Database Tools Whitepaper

MAKING TIME USING TEMPORAL DATA

BY JOE CELKO

TABLE OF CONTENTS

MAKING TIME USING TEMPORAL DATA

SQL is the first programming language to have temporal data types in it. If COBOL had done this, then we would never have had the 'Y2K Crisis' that we survived a few years ago. SQL-92 added temporal data to the language, acknowledging most of what was already in most SQL product by that time. The problem is that each vendor made a trade-off internally. We get into SQL code later. However, since this is an area where people do not have a good understanding, it is better to start with foundations.

The current calendar is known as the Common Era Calendar, and not the Western, Christian or Gregorian Calendar. We want to use a universal, non-ethnic, non-religious name for it. The abbreviations for postfixes on dates are CE and BCE for 'Common Era' and 'Before Common Era' respectively. The abbreviations A.D. (that is, Anno Domini - Latin for 'in the year of Our Lord') and B.C. (that is, 'Before Christ') were dropped to avoid religious references.

Unfortunately, the solar year is not an even number of days. There are 365.2422 days in a year, and the fractional day adds up over time. The uneven number of days is why we have a leap year in the Common Era Calendar. Leap years did not exist in the Roman or Egyptian solar calendars before the year 708 AUC (that is, 'ab urbe condita', Latin for 'from the founding of the City of Rome). Consequently, they became useless for agriculture, so that the Egyptians relied on the stars to predict the flooding of the Nile. To realign the calendar with the seasons, Julius Caesar decreed that the year 708 (that is, the year 46 BCE to us) would have 445 days. Caesar, on the advice of Sosigenes, also introduced leap years (known as bissextile years) at this time. Many Romans referred to 708 AUC as the 'year of confusion' and thus began the Julian Calendar that was the standard for the world from that point forward.

The Julian calendar had a leap year day every four years. Therefore, it was reasonably accurate in the short or medium range. However, it drifted by approximately three days every 400 years. The drifting is a result of the fraction of a day of 0.0022 adding up.

It had gotten ten days out of step with the seasons by 1582. A calendar without a leap year would have drifted completely around slightly more than once between 708 AUC and 2335 AUC (that is, 1582 CE to us). The Summer Solstice, so crucial to planting crops, had no relationship to 21 June. Scientists finally convinced Pope Gregory to realign the calendar by dropping almost two weeks from October in 1582 CE. The years 800 CE and 1200 CE were leap years anywhere in the Christian world. But whether 1600 CE was a leap year depended on where you lived. European countries did not move to the new Calendar at the same time or follow the same pattern of adoption.

The calendar corrections had economic and social ramifications. In Great Britain and its colonies, 1752 September 02 was followed by 1752 September 14. The calendar reform bill of 1751 was entitled ‘An Act for Regulating the Commencement of the Year and For Correcting the Calendar Now in Use’. The bill included provisions to adjust the amount of money owed or collected from rents, leases, mortgages, and similar legal arrangements. Consequently, rents and so forth were prorated by the number of actual elapsed days in the period affected by the calendar change. Nobody had to pay the full monthly rate for the short month of September in 1752, and nobody had to pay the total yearly price for the short year.

The severe, widespread, and persistent rioting was not due to the economic problems that resulted. Instead, the rioting was due to the common belief that the days of each person were numbered and that everyone was preordained to be born and die at a divinely determined time that no human agency could alter in any way.

Thus the removal of 11 days from September shortened the lives of everyone on Earth by 11 days. And there was also the matter of the missing 83 days due to the change of the New Year’s Day from March 25 to January 01, which was believed to have a similar effect.

If you think this behavior is insane, consider the number of people today who get upset about the yearly one-hour clock adjustments for Daylight Saving Time.

To complicate matters, the beginning of the year also varied from country to country. Great Britain preferred to begin the year on March 25, while other countries started at Easter, December 25, or perhaps March 01 and January 01 -- all essential details for historians to keep in mind.

In Great Britain and its colonies, the calendar year 1750 began on March 25 and ended on March 25 -- that is, the day after 1750 March 24 was 1751 March 25. The British added the leap year day to the end of the last full month in the year, which was then February. The extra leap year day comes at the end of February since this part of the calendar structure was not changed.

In Latin, ‘septem’ means seventh, from which we derived September. Likewise, ‘octem’ means eighth, ‘novem’ means ninth, and ‘decem’ means tenth. Thus, September should be the seventh month, October should be the eighth, November should be the ninth and December should be the tenth.

So, how come September is the ninth month? September was the seventh month until 1752 when authorities changed the New Year from March 25 to January 01.

Until fairly recently, nobody agreed on the proper display format for dates. Every nation seems to have its commercial conventions. Most of us know that Americans put the month before the day, and the British do the reverse. However, do you know any other national conventions? National date formats may be confusing when used in an international environment. When it was ‘12/16/95’ in Boston, it was '16/12/95' in London, '16.95' in Berlin and '95-12-16' in Stockholm. Then there are conventions within industries within each country that complicate matters further.

Today, we have a standard for this: ISO-8601 'Data Elements and Interchange Formats - Information Interchange — Representation of Dates and Times' that is part of Standard SQL and other ISO standards.

The full ISO-8601 timestamp can be either local time or UTC. UTC is the code for 'Universal Coordinated Time,' which replaced the older GMT, which was the code for 'Greenwich Mean Time', which is still improperly used in popular media.

