Dealing with time in databases and data models: implementation

Prof. dr. Bas van Gils

Bas.vanGils@strategy-alliance.com
Managing partner @ Strategy Alliance
Professor @ Antwerp Management School

Introduction

This is a follow-up to an earlier earlier article where I explored the notion of time in relational databases from a design perspective. In this article I aim to explore how a sound design can be implemented in an SQL-based database.

I have been interested in this topic for several years, both from a theory and practical perspective. After reading up on the topic, I decided to collect my thoughts and start writing. The previous article was based heavily on the book *Time and Relational Theory: Temporal Databases* by C. J. Date and others (see e.g. Amazon). In that article, I explored: (1) the notion of valid time versus transaction time and (2) the notion of what time is/how it can be represented in relational databases. I made a distinction between the conceptual level (where we structure data, often using modeling techniques such as ERD) and the implementation level (which deals with the representation in the underlying platform). In short, my conclusion was as follows:

For the time being, I have learned that dealing with time (and particularly with time intervals) should be trigger alarm bells. It seems so simple to simply add some attributes to your entity types and then build/generate your physical data model. Hopefully this short exploration convinces you that such topics require more careful thought.

Shortly after writing that article, I shared it with a dear friend to see what she thought. She responded as follows:

I have many thoughts about the concept of time in databases. We experience trouble with employee benefits and salaries all the time. Conceptually an employee could have zero or many benefits (insurance, retirement, tuition, wellness stipends, more). Those benefits may start and end at different times, so it is always a struggle to use effective dating of 'what the current benefits are' at any given time, if they even exist.

And... Have you thought about (and/or does Date address it) about the dimension of time in a warehouse? Here's a horrible example of time that causes us issues - we like to count how many applications/admits/deposits we have on any given date, compared to last year's count. It's a different concept but a major date struggle for us.

I thought it would be fun to tackle the former question (and set the latter aside, for now). The goal of this article, therefore, is to see if we can come up with a way to build and SQL-database using the theory as described in the previous article. I hope to give the interested reader some insight in the mechanics of more advanced database design topics. Lector caveat.

PostgreSQL

I'm going to play with this problem on my Mac Mini with a fresh install of the PostgreSQL DBMS. Note that this is an SQL-based dbms. It isn't a fully relational database (tables instead of relations, allowing null-"values", etc.) but it does a pretty good job.

Step one is to create a new database. We don't need any of the fancy options around templates and encoding. Therefore, we start simple:

```
CREATE DATABASE benefits;
```

We must also connect to the database using the `\c` command. After that, we are good to go.

Big picture analysis

If you're not careful, even this "simple" problem can become pretty big and messy. I'm going to focus on the first question here: "How do we deal with benefits that vary over time and figure out how to do that in a relational database?" I'm also going to make some assumptions to simplify the problem somewhat:

- I'm going to assume that benefits are for an EMPLOYEE and that we can uniquely identify EMPLOYEES with a number.
- Further, I'm going to assume that numbers are never re-used. So far, all the time a specific number, say 007, is used once and only once. If that weren't the case, we'd have to make a more complex key for EMPLOYEE, namely the combination of a number and a date.
- I'm also going to assume that BENEFITs of some type are of a fixed amount per month (i.e. if you have a retirement benefit then that amounts to X per month. We can later change this with a percentage if we want: if you have a 60% retirement benefit then you get 60% of X per month).
- Along the same lines as with EMPLOYEE, I am going to assume that the amount doesn't change over time. If we really want to, we can always introduce a timing mechanism but for the time being that would steer us away from the key point that we are trying to solve.

I will probably end up with at least the following relvars (tables): EMPLOYEE, BENEFIT, EMPBENEFIT. When we start playing with *dates*, this is likely to get even more tricky. We'll see.

Settting up EMPLOYEE

Since we're dealing with a dummy example, I'm not going to do anything fancy here. It is likely that an unique number, first name, last name etc. is enough.

Please note that we could work with transaction time here: when was a specific row (more formally: a tuple / proposition about our domain) added to the database. For the fun of it I'll add it, just so we can see what that looks like. As a reminder, the DATE data type uses a YYYY-MM-DD format.

