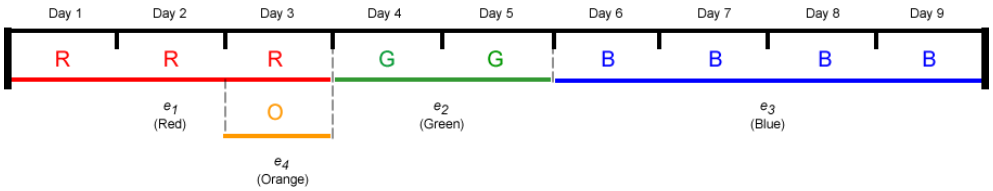


Figure 3. Examples of an overriding event (orange – day 3) and a shadow event (red – day 3).

The time span approach must deal with the problem of **overriding events**. That is, the situation in the example where the red code becomes unusable at the end of Day 2 and an orange code must now be used on Day 3 (rather than the scheduled red code). Figure 3 graphically depicts this situation. The database entries remain e_1 - (t_1, t_3), e_2 - (t_4, t_5) and e_3 - (t_6, t_9) corresponding to red, green and blue respectively. However, we now include e_4 - (t_3, t_3) for the orange code overriding event.

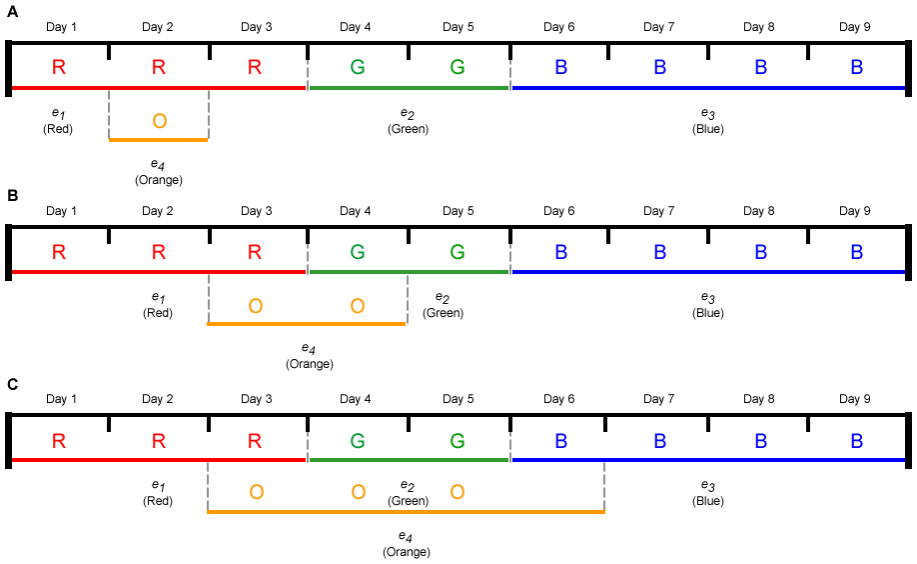


Figure 4. Examples of more complicated overriding event and shadow event scenarios.

A **shadow event** (or overridden event) refers to an event (or the section of an event) that has been overridden. Figure 4 shows some further examples of overriding events for the treatment code example. Figure 4A illustrates an overriding event that occurs entirely within the time span of a shadow event. Figure 4B illustrates an overriding event that spans two shadow events. Figure 4C illustrates an overriding event that spans three shadow events, one of which is completely overwritten by the overriding event. A side effect from the inhibitor drug could be the cause of these overriding events where the drug is discontinued or the dosage changed.

Let e denote the shadow event and ε the overriding event. e_0 and e_n denote the starting and ending times for e. ε_0 and ε_n denote the starting and ending times for ε. The following scenarios exist for overriding events:

1. ε is within e, $(e_0 \leq \varepsilon_0 \wedge \varepsilon_n \leq e_n)$
2. ε starts within e but finishes after e, $(e_0 \leq \varepsilon_0 \wedge e_n \leq \varepsilon_n)$
3. ε starts before e but ends within e, $(\varepsilon_0 \leq e_0 \wedge \varepsilon_n \leq e_n)$
4. ε completely overrides e, $(\varepsilon_0 \leq e_0 \wedge e_n \leq \varepsilon_n)$
5. ε completely overrides more than one e

At present, the time span model cannot satisfy the types of queries outlined in Section 1. That is, given the scenario in Figure 4, if the database was queried for which days a red code was used, the result returned would include e_1 in its entirety (including the shadow event portion).

D. Cascading Overriding Events

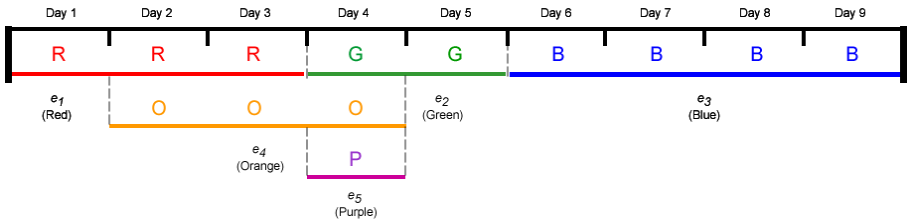


Figure 5. An example of a cascading overriding event (purple – day 4) and shadow event scenarios (orange and red – day 4).

The problem is compounded with the presence of **cascading overriding events**. Figure 5 illustrates an example of a cascading overriding event. Here, orange overrides red, however, purple overrides part of the orange event. An orange overriding event could be the discontinuation of the inhibitor drug, and a purple overriding event could be the prescription of a drug to manage the symptoms of the side effects. In this situation, the database must retain knowledge of the red shadow event, knowledge of the orange overriding event, knowledge of the purple overriding event, and also knowledge of the portion of the orange shadow event. It is important to note that *overriding events can be nested up to any level*.

Consider the special scenario where red is overridden by orange, which is in turn overridden by red again. What happens in this case? Orange could be scaled back, i.e., removed from the database so that the event defaults back to red. However, we want to retain knowledge that orange was included (otherwise there would be no record that a side effect had occurred). In addition, scaling back events may become difficult when the level of nesting is large, and where events span portions of multiple shadow events.

