History of Change

INTENT

Model attributes changing at irregular date/time intervals.
"The nature of the attributes is such that it holds a number of discrete values for intervals of time (as opposed to properties such as temperature that can change continuously)" [PLoP99].

ALSO KNOWN AS

Audit Trail, Valid-Time StateTable

MOTIVATION

Time modeling could seem trivial in a first view. If modeled by taking an attribute "snapshot" (tuple version) at all the possible dates and/or times, the model is quite simple (Time Series).
The drawback of this approach is that the number of time points could not be finite or anyway waste an excessive storage space. Suppose for example a relation Employee->Employer. You will not create a time series table containing a record every day for each employee, stating which employer he or she is related to.

Suppose we have a simple "Employee" entity, and we want to add the time dimension.

At the conceptual level, we typically have an instant (or event [Simsion01]) associated with an attribute change.
At the physical level, if we translate this conceptual model as is, on a one-to-one basis, in a normalized form, the following issues appear:
1 – Because each attribute can change at a different instant, we need a separate relation instant-attribute for each employee attribute. That implies that each attribute becomes a new entity and a new table. That increases dramatically the number of table, joins and degrades performances.
2 – When we query an entity to know the attribute value at a given instant, we always need to retrieve at least two rows: the rows enclosing the instant. In practice, the query often requires the max(…) and min(…) functions, which are quite difficult for the RDBMS to optimize.

The first issue is of course solved by creating a “snapshot” of all attributes at each instant when at least one attribute changes (Tuple Versioning).

The second issue is solved by modeling periods, instead of instants. Each row has a start and end time attributes, specifying in a single row which period the tuple is valid.

Some rare relational databases offer the SQL3 PERIOD data type (which is part of the temporal extension), with specialized indexing approaches.

After all, a period is “just” a pair of date/time. It looks pretty simple, and is very common, but experience shows that period handling is a source of bugs and performance problems. Solid integrity checks will limit such bugs. An accurate indexing strategy will improve the performance.

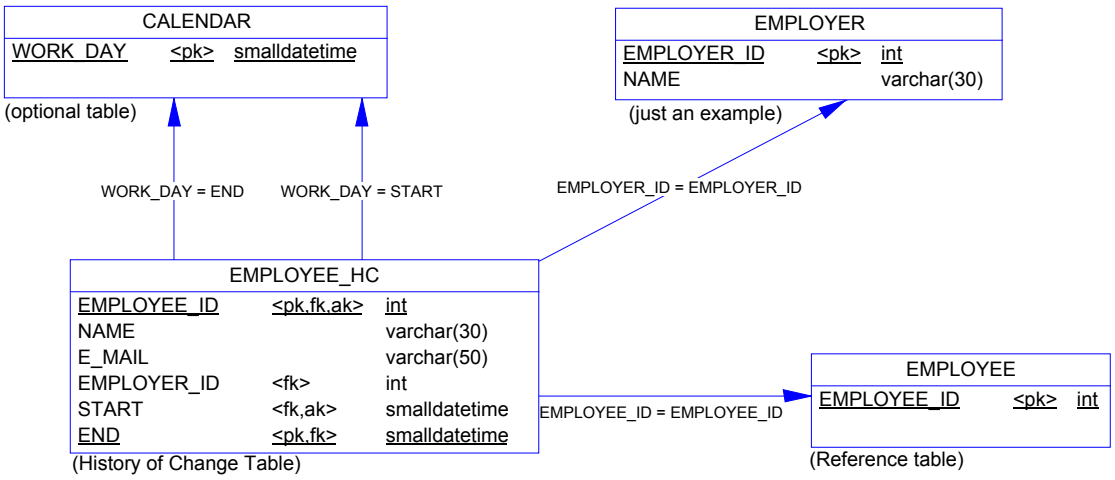
APPLICABILITY

Use the History of Change Pattern when:

- Attribute values changes over time irregularly and not at fixed predictable date/times and not continuously.
- A “Period” data type is not available
- There is enough storage space to apply some slight denormalization.
- The update performance is not critical

STRUCTURE

An example:



CONSEQUENCES

- All the attributes valid at a given instant (any instant) can be retrieved from a single table, in a “one-shot” read.
- The denormalization costs some slower updates and some waste of space.

