NORM Hierarchies

Norm

1 Introduction

This section briefly outlines the NORM approach and the problems that it addresses. More detailed descriptions can be found in numerous conference talks, blogs, posts, and a book on PostgreSQL query optimization.

The origin of NORM is not clear as it goes back into the dark ages. Some sources insist it stands for 'No ORM,' where ORM is 'an object-relational mapper,' but this looks doubtful even though the authors insist.

It is more likely that NORM addresses the impedance mismatch, first identified in the dark ages (before the Internet) when powerful database query languages started becoming practical.

The models and languages widely used in the database world significantly differ from the application models and languages. Consequently, the interactions between an information system's database and application parts result in severe problems affecting development, maintenance, and performance. Several attempts were made to resolve this, but they have failed. Any attempt to adjust one of the models to the needs of others results in poor design on the adjusted side. For example, ORMs attempt to enforce application models on the database side, effectively turning off the powerful capabilities of database management systems.

Thus, any serious attempt to resolve the impedance mismatch must accept that models differ. The approach of NORM is to leave both interacting sides with their favorite models and define yet another model for their interaction. Thus, NORM requires two mappings instead of one: the intermediate transfer model must be mapped to both the application and the database.

The advantages are not so obvious. Neither of the mappings must be complete in any sense. The model and language used for the interaction can be tailored to data transfer. Both mappings may preserve (map) complex data structures on both sides.

NORM improves the application performance by transferring as much data as possible in any interaction and thus reducing the number of network round trips.

The data transfer unit in NORM is a hierarchical document called hierarchy for brevity. An example of a hierarchy is an airfare booking that includes some heare information, a list of passengers, and a list of booking legs linking the booking to scheduled flights. Another example is the manifest, which contains information on a flight and everything loaded onto the aircraft (passengers and/or cargo).

A hierarchy is the most straightforward data structure that is more complex than flat files (lick CSV or plan SQL output). At the same time, it is the most complex structure that can be easily represented sequentially. That is why it is widely accepted for formats used to transfer data over the network (SGML, XML, JSON).

2 Defining Hierercheis with JSON Schemas

Norm hierarchies are built from typed objects. Any object in the hierarchy may have child objects of other types. Any single instance of the parent object may have exactly one child object or an array of child objects of the same type as an attribute and may have multiple types of child objects as its attributes.

For example, `booking` object has an array of `booking_leg` objects and an array of `passenger` objects and its child objects. The `booking_leg` object has a single `flight` object as its attribute.

The NORM hierarchies are formally specified with JSON schemas. Many features of JSON schema language are not used in NORM. A hierarchy is defined as an array. The table˜1 describes the keys that NORM can interpret.

Table 1: JSON schema keys for NORM hierarchies

key	description
title	name identifying the hierarchy.
description	Informal description of the hierarchy
items	contains the reference to the root object type
definitions	contains definitions of all objects in the hierarchy (including the root object)

An example of a hierarchy definition header:

```
{"type": "array",
 "title": "booking_hierarchy",
 "description": "all booking details",
 "items": {"$ref": "#/definitions/booking"},
 "definitions": { . . . } }
```

All objects in the `definitions` key are defined at the same level. The hierarchical structure is specified with references to other objects using the `$ref` key and relative addressing within the JSON schema.

The `definitions` key of the JSON schema contains a list of objects where the object type name is represented as a key. The value of this key is a complex JSON object. Table˜2 describes keys used in the object definitions.

A list of object attributes is placed as a value of /tt properties key in the object definition.

Table 2: JSON schema keys for NORM objects

key	description
type	The value of the defined object is usually `object`.
properties	A list of object attributes. The JSON key specifies the name of an attribute.

The attribute specification can contain the key `type`. The type of an attribute can be scalar (`string` or `number`), `object`, or `array`. A key `items` must reference another object's definition for attributes that are not scalar. The syntax of JSON schemas allows the specification of sub-objects in-line, but NORM currently does not accept this option. NORM also currently ignores additional specifications (such as `format`.