V. Simple Time Span Solution that Replaces Overridden Events

This section describes an approach for implementing the time span where overriding events replace shadow events in the database. That is, consider the situation where ε is contained entirely within e. e is split into two separate database entries e_1 and e_2, which correspond to the time spans before and after ε. ε is inserted directly between e_1 and e_2, so that the time spans appear contiguous (i.e., e_1, ε, e_2) and there is no overlap. While this solution loses the information regarding shadow events, it is used to aid in the construction of the comprehensive approach (described in Section VI). The simple time span approach can be implemented using a relational database.

A. One Overriding Event

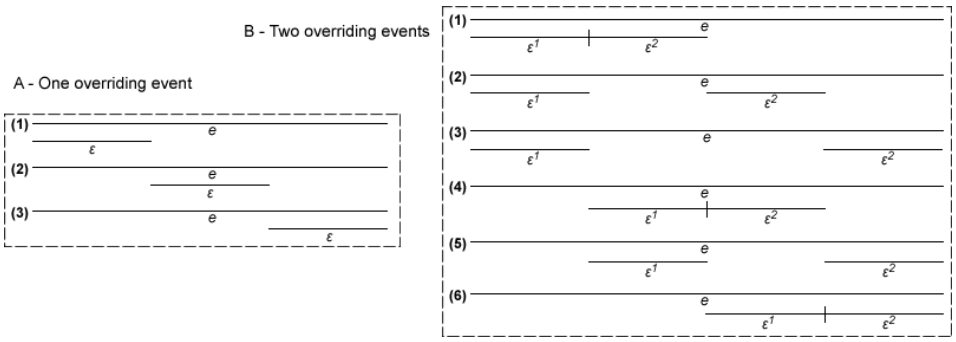


Figure 6. Scenarios for (A) one overriding event and (B) two overriding events.

This section examines the scenarios for **one** overriding event (denoted ε) that is contained entirely within a shadow event e. Figure 6A graphically illustrates the three scenarios that can occur. These extend scenario 1 in Section IV C. The following describes the algorithms that are used to update the database for each scenario:

Algorithm 1

ε begins at the same time as e, but ends before e:

$$\varepsilon_0 = e_0 \wedge \varepsilon_n < e_n$$

1. Update e
   (a) Set $e_0 = \varepsilon_n + 1$
2. Insert ε

Algorithm 2

ε begins after e and ends before e:

$$e_0 < \varepsilon_0 \wedge \varepsilon_n < e_n$$

1. Split e into e_1 and e_2
   (a) Set $e^1_0 = e_0$
   (b) Set $e^1_n = \varepsilon_0 - 1$
   (c) Set $e^2_0 = \varepsilon_n + 1$
   (d) Set $e^2_n = e_n$
   (e) Delete e
2. Insert e_1
3. Insert e_2
4. Insert ε

Algorithm 3

ε begins after e and ends at the same time as e:

$$e_0 < \varepsilon_0 \wedge \varepsilon_n = e_n$$

1. Update e
(a) Set $e_n = \varepsilon_0 - 1$
2. Insert ε

The scenario where ε completely overrides e may also occur, and will be discussed later.

B. Generalising the Number of Overriding Events

First let us examine the scenarios for two overriding events (denoted ε^1 and ε^2) that are contained entirely within a shadow event e. Figure 6B graphically illustrates the five scenarios that can occur. These extend scenario 1 in Section IV C.

In general, given n overriding events the number of possible combinations for overriding events that are within an event appears to be bounded by the following recurrence:

Proof: Assume that for $n = k$

$$f_k = \sum_{j=0}^{k} f_j = 2^{k-2}(f_0 + f_1),\ k = 2$$

$$f_{k-1} = \sum_{j=2}^{k} 2^{k-2}\ (f_0 + f_1) + f_0 + f_1$$

$$= (f_0 + f_1)\ x \sum_{j=2}^{k} 2^{k-2} + (f_0 + f_1)$$

$$= (f_0 + f_1)\ [\sum_{j=0}^{k-1} 2^{j-1} + 1]$$

$$= (f_0 + f_1)\ \{\frac{2^{k-1}-1}{2-1} + 1\}$$

$$= (f_0 + f_1)\ \{2^{k-1}-1 + 1\}$$

$$= 2^{k-1}(f_0 + f_1)$$

In order to generalise the algorithm for overriding events completely contained within *e*, there are four important factors:

- Start time;
- End time;
- Space between overriding events; and
- Consecutive overriding events (absence of space between two overriding events).

Contrasting the algorithms for *one overriding event* (Section V B) and *two overriding events*, there is an apparent trend. The algorithms for two overriding events simply extend those of one overriding event, and also incorporate the unique cases that occur with the increased level of complexity. Rather than defining a *generalised* algorithm to take into account the complexity with increasing numbers of overriding events, a simple generalised algorithm can be deduced. This algorithm is based on the idea that multiple overriding events are just a special case of one overriding event. There is no need to define complex procedures to apply all overriding events at once when it makes no difference whether the overriding events are applied individually.

Consider the following scenario with three overriding events (ε^1, ε^2, ε^3). Apply ε^1 using the algorithms in Section V A, then do the same for ε^2 and ε^3. Once one overriding event has been applied, *e* is either updated or split into two new events e^1 and e^2. In the first instance, the second overriding event can be applied to *e* without any regard for the changes resulting from the first overriding event. In the second instance, the second overriding event can be applied to either e^1 or e^2 (as if it were the original *e*), without any regard for the changes resulting from the first overriding

event. All that needs to be determined before each ε is applied is which e the overriding event belongs to.

Given n overriding events, the generalised algorithm is defined as follows:

1. For each ε^i from 1 to n do
    (a) Determine which e is overridden by ε^i
    (b) If $\varepsilon^i_0 = e_0 \wedge \varepsilon^i_n < e_n$
        i. Apply Algorithm 1
    (c) Else If $e_0 < \varepsilon^i_0 \wedge \varepsilon^i_n < e_n$
        i. Apply Algorithm 2
    (d) Else $e_0 < \varepsilon^i_0 \wedge \varepsilon^i_n = e_n$
        i. Apply Algorithm 3
    (e) $i = i + 1$

C. Overriding Events that Span Multiple Shadow Events

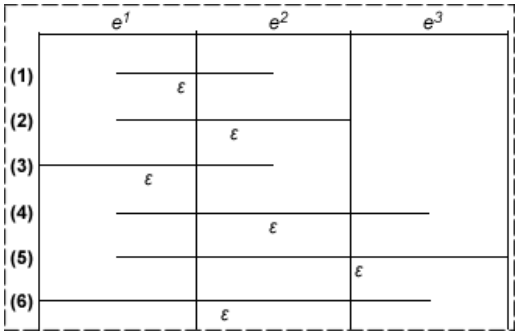


Figure 7. Basic scenarios for overriding events that span multiple shadow events.