IMPLEMENTATION

1- Tables

- The Employee (History of Change) table keeps snapshots of all the entity property, and the period defining when the properties are valid.
  Note that the period does not necessarily imply date/time, it can for example be an integer representing the start and end range of version of a system.
- If another table needs to reference the Employee table, then an Employee reference table is needed.
- A calendar table can be useful, when a period cannot start and end at any instant (for example only work days).

2- Indexing

Conceptually, the primary key is composed by the time-independent PK plus the date/time period. The “time-independent PK” is the PK we would use if we were modeling without taking the time dimension into account. We want this “time-independent PK” to be unique at any instant.

The choice, is about including in the PK the start instant or the end instant.
Because usually most queries search for recent records, an index on the end instant is on average much more selective than an index on the start instant (typically all the records have a start smaller than the current instant, but only one has the end greater than the current instant). Therefore the end instant is the preferred choice for the index specifying the physical order (typically the clustered index).

Additionally, an AK (alternate key or unique constraint) can be created, composed by the time-independent PK and the start instant. That can help queries on historical data.

A trigger can enforce the temporal key integrity, by checking that no overlapping periods exist. It will ensure that the time-independent PK is unique at any instant.

3- Nullability

The choice, is about how to code a record which is valid from “always” or is valid “forever”.
Implementing START and END not null brings the following advantages:

- better indexes
- the query optimizer can find a better access plan
- on some RDBMS, more efficient updates because the record never changes in size
- queries simpler and clearer because they do not need to check for nulls

The consequence is that when the attributes are valid forever, END must contain the maximum value allowed by the system (e.g. 31 December 9999) and START the minimum. It can be comfortable to define these values as defaults.