The statement to create the table is:

```
benefits=# CREATE TABLE EMPLOYEE(
  EMPNR    INTEGER     NOT NULL,
  FNAME    VARCHAR(25) NOT NULL
  LNAME    VARCHAR(25) NOT NULL
  MODDATE  DATE        NOT NULL,
  PRIMARY KEY (EMPNR) );
```

Note that the NOT NULL part of the empnr attribute is probably superfluous since we're also making this our primary key. I've simply made it a habit to always add these constraints without fail: any design that allows NULL "values" (particularly in base relations/tables) is a poor design in my opinion. The predicate of this table is: the EMPLOYEE with number EMPNR, first name FNAME, and last name LNAME was added on date MODDATE. Seems reasonable enough.

All in all this looks good. We can start populating our EMPLOYEE table with some SQL INSERT statements, ending up with the following:

```
SELECT * FROM EMPLOYEE;
 empnr | fname |  lname   |  moddate
-------+-------+----------+------------
     7 | Bas   | Van Gils | 2002-02-01
     1 | Becky | Frieden  | 2001-01-01
   666 | Lisa  | Gaudette | 2000-06-01
(3 rows)
```

Aside: I probably should have made empnr a VARCHAR too just so I could give myself my usual number, being 007. End of a side.

Setting up benefits

With the assumptions that I made, this should also be straightforward, perhaps even more so. This time we don't even need a unique number to identify benefits. The set of attributes {BNAME} for benefit name is probably sufficient (note: a set with 1 element is still a set, hence the curly brackets).

```
CREATE TABLE BENEFIT(
  BNAME    VARCHAR(50) NOT NULL,
  BAMOUNT  INTEGER NOT NULL,
  MODDATE  DATE NOT NULL,
  PRIMARY  KEY (BNAME) );
```

After adding some data to this table, we end up with:

```
SELECT * FROM BENEFIT;
   bname    | bamount |  moddate
------------+---------+------------
 insurance  |     100 | 2000-01-01
 retirement |      80 | 2000-01-01
 tuition    |      80 | 2000-01-01
(3 rows)
```

In my simple setup, all the benefits have been added to our database on 1 January of 2000. Not too fancy, but it serves our purposes for now. At the very least the mechanism will allow us to query when certain benefits were added.

For the fun of it, we can now check what the database consists of:

```
           List of relations
 Schema |   Name   | Type  |   Owner
--------+----------+-------+------------
 public | benefit  | table | basvangils
 public | employee | table | basvangils
(2 rows)
```

I am going to be somewhat cheeky here and point out that the \d command apparently lists the relations in this database, yet in the Type column of the output we see that we're really dealing with tables. For the fun of it, I looked in the PostgreSQL specification and found this:

PostgreSQL is a relational database management system (RDBMS). That means it is a system for managing data stored in relations. Relation is essentially a mathematical term for table.

I very much disagree with this and I'll just conclude for the time being that the PostgreSQL spec is wrong and that "list of relations" should simply have said "list of tables" for that is what we're dealing with. Incidentally, asking ChatGPT doesn't give me any better answers.

Exporing date-intervals

Conceptually, date-intervals are tricky enough. I'm following the convention of C. J. Date to say that [a,b] is a date-interval that starts with date a (inclusive) and ends with dates b (inclusive). Had we used rond brackets then the date would be exclusive. For example, [c, d) would start at c (inclusive) and stop at d (exclusive). When working with intervals, we also have to be very precise in the grain that we use. For example:

- We use years only. Then we can say [2000, 2022) to indicate a period that starts at (including) the year 2000 and ends at (But excluding) the year 2021. It would be equivalent to [2000,2021].
- We use months and years. Then we can say [Jan-2000, Dec-2022) to indicate a period that starts at (including) January of 2000 and ends at (but excluding) December of 2022. It would be equivalent to [Jan-2000, Nov-20222]
- Etc.

Checking the PostgreSQL specification, I noticed that there is such a thing as an INTERVAL type. However, this type functions more as a "duration". The examples for interval input in the spec are shown in the table.

1-2	SQL standard format : 1 year and 2 months
3: 4:05:06	SQL standard format : 3 days 4 hours 5 minutes 6 seconds
1 year 2 months 3 days 4 hours 5 minutes 6 sseconds	Traditional Postgres format: 1 year 2 months 3 days 4 hours 5 minutes 6 seconds
P1Y2M3DT4H5M6S	ISO 8601 “format with designators”: same meaning as above
P0001-02-03T04:05-06	ISO 8601 “alternative format”: same meaning as above

While probably useful in various contexts, this is not really what I wanted. If I were to use this, I'd have to specify the startdate and the interval to get anything meaningful out of it.