Figure 7 shows the scenarios for overriding events that span multiple shadow events. Here there are only three shadow events, (e^1, e^2, e^3). Once the completely overridden events are removed, only the beginning and ending shadow events are important. These can be reduced to the overriding event scenarios from Section 7. The algorithm for dealing with overriding events that span multiple shadow events is as follows:

1. Work out how many events ε spans
    (a) Query for events where $e_0 < \varepsilon_0 \wedge e_n < \varepsilon_n$
2. For each e completely overridden by ε
    (a) Delete e
3. For two partially overridden events
    (a) Determine which case 1, 2, or 3
    (b) Apply the appropriate algorithm

D. Cascading Overriding Events

Each ε in a cascading overriding event sequence has an ordering associated with it. Run the algorithm from Section V C on the first ε in the sequence. Then run the algorithm on the next ε, etc.

E. Self-Overriding Events

In the situation where an event overrides itself, the event is referred to as *self-overriding*. A self-overriding event either extends or reduces its time span. When inserting a self-overriding event, the application logic must ensure that the database does not lose *temporal cohesion*. That is, span times for two or more events cannot overlap as a result of the insertion/deletion process.

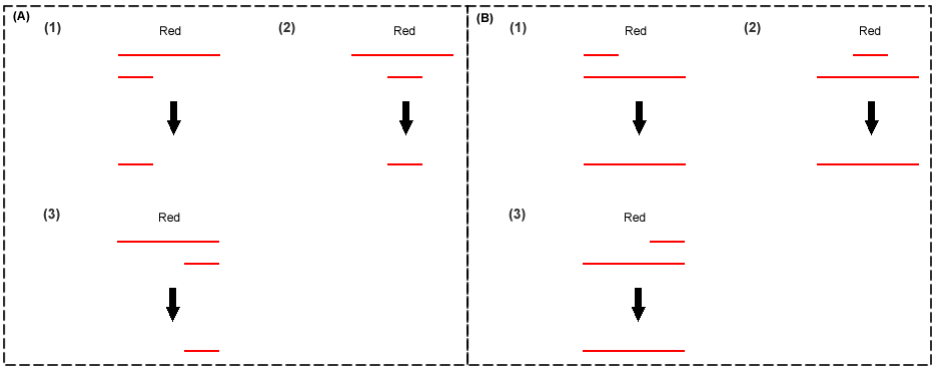


Figure 8. Scenarios for (A) a self-overriding event that reduces its time span and (B) a self-overriding event that extends its time span.

Figure 8A illustrates the basic scenarios for a self-overriding event that reduces its time span. Figure 8B illustrates the basic scenarios for a self-overriding event that extends its time span. There are three major points for these scenarios:

- ε starts at the same time as *e*;
- ε starts within *e* (for reduction), or before and after *e* (for extension); and
- ε ends at the same time as *e*.

Figure 9 illustrates the basic scenarios for a self-event that simultaneously reduces and extends its time span. That is, (a) the event amends its start time to be later (*reduction*), while at the same time it delays its end time (*extension*), or (b) the event amends its start time to be earlier (*extension*), but finishes earlier (*reduction*). These scenarios are more complicated than the aforementioned self-

overriding scenarios as both reduction and extension operations must occur. Furthermore, this figure illustrates how neighbouring events are affected. The extension operation must update the overridden portion of the neighbouring event, whereas the reduction operation yields a null event proportional to the reduction.

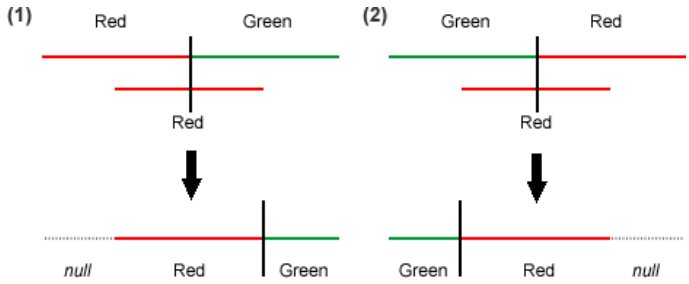


Figure 9. Scenarios for self-overriding events involving reduction and extension.

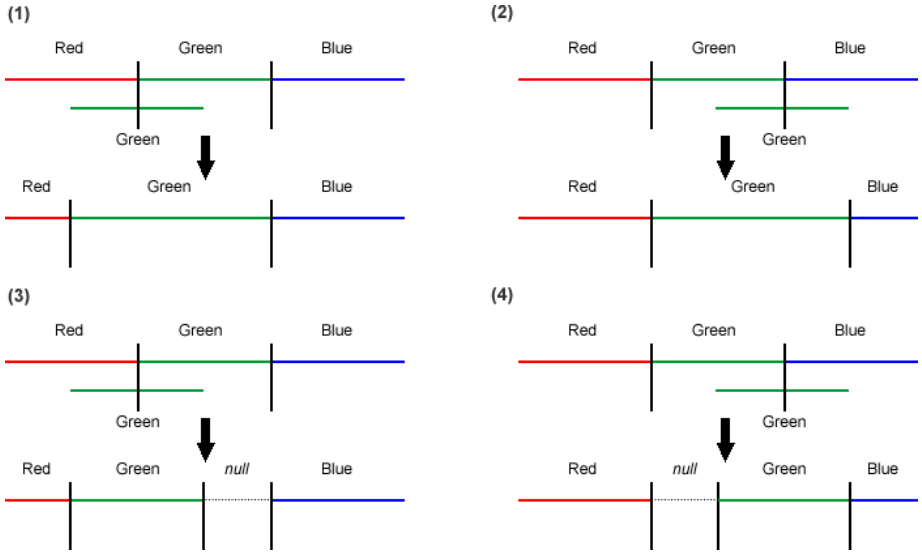


Figure 10. Clarification of reduction and extension scenarios involving multiple events. Scenarios (1) and (2) are incorrect.

