Database design : normalization

Goals of a good data structure design

- **Data Integrity** aka **Truth**: consistency, accuracy, avoiding anomalies
- **Query performance**: fast data retrieval
- Allow for future **expansion** of data types or relationships, evolution business requirements
- **Scalability** : growth in data volume
- **Storage** : minimize data redundancy (also important for integrity)

and most important:

- **Simplicity**: Create an understandable structure for developers and analysts

as we will see it always comes down to balancing between read and write performance

So the question is: **How to design the data structure of a database ?**

Scope

We start with Entity-Relationship Diagrams: ERDs

and then:

- OLAP vs OLTP databases and design strategies
- Normalization
  - anomalies to detect the need for normalization
  - normalization criteria: normal forms: 1NF, 2NF, 3NF

- Denormalization when needed (OLAP)

Practice: we normalize the treesdb database

Entity Relation Diagrams

Peter Chen developed the ER diagram in 1976.

> Chen's main contributions are formalizing the concepts, developing a theory with a set of data definition and manipulation operations, and specifying the translation rules from the ER model to several major types of databases (including the Relational Database)

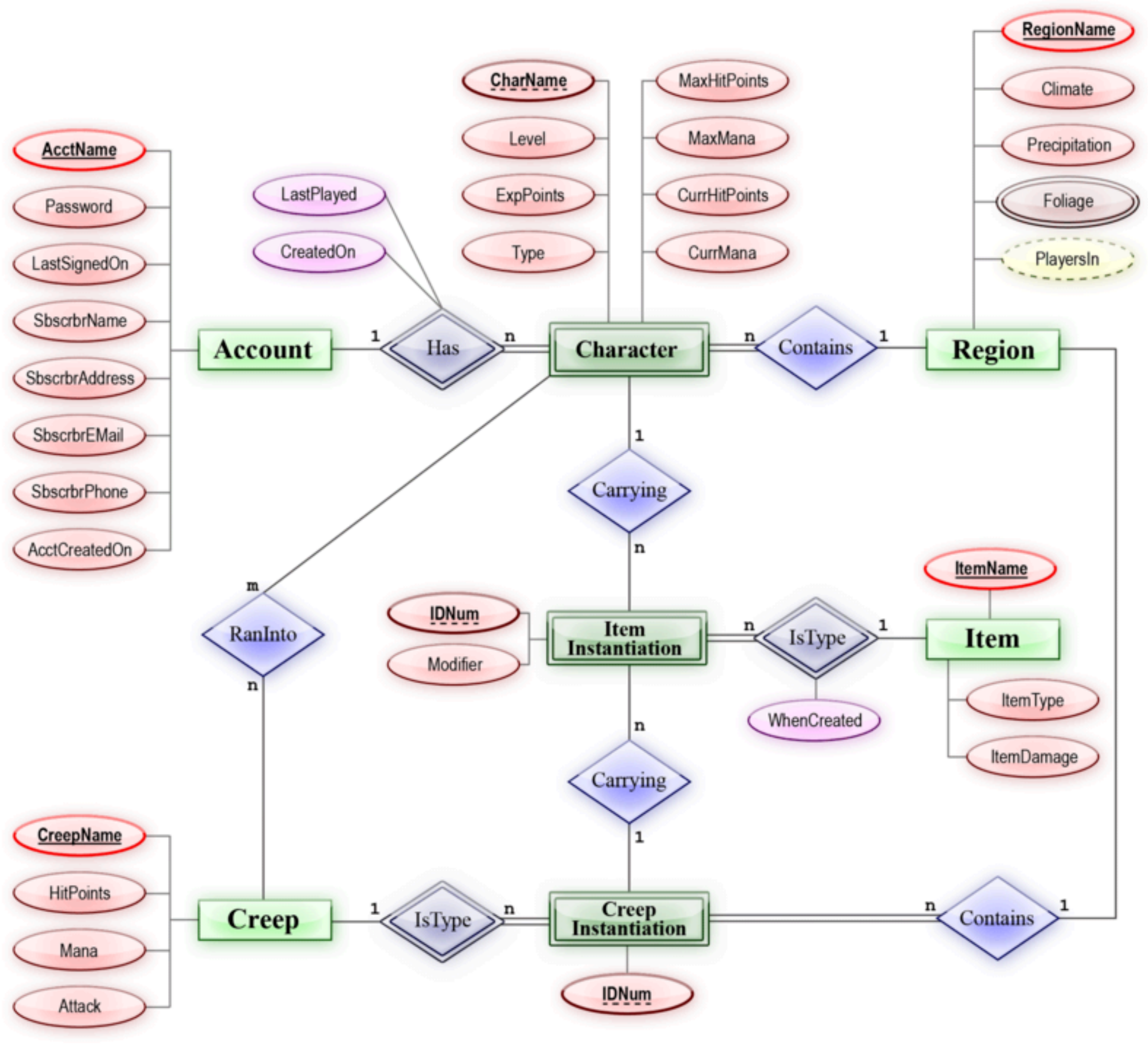
1976

That's what the Mac looked like in 1976 🤪🤪🤪

The ER model was created to visualize data structures and relationships in many situations.

- Object-Oriented Systems
- Software Architecture
- Business Process Modeling
- Data Flow in systems
- and Relational Databases

ER diagrams help in system design, information architecture, and application workflows.



The components of an ER diagram are :

- entities such as tasks, real world object, etc...
- attributes,
- and relations between the entities.

Check out this article for a complete explanation of ER diagrams: Introduction of ER model

As you can see there are many types of entities and attributes : strong, weak, key, composite, etc ...

Read also the wikipedia page

ER diagram for relational databases

In the context of relational databases, **the ER Diagram represent the structure of the database.**

- entities are **tables**

- attributes are table **columns**
- **relations** between entities can be

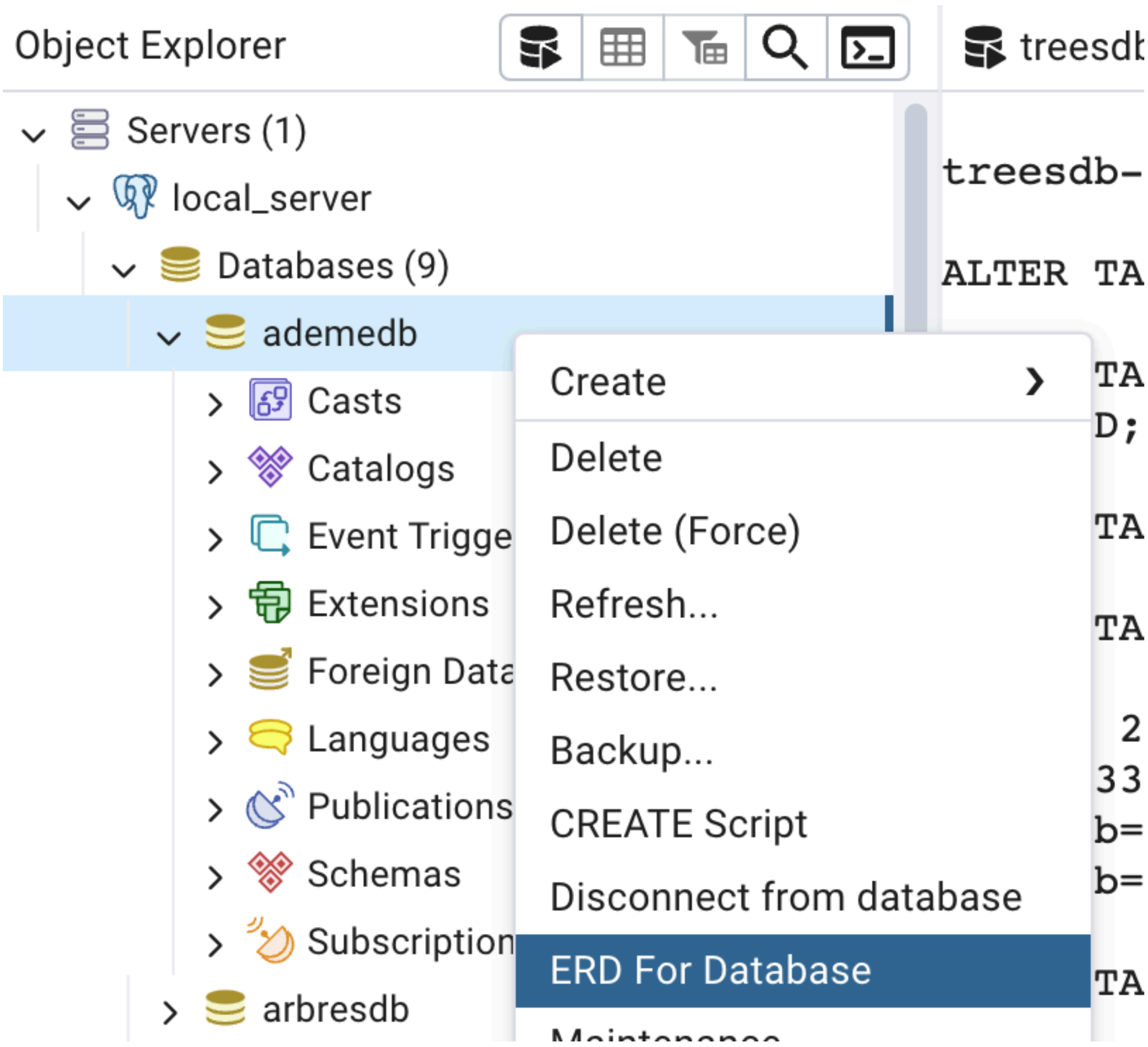
    - one to one
    - one to many
    - many to many