In 1970 the Coordinated Universal Time system was devised by an international advisory group of technical experts within the International Telecommunication Union (ITU). The ITU felt it was best to designate a single abbreviation for use in all languages to minimize confusion. The two alternative, original abbreviation proposals for the 'Universal Coordinated Time' were CUT (English: Coordinated Universal Time) and TUC (French: Temps Universel Coordonné). UTC was selected both as a compromise between the French and English proposals, and because the C at the end looks more like an index in UT0, UT1, UT2, and a mathematical-style notation is always the most international approach.

Universal Coordinated Time is not quite the same thing as astronomical time. The Earth wobbles a bit, and authorities had to adjust the UTC to the solar year with a leap second added or removed once a year to keep them in synch. As of this writing, authorities will base Universal Coordinated Time on an atomic clock without a leap second adjustment soon.

This extra second has screwed up software. In 1998, the leap second caused a mobile-phone blackout across the southern United States because different regions were suddenly operating with time differences outside the error tolerances. Then in 2012, the booking system of an airline went belly-up for hours after a leap second insertion. Most nations want to move to an atomic clock (International Atomic Time, TAI) standard. However, it has not happened yet. Here is a short history of the leap second:

TAI versus UTC after adding the leap seconds

UTC Date	UTC Time	Difference
1998-12-31	23:59:60	32 secs
2005-12-31	23:59:60	33 secs
2008-12-31	23:59:60	34 secs
2012-06-30	23:59:60	35 secs
2015-06-30	23:59:60	36 secs
2016-12-31	23:59:60	37 secs

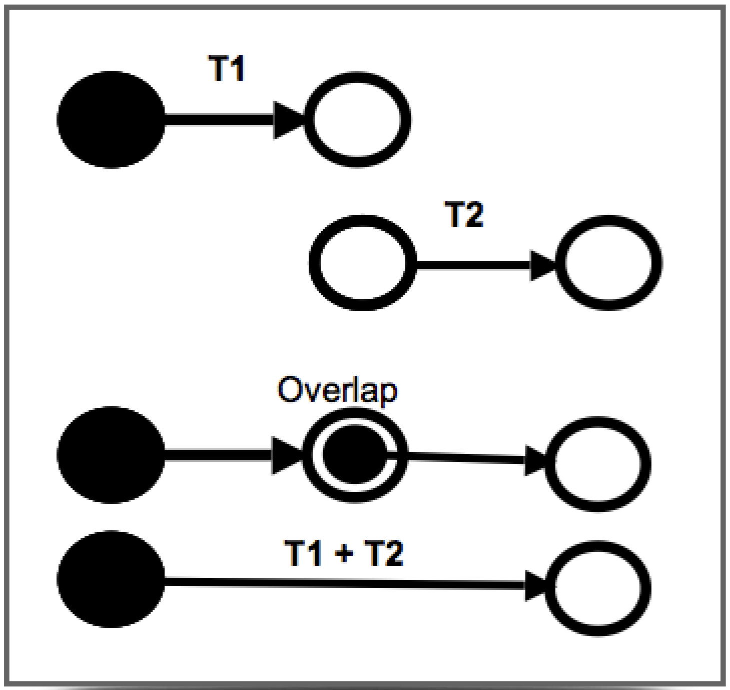
Further leap seconds not yet announced.

Another problem is the use of local time zones (four of them in the United States) and having to worry about ‘lawful time’. The ‘lawful time’ is the technical term for the time required by law for commerce. Usually, this means whether or not you use Daylight Saving Time (DST) and how it is defined locally. A date without a time zone is ambiguous in a distributed system. A transaction created with DATE ‘1995-12-17’ in London may be physically younger than a transaction created with DATE ‘1995-12-16’ in Boston.

THE ISO TEMPORAL MODEL

The ISO model for temporal values is based on half-open intervals. The half-open intervals mean there is a starting point. However, the interval never gets to the ending point. For example, the day begins at ‘2016-01-01 00:00:00’ exactly. However, it does not end at ‘2016-01-02 00:00:00’. Instead, it approaches the start of the next day as the limit. Depending on how much decimal precision we have, ‘2016-01-01 23:59:59.999..’ as the endpoint approximation. I need to have at least one more fractional part than my data in my DDL.

Half-open intervals can be abutted to each other to produce another half-open interval. Two overlapping half-open intervals provide a half-open interval. Likewise, if you remove a half-open interval from another half-open interval, then you get one or two half-open intervals. The removal is called closure, and it is a nice mathematical property to have.



SQL TEMPORAL DATA TYPES

Standard SQL has a complete description of its temporal data types. Rules exist for converting from numeric and character strings into these data types. Also, a schema table exists for global time-zone information that is used to make sure that temporal data types are synchronized. It is so complete and elaborate that smaller SQLs have not implemented it yet. As an international standard, SQL has to handle time for the whole world, and most of us work with only local time. If you have ever tried to figure out the time in a foreign city to place a telephone call, then you have some idea of what is involved.

The common terms and conventions related to time are also confusing. We use the term 'an hour' to mean a particular point within the cycle of a day (such as 'The train arrives at 13:00 hrs.') or to indicate an interval of time (such as 'The train takes three hours to get there.'). The number of days in a month is not uniform. The number of days in a year is not uniform. Weeks are not easily related to months. And so on.