Figure 10 illustrates the reduction and extension scenarios with three events. Scenarios (1) and (2) incorrectly show the outcome of extending the green event. These scenarios have failed to perform the reduction operation. Scenarios (3) and (4) show the correct outcomes whereby a null event is introduced into the portion of the self-overriding event that was reduced. Note that it is permissible to have gaps between events (i.e., null events), but not overlapping events (i.e., loss of temporal cohesion).

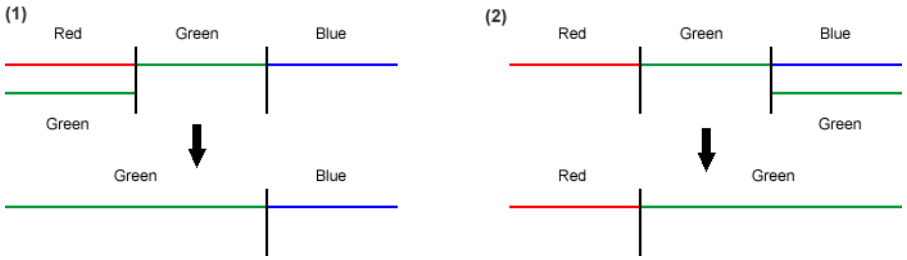


Figure 11. Self-overriding events that join with neighbouring events.

Figure 11 illustrates two new special cases of self-overriding events where an overriding event joins together with an existing "neighbouring" event of the same type. In this situation, the neighbouring event's time span must be extended to incorporate the overriding event's time span. The end result should be that only the existing event is updated, and that the overriding event is not stored (i.e., there is only one contiguous record in the database). Obviously, any overridden event must also be updated proportional to the time span of the overriding event (e.g., the red and blue events in Figure 11 (1) and (2) respectively).

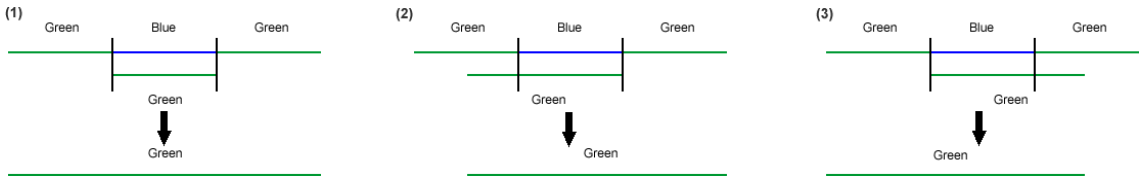


Figure 12. Scenarios for self-overriding events that bridge together self-overriding events.

Figure 12 illustrates an extended situation of the previous case where a self-overriding event neighbours two events of the same type. This is referred to as a bridging event. The bridging event must essentially join both neighbouring events into one contiguous event.

To create one contiguous event from scenario (1) in Figure 12 involves updating the (chronological) first event's end time to be equal to the end time of the third event. The bridging event is not stored (as its time span is now reflected by the first event), and the third event is deleted. The outcome should result in only one record being stored in the database which contains the time spans of all three events.

Scenarios (2) and (3) require a reduction operation to be performed. In Scenario (2), the first event's start time is updated to reflect the bridging event's start time (reduction), and the end time is updated to reflect the third event's end time (extension). In Scenario (3), the first event retains its start time (extension in a sense), but its end time must be updated to only reflect the bridging event's end time rather than the third event's end time (reduction).

VI. Comprehensive Time Span Solution that Retains Overridden Events

The simple time span approach presented so far is unable to keep track of shadow events, as they are removed as soon as an overriding event is entered. This section describes an implementation for the comprehensive time span approach, which retains knowledge of the shadow events.

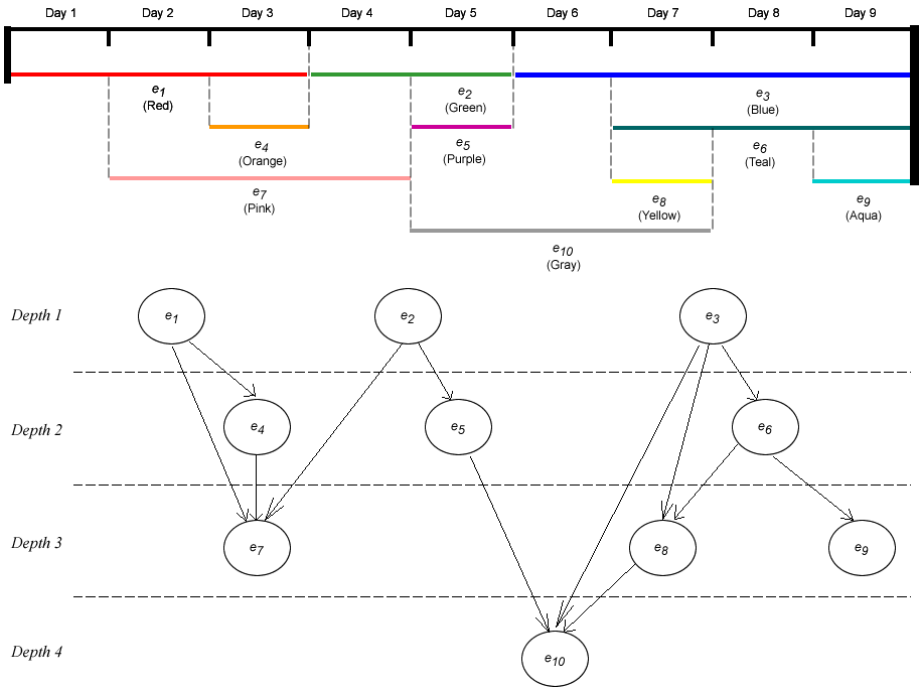


Figure 13. An example of how time spans can be represented using a graph. Shadow events point to their immediate relevant overriding events (i.e., first level overriding events).

A. Modelling the Problem

The problem can be considered as a directed graph $G=(V,E)$, where V is the set of events and E is the set of edges between events (refer to Figure 13). An edge between two events v_1, v_2 indicates that v_2

directly overrides some portion of v_1 (or v_1 entirely). The direction of an edge indicates which event is the overriding event and which is the shadow event.