We'll either have to use the classic solution with a table that has a start date and end date, or create a type of our own. I haven't done that very often yet, so after playing with the specification for a bit, I decided to ask my friend ChatGPT to generate the code for me. The query I gave ChatGPT is:

In postgresql, create a type called DINTERVAL that has a start date of type DATE and an end date of type DATE. Make sure that the start date is always before the end date. Also create a function that takes a date and an DINTERVAL as input and returns TRUE if the date is in the interval and FALSE otherwise. Also add a function that takes a DINTERVAL and computes the duration, defined as the difference between end date and start date.

In a few seconds it came up with the following:

```
-- create the type itself.
CREATE TYPE DINTERVAL AS (
    start_date DATE,
    end_date DATE
);

-- Create a function to check if a date
-- is within the interval:
CREATE OR REPLACE FUNCTION
date_in_dinterval(
  date_to_check DATE,
  dinterval DINTERVAL)
RETURNS BOOLEAN AS $$
BEGIN
  RETURN (date_to_check >=
    dinterval.start_date AND
    date_to_check <= dinterval.end_date)
END;
$$ LANGUAGE plpgsql;
```

```
-- Create a function to compute the
-- duration of the DINTERVAL:
CREATE OR REPLACE FUNCTION
dinterval_duration(dinterval DINTERVAL)

RETURNS INTERVAL AS $$
BEGIN
    RETURN dinterval.end_date -
      dinterval.start_date;
END;
$$ LANGUAGE plpgsql;
```

It also gave some suggestions for using the type/ functions as follows:

```
-- Check if a date is in the DINTERVAL
-- Returns TRUE
SELECT date_in_dinterval('2023-01-05',
my_dinterval);

-- Compute the duration of the DINTERVAL
-- Returns '9 days'
SELECT dinterval_duration(my_dinterval);
```

This actually looks pretty useful. It did omit the integrity constraint that the start date of a DINTERVAL should be before the end date, but I can live with that for now.

Setting up the benefits for employees

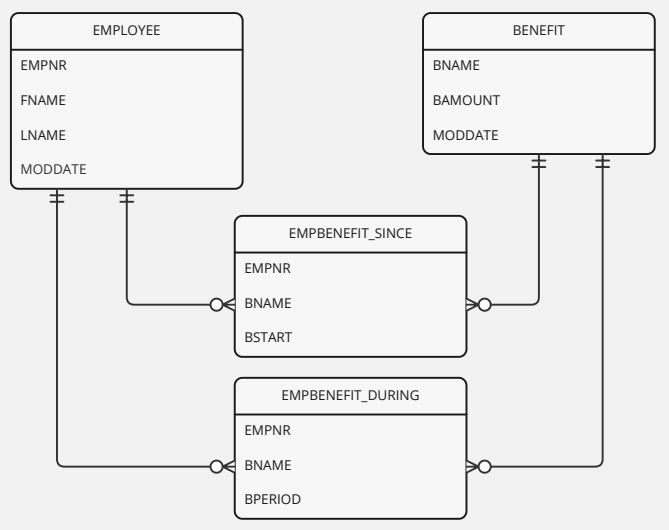
The theory suggests that we should distinguish between two relvars for storing data about BENEFITs of EMPLOYEEs:

- EMPBENEFIT_SINCE should reflect that an EMPLOYEE has a BENEFIT as of (and ever since) a certain start date. The predicate would be: the Employee with number EMPNR has benenfit with name BNAME since date BSTART.
- EMPBENEFIT_DURING should reflect that an EMPLOYEE had a BENEFIT during a certain period. The predicate would be: the Employee with number EMPNR had benenfit with name BNAME during period BPERIOD.

This would lead to the design shown below (incidentally, I created this with MIRO which now seems to have basic ERD modeling capabilities).

Note that there are foreign key constraints (FKs) between EMPBENEFIT_SINCE and EMPLOYEE and BENEFIT, as well as between EMPBENEFIT_ DURING and EMPLOYEE and BENEFIT. These basically specify that an EMPBENEFIT can only be registered if both the EMPLOYEE and the BENEFIT actually exist.