The definition of `booking` object is included below as an example.

```
"booking": {
    "type": "object",
    "properties": {
        "booking_id": {
            "type": "number"
        },
        "passengers": {
            "type": "array",
            "items": {
                "$ref": "#/definitions/passenger"
            }
        },
        "booking_legs": {
            "type": "array",
            "items": {
                "$ref": "#/definitions/booking_leg"
            }
        },
        "booking_reference": {
            "type": "string"
        }
    }
},
```

The above definition specifies that `booking` object has `booking_id`, `passengers`, `booking_legs`, and `booking_reference` as its attributes. The attributes `passengers` and `booking_legs` arrays of objects of types `passenger` and `booking_leg`, respectively. These objects are defined separately as separate keys under the `definitions` key.

The definition of `booking_leg` includes the attribute `flight` of type `object` tat refers to `flight` object definition:

```
"booking_leg": {
    "type": "object",
    "properties": {
        "flight_id": {
            "type": "number"
        },
        "flight": {
            "type": "object",
            "items": {
                "$ref": "#/definitions/flight"
            }
        },
        "leg_num": {
            "type": "number"
        },
        "booking_leg_id": {
            "type": "number"
        }
    }
}
```

The complete `booking_hierarchy` schema and more examples can be found in the eNORM repository.

3 Designing Hierarchies

All data transfers between an application and a database operate on sets of hierarchically structured objects. In terms of old classical database textbooks (does anyone read them?), NORM provides a hierarchical data model, although the data structures in the database itself can be more complex.

Typically, an application needs multiple hierarchies supporting different business functions that might be implemented even as different microservices can still share some parts of stored data.

In order to enable NORM performance benefits, each hierarchy should include all data needed for a specific business function. For example, the booking service includes complete flight information. The same flight information is also included in the timetable service.

The designer should remember that hierarchies are assembled when requested rather than stored anywhere in the database. Consequently, there is no need to avoid redundancy in the definitions of hierarchies. There are better ideas than just using hierarchies that only partially meet application needs and combining data from different hierarchies on the application side. A new hierarchy that perfectly meets application requirements should be defined instead. Of course, an application function may require data from different hierarchies. For example, to re-book a passenger to another flight, access to the timetable is needed to find a new flight, as the booking hierarchy includes flights for the booking rather than a complete timetable.

However, the number of hierarchies and data transfers must be kept reasonably small. A proven method to get extremely poor performance with NORM is to define a hierarchy for each database table and transfer only one row in each request.

4 Mapping Hierarchies to the Database

The JSON schema (contract) mapping to the database is specified with an additional JSON key `db_mapping`.

The JSON schema language does not specify this key, so hopefully, third-party validators of JSON schemas will ignore it. Mapping is defined at different levels: the hierarchy, objects, and components. The mapping definition can use several keys under the `db_mapping` key.

4.1 Mapping Hierarchy

The following keys can be used at the hierarchy level:

`db_schema` specifies the PostgreSQL schema name where the tables containing data for the hierarchy are stored. This value is default and can be overridden for any object.

`db_prefix` specifies the character string that is concatenated with names of generated database objects (type definitions and functions). This key is optional; the default value is the hierarchy title. The prefix is needed to ensure that objects generated for different hierarchies have different names.

`norm_schema` The PostgreSQL schema where the generated object will be stored. The default is the same schema as for tables.

4.2 Mapping Objects

The following keys can be specified at the object level:

`db_schema` specifies the PostgreSQL schema where the table is located. Need only if it differs from the schema specified for the hierarchy.

`db_table` is this object's base table name.

`pk_col` the column n that references the parent from the child objects. Typically, it is the primary key of the base table.

`parent_fk_col` the column referencing the parent object. Usually, it is a foreign key, but NORM does not check that.

`db_expression` specifies that the value of `db_table` is a table expression. The correct values of this key are "Y" and "N" only; the default is "N".

`record_type` is the name of the generated type definition. It is used for outgoing types only and is needed for historical reasons.

`embedded` contains a JSON array of specifications of additional tables used as a data source for this object. The format of these specifications is described below.

An object may include fields extracted from tables other than the base table. The mapping relies on aliases in order to identify these additional tables. Each table is defined as a separate JSON object in the `embedded` key and includes the following keys:

`alias` identifies the additional data source.

`db_schema` is the schema name needed if it differs from the schema for a base table for this object.

`db_table` The name of the table or table expression,

`pk_col` is the column identifying the row, typically the primary key.

`fk_col` the column in the base table referencing the row of the additional table.