To implement the graph structure, a new table is required. However, this table must be as efficient as possible to reduce storage space and make querying easy for insertions, updates and deletions of time spans. The table is referred to as *OverrideMeta* and has the following schema:

OverrideMeta(Overridden, Overrider)

Here *Overridden* is a foreign key for the shadow event and *Overrider* is the foreign key for the overriding event. The primary key is the composite of *{Overridden, Overrider}*. This will always be unique. The number of entries in the *OverrideMeta* is denoted as $|E|'$.

The basic procedure for recording information about an overridden event *e* is as follows:

1. Insert *(e, ε)* into the *OverrideMeta* table

To determine which event(s) directly override e, the system just needs to perform a query on the *OverrideMeta* table and search for *e*. Note that this only keeps track of directly overriding events for *e* (referred to as **first level overriding events**), it does not specifically record information about cascading overriding events that override *e* at deeper levels of the graph. This scenario will be addressed later.

B. Inserting New Events

i) First Level Overriding Events

A new event to be inserted is denoted as ε'. At this point it is unknown whether it is an overriding event or not, hence the distinction from the previous definition of ε.

The first step for inserting ε' is to query the Record table for all the events that might occupy part of the time span for ε'. If none do, then ε' does not override anything and we can go ahead and insert it.

Now consider the case where there is one *e* for which ε' either partially or entirely overwrites. The system must then follow the procedure outlined in Section VI A by inserting *(e, ε')* into the *OverrideMeta* table. This approach can also be generalised for any number of shadow events for ε'.

That is, if $\{e_1, ..., e_m\}$ (where $2 \leq m \leq n-1$) denotes the set of shadow events for ε', then insert the corresponding records into the *OverrideMeta* table as follows: $\{(e_1, \varepsilon'), (..., \varepsilon'), (e_m, \varepsilon')\}$.

ii) Cascading Overriding Events

There are two main problems when dealing with cascading overriding events:

- We need a way of determining which events have been entirely overridden and exclude them from further search results.
- We need to work out which shadow events may contain time span portions that are not overridden and return them as candidates in the search results.

When an event is overridden and a new entry is added to the *OverrideMeta* table, the system needs to determine whether the event has been completely overridden (referred to as a buried event). A **buried event** is no longer visible to any new events being inserted. In order to keep track of whether a shadow event is buried, the Record table schema is modified as follows:

Record(RecordID, Event, StartDate, EndDate, IsBuried, DateTimeStamp)

where *IsBuried* indicates whether the record is visible to any future ε'.

The process for determining whether e is a buried event is as follows:

1. After inserting (e, ε') into OverrideMeta
   (a) Return all records from OverrideMeta for *e*
   (b) Coalesce the time periods for each ε associated with *e* (ignoring overlaps between εs) to form one contiguous time span
   (c) If the coalesced time span is larger than or equal to *e*'s time span
      i. Set IsBuried to true for *e*

Discussions and algorithms for coalescing time spans can be found in [6, 12].

iii) Generalised Insertion

The procedures from Sections VI B i and VI B ii can be combined to create a compact generalised algorithm for inserting records that override other events.

Algorithm - Override

1. For each e_i from *1* to *m* do
   (a) Insert (e_i, ε') into *OverrideMeta*
      i. Return all records from *OverrideMeta* for e_i

ii. Coalesce the time periods for each ε associated with *ei*
iii. If the coalesced time span is larger than or equal to e_i's time span
A. Set *IsBuried* to true for e_i

iv) The Final Insertion Procedure

The following procedure combines the previous approaches to handle all insertion scenarios for the comprehensive time span solution:

1. Insert ε' into Record
2. Query for all *e*s where *IsBuried* is false and the time span overlaps ε'
3. If results are returned
(a) Apply Algorithm – Override

C. Updating Existing Events

There are several scenarios to take into account when updating an existing record for an event. Each is dealt with in turn below.

i) No Effect

If the record being updated does not affect any other records, then there is nothing more to do.

ii) Creates New Shadow Events

A record must be inserted into *overrideMeta* for each event that becomes a shadow event to ε'. We must then check to see whether the e becomes buried. Fortunately, this is as simple as running Algorithm – Override. The only check in place is to ensure that existing shadow events for ε' aren't modified.

iii) Removes Existing Shadow Events

Note here we use ε rather than ε' as it is known that this is an overriding event. It must be determined if *e* is no longer a shadow event for anything, and also if *e* is no longer buried. Determining the former will imply the latter, but the relationship is not symmetric.

1. Query *OverrideMeta* for all events associated with ε
2. For each e_i from *1* to *m* do
(a) Return all records from *OverrideMeta* for e_i
(b) If no records are returned
i. Delete record for e_i, ε from *OverrideMeta*
ii. Set *IsBuried* to false for e_i
iii. exit
(c) else

i. Coalesce the time periods for each ε associated with *ei*
ii. If the coalesced time span is smaller than e_i's time span
   A. Set *IsBuried* to false for e_i

iv) Deleting Events

Deleting an event is a similar process to that described in Section VI C iii. First it must be determined whether this is an overriding event. If so, any affected shadow events should be examined to see if they are no longer shadow events, or buried events. The event can then be deleted from *Record*.

Note that a deletion will usually only happen if a record is erroneous or not required. Otherwise, events will get buried over time if the database is to be used in the manner consistent with the theme of this paper where knowledge of overridden events is retained.

E. Querying Events

There are three main types of queries that will occur within the context of the database this paper is proposing:

1. On which days did a particular event occur?
2. How many days did a particular event span?
3. Which events were originally scheduled for a particular time period but were overridden by a new event?

Table 2. Summary of the Result Sets and Operations for Queries 1 and 3.

Query 1		Query 3	
RS1	All events that are not buried	RS1	All events within the time span
RS2	List of overriding events	RS2	List of overriding events
RS3	Time spans for overriding events	TRS	Results in RS2 that are not in RS1
		RS3	Overridden events and originally scheduled events that have not been overridden

With regard to query 1, we are only interested in the event that actually occurred on the days in question – not any events that were scheduled, but were overridden. To perform this query, the database must scan the Record table for the event, and omit any instances of the event where *IsBuried* is true. We will refer to this initial result set as *RS1*. Next, the *OverrideMeta* table must be

queried to see whether any of the event instances from *RS1* have overriding events. We will refer to this second result set as RS2. Any overriding event results returned in the *RS2* query need to be individually searched for in the *Record* table. The portions of the time for the overriding event are then deducted from *RS1*. This third result set, *RS3*, gives the results for query 1. (Refer to Table 2 for a summary of the operations.)