There is, however, an additional constraint that is missing here: it seems awkward for an employee to have an ongoing benefit as of some start date



listed in EMPBENEFIT_SINCE while, at the same time, that start date falls smack in the middle of a benefit period in EMPBENEFIT_DURING. This could potentially lead to the situation where we mistakenly conclude that the employee should be paid this benefit twice! The table creation is straightforward:

```
CREATE TABLE EMPBENEFIT_SINCE(
  EMPNR   INTEGER      NOT NULL,
  BNAME   VARCHAR(50)  NOT NULL,
  BSTART  DATE         NOT NULL,
  PRIMARY KEY (EMPNR, BNAME),
  CONSTRAINT
    FK_EMP FOREIGN KEY (EMPNR)
      REFERENCES EMPLOYEE(EMPNR) ,
  CONSTRAINT
    FK_BEN FOREIGN KEY (BNAME)
      REFERENCES BENEFIT(BNAME) );


CREATE TABLE EMPBENEFIT_DURING(
  EMPNR   INTEGER      NOT NULL,
  BNAME   VARCHAR(50)  NOT NULL,
  BPERIOD DINTERVAL    NOT NULL,
  PRIMARY KEY (EMPNR, BNAME,BPERIOD),
  CONSTRAINT
    FK_EMP FOREIGN KEY (EMPNR)
      REFERENCES EMPLOYEE(EMPNR) ,
  CONSTRAINT
    FK_BEN FOREIGN KEY (BNAME)
      REFERENCES BENEFIT(BNAME) );
```

Note that the DURING version is all-key (meaning: all attributes are part of the key). This is because we can have multiple periods in which an employee has a benefit. In the SINCE version that is not the case.

In TutorialD, the lanuage that C. J. Date uses in his book, it is easy to specify a constraint over two relations. In SQL it is less straight forward. I figured that, on top of the FKs, I'd probably need some fancy constraint that is implemented through a function that is triggered upon updates. Here's the query for ChatGPT (after it did such a good job the first time, I figured I'd try this again):

Assume that I continue to work in postgresql. Suppose I create an EMPBENEFIT_SINCE table that has three attributes: an EMPNR (integer), a BNAME (varchar) and a BSTART (date). I also create an EMPBENEFIT_DURING table with three attributes: an EMPNR (integer), a BNAME (varchar) and a BPERIOD (DINTERVAL). I want to implement an integrity constraint that specifies that the BSTART in EMPBENEFIT_SINCE must always after the end date of the BPERIOD in EMPBENEFIT_DURING (if it exists). How can I do this?

Surely enough it first spits out the creation of the DINTERVAL type and simplified versions of the appropriate relations (it couldn't have figured out the FK constraints because I didn't list them in my query). It then indeed generates a nice function that does the appropriate check and is triggered upon inserts and updates:

```
-- define the function that does the check
CREATE OR REPLACE FUNCTION
  check_benefit_start_date()
RETURNS TRIGGER AS $$
BEGIN
    -- it included this but it doesn't
    -- seem necessary we certainly do not
    -- want to allow NULL "values"
    -- IF NEW.BSTART IS NULL THEN
    --    RETURN NEW;
    -- END IF;

    IF EXISTS (
        SELECT 1
        FROM EMPBENEFIT_DURING b
        WHERE
          b.EMPNR = NEW.EMPNR
          AND b.BNAME = NEW.BNAME
          AND (b.BPERIOD).end_date >=
            NEW.BSTART
    ) THEN
      RAISE EXCEPTION 'BSTART must be
        after the end of the BPERIOD.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;


-- create the trigger that enforces the
-- constraint
CREATE TRIGGER
  enforce_benefit_start_constraint
BEFORE INSERT OR UPDATE ON
  EMPBENEFIT_SINCE
  FOR EACH ROW
    EXECUTE FUNCTION
      check_benefit_start_date();
```

I will admit that I'm pretty impressed at this point: ChatGPT's code may not be perfect but with some minor tweaking it gets the job done.

Additional constraints

Before diving into data entry, I realized that there is a set of constraints that we currently cannot implement. It has to do with transaction time versus valid time in the EMPLOYEE and BENEFIT tables (relations). Note that, currently, we only have transaction times in these tables. We could interpret this as "an employee/benefit exists as soon as it is entered in the system and until it is removed from the system" but that is not how the real world works as the following example shows. It could very well be the case that someone started working for us as of 1-feb-2022 but we didn't have time to enter this into the system until 03-mar-2023. The same goes for benefits.