Standard SQL has a set of date, time (that is, DATE, TIME and TIMESTAMP) and INTERVALs (that is, DAY, HOUR, MINUTE, and SECOND with a decimal fraction) data types. They are made up of fields that are ordered within the value. They are YEAR, MONTH, DAY HOUR, MINUTE, and SECOND. These fields are the only place in SQL that we use the term 'field'. New SQL programmers too often confuse the term field as used in file systems with the column concept in RDBMS. They are different.

Both of these are temporal data types. However, datetimes represent points in the timeline, while the interval data types are durations of time, not anchored at a position on the timeline. Standard SQL also has a full set of operators for these data types. But you still find vendor syntax in most SQL implementations today.

TIPS FOR HANDLING DATES, TIMESTAMPS AND TIMES

The syntax and power of date, timestamp and time features vary so much from product to product that it is impossible to give anything but general advice. This chapter assumes that you have simple date arithmetic in your SQL. However, you might find that some library functions would let you do a better job than what you see here. Please continue to check your manuals until the implementation of the Standard SQL operators.

As a general statement, there are two ways of representing temporal data internally. The 'UNIX representation' is based on keeping a single binary string of 64 or more bits that counts the computer clock ticks from a base starting date and time. The other representation I call the 'COBOL method' since it uses separate fields for the year, month, day, hours, minutes, and seconds. These fields can be characters, BCD or other another internal format.

The UNIX method is suitable for calculations. However, the engine must convert from the external ISO-8601 format to the internal format and vice versa. The COBOL format is the opposite. That format is ideal for display purposes but weaker on calculations.

Do not write code that depends on any internal format. You want portable code, and you want to do any display formatting in a presentation layer, not the database. To give you an idea just how different something as simple as the names of the months can be, here is a comparison of Czech and Slovak. Czech and Slovak they used to be in the same country. However, Slovak uses Latin names, and Czech uses old Slavic.

	Czech	Slovak
1	leden	Január
2	únor	Február
3	březen	Marec
4	duben	Apríl
5	květen	Máj
6	červen	Jún
7	červenec	Júl
8	srpen	August
9	září	September
10	říjen	Október
11	listopad	November
12	prosinec	December

Want to ADD Chinese to this?

DATE FORMAT STANDARDS

There are three basic display formats in the ISO standards. The three formats are all digits separated by the punctuation of some kind. You do not want to use language-dependent names as they do not alphabetize or machine sort efficiently.

CALENDAR DATE

It is a string of digits that are made up of the four-digit year, a dash, two-digit month, dash and a two-digit day within the month. For example: ‘2015-06-25’ for June 25th of 2015. That date display format is the only one allowed in ANSI/ISO Standard SQL. However, the full ISO-8601 standards allow you to drop the dashes and write the data as a string of all digits.

There is a delightful cartoon about the ISO-8601 Standard at ‘https://xkcd.com/1179/’ that people ought to use as a poster to keep the developers in line. Always avoid dialect in favor of ANSI/ISO Standard SQL.

ORDINAL DATE

It is a string of digits that are made up of the four-digit year, a dash, and the three-digit ordinal number of the day within the year expressed as ‘001’ thru ‘365’ or ‘366’ as appropriate. For example: ‘2015-176’ for June 25th of 2015. You have to implement this as a string with a CHECK() constraint that enforces the regular expression.

WEEK DATE

It is a string of digits that are made up of:

- the four-digit year,
- a ‘W’,
- the two-digit ordinal number of the week within the year expressed as ‘01’ thru ‘52’ or ‘53’ as appropriate,
- a dash, and
- a single digit from 1 to 7 for the day within the week (that is, 1= Monday, 7= Sunday).

Very often, the weekday is not used. For example: ‘2015W26’ is the entire week from ‘2015-06-22’ to ‘2015-06-28’ , which includes ‘2015W26-5’ for June 25th of 2015.

Weeks do not align to calendar dates so that a week can cross over year boundaries. The first week of the year is the week that contains the first Thursday of that year (that is, the first four-day week of the year). The highest week number in a year is either 52 or 53. Again, you have to implement this as a string with a CHECK() constraint that enforces the regular expression.

You can download both the ordinal and week dates from the Internet or built-in spreadsheet using their functions.

TIME PERIODS

Model longer periods by truncating the finer fields in these display formats. Thus a whole year can be shown with four digits, the year-month period with ‘yyyy-mm’ and we also ready showed a whole week. But there are no conventions for quarters and other fiscal periods. I like the MySQL convention of using double zeroes for months and years. That is ‘yyyy-mm-00’ for a month within a year and ‘yyyy-00-00’ for the whole year. The advantages are that it sorts with the ISO-8601 date format and it is language independent. The regular expression patterns for validation are ‘[12][0-9][0-9][0-9]-00-00’ and ‘[12][0-9][0-9][0-9]-[01][0-9]-00’ respectively. You need to create a look-up table with the period name and the start and end timestamps for it.

TIME FORMAT STANDARDS

TIME(n) is made up of a two-digit hour between ‘00’ and ‘23’, colon, a two-digit minute between ‘00’ and ‘59’, colon, and a two-digit second between ‘00’ and ‘59’ or ‘60’, if the leap second is still in use. Seconds can also have decimal places shown by (n) from zero to an implementation-defined accuracy. The FIPS-127 standard requires at least five decimal places after the second and modern products typically go to seven decimal places.