Query 2 is essentially the same process as described above for query 1. However, for the basic approach, a count operation is performed on the records returned for an event to determine how many days the event spanned. For the simple and comprehensive time span approaches, a date operation to undertaken on *RS3* to determine the number of days in for which an event occurred on.

Query 3 is interested in any event that was partially or completely overridden by a new event. To perform this query, the database must scan the *Record* table for any event instances that fall within the desired time span. We will refer to this initial result set as *RS1*. Next, the *OverrideMeta* table must be queried to see whether any of the event instances from *RS1* have overriding events. We will refer to this second result set as *RS2*. Any events in *RS1* that are not found in *RS2* are removed from *RS1*, and are placed in a temporary result set *TRS*. However, it may be possible that an event does not have an entry in the *OverrideMeta* table as it is the originally schedule event, and it has no overriding events. Therefore, the database must make this final check on all events in *TRS* for this condition. Any events from *TRS* that satisfy this condition are returned to *RS1* to give the final answer to query 3 (*RS3*). (Refer to Table 2 for a summary of the operations.)

VII. Complexity Analysis

This section provides a complexity analysis of the three approaches for the time span problem presented in this paper. The algorithms are referred to as the *Basic* (Section II), *Simple Time Span* (Section V) and *Comprehensive Time Span* (Section VI) respectively. Each algorithm is assessed on its storage and computational complexity, and its suitability for being implemented by a relational

database with SQL. The following sections describe how the analysis was conducted for each performance indicator.

A. Storage

Storage complexity is based on the number of records that must be stored to represent a series of time spans. The number of event records is denoted as $|E|$ (i.e., cardinality of event set, E) and is bounded by the total number of time units $O(|T|)$ (i.e., cardinality of time set, T).

Using the basic approach of one record per time unit gives an overall storage complexity of $O(|T|)$. Both time span approaches result in a storage complexity of $O(|E|)$, where $0 \leq |E| \leq |T|$ for the simple time span approach, and $0 \leq |E| \leq \infty$ for the comprehensive approach. The only situation where either time span approach would perform as poorly as the basic approach is when there are as many events as there are time units. It is conceivable that the comprehensive approach may perform worse than the basic approach and the simple time span approach when the depth of cascading overriding events becomes extremely deep (hence the bounding by ∞). However, this storage is required to retain knowledge of the shadow events, which is not captured by the other approaches (nor a temporal database).

The average storage complexity for the simple time span and comprehensive time span approaches will significantly differ. In the simple time span approach, when e is buried, it is deleted from the database. This reduces the simple time span approach's storage requirements in this respect to the comprehensive approach. However, knowledge of the originally scheduled event is lost, which is the reason for constructing the comprehensive approach.

Now, consider the situation in the simple time span approach where ε is contained entirely within e. e is split into two separate database entries e^1 and e^2 which correspond to the time spans before and after ε. ε is inserted directly between e^1 and e^2 so database appears as e^1, ε, e^2. In this situation, three entries are required as compared to two for the comprehensive approach. When the depth of cascading overriding events is minimal, the comprehensive approach will generally outperform the simple time span approach.

B. Insert/Update

Insert/Update complexity is determined by the number of database entries that must be altered to insert or update an individual ε.

For the basic approach, this is analysed in terms of n which is the number of time units in *T* that the time span represents. All n individual records must be updated. That is, one record per time unit. This gives the basic approach an insertion complexity of *O(n)* updates, where $n \leq |T|$. However, it may be the case that when an insertion or update occurs, an existing event may have to be either partially or entirely deleted. Therefore, we bound the basic scenario by *O(2n)* in the worst case where all the records from an existing event must be deleted to accommodate a new event.

For the time span approaches, inserting a single record for ε reduces this complexity to *O(1)* per update. However, we must now take into account the modifications that are made to the overridden events. In the simple time span approach, when a single ε is contained entirely within e, there are *O(3)* updates required. Where ε spans multiple events, there are *O(d+3)* updates required, where *d* is the number of deletions for any shadow events that are completely overridden by ε.

The comprehensive time span approach's insert complexity only depends on the number of events directly overridden by ε. This is regardless of the whether ε spans multiple events, or its depth in cascading overriding events (buried events are irrelevant as they are already pointing to their immediately overriding events). Let ι denote the number of first-level events overridden by ε. The comprehensive approach has an insertion complexity of $O(\iota + 1)$, where ι is the cost of recording that ε now overrides the first level events, and *1* is the cost of inserting ε itself.

When $d+3 < n$ the simple time span approach will always outperform the basic approach for the number of insertions required. In terms of the simple and comprehensive time span approaches, *d* and ι are referring to two separate concepts. *d* is the number of shadow events completely overridden by ε, whereas ι is the number of first-level events overridden by ε (either completely or partially). It is interesting to note that $(d+3) = (\iota + 1)$. Both the simple time span and comprehensive time span approaches modify the same number of records. However, the performance of both

approaches are not entirely equivalent. For the comprehensive time span approach, the complexity of the querying for shadow events and the coalescing algorithm needs to be taken into account. Therefore, both the time span approaches have higher transactional costs than the basic approach, with the comprehensive approach being largely dominated by the complexity of the coalescing algorithm.

C. Delete

Delete complexity is determined by the number of database entries that must be altered to delete an event. For the basic approach, n records must be removed proportional to the number of time units in the event. The simple time span approach only requires *1* entry to be removed. As knowledge of overridden events is not retained in the simple time span approach, there are no other records/events affected by the deletion.

The comprehensive time span approach requires ι deletions from the *OverrideMeta* table for any first level events that are no longer being completely or partially overridden. The event being deleted must also be removed from the Record table, which gives the comprehensive approach an overall complexity of $\iota+1$. However, consider the cascading events scenario where e_1 is overridden by e_2, which is overridden by e_3 (assuming some portion of e_1 and e_3 overlap). If e_2 is being deleted, then e_1's record in the *OverrideMeta* table must be updated to point to the portion of its timespan that e_3 now overrides in the absence of e_2. Therefore, there may be some additional processing required by the comprehensive approach depending on the scenario for where in the directed graph the event is being deleted.