The ER diagram displays the **relations** between the **entities** (tables) present in the database and lists their **attributes** (columns).

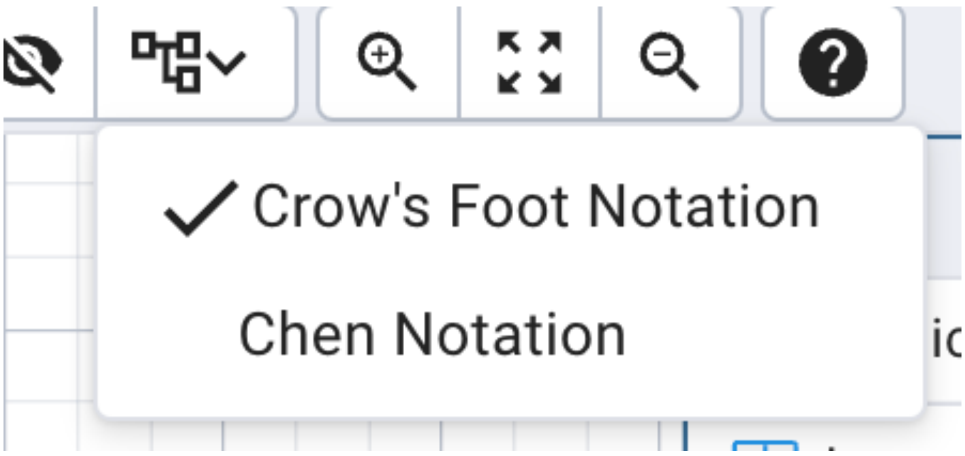
Generate an ERD in pgAdmin

- connect to the remote server
    - host:
    - username: epita
    - password:

- click right on the **airdb** database name
- click on **ERD for database**



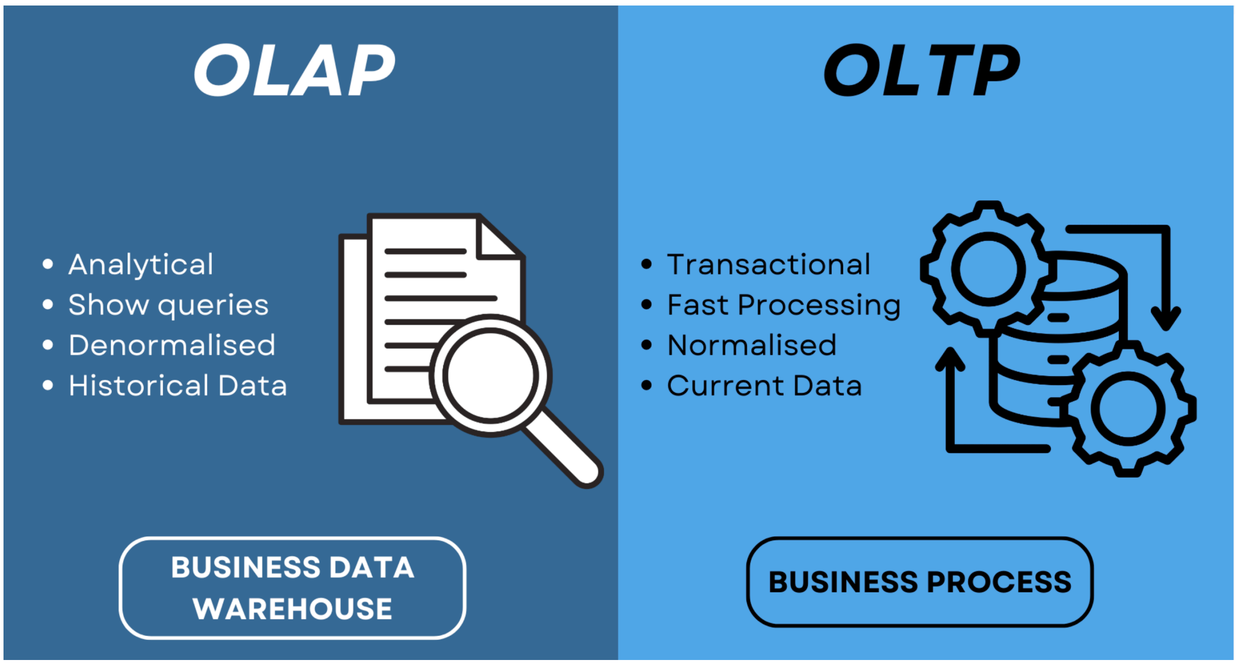
You can change notation for the relation type with



2 main types of databases: OLAP vs OLTP

The end usage of a database drives its data structure.

We consider two types of relational databases: **analytical** databases (OLAP) vs **transactional** databases (OLTP)



OLAP : Online Analytical Processing

- analysis, BI, reporting, dashboards,
- optimized for high **read** volume
- complex queries (lots of joins and calculations)
- can be **asynchronous**, query execution does not have to be lightning fast

OLAP does not have to be super fast

OLTP: Online Transaction Processing,

- applications, transactions, high **write** volume
- optimized for high write volume: **data integrity**, fast updates and inserts
- ACID properties for transactions (all or nothing) (ACID: (Atomicity, Consistency, Isolation, Durability))
- **synchronous**, real time, speed is super important

OLTP has to be super fast

Further reading : difference between olap and oltp in dbms.

Look at the table of differences and the Q&A at the end of the article.

Quiz

For each scenario, determine whether it's more suited for an OLTP (Online Transaction Processing) or OLAP (Online Analytical Processing) system.

- Liking a friend's post on Instagram.
- Analyzing trending hashtags on Mastodon over the past month.
- Sending a Snapchat message to a friend.
- Netflix recommending shows based on your viewing history.
- Ordering food through a delivery app.
- Making an in-app purchase.
- TikTok or Youtube analyzing which video types keep users watching longer.
- A fitness app calculating your average daily steps for the past year.