The implication for entering the benefit of an employee in the EMBENEFIT tables is that we must check whether the start date in EMPBENEFIT (in either the SINCE or DURING version) is after the valid date in both EMPLOYEE and BENEFIT, regardless of the transaction date. To put it differently: we have to check if the employee is really a valid employee and a benefit is really a valid benefit. This will be a tricky constraint to implement. It will probably require another function and associated trigger. Given that I haven't implemented the valid time attributes, I'll leave this for the time being.

Adding benefits of employees

Adding data isn't too hard. In the DURING-version of the table, you have to take care to use periods rather than simple dates but that is doable. As an example is as follows:

```
INSERT INTO EMPBENEFIT_DURING
  ( EMPNR, BNAME, BPERIOD )
VALUES ('7', 'insurance',
  ('2003-01-01'::DATE,
    '2003-12-31'::DATE) );
```

After a few INSERTs, we end up with the following:

```
SELECT * FROM EMPBENEFIT_DURING;
```

```
 empnr |   bname   |          bperiod
-------+-----------+---------------------
     7 | insurance | (2003-01-01,
       |           |  2003-12-31)
     7 | insurance | (2005-01-01,
       |           |  2005-12-31)
     7 | tuition   | (2004-01-01,
       |           | 2004-06-30)
   666 | tuition   | (2004-01-01,
       |           |  2004-12-31)

(4 rows)
```

Now we can check our constraint for adding data to the EMPBENEFIT_SINCE. Recall that the whole point of this constraint is to ensure that the start date in the SINCE table is after all of the "end of period" in each of the associated PERIODs in the DURING table, if any. We'll try to add an benefit of insurance for person with number 7, starting in

2007 (which is well after the last period was closed).

```
INSERT INTO EMPBENEFIT_SINCE
  (EMPNR, BNAME, BSTART)
VALUES
  ('7', 'insurance', '2007-01-01'::DATE)
INSERT 0 1
```

That went well: 0 errors and 1 row inserted. Now lets try for failure. We're going to add a tuition benefit for the employee with number 666 that starts in the middle of 2004:

```
INSERT INTO EMPBENEFIT_SINCE
  (EMPNR, BNAME, BSTART) VALUES
  ('666', 'tuition', '2004-07-01'::DATE);
ERROR:  BSTART must be after the end of
the BPERIOD.
CONTEXT: PL/pgSQL function check_benefit_
start_date() line 17 at RAISE
```

That seems to work really nicely: we get an error. When we change 2004 to 2007, the insert statement does work, as expected:

```
INSERT INTO EMPBENEFIT_SINCE
  (EMPNR, BNAME, BSTART) VALUES
  ('666', 'tuition', '2007-07-01'::DATE);
INSERT 0 1
```

To be clear, the current population of this table is as follows:

```
SELECT * FROM EMPBENEFIT_SINCE;
 empnr |   bname   |   bstart
-------+-----------+------------
     7 | insurance | 2007-01-01
   666 | tuition   | 2007-07-01
(2 rows)
```

Let's say that the tuition-benefit of 666 ends at the end of 2007. We can simply add the row to EMPBENEFIT_DURING:

```
INSERT INTO EMPBENEFIT_DURING
  (EMPNR, BNAME, BPERIOD)
VALUES
  ('666', 'tuition',
    ('2007-07-01'::DATE,
     '2007-12-31'::DATE));
INSERT 0 1
```

This gives the following population of that table:

```
SELECT * FROM EMPBENEFIT_DURING;
 empnr |   bname   |         bperiod
-------+-----------+---------------------
     7 | insurance | (2003-01-01,
       |           |  2003-12-31)
     7 | insurance | (2005-01-01,
       |           |  2005-12-31)
     7 | tuition   | (2004-01-01,
       |           |  2004-06-30)
   666 | tuition   | (2004-01-01,
       |           |  2004-12-31)
   666 | tuition   | (2007-07-01,
       |           |  2007-12-31)
(5 rows)
```

This should not be possible! Note that we have a row in the DURING Table for a tuition benefit of employee 666 that starts at 1 july 2007, but we also have a row in the SINCE table that has a tuition benefit of the employee 666 that starts at 1 july 2007. Our trigger-constraint should have prevented this.