D. Query

Query complexity is the computational cost of retrieving specific information from the database. This complexity is being analysed in terms of how many records must be accessed and manipulated during the query. The three queries from Section VI E being analysed are:

1. On which days did a particular event occur?
2. How many days did a particular event span?

3. Which events were originally scheduled for a particular time period but were overridden by a new event?

For query 1, the basic approach requires all the records to be searched in the worst case scenario. This is bounded by the $|T|$, therefore the complexity is $O(|T|)$. The simple time span approach must inspect all events in E to determine which are the correct instances of the event being sought. This gives the simple time span approach a complexity of $O(|E|)$. The comprehensive time span approach must inspect all events in E to obtain *RS1*. Then all records must be searched in the *OverrideMeta* table to remove those events that have been overridden from the result set (i.e., *RS2*). After this a final scan of the Record table must take place to find the time spans for the overriding events to obtain *RS3*. This gives the comprehensive approach an overall complexity of $O(2|E|+|E|')$.

In query 2, the same operations that occurred for query *1* are largely needed for all three algorithms. The basic approach performs a count on the number of records returned, and has a complexity is $O(|T|)$. The simple time span approach performs a date operation on the number of records returned in *RS3*, and its complexity is dominated by $O(|E|)$. The comprehensive time span approach also performs a date operation on the number of records returned in *RS3*, and its complexity is bounded by $O(2|E|+|E|')$.

Neither the basic approach or the simple time span approach are capable of performing query 3, as they do not retain knowledge of overridden events. The comprehensive time span approach must first scan all $|E|$ records to determine the events falling within the desired time span (i.e., *RS1*). Next, all $|E|'$ records of the *OverrideMeta* table are searched to determine the list of overriding events (i.e., *RS2*). The *TRS* is determined by searching for events that are contained in *RS1* but are not in *RS2*. Finally, all $|E|$ events in the *Record* table must be examined again to determine which events are originally scheduled, but do not have overriding events. This gives the comprehensive approach an overall complexity of $O(2|E|+|E|')$. We feel that the TRS operation is negligible and will be dominated by the $2|E|+|E|'$ operations.

E. Discussion and Design Justification

Table 3. Summary of the Complexity Analysis for the Three Approaches to Recording Time Span Information Using a Relational Database.

Approach	Storage	Insert/Update	Deletion	Query		
				Query 1	Query 2	Query 3
Basic	$O(\|T\|)$	$O(2n)$	$O(n)$	$O(\|T\|)$	$O(\|T\|)$	-
Simple TS	$O(\|E\|)$	$O(d+3)$	$O(1)$	$O(\|E\|)$	$O(\|E\|)$	-
Comprehensive TS	$O(\|E\|+\|E'\|)$	$O(\iota+1)$	$O(\iota+1)$	$O(2\|E\|+\|E'\|)$	$O(2\|E\|+\|E'\|)$	$O(2\|E\|+\|E'\|)$

Table 3 presents the complexities for each algorithm across the various performance indicators. Prior to the analysis, we assumed that the simple and comprehensive time span approaches would result in storage gains over the basic approach, at the expense of increased transaction costs. However, the analysis shows that for the insert/update and delete transactions, the basic approach actually performs poorly. Therefore, the time span approaches are more successful in terms of storage costs and basic transaction costs. With regard to the query complexity, there is increased overhead for both time span approaches in contrast to the basic approach. However, the comprehensive approach is the only algorithm that can perform the types of queries desired by the objectives of this paper. As such, we believe we have succeeded in our goals of showing that temporal functionality can be achieved in a relational database.

Each of the three approaches were implemented using Microsoft SQL Server and its TransactSQL language. The implementation drove our development of the two time span approaches as practical challenges were raised and overcome. The goal of the implementation exercise was to determine how easy it is for each approach to be implemented using the common tools provided with a relational database, and the difficulty for a *Database Administrator* (DBA) to maintain the information.

Our anecdotal experience suggests that basic approach is the easiest to implement using a standard relational database. It is conceptually simple. However, its storage size presents a major problem, and as was shown in the complexity analysis, implementing some of the transactions is not overly efficient/ideal. From a DBA perspective, understanding and maintaining a database using this

approach soon becomes cumbersome for a large number of records. However, the advantages of this approach are that no sophisticated application logic is required, and a DBA can easily comprehend the meaning of the raw table entries.

The simple time span approach is more powerful than the basic approach. It is also conceptually simple, and a DBA can easily comprehend the raw table entries. However, this approach requires a set of procedures to be run for automated entry, updating, and querying of time span data. This can reasonably be implemented using programming extensions to SQL (i.e., transact SQL or PL/SQL).

The comprehensive time span approach is not conceptually simple, and is more difficult for a DBA to comprehend by observing the raw table entries. Furthermore, it requires quite extensive stored procedures to be created. However, the comprehensive approach is the most powerful out of all three approaches and does show that temporal functionality can be added to a relational database at the stored procedure level.

VIII. Conclusions

This paper described a problem where a RDBMS must record a series of events that occurs over a given time span. The database is able to perform queries that indicate during which times the events occurred, and can cope with overriding events, which take precedence over previously scheduled events. This is a practical problem that cannot use a temporal database management system (as there are no commercially available systems), nor can the user make any modification to the database engine itself. The solution is implemented in a software layer above the DBMS, and is seamlessly integrated as a set of stored procedures. The feasibility of such an approach has not been previously explored in the literature.

Three solutions of increasing complexity were proposed. A basic approach to the problem is discussed which is conceptually simple. However, the basic approach also cannot efficiently record the required information, nor can it cope with complex overriding event scenarios. A second more sophisticated scheme was proposed that models the events using time spans. This approach results

in storage gains and can be implemented as a set of SQL stored procedures without any modification to the DBMS (as other solutions in the literature have done). However, the simple time span approach does not retain knowledge of overridden events. The third comprehensive time span approach extends the simple time span approach to support complex overriding event scenarios. The comprehensive solution is based on a directed graph and uses an additional table to keep track of events' temporal relationships amongst each other. We believe that the proposed comprehensive solution fulfils all of the paper's stated objectives.