We do not use the old AM and PM postfix any ISO Standards. There is no such time as 24:00:00. Instead, the time is 00:00:00 of the next day. Remember the half-open interval model? However, some SQLs accept 24:00:00 as input and put it in the proper format.

TIMESTAMP(n) values are made up of a date, a space, and a time. The ISO standards allow replacing of the space to by the letter ‘T’ to put the timestamp into a single unbroken string and to remover the punctuation. The SQL standard does not.

Remember that a CURRENT_TIMESTAMP reads the system clock once and use that same time on all the items involved in a transaction. It does not matter if the actual time it took to complete the transaction was days. A transaction in SQL is done as a whole unit or is not done at all. Such time delays are not usually a problem for small transactions. However, it can be in large batched ones with complex updates.

TIME IN A DATABASE

You should use a ‘24-hour’ time format, which is less prone to errors than 12-hour (AM/PM) time since it is less likely to be misread or miswritten. This format can be manually sorted more efficiently and is less prone to computational errors. Americans use a colon as a field separator between hours, minutes, and seconds just SQL. However, some Europeans use a period in their local display format.

TIME ZONES

Older, smaller databases live and work in a single time zone. The DBA sets the system clock to local time, and the DBA ignores the complications like leap seconds, DST, and time zones. Standard SQL uses only UTC and converts it to local time with TIMEZONE_HOUR and TIMEZONE_MINUTE fields at the end. These fields give the time zone displacement for local times in that column or temporal variable.

There are also three-letter and four-letter codes for the time zones of the world, such as EST, for Eastern Standard Time, in the United States. But these codes are not universal. For example, all of these time zones are UTC-3 hours.

Time Zone Code	Time Zone Name	Where used
ADT	Atlantic Daylight Time	Atlantic
BRT	Brasília time	South America
CLST	Chile Summer Time	South America
GFT	French Guiana Time	South America
WGT	West Greenland Time	North America

The closest thing to a universal naming convention for time zones is the Military alphabet code.

Time Zone Code	Time Zone Name	Displacement from UTC
A	Alpha Time Zone	+ 1 hour
B	Bravo Time Zone	+ 2 hours
C	Charlie Time Zone	+ 3 hours
D	Delta Time Zone	+ 4 hours
E	Echo Time Zone	+ 5 hours
F	Foxtrot Time Zone	+ 6 hours
G	Golf Time Zone	+ 7 hours
H	Hotel Time Zone	+ 8 hours
I	India Time Zone	+ 9 hours
K	Kilo Time Zone	+ 10 hours
L	Lima Time Zone	+ 11 hours
M	Mike Time Zone	+ 12 hours
N	November Time Zone	- 1 hour
O	Oscar Time Zone	- 2 hours
P	Papa Time Zone	- 3 hours
Q	Quebec Time Zone	- 4 hours

R	Romeo Time Zone	- 5 hours
S	Sierra Time Zone	- 6 hours
T	Tango Time Zone	- 7 hours
U	Uniform Time Zone	- 8 hours
V	Victor Time Zone	- 9 hours
W	Whiskey Time Zone	- 10 hours
X	X-ray Time Zone	- 11 hours
Y	Yankee Time Zone	- 12 hours
Z	Zulu Time Zone	UTC

That Military alphabet code is why UTC is sometimes called ‘Zulu time’, and the letter Z is used as punctuation between the timestamp and the displacement in parts of the full ISO-8601.

The offset is usually a positive or negative number of hours. However, there are still a few odd zones that differ by 15 or 30 minutes from the expected pattern.

The TIMESTAMP data type is a DATE and a TIME put together in a single value (such as ‘2017-05-03 05:30:06.123’). There are some variations from DBMS to DBMS though. For example, the time component of DB2 TIMESTAMP data is configurable and can be more precise than DB2 TIME data. The DB2 TIMESTAMP data is what CURRENT_TIMESTAMP returns from the system clock in a program, query, or statement. However, SQL dialects still use NOW, getdate(), and other proprietary reserved words.

TIMESTAMP WITH TIME ZONE or TIMESTAMPTZ data types are a TIMESTAMP. However, include the displacement of the time zone from UTC. The standards allow for TZD (time zone designator), ‘Z’, or a positive or negative hour to minute interval (that is, +hh:mm or -hh:mm). Standard SQL uses the last option.

CURRENT_TIMESTAMP is a representation of the current date and time with the time zone. LOCALTIMESTAMP is a representation of the current date and time but without a time zone.

Now you have to factor in Daylight Saving Time on top of that to get what is called ‘lawful time’ which it is the basis for legal agreements. The US government uses DST on federal lands inside of states that do not use DST. You can get a mix of gaps and duplicate times in the local lawful time display over a year. That mix is why Standard SQL uses UTC internally.

Vendors often have a system configuration parameter to set the local time zone and other options. You need to know your SQL and not get caught in this. My advice for multi-national users is to keep the databases in UTC and handle local time in the presentation layers.

INTERVAL DATA TYPES

INTERVAL data types are used to represent temporal duration. They come in two basic types. Intervals that deal with the calendar and those that deal with the clock. The year-month intervals have an express or implied precision that includes no fields other than YEAR and MONTH, though it is not necessary to use both. The other class, called day-time intervals, has an express or implied interval precision that can include any fields other than YEAR or MONTH — that is, DAY, HOUR, MINUTE and SECOND (with decimal places).