`db_expression` The value should be "Y"if the `db_table` is a table expression. The default is "N".

The design of `embedded` key may look overcomplicated. However, the same additional table may be needed multiple times. For example, the `flight` object contains `departure_city` and `arrival_city`. Both reside in the `airport` table but in different rows. For this reason, the mapping for flight contains aliases `departure` and `arrival` that provide links to the `airport` table with different `fk_col` columns.

4.3 Mapping Object Attributes

The mapping at the property (field) level may include the following keys:

`db_col` column name of the base table or scalar expression. This key is optional. It is unnecessary if the JSON object property's name coincides with the table column's name.

`db_type` the PostgreSQL type of the value of `db_col`. NORM uses this value only if it cannot find the type automatically (from the PostgreSQL catalog).

`db_expression` Should be "Y" if the value of `db_col` is an expression. The default is "N".

`db_source_alias` is an alias of the additional table.

Mapping of composite JSON fields (that is, objects and arrays) requires additional clarifications. If a JSON field is an array, it refers to another object and its mapping. In contrast, for fields that have JSON type `object`, the key `db_col` specifies the column that refers to a single object instance (that is, to a row in the base table of the object.

The inclusion of objects is similar to the inclusion of columns from additional tables. The differences are:

- The data in the composite object is surrounded with braces (so that it is a JSON object, while the data from additional tables reside on the same level as other scalar fields.
- The nested object may have a complex structure, e.g., containing arrays and objects.

An extended example is needed here.

Several examples of mappings can be found in the NORM repository.

4.4 Expressions

An application may require data that are not stored in the database but can be calculated from the stored data. For example, a booking includes a departure date and time in the local time zone of the departure airport. However, the `flight` table contains only the departure timestamp in the UTC. The calculation of the departure date also requires a time zone from the `airport` table. The following mapping specifies the calculation on the database side:

```
"departure_date": {
    "type": "text",
    "db_mapping": {
        "db_type": "text",
        "db_expression": "Y",
        "db_col":"to_char(
           scheduled_departure
               at time zone departure.airport_tz,
           'YYYY-MM-DD')"
     }
 },
```

Note that `scheduled_departure` and `airport_tz` are extracted from different tables: the timestamp `scheduled_departure` is a column of `flight` table (the base table for the object), while `airport_tz` comes from the `airport` table specified with the alas `departure`.

The mappings for departure time, arrival date, and arrival time are similar.

The table expressions can be specified instead of table names as values of db_table or in the list of aliases at the object level. For example, if the /tt flight object includes the following object definition, an application can receive summary values for a flight.

```
"summary":{
       "type": "object",
       "db_mapping": {
           "db_table":"(
               select flight_id, count(*) as passenger_cnt
               from boarding_pass
               group by flight_id)",
           "db_expression": "Y",
           "pk_col": "flight_id",
           "parent_fk_col":"flight_id",
           "record_type": "pass_summary"
       }, ...}
```

However, such inclusion of the table expressions may cause poor performance of generated queries. The problem is that generated filtering criteria are applied to the table expression. Pushing the filtering conditions inside the table expression can resolve this issue, but the generator cannot. In some cases, this issue can be resolved using database views instead of table expressions in the mapping.

5 Installing NORM

All needed components (type definitions, tables, and functions) are placed in the `NORM_GEN` database schema. Currently, this schema name is hard-coded. The repository contains a SQL script `load_all` that creates all required database objects.

6 Preparing the Schema with Mappings for Generation

Any JSON schema defining NORM hierarchy must be compiled into NORM internal tables using `norm_gen.ts_all` function. The following `psql` script compiles the JSON schema stored as a file `booking_hierarchy.json`:

```
\set  p_schema ‘cat booking_hierarchy.json‘
select norm_gen.ts_all(:’p_schema’::json);
```

This SELECT statement returns a value of the surrogate key assigned to the compiled schema. Note that the value of the surrogate key is changed after re-compilation and, therefore, can hardly be used for identification. It is safe to use the schema title instead. The internal NROM tables can store multiple hierarchies with different titles simultaneously.

7 Queries

The NORM queries follow the same pattern. Any query returns a single value, an array of complete hierarchies satisfying the contract. Filtering conditions can be specified for attributes for attributes of nested objects at any level of the hierarchy. If at least one instance of nested object satisfies the condition, then the whole hierarchy is included in the output.