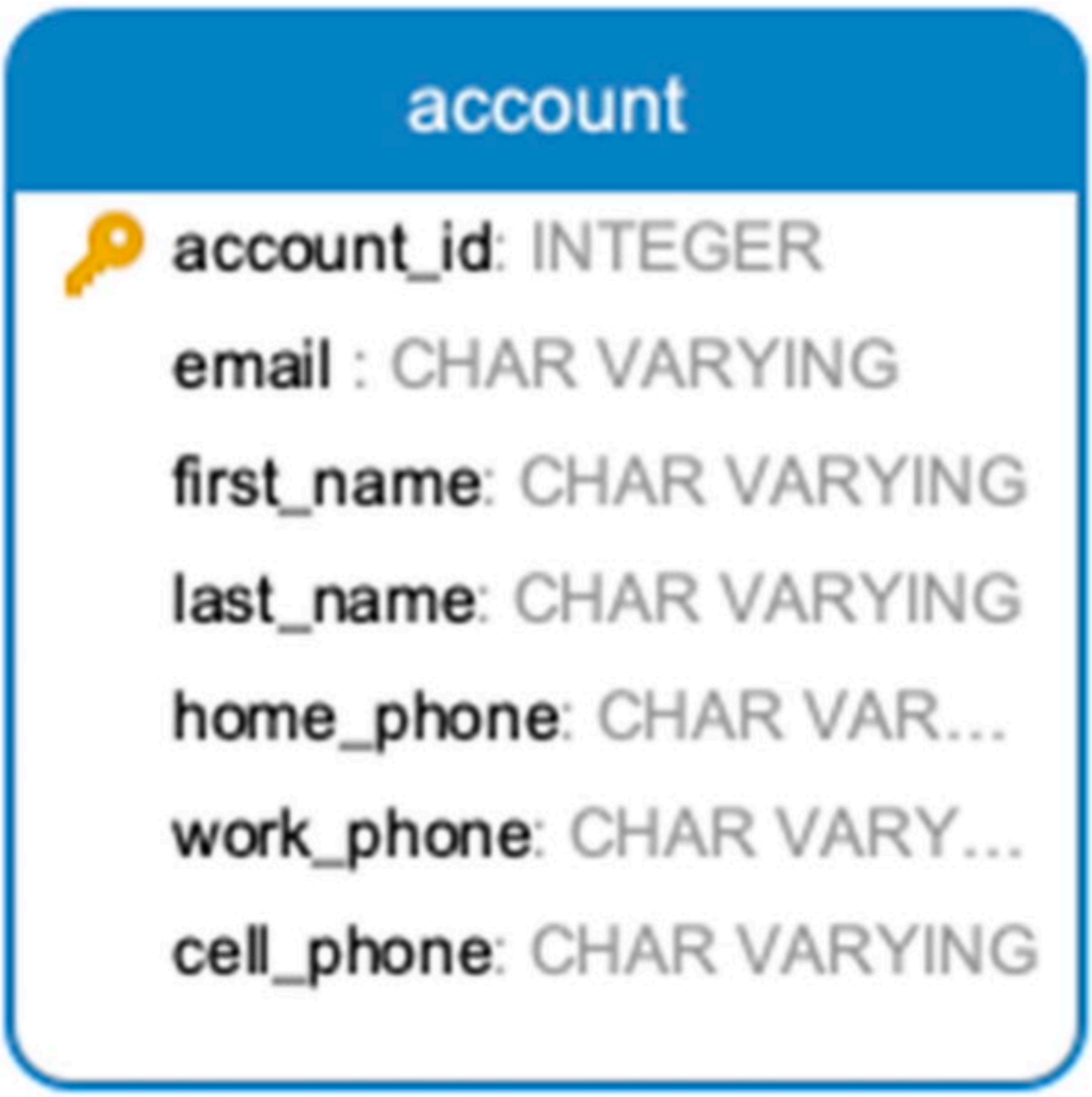
Solution

- OLTP: Liking a post is a simple, immediate transaction.
- OLAP: Analyzing trends over time involves processing large amounts of historical data.
- OLTP: Sending a message is a quick, individual transaction.
- OLAP: Recommendations are based on analyzing viewing patterns across users and time.
- OLTP: Placing an order is a specific, time-sensitive transaction.
- OLTP: An in-app purchase is a single, immediate financial transaction.
- OLAP: This involves analyzing user behavior patterns across many videos and users.
- OLAP: Calculating long-term averages involves processing historical data over time.