Looking back, I it is easy to see what went wrong: the trigger is on an insert in the SINCE table only, and not the DURING table - so if we really want to get this correct, we'll have to expand our constraints once more. As it stands, we'll have to remove the corresponding row from the SINCE table anyway. This illustrates a) why good design is so important, and b) some of the challenges around the way these constraints must be implemented in SQL-based databases. Expanding this point would require another blog post, so I'll set it aside for now (and perhaps write that blog in a little while).

In a realistic scenario, the two updates (one insert, one delete) should occur as one whole. In TutorialD this is done with a compound statement. In any SQL database, we are stuck wrapping this in a transaction if we really want to treat the two as one whole.

Querying the database

The last step to explore here is querying the database. With the amount of tables (relations) that we have, that may seem daunting. Luckily, though, it is actually the inverse: querying the database should be mostly straightforward thanks to our good design.

To substantiate that claim, I will present with two observations and an example. The observations are:

- The data about past/completed benefit periods are stored in the _DURING table. If we want to know something about the past, we only have to look at this table.

- The data about ongoing benefit periods are stored in the _SINCE table. If we want to know something about the present, we only have to look at this table.

So far so good! This leaves the matter of queries that touch upon both tables. This would be the case where we combine data about the past and current situation. An example would be: show me the benefits of a specific person and make sure to include both past and ongoing benefits. Observe that the _DURING table (past benefits) has a column (attribute) that holds periods with

a start date and an end date. However, the _SINCE (ongoing) table has a column (attribute) with a start date only. The two do not match so we cannot simply perform a union on two result sets. The solution lies in the combination of:

- Unpack the period from the _DURING table in a start_date and an end_date.
- Extend the results from the _SINCE table with a column that has NULL "values" for the end_date. Note: I am very much against using NULLs in (base) relations, but for such a query it is actually useful and perhaps even necessary.

The code would be as follows:

```
SELECT EMPNR AS NR, BNAME AS NAME,
  (BPERIOD).start_date as START,
  (BPERIOD).end_date as END
  FROM EMPBENEFIT_DURING
  WHERE EMPNR='7'
UNION
  SELECT EMPNR AS NR, BNAME AS NAME,
  BSTART AS START,
  NULL AS END
  FROM EMPBENEFIT_SINCE
  WHERE EMPNR='7';
```

And the result of this query would be:

```
nr |  name     | start      |  end
---+-----------+------------+----------
 7 | tuition   | 2004-01-01 | 2004-06-30
 7 | insurance | 2005-01-01 | 2005-12-31
 7 | insurance | 2003-01-01 | 2003-12-31
 7 | insurance | 2007-01-01 |

(4 rows)
```

That looks fairly painless. On top of the unpacking and addition of a column for purposes of the JOIN, I also did some renaming to make sure the table has a "friendly" look.

Conclusion

The objective of this blog post was to 'play' with the theory around (a) transaction/valid time and (b) time intervals in a relational / sql database. As pointed out in several places, relational databases aren't really relational. Based on a relatively simple use case from the real world, I set out to see how to implement things properly in an SQL databases. My lessons learned is as follows:

- I still like playing with PosgreSQL. It's a fun database with good documentation.
- I must admit that my SQL skills (particularly for defining triggers and functions) is a bit rusty. This exploration did give me the opportunity to play with ChatGPT a little. ChatGPT gave me surprisingly good results (despite the fact that I had to tweak the code just a little).
- The conceptual model for this case is not super complex. Implementing it is another story.
- Defining my own type for DINTERVAL (date intervals) was straightforward, but setting up the appropriate integrity constraints was far from easy. Even when doing this carefully, step by step, I still missed some constraints.
- Having to work with functions/triggers makes it very difficult to implement relatively simple / obvious integrity constraints.
- I still wish a relational database system (e.g. PostgreSQL) were truly relational ;-)

All in all, I do believe that a "proper" implementation is possible, particularly for developers that have a lot more experience in implementing constraints using functions and triggers. I also think it is worth it to go through these steps. You're building a database for a reason: because you want to have data that captures your understanding of a domain such that it can stand for that domain. Why would you not spend the time to come up with a good design?

I hope you find this article interesting. If you have some thoughts or comments, please feel free to drop me a note. I'll be thinking about this topic for a while longer. Thanks!