For example, suppose the conditions specify a passenger's last name, departure date, and departure airport. In that case, the query will return all bookings that contain such passenger and flight on that date from the specified airport, including all passengers and all flights on these bookings.

NORM queries should not use explicit joins because all object relationships should be represented in the hierarchical structure.

The query output is produced as a PostgreSQL array of composite objects that can contain other composite objects and arrays as attributes. The structure of these objects is defined with PostgreSQL types generated from the JSON schema. The purpose of these types is twofold:

- to enforce the correct structure of the output
- to provide proper JSON key names.

7.1 Building Query Output

The function

```
 norm_gen.generate_types(p_schema_title text)
returns text
```

returns the SQL script containing CREATE TYPE statements from the contract (JSON schema) identified by its title. The types are created in the bottom-up order.

The function

```
 norm_gen.nested_root( p_hierarchy text) returns text
```

returns SELECT and FROM clauses of a query. These clauses depend on the JSON schema, as any NORM query returns an array of full hierarchical objects of the same (root) type. To produce a complete query, the output of this function must be appended with the WHERE clause.

The following code builds `booking_hierarchy`:

```
/* selecting booking_hierarchy booking */
select
          array_agg(
   /* Entering booking_record */
   row(top.booking_id  ,
      (
    select array_agg(  /* Entering passenger_record */
    row(passengers.last_name  ,
    passengers.account_id,
    passengers.first_name,
    passengers.passenger_id,
    passengers.passenger_no)::norm.bh_passenger_record)
from  postgres_air.passenger passengers
were top.booking_id = passengers.booking_id
)
  ,
       (
    select array_agg(  /* Entering booking_leg_record */
    row(booking_legs.flight_id  ,
    (
select  (  /* Entering flight_record */
    row(flight.flight_no  ,
    arrival.city  ,
    departure.city  ,
    flight.scheduled_arrival  ,
    flight.scheduled_departure  ,
    flight.arrival_airport  ,
    arrival.airport_name  ,
    flight.departure_airport  ,
    departure.airport_name)::norm.bh_flight_record)
from  postgres_air.flight  flight
   join postgres_air.airport departure on departure.airport_code = flight.departure_airp
   join postgres_air.airport arrival on arrival.airport_code = flight.arrival_airport
where  booking_legs.flight_id = flight.flight_id
)
  ,
    booking_legs.leg_num  ,
    booking_legs.booking_leg_id)::norm.bh_booking_leg_record)
from  postgres_air.booking_leg  booking_legs
where  top.booking_id = booking_legs.booking_id
)
  ,
    top.booking_ref)::norm.bh_booking_record)
from  postgres_air.booking  top
```

NORM promotes the use of stored functions. The function

```
norm_gen.generate_select_by_id_function(
  p_schema_name text,
  p_root_object_name text default null)
returns boolean
```

generates and stores in the database a function that combines generated SE-

LECT and FROM clauses with filtering criteria built from an array of primary key values for the root of the hierarchy and then executes the resulting SQL dynamically. The returned result indicates the success or failure of the generation. It is not an easy task to find out the name of the generated function.

7.2 Specifying Filtering Conditions

NORM generates filtering conditions from a search specification expressed in JSON format from simple conditions on attributes of any object in the hierarchy. The simple condition can be in one of two forms:

```
{"<object-key>":{"<predicate>":"<value>"}}
{"<object-key>":"<value>"}
```

Currently, available predicates are listed in the table 3. F, the second form of the simple condition "$eq" is the predicate.

Table 3: Binary predicaates

key	SQL op	description
$eq	=	equals to
$lt	¡	less
$le	¡=	less or equal
$ne	¡¿	not equals
$ge	¿=	greater or equal
$gt	¿	greater
$like	LIKE	SQL LIKE predicate
other	UNKNOWN	Forces SQL error

Simple conditions are placed into a hierarchy defined in the JSON schema of the query. The top key is the schema title; other keys represent object types for which simple conditions are set. The object types are needed only to resolve ambiguities (same attribute name in different object types).