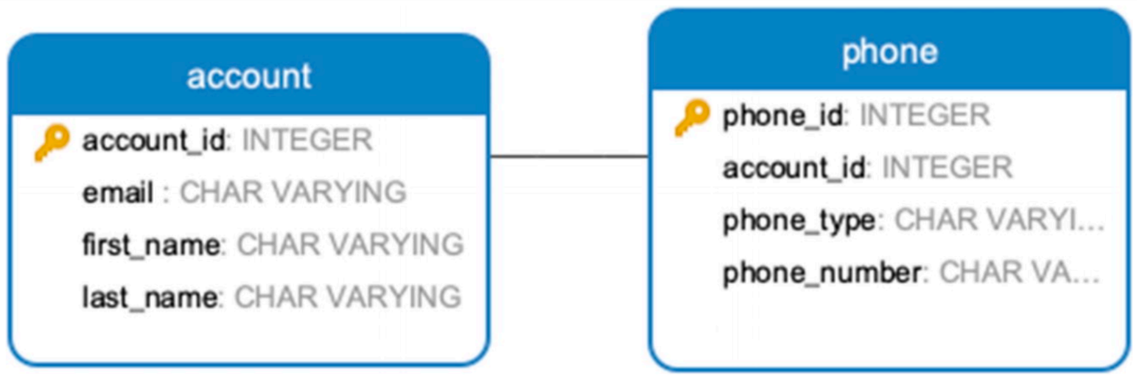
An account with multiple phones

Consider the 2 database designs:

- **1 account table with multiple phones**



- **1 account table and 1 dedicated phone table**



which design (1 or 2 tables) is better in terms of faster or simpler query for:

- dealing with missing phone type (no work phone)
- adding a new phone type (new secret phone)
- flagging a phone as primary
- handling a user with no phone

- displaying all the phones of an account in a UX
- fast retrieval search over phone number(s)

Normalization

The general goal of **normalization** is to reduce data **redundancy** and **dependency** by organizing data into **separate, related tables**.

This helps maintain data integrity and flexibility:

- logical entities
- independence between tables
- uniqueness of data

Normalized databases are

- easy to update
- easy to maintain

More formally, a database is normalized if:

> **all column values depend only on the table primary key,**

In the *1 table design* for the account and its phone numbers, a phone number value depends on the name of the phone column (home*phone, work*phone, ...) not just the account_id key: We can say it's not normalized

With a table dedicated to the phones, (design with 2 tables), the phone value depends only on the phone_id key: the tables are normalized.

Denormalization

The idea of denormalization is to have data **redundancy** to simplify queries and make OLAP queries faster.

Redundant data : the same data / info exists in multiple tables

SELECT queries involve fewer JOINs.

However INSERT, UPDATE, DELETE queries are more complex as multiple tables must be accounted for. Therefore data integrity is more complex to preserve.

Scenario:

In a social network, imagine that you have two tables:

1. The **Users table**: Contains user information like `user_id` `name` and `email`.

user_id	user_name	email
101	Ulaf	....@gmail.com
102	Birgitte	....@gmail.com
103	Inge	....@gmail.com
114	Boris	....@gmail.com

1. The **Posts table**: Contains posts made by users, with fields like `post_id` `content`, `publication_date` ... and the post's author information through the `user_id`.

post_id	user_id	content	pub_date
1	101	I ❤️ postgreSQL	2022-10-01
2	103	Singing in the rain	2022-10-02
3	102	Nerds unite!	2022-10-02
4	114	Guys? i'm bored 😒 when's the break?	2022-10-02
5	103	Taylor swift omg! #sweet	2022-10-03

In a **normalized** database: users and their posts are only linked by the `user_id` which is a foreign key in the posts table.

The users table has no information about posts. And similarly the posts table is all about the posts and not their author (besides the user_id foreign key).