SECOND are integers and have precision 2 when not the first field. SECOND, however, can be defined to have an <interval fractional seconds precision> that indicates the number of decimal digits maintained following the decimal point in the seconds value. When not the first field, SECOND has a precision of 2 places before the decimal point.

The fields in the interval have to be in high to low order without missing fields.

Field	Inclusive value limit
YEAR	‘0001’ to ‘9999’; follows the ISO-8601 Standard
MONTH	‘01’ to ‘12’; may round the value 12 to one year
DAY	‘01’ to ‘31’; must be valid for month and year
HOUR	‘00’ to ‘23’; may round the value 24 to the next day
MINUTE	‘00’ to ‘59’; watch for leap seconds!
SECOND	‘00’ to ‘59.999..’ ; the implementation defines the precision

The datetime literals are not surprising in that they follow the syntax used by ISO-8601 Standards with the dashes between the fields in dates and colons between the fields times. Always quote the strings. The interval qualifier follows the keyword INTERVAL when specifying an INTERVAL data type.

The following table lists the valid interval qualifiers for YEAR-MONTH intervals:

Interval Qualifier	Description
YEAR	An interval class describing a number of years
MONTH	An interval class describing a number of months
YEAR TO MONTH	An interval class describing a number of years and months

The following table lists the valid interval qualifiers for DAY-TIME intervals:

Interval Qualifier	Description
DAY	Plus or minus a number of days
HOUR	Plus or minus a number of hours
MINUTE	Plus or minus a number of minutes
SECOND(s)	Plus or minus a number of seconds (decimals are allowed)
DAY TO HOUR	Plus or minus a number of days and hours
DAY TO MINUTE	Plus or minus a number of days, hours and minutes
DAY TO SECOND(s)	Plus or minus a number of days, hours, minutes and seconds
HOUR TO MINUTE	Plus or minus a number of hours and minutes
HOUR TO SECOND(s)	Plus or minus a number of hours, minutes and seconds
MINUTE TO SECOND(s)	Plus or minus a number of minutes and seconds

Here is a sample query that shows all of the INTERVAL types in use:

```
SELECT CURRENT_TIMESTAMP + INTERVAL '+7' YEAR,
 CURRENT_TIMESTAMP + INTERVAL '-3' MONTH,
 CURRENT_TIMESTAMP + INTERVAL '0007 03' YEAR TO MONTH,
 CURRENT_TIMESTAMP + INTERVAL '+5' DAY,
 CURRENT_TIMESTAMP + INTERVAL '-5' HOUR,
 CURRENT_TIMESTAMP + INTERVAL '12' MINUTE,
 CURRENT_TIMESTAMP + INTERVAL '3' SECOND,
 CURRENT_TIMESTAMP + INTERVAL '1 12' DAY TO HOUR,
 CURRENT_TIMESTAMP + INTERVAL '1 12:35' DAY TO MINUTE,
 CURRENT_TIMESTAMP + INTERVAL '1 12:35:45' DAY TO SECOND,
 CURRENT_TIMESTAMP + INTERVAL '01:12' HOUR TO MINUTE,
 CURRENT_TIMESTAMP + INTERVAL '01:12:35' HOUR TO SECOND,
 CURRENT_TIMESTAMP + INTERVAL '01:12' MINUTE TO SECOND
 FROM Dummy;
```

Notice that the quoted strings in the HOUR TO MINUTE and MINUTE TO SECOND example are the same, but they have different meanings. A timestamp literal can also include a time zone interval to change it from a UTC to local time.

QUERIES WITH DATE ARITHMETIC

Almost every SQL implementation has a DATE data type. However, the proprietary functions available for them vary quite a bit. The most common ones are a constructor that builds a date from integers or strings. Others are extractors to pull out the month, day, or year. And some display options to format output.

You can assume that your SQL implementation has simple date arithmetic functions, although with different syntax from product to product, such as

1. A date plus or minus a number of days yields a new date.

2. A date minus a second date yields an integer number of days between the dates.

Here is a table of the valid combinations of <datetime> and <interval> data types in the Standard SQL standard:

```
<datetime> - <datetime> = <interval>
<datetime> + <interval> = <datetime>
<interval> (* or/) <numeric> = <interval>
<interval> + <datetime> = <datetime>
<interval> + <interval> = <interval>
<numeric> * <interval> = <interval>
```

There are other rules, which deal with time zones and the relative precision of the two operands, that are intuitively obvious.

USE OF NULL FOR ‘ETERNITY’

The temporal model in SQL does not have a symbol for ‘eternity in the future’ or ‘eternity in the past’. Consequently, you have to work around it for some applications. The IEEE floating-point standard does have both a ‘-inf’ and ‘+inf’ symbol to handle this problem in the continuum model for real numbers. SQL can ‘only’ represent timestamps in the range of years from 0001 CE up to 9999 CE. Usually, this range is good enough for most applications outside of archeology.

