Data modeling for modern SQL applications: 3NF? ARRAY? JSONB?

Franck Pachot, Developer Advocate

yugabyteDB

Franck Pachot

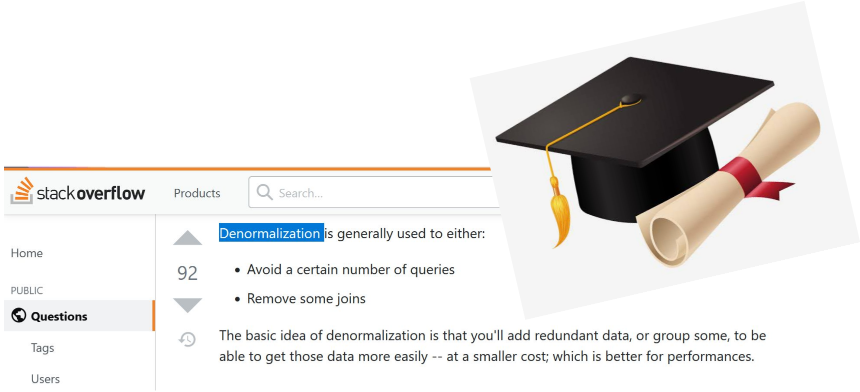
Developer Advocate on YugabyteDB
(PostgreSQL-compatible distributed database)

Past:

- 20 years in databases, dev and ops
- Oracle ACE Director, AWS Data Hero
- Oracle Certified Master, AWS Database Specialty

fpachot@yugabyte.com
dev.to/FranckPachot
@FranckPachot

Normalization



Denormalization is generally used to either:

- Avoid a certain number of queries
- Remove some joins

The basic idea of denormalization is that you'll add redundant data, or group some, to be able to get those data more easily -- at a smaller cost; which is better for performances.

I have heard a lot about normal forms at university

I've mostly heard about denormalization once at work

Normalization... why? Wrong answers only 👇

According to MongoDB:
avoid data duplication because of the cost of storage
https://www.mongodb.com/nosql-explained

According to DynamoDB:
same words: optimize of storage so not needed today
AWS re:Invent 2018: Amazon DynamoDB Deep Dive

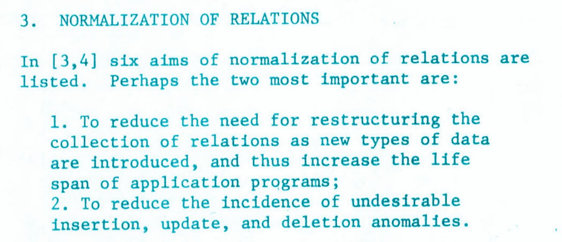
🤔 Relational theory, invented by a mathematician (Codd)
was driven by storage obsession?

Normalization... why?   Better ask Codd

E. F. Codd, Recent investigations in relational data base systems

https://dl.acm.org/doi/10.1145/1734714.1734716

https://purl.stanford.edu/ys277xx1104



3. NORMALIZATION OF RELATIONS

In [3,4] six aims of normalization of relations are listed. Perhaps the two most important are:

1. To reduce the need for restructuring the collection of relations as new types of data are introduced, and thus increase the life span of application programs;
2. To reduce the incidence of undesirable insertion, update, and deletion anomalies.

- Data Integrity
  (undesirable insertion, update and deletion dependencies)
- Agility
  (reduce the need for restructuring as new type of data is added)
- Be more informative to users
- Logical - Physical independence

Normalization... How

Forget about normal forms...

- Separate the business concepts that can be queried / updated independently in your system (*)
- Group into same table those that are tightly linked

(*) Example: Address + ZIP code + City + Country

- may be attributes of same entity in social media application
- is probably normalized to multiple tables in a Post Office application

Denormalization... When

- Want a simple data structure, that will not evolve
  👉 microservice with one use case only

- Got the impression that "Joins don't scale"
  👉 pre-join data for the main use-case

- Use more cheap storage? 😂
  No! You will need more indexes and foreign keys on a normalized data model

Ok, enough theory... facts and examples

Let's build a messenger, with tags and groups

- a post from a user, with content, at timestamp
- it has a list of tag_id and a list of group_ids

Access patterns:

- put a post into the database, with all related information
- get posts by tag, ordered by last timestamp
- get posts by group, ordered by last timestamp

Relational design: Entities and Relationships

Let's build a messenger, with tags and groups

- a post from a user, with content, at timestamp
- it has a list of tag_id and a list of group_ids

Primary keys: user_id, tag_id, group_id, post_id

we will not detail reference tables here (users, tags, groups)

To record a post, we need the following tables:

- "**posts**" records (post_id) -> user_id, content, timestamp
- "**post_tags**" lists (tag_id, post_id)
- "**post_groups**" lists (group_id, post_id)

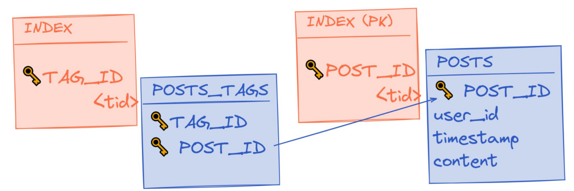
Relational design: Heap tables and B-Tree indexes

Get post by tag:

- index tag_id -> table **post_tags** *tid*
- table **post_tags** *tid* -> (post_id, tag_id)
- index post_id -> table **posts** *tid*
- table **posts** *tid* -> (post_id, user_id, content, timestamp)