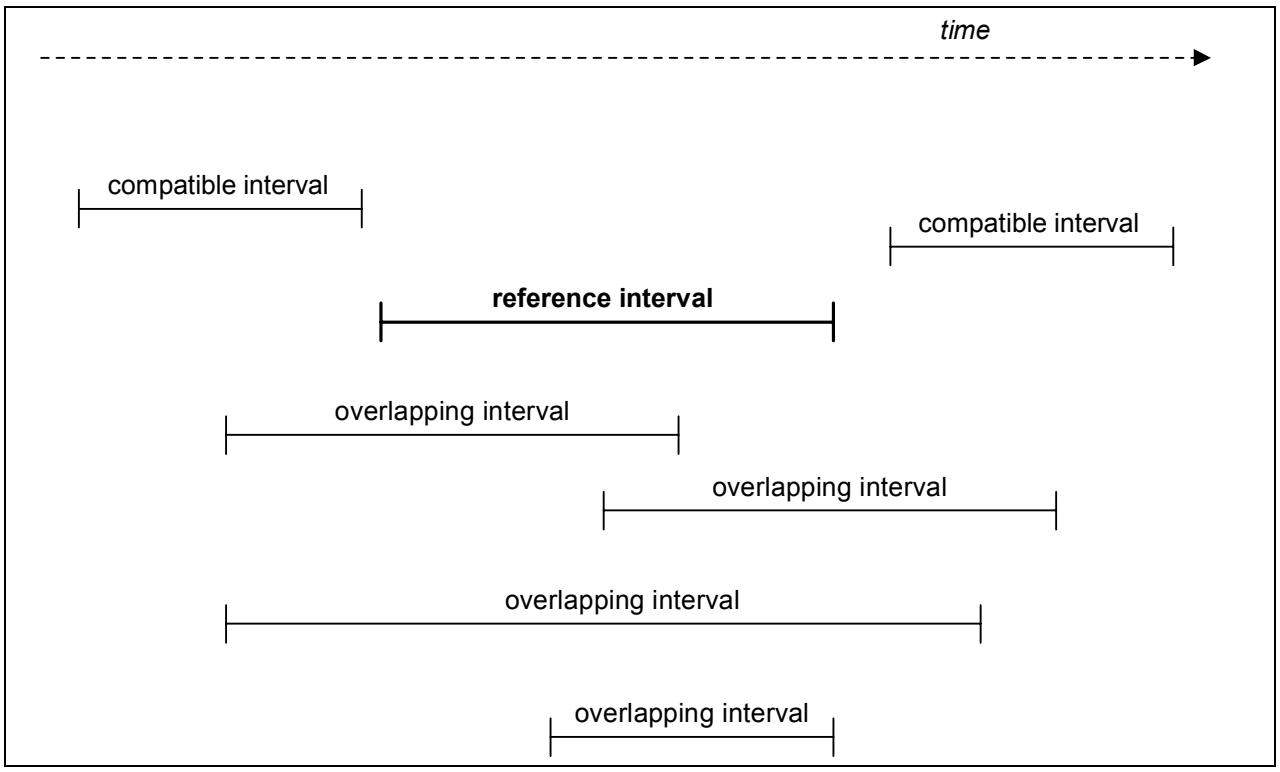
4- Integrity

A check constraint (or the trigger) can check that the end instant is greater that the start instant. The check constraint is slightly more efficient, the trigger usually allows a customized eloquent error message.

Often, start and/or end instants are allowed for example only at the beginning of a day or month, or only in business days, and so on. In this case, a calendar table contains the valid start and/or end instants and a referencing FK then enforces the rule.

Sometimes, each new record is associated on a FK to an event, which is the origin of the change of the attribute(s).

The key question is how to efficiently write the trigger avoiding overlapping periods.



All the overlapping periods have the common property:
- overlapping.END > reference.START AND overlapping.START < reference.END

However this is not enough. Suppose that the EMPLOYER table is also an History of Change with a START and END fields. In this case you would like that the referential integrity (EMPLOYEE.EMPLOYER ==> EMPLOYER.ID) takes the periods in both tables into account.
For each period on the employee table, you need that the referenced employer be defined in at least the same period.
Unfortunately, this cannot be enforced by a trigger, because when referenced intervals are updated, they are inconsistent for a short time, even when a transaction encloses all the changes. That's because the triggers are fired at each statement, even if the transaction is not finished.

Actually, if you have a multi-tier architecture, one elegant solution consists in implementing the periods integrity checks in the middle tier in an object oriented language. That allows code redundancy to be avoided by delegating time-related operations to a specialized set of classes.

5 – Deducible field maintenance.

There is an interesting option. Because the end instant can be deduced from the start instant and the subsequent record (and in a symmetric way the start instant from the end instant), the end instant can by maintained by a trigger. The benefit could be a more encapsulated logic. The drawbacks are some performance degradation (index updated twice, problems ensuring index values uniqueness).

SAMPLE CODE

Example of trigger checking basic temporal consistency (here in Sybase/Microsoft T-SQL):

```
create trigger EMPLOYEE_HC_IU  on EMPLOYEE_HC
   for insert, update
as
------- check that END > START --------------------------------
if exists (
     select 1
     from   inserted
     where  END <= START
) begin
     raiserror 20000 "the END <= START error message"
     rollback
     return
end
------- check for overlapping periods ------------------------
if exists (
     select 1
     from   inserted i
     ,      EMPLOYEE_HC e
     where  I.EMPLOYEE_ID = E.EMPLOYEE_ID
     and    I.END   > E.START
     and    I.START < E.END
     and    I.END   != E.END  -- needed for "after" transaction triggers
                              -- assumes that a unique index avoids duplicates
) begin
     raiserror 20001 "your overlapping periods error message"
     rollback
     return
end
go
```

[Snodgrass00] gives some similar examples in SQL dialects other than Sybase/Microsoft

RELATED PATTERNS

Time Series

REFERENCES

[Snodgrass00] R.T.Snodgrass, Developing Time-Oriented Database Applications in SQL, Morgan-Kaufmann, 2000

[PLoP99] N.Harrison, B.Foote, H.Rohnert, Pattern Languages of Program Design 4, *Temporal Patterns*, Addison Wesley, 1999

[EN99] R.A.Elmasri, S.B.Navathe, Fundamentals of Database Systems, Addison-Wesley, 1999

[Simsion01] Graeme C. Simsion, Data Modeling Essentials 2nd Edition, Coriolis, 2001