For example, when someone checks into a hotel, we know their arrival date. However, we do not know their departure date (since an expected departure date is not the same thing as an actual one). All we know for certain is that it has to be after their arrival date. A NULL acts as a ‘place holder’ until we get the actual departure date. The skeleton DDL for such a table would look like this:

```
CREATE TABLE Hotel_Register
 (patron_id INTEGER NOT NULL
REFERENCES Patrons (patron_id),
arrival_date TIMESTAMP(0) DEFAULT CURRENT_TIMESTAMP NOT NULL,
departure_date TIMESTAMP(0), -- null means guest still here
CONSTRAINT arrive_before_depart
CHECK (arrival_date <= departure_date),
..);
```

When getting reports, you need to use the current timestamp in place of the NULL to accurately report the current billing.

```
SELECT patron_id, arrival_date,
  COALESCE (CURRENT_TIMESTAMP, departure_date)
  AS departure_date
 FROM HotelRegister
WHERE ..;
```

THE OVERLAPS() PREDICATE

The OVERLAPS() predicate is a feature that is still not available in most SQL implementations. The reason is that it requires more of the Standard SQL temporal data features than most implementations have. You can 'fake it' in many products with the BETWEEN predicate and careful use of constraints.

The result of the <OVERLAPS predicate> is formally defined as the result of the following expression:

```
(S1 > S2 AND NOT (S1 >= T2 AND T1 >= T2))
OR (S2 > S1 AND NOT (S2 >= T1 AND T2 >= T1))
OR (S1 = S2 AND (T1 <> T2 OR T1 = T2))
```

Here, S1 and S2 are the starting times of the two periods, and T1 and T2 are their termination times. The rules for the OVERLAPS() predicate sound like they should be intuitive. However, they are not. The principles that we wanted in the Standard were:

1. A time period includes its starting point but does not include its endpoint. We have already discussed this model and its closure properties.

2. If the periods are not 'instantaneous', then they overlap when they share a common period.

3. If the first term of the predicate is an INTERVAL and the second term is an instantaneous event (that is, a <datetime> data type), then they overlap when the second term is in the period but is not the endpoint of the period. That follows the half-open model.

4. If the first and second terms are both instantaneous events, then they only overlap when they are equal.

5. If the starting time is NULL, and the finishing time is a <datetime> value, then the finishing time becomes the starting time, and we have an event. If the starting time is NULL and the finishing time is an INTERVAL value, then both the finishing and starting times are NULL.

Please consider how your intuition reacts to these results when the granularity is at the YEAR-MONTH-DAY level. Remember that a day begins at 00:00:00 Hrs.

```
(today, today) OVERLAPS (today, today) = TRUE
(today, tomorrow) OVERLAPS (today, today) = TRUE
(today, tomorrow) OVERLAPS (tomorrow, tomorrow) = FALSE
(yesterday, today) OVERLAPS (today, tomorrow) = FALSE
```

Alexander Kuznetsov wrote this idiom for History Tables in T-SQL. However, it generalizes to any SQL. It builds a temporal chain from the current row to the previous row. With a self-reference. The process is easier to show with code:

```
CREATE TABLE Tasks
(task_id INTEGER NOT NULL,
 task_score CHAR(1) NOT NULL,
 previous_end_date DATE, -- null means first task
 current_start_date DATE DEFAULT CURRENT_TIMESTAMP NOT NULL,
 CONSTRAINT previous_end_date_and_current_start_in_sequence
   CHECK (prev_end_date <= current_start_date)
 DEFERRABLE INITIALLY IMMEDIATE,
 current_end_date DATE, -- null means unfinished current task
 CONSTRAINT current_start_and_end_dates_in_sequence
   CHECK (current_start_date <= current_end_date),
 CONSTRAINT end_dates_in_sequence
   CHECK (previous_end_date <> current_end_date),
 PRIMARY KEY (task_id, current_start_date),
 UNIQUE (task_id, previous_end_date), -- null first task
 UNIQUE (task_id, current_end_date), -- one null current task
 FOREIGN KEY (task_id, previous_end_date)  -- self-reference
   REFERENCES Tasks (task_id, current_end_date));
```

Well, that looks complicated! Let us look at it column by column. Task_id explains itself. The previous_end_date does not have a value for the first task in the chain so that it is NULL-able. The current_start_date and current_end_date are the same data elements, temporal sequence, and PRIMARY KEY constraints we had in the simple history table schema.

The two UNIQUE constraints allow a single NULL in their pairs of columns and prevent duplicates. Remember that UNIQUE is NULL-able, not like PRIMARY KEY, which implies UNIQUE NOT NULL.

Finally, the FOREIGN KEY is the real trick. The previous task has to end when the current task started for the two tasks to abut so that there is another constraint. This constraint is a self-reference that makes sure this is true. Modifying data in this type of table is easy. However, it requires some thought.

There is just one little problem with that FOREIGN KEY constraint. It does not let you put the first task into the table. There is nothing for the constraint to reference. In Standard SQL, we can declare constraints to be DEFERRABLE with some other options. The idea is that you can turn a constraint ON or OFF during a session so that the database can be in a state that would otherwise be illegal. But at the end of the session, all constraints have to be TRUE or UNKNOWN.

CALENDAR TABLES

Because the calendar and temporal math are so irregular, build axillary tables for various single day calendars and period calendars. One column for the calendar date and other columns to show whatever your business needs in the way of temporal information. Do not try to calculate holidays in SQL -- Easter alone requires too much math. Oh, which Easter? Catholic or Orthodox?