An example of search specification looks as follows:

```
{
"booking_hierarchy":{
"departure_airport_code":"ORD",
"arrival_city":{"$like":"NEW%"},
"last_name":"Smith"}
}
```

This specification is converted into the following SQL conditions that can be placed in WHERE clause appended to a generated SELECT-FROM and potentially combined with hand-crafted conditions:

```
  booking_id IN (
    select booking_id from postgres_air.booking_leg where
    flight_id IN (
   select flight_id from postgres_air.flight where
    arrival_airport IN (
   select airport_code from postgres_air.airport where
    city  LIKE  ('NEW%'::text) )
AND  departure_airport  =  ('ORD'::bpchar) ) )
AND  booking_id IN (
   select booking_id from postgres_air.passenger where
    last_name  =  ('Smith'::text) )
```

The function `build_conditions` iconverts JSON specification into SLQ conditions;

```
norm_gen.build_conditions (p_in json)
returns text
```

Finally, the following function generates an executable function that combines generated SELECT-FROM with search conditions and executes dynamic SQL:

```
norm_gen.generate_search_generic_function(
  p_schema_name text,
  p_root_object_name text default null)
returns boolean
```

Note that although the NORM search specification may resemble other languages designed for querying JSON documents, no attempt is made in NORM to implement any of these languages or any subset of them.

8 Modifications

NORM provides means for automatic modification under the assumption that all base tables of the objects in the hierarchy have automatically assigned surrogate primary keys (selected from a database sequence or produced by a trigger).

An application forms an update request as an array of JSON hierarchies. The structure of these hierarchies follows the JSON schema with some additions. The following rules guide the interpretation of an update request:

- If an object at any level of the hierarchy contains an attribute's value mapped to the base table's primary key, this object is updated with values of other attributes specified in the request. Only attributes mapped to an object's base table may be updated.
- If an object contains a value of attribute mapped to the primary key and additional key "cmd" (probably should be "$cmd') with value "DELETE" then this object and all its descendants are deleted from the database.

- If an object does not contain a value mapped to the primary key, then all its descendants should not contain values for primary keys, and the object is inserted together with its descendants. New surrogate values of primary keys are assigned to inserted objects.

An example of an update request is shown below:

```
[
    {
        "booking_id": 556470,
        "cmd": "DELETE"
    },
    {
        "passengers": [
            {
                "last_name": "Jones",
                "account_id": 238648,
                "first_name": "Lucy"
            }
        ],
        "booking_legs": [
            {
                "leg_num": 1,
                "flight_id": 558238,
            },
            {
                "leg_num": 2,
                "flight_id": 563410
            }
        ],
        "booking_reference": "IYZI42"
    },
    {
        "booking_id": 3974917,
        "passengers": [
            {
                "cmd": "DELETE",
                "passenger_id": 11479596
            },
            {
                "last_name": "SCOTT",
                "first_name": "MILES",
                "passenger_id": 11479599
            }
        ],
        "booking_legs": [
            {
                "flight_id": 432724,
                "booking_leg_id": 11453272
            },
            {
                "flight_id": 427273,
                "booking_leg_id": 11453273
            }
        ]
    },
    {
        "booking_id": 2733047,
        "booking_reference": "Q8JX22"
    }
]
```

All modifications are performed with bulk operations. For any update request, NORM executes at most one DELEtE, one UPDATE, and two INSERT SQL statements from each object type in the JSON schema, no matter how many instances of the objects are included in the update request.

The following function generates several PostgreSQL types and functions needed to perform modifications of hierarchies defined in a JSON schema:

```
norm_gen.build_to_db (p_hierarchy text) returns text
```

Finally, NORM provides a high-level function that executes the script generated by `build_to_db`:

```
norm_gen.generate_to_db_function(
  p_schema_name text)
returns boolean
```

9 Known Issues

'key names The JSON keys are case-sensitive but are derived from PostgreSQL-type attribute names that are case-insensitive. As a result, the key in the outgoing JSON objects may differ from those specified in the contract. The fix is easy, but the repository owner hates double-quoted names in SQL.

NORM_GEN The schema name NORM_GEN is hard-coded.

10 Conclusions

NORM provides functionality similar to ORMs but avoids several pitfalls due to implementation on the database rather than the application side of the interaction. The generated code is sufficient for most common kinds of interactions between application and database.

In contrast with other ORMs, NORM encourages bulk operations on the database and thus provides for unbeatable performance benefits.

References

NORM github
The book
Other posts and presentations