Results from a complexity analysis indicates there is a trade-off between storage space and computational complexity. The time span approach saves database storage space, but increases the costs of database operations for insertion, updating and deletion. Furthermore, there is a degree of 'involvedness' to establish and use the system, which might make implementation initially difficult for database practitioners. However, overall this paper shows that it is feasible to add temporal operations to an existing RDBMS without modification to its engine. Furthermore, the proposed system has functionality that is not supported within the existing temporal database theory.

Future work involves determining whether a more efficient solution can be found for implementing the comprehensive time span approach. Furthermore, it would be useful to see what other types of temporal operations could be implemented via this approach such as union, merge, intersect, expand, collapse, etc.

Acknowledgement

The authors would like to thank Rui Zhang and Bruce Litow for their insightful comments.

References

[1] J.F. Allen, Maintaining Knowledge about Temporal Intervals, Communications of the ACM, 26(11), 1983, 832-843.

[2] C.H. Ang and K.P. Tan, The Interval B+tree, Information Processing Letters, 53(2), 1995, 85-89.

[3] G. Arumugam and M. Thangaraj, An efficient multiversion access control in a Temporal Object Oriented Database, Journal of Object Technology, 5(1), 2006, 105-116.

[4] B. Becker and S. Gschwind and T. Ohler and B. Seeger and P. Widmayer, An Asymptotically Optimal Multiversion B+tree, VLDB, 5(4), 1996, 264-275.

[5] M.H. Bohlen and R. Busato and C.S. Jensen, Point-Versus Interval-Based Temporal Data Models, Fourteenth International Conference on Data Engineering, 1998, 192-200.

[6] M.H. Bohlen and R.T. Snodgrass and M.D. Soo, Coalescing in Temporal Databases, 22nd International Conference on Very Large Data Bases, 1996, 180-191.

[7] T. Bozkaya and M. Ozsoyoglu, Indexing Valid Time Intervals, International Conference on Database and Expert Systems Applications, 1998, 541-550.

[8] J. Chomicki, Temporal Query Languages: A Survey, Temporal Logic, First International Conference, 1994, 506-534.

[9] J. Chomicki and D. Toman and M.H. Bohlen, Querying ATSQL databases with Temporal Logic, ACM Transactions on Database Systems, 26(2), 2001, 145-178.

[10] J. Clifford and C. Dyreson and T. Isakowitz and C. Jensen and R. Snodgrass, On the Semantics of “Now” in Databases, ACM Transactions on Database Systems, 22(2), 1997, 171-214.

[11] E.F. Codd, A Relational Model of Data for Large Shared Data Banks, Communications of the ACM, 13(6), 1970, 377-387.

[12] C. Dyreson, Temporal Coalescing with Now Granularity, and Incomplete Information, SIGMOD, 2003, 169-180.

[13] R. Elmasri and G. Wun and V. Kouramajian, The Time Index and the Monotonic B+tree, Chapter 18, 1993, 433-456.

[14] H. Gunadhi and A. Segev, Efficient Indexing Methods for Temporal Relations, IEEE Transactions on Knowledge and Data Engineering, 5(3), 1993, 496-509.

[15] C. He and S. Li, Construction of Temporal Data Management Platform Based on Extensions to Non-Temporal RDBMS, 2nd International Workshop on Database Technology and Applications (DBTA), 2010, 1-5.

[16] C.S. Jensen, Introduction to Temporal Database Research, PhD Thesis, 2000.

[17] C. Jensen and R. Snodgrass, Temporal Data Management, IEEE Transactions on Knowledge and Data Engineering, 11(1), 1999, 36-44.

[18] C. Jensen and D. Lomet, Transaction Timestamping in (Temporal) Databases, International Conference on Very Large Data Bases, 2001, 441-450.

[19] N. Kline, An Update of the Temporal Database Bibliography, SIGMOD Record, 1993, 66-80.

[20] V. Kouramajian and I. Kamel and R. Elmasri and S. Waheed, The Time Index+: An Incremental Access Structure for Temporal Databases, Third International Conference on Information and Knowledge Management (CIKM`94), 1994, 296-303.

[21] D. Lomet and R. Barga and M. Mokbel and G. Shegalov and R. Wang and Y. Zhu, Immortal DB: Transaction Time Support for Sql Server, SIGMOD, 2005, 939-941.

[22] D. Lomet and R. Barga and M. Mokbel and G. Shegalov and R. Wang and Y. Zhu, Transaction Time Support Inside a Database Engine, IEEE International Conference on Data Engineering (ICDE), 2006, 35.

[23] D. Lomet, M. Hong, R. Nehme and R. Zhang, Transaction Time Indexing with Version Compression, VLDB Endowment, 2008, 870-881.

[24] G. Ozsoyoglu and R. Snodgrass, Temporal Data Management, IEEE Transactions on Knowledge and Data Engineering, 7(4), 1995, 513-532.

[25] Plattner and A. Wapf and G. Alonso, Searching in Time, SIGMOD, 2006, 754-756.

[26] B. Salzberg and V. Tsotras, Comparison of access methods for time-evolving data, ACM Computing Surveys, 31(2), 1999, 158-221.

[27] M. Sao, Bibliography on Temporal Databases, SIGMOD Record, 1991, 14-23.

[28] H. Shen and B-C Ooi and H. Lu, The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases, International Conference on Data Engineering, 1994, 274-281.

[29] R. Snodgrass, Developing Time-Oriented Database Applications in SQL, Morgan Kaufmann Series in Data Management Systems, 504 pages. ISBN 1-55860-436-7, 1999.

[30] A. Steiner, A Generalisation Approach to Temporal Data Models and their Implementations, PhD thesis, 1998.

[31] Y. Tang and L. Lian and R. Huang and Y. Yu, Bitemporal Extensions to Non-temporal RDBMS in Distributed Environment, 8th International Conference on Computer Supported Cooperative Work in Design, 2003.

[32] U. Tansel and J. Clifford and S. Gadia and A. Segev and R. Snodgrass, Temporal Databases: Theory, Design, and Implementation, Benjamin/Cummings, 1993.

[33] V. Tsotras and A. Kumar, Temporal Database Bibliography Update, SIGMOD Record, 25(1), 1996, 41-51.