It is possible to build a calendar table for US Secular holidays from available data.

You probably want a fiscal calendar in this table. Which fiscal calendar? The GAAP (General Accepted Accounting Practices) lists over a hundred of them.

I would add the ordinal date and week date, which we discussed earlier. These two dates make temporal math much more accessible.

The Julian business day is a good trick. Number the days from whenever your calendar starts and repeat a number for a weekend or company holiday. The following code counts business days.

```
CREATE TABLE Calendar
(cal_date DATE NOT NULL PRIMARY KEY,
 julian_business_nbr INTEGER NOT NULL,
 ...);
INSERT INTO Calendar
VALUES ('2007-04-05', 42),
 ('2007-04-06', 43), -- good Friday
 ('2007-04-07', 43),
 ('2007-04-08', 43), -- Easter Sunday
 ('2007-04-09', 44),
 ('2007-04-10', 45); --Tuesday
```

To compute the business days from Thursday of this week to next Tuesdays:

```
SELECT (C2.julian_business_nbr - C1.julian_business_nbr)
 FROM Calendar AS C1, Calendar AS C2
 WHERE C1.cal_date = '2007-04-05',
 AND C2.cal_date = '2007-04-10';
```

REPORT PERIOD TABLES

Since SQL is a database language, we prefer to do lookups and not calculations. They can be optimized while temporal math messes up optimization. A useful idiom is a report period calendar that everyone uses so that there is no way to get disagreements in the DML.

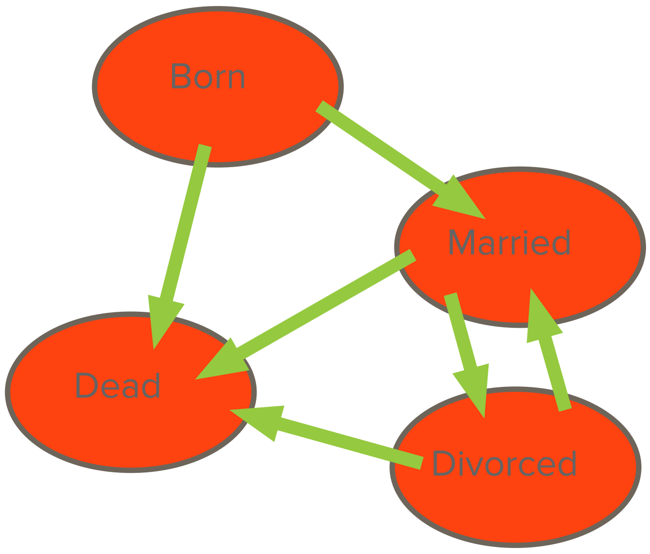
The report period table gives a name to a range of dates that is common to the entire enterprise.

```
CREATE TABLE Something_Report_Periods
(something_report_name CHAR(10) NOT NULL PRIMARY KEY
 CHECK (something_report_name LIKE <pattern>),
 something_report_start_date DATE NOT NULL,
 something_report_end_date DATE NOT NULL,
 CONSTRAINT date_ordering
 CHECK (something_report_start_date <= something_report_end_date),
etc);
```

These report periods can overlap or have gaps. Avoid period names that are language-dependent. They have trouble porting. If possible, the name of the periods should sort in the temporal order. Again, I like the MySQL convention of using double zeroes for months and years. That is ‘yyyy-mm-00’ for a month within a year and ‘yyyy-00-00’ for the whole year. The advantages are that it sorts with the ISO-8601 date format and goes to the top of each year and month within the year.

STATE TRANSITION CONSTRAINTS

Transition constraints force status changes to occur in a particular order over time. A state-transition diagram is the best to show the rules. There is at least a single initial state, flow lines that show what the next legal states and one or more termination states are. Here is a simple state change diagram of possible marital states:



This state transition diagram was deliberately simplified. However, it is good enough to explain the principles. My table is for the marital status of only a single person over his or her life to keep the discussion as simple as possible. Here is a skeleton DDL with the needed FOREIGN KEY reference to valid state changes and the date that the current state started.

```
CREATE TABLE MyLife
(previous_state VARCHAR(10) NOT NULL,
 current_state VARCHAR(10) NOT NULL,
 CONSTRAINT Improper_State_Change
 FOREIGN KEY (previous_state, current_state)
 REFERENCES StateChanges (previous_state, current_state),
 start_date DATE NOT NULL PRIMARY KEY,
 --etc.
);
```

It does not show which nodes are initial states (in this case, 'Born') and which are terminal or final states (in this case, 'Dead', a terminal state of being). A terminal node can be the current state of a middle node. However, not a former state. Likewise, an initial node can be the previous state of a central node, but not the current state. I did not write any CHECK() constraints for those conditions. It is easy enough to write a quick query with an EXISTS() predicate to do this, and I leave that as an exercise for the reader. Let us load the diagram into an auxiliary table with some more constraints.

```
CREATE TABLE StateChanges
(previous_state VARCHAR(10) NOT NULL,
 current_state VARCHAR(10) NOT NULL,
 PRIMARY KEY (previous_state, current_state),
 state_type CHAR(1) DEFAULT 'M' NOT NULL
 CHECK (state_type IN ('I', 'T', 'M')), -- initial, terminal, middle
 CONSTRAINT Node_Type_Violations
 CHECK (CASE WHEN state_type IN ('I', 'T')
 AND previous_state = current_state
 THEN 'T'
 WHEN state_type = 'M'
 AND previous_state <> current_state
 THEN 'T' ELSE 'F' END = 'T'));
INSERT INTO StateChanges
 VALUES ('Born', 'Born', 'I'), -- initial state
 ('Born', 'Married', 'M'),
 ('Born', 'Dead', 'M'),
 ('Married', 'Divorced', 'M'),
 ('Married', 'Dead', 'M'),
 ('Divorced', 'Married', 'M'),
 ('Divorced', 'Dead', 'M'),
 ('Dead', 'Dead', 'T'); -- a terminal state
```