To record a post, we need the following tables:

- **posts** to record (post_id) -> user_id, content, timestamp
- **post_tags** to list (tag_id, post_id)
- **post_groups** to list (group_id, post_id)



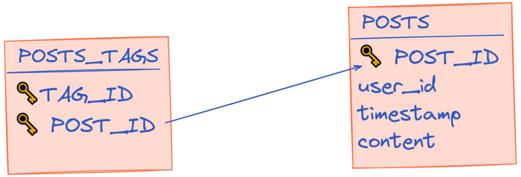
Relational design: Index Organized Tables (LSM Trees)

Get post by tag:

- primary index on **post_tags** tag_id -> post_id
- primary index on **posts** post_id -> (post, user_id, content, timestamp)

To record a post, we need the following tables:

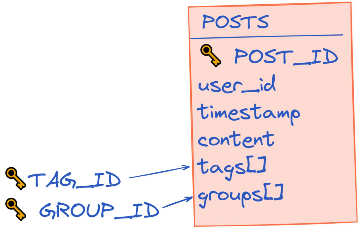
- **posts** to record (post_id) -> user_id, content, timestamp
- **post_tags** to list (tag_id, post_id)
- **post_groups** to list (group_id, post_id)



Single Table design: ARRAY

Do you need so many tables?

- (post_id, tag_id)&(post_id, group_id) can be stored as with each post_id as (post_id) -> array of post_id's, array of group_id's
- but only if you can still lookup by tag_id and group_id



To record a post, we need the following tables:

- **posts** to record (post_id) -> user_id, content, timestamp
- **post_tags** to list (tag_id, post_id)
- **post_groups** to list (group_id, post_id)

Single Table design: ARRAY

Do you need so many tables?

- (post_id, tag_id) & (post_id, tag_id) can be stored with each post_id as (post_id) -> array of post_id's, array of group_id's
- but only if you can still lookup by tag_id and group_id

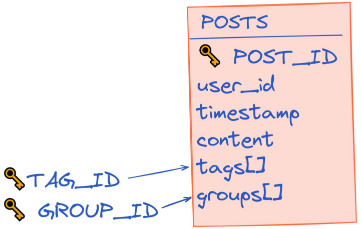


table (post_id,user_id, content, timestamp, int[] group_ids , int[] tag_ids)

- index on posts using gin (group_ids)
- index on posts using gin (tag_ids)

ARRAY.... or JSON?

Do you need so many tables?

- (post_id, tag_id) & (post_id, tag_id) can be stored with each post_id as (post_id) -> array of post_id's, array of group_id's
- but only if you can still lookup by tag_id and group_id

This can also be JSONB (and GIN index)

```
{
 tags: [ tag1, tag2, ...],
 groups: [ group1, group2, ...]
}
```

Finally... it is not very different

If tables are stored in the index structure (like YugabyteDB LSM tree)

👉 a GIN index references the row via the PK (hash)

🤔 *like association table in a normalized model with FK*

If tables are stored in heap tables (like PostgreSQL B-Tree)

👉 The GIN index references the row (tid)

🤔 faster than an association table?

Index Only Scan, Heap with Bitmap Scan optimizes the index-to-heap

Takeout

we have the choice: Table, ARRAY, JSONB

🤔 data integrity, performance, evolution

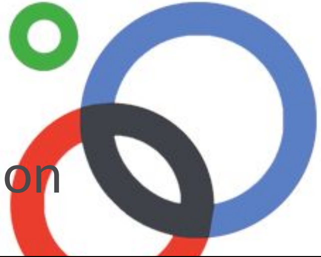
you must understand the access patterns

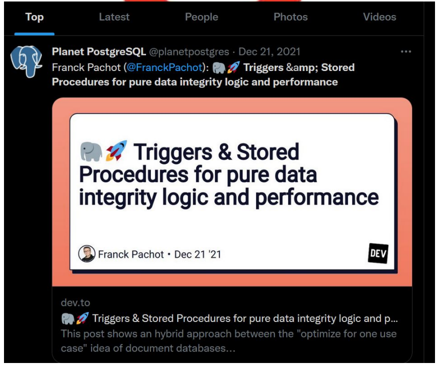
and think tables and indexes in the same way

Let's get a a bit more complex: demo

Access patterns:

- put a post into the database, with all related information
- get posts by tag, **ordered by last timestamp**
- get posts by group, **ordered by last timestamp**





Planet PostgreSQL @planetpostgres · Dec 21, 2021
Franck Pachot (@FranckPachot): 🐘🚀 **Triggers &amp; Stored Procedures for pure data integrity logic and performance**

🐘🚀 **Triggers & Stored Procedures for pure data integrity logic and performance**

Franck Pachot • Dec 21 '21

DEV

dev.to
🐘🚀 Triggers & Stored Procedures for pure data integrity logic and p...
This post shows an hybrid approach between the "optimize for one use case" idea of document databases...

GIN + B-Tree (btree_gin extension)

custom table maintained by trigger

https://dev.to/yugabyte/triggers-stored-procedures-for-pure-data-integrity-logic-and-performance-1eh8

fpachot@yugabyte.com
dev.to/FranckPachot
@FranckPachot

Join us on Slack:
www.yugabyte.com/slack
Star us on GitHub:
github.com/yugabyte/yugabyte-db