We want to see a temporal path from an initial state to a terminal state. State changes do not happen all at once. However, they are spread over time. Time controls some of the changes, while agents control others. I cannot get married immediately after being born. However, have to wait to be of legal age. Then I have to consent.

For a real production system, you would need more state pairs. However, it is easy to expand the table.

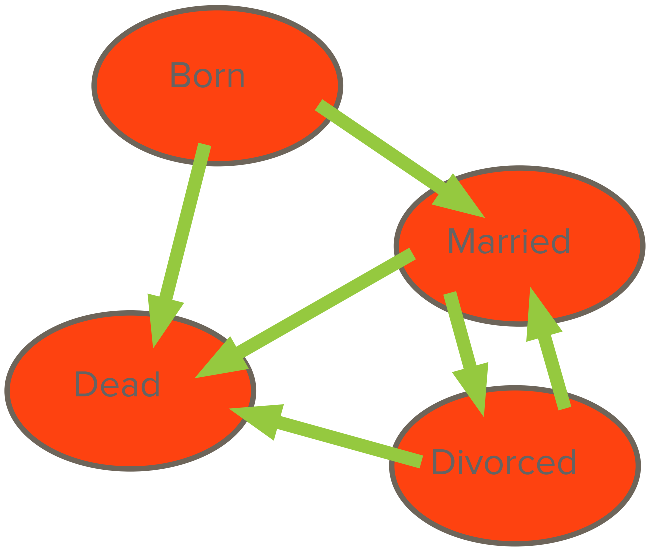
```
CREATE PROCEDURE Change_State
(IN change_date DATE,
 IN change_state VARCHAR(10))
LANGUAGE SQL
DETERMINISTIC
BEGIN
DECLARE most_recent_state VARCHAR(10);
SET most_recent_state
 = (SELECT current_state
 FROM MyLife
 WHERE start_date
 = (SELECT MAX(start_date) FROM MyLife));
-- insert initial state if empty
IF NOT EXISTS (SELECT * FROM MyLife)
 AND in_change_state
 IN (SELECT previous_state
 FROM StateChanges
 WHERE state_type = ‘I’)
THEN
INSERT INTO MyLife (previous_state, current_state, start_date)
VALUES (in_change_state, change_state, change_date);
END IF;
-- must be a real state change
IF change_state = most_recent_state
THEN SIGNAL SQLSTATE ‘75002’
 SET MESSAGE_TEXT = ‘This does not change the state.’;
END IF;
-- must move forward in time
IF change_date <= (SELECT MAX(start_date) FROM MyLife)
THEN SIGNAL SQLSTATE ‘75003’
 SET MESSAGE_TEXT = ‘Violates time sequence.’;
END IF;
END;
INSERT INTO MyLife (previous_state, current_state, start_date)
VALUES (most_recent_state, change_state, change_date);
END;
```

The first block of code locates the most recent state of my life, based on the date. The second block of code inserts an initial state if the table is empty. That initial state is a safety feature. However, there probably ought to be a separate procedure to create the set of initial states. The new state has to be an actual change so that there is a block of code to be sure. The changes have to move forward in time. Finally, we build a row using the most recent state as the new previous state, the input change state, and the date. If the state change is illegal, then this violates the FOREIGN KEY and we get an error.

If you had other business rules, you could also add them to the code in the same way. You should have noticed that if someone makes changes directly to the MyLife Table, then they are pretty much free to screw up the data. The first question is where to check for temporal violations, either during insertion or with validation procedures? My answer is both. Whenever possible, do not knowingly put bad data into a schema so that this should be done in the ChangeState() procedure. But someone or something subverts the schema, and you have to be able to find and repair the damage.

A lot of commercial situations have a fixed lifespan. Warranties, commercial offers, and bids expire in a known number of days. Such fixed lifespans mean adding another column to the StateChanges table. That added column tells the insertion program if the expiration date is optional (that is, shown with a NULL) or mandatory (that is, computed from the duration).

Overlapping periods are useful for reporting things like sales promotions. You can quickly see if the overlap between your 'Bikini Madness Week' and 'Three Day Suntan Lotion Promotion' helped increase total sales.



ABOUT THE AUTHOR

Mr. Joe Celko serves as Member of Technical Advisory Board of Cogito, Inc. Mr. Celko joined the ANSI X3H2 Database Standards Committee in 1987 and helped write the ANSI/ISO SQL-89 and SQL-92 standards. He is one of the top SQL experts in the world, writing over 700 articles primarily on SQL and database topics in the computer trade and academic press. The author of six books on databases and SQL, Mr. Celko also contributes his time as a speaker and instructor at universities, trade conferences, and local user groups.