So when you need to display the user's name next to their post, you need to **JOIN** `Users` and `Posts` tables.

```SQL
SELECT p.*, u.name
FROM POSTS p
JOIN users u on p.user_id = u.id
WHERE p.pub_date = '2022-10-02';
```

In order to improve performance, you can **denormalize** the posts table by adding the `user_name` to the `Posts` table.

A Denormalized `Posts` table would then look like:

post_id	user_id	user_name	content	pub_date
1	101	Ulaf	I ❤️ postgreSQL	2022-10-01
2	103	Inge	Singing in the rain	2022-10-02
3	102	Birgitte	Nerds unite!	2022-10-02
4	114	Boris	Guys? i'm bored 😒 when's the break?	2022-10-02
5	103	Inge	Taylor swift omg! #sweet	2022-10-03

SQL
```sql
SELECT p.*
FROM POSTS p
WHERE p.pub_date = '2022-10-02';
```

Faster read performance since you can fetch the `user_name` along with the post data without needing to perform a join between the `Users` and `Posts` tables.

But

Data redundancy: If *Ulaf* changes his name, you will need to update it in both the `Users` table and every row in the `Posts` table that references him. This increases the complexity of updates.

Anomalies

So, given a database, how do you know if it needs to be normalized?

There are 3 types of *anomalies* that you can look for:

- insertion
- update

- deletion

A normalized database will solve these anomalies.

Insertion anomalies

Consider the table of teachers and their courses

id	teacher	course
1	Bjorn	Intermediate Database
2	Sofia	Crypto Currencies
3	Hussein	Data Analytics

Now a new teacher (Alexis), is recruited by the school. The teacher does not have a course assigned to yet. If we want to insert the teacher in the table we need to put a `NULL` value in the course column.

and NULL values can cause problems!!! (more on that later)

More generally:

- in a relation where an item (A) has many (or has one) other items (B), but A and B are in the same table.
- so for a new item A without items B, inserting the new A means the value for B has to be null

Update anomaly

Consider now the following movies, directors, year and production house

id	Film	Director	Production House
1	Sholay	Ramesh Sippy	Sippy Films
2	Dilwale Dulhania Le Jayenge	Aditya Chopra	Yash Raj Films
3	Kabhi Khushi Kabhie Gham	Karan Johar	Dharma Productions
4	Kuch KuchwHota Hai	Karan Johar	Dharma Productions
5	Lagaan	Ashutosh Gowariker	Aamir Khan Productions
6	Dangal	Nitesh Tiwari	Aamir Khan Productions
7	Bajrangi Bhaijaan	Kabir Khan	Salman Khan Films
8	My Name Is Khan	Karan Johar	Dharma Productions
9	Gully Boy	Zoya Akhtar	Excel Entertainment
10	Zindagi Na Milegi Dobara	Zoya Akhtar	Excel Entertainment

(chatGPT knows Bollywood 😁)

If one of the production house changes its name, we need to update multiple rows.

In a small example like this one, this is not a problem since the query is trivial but in large databases some rows may not be updated.

- **human error** when writing the update query or doing manual updates
- **Redundant Data**: updating multiple rows *at the same time* can cause performance issues in large databases with millions of records.
- **Data Locking**: In a multi-user environment, updating several rows at once may lock parts of the database, affecting performance and potentially leading to delays in accessing data for other users or systems.
- if changes are made in stages or **asynchronously**, data can temporarily or permanently enter an inconsistent state.

By moving the production house data in its own separate table, then updating its name would only impact one row!

Deletion errors

Here is a table about programmers, languages and (secret) projects

Developer_Name	Language	Project_Name
Lisa	Python	Atomic Blaster
Bart	Java	Invisible tank
Montgomery	Python	Atomic Blaster
Maggie	JavaScript	Exploding Squirrel
Ned	C++	War AI
Homer	Python	Atomic Blaster
Marge	Ruby	War AI

Let's say that for imperative reasons we need to cover up the existence of the *War AI* project. And we will delete all rows that mention that project in the databse. Then an unfortunate consequence is that we will also also delete all mentions of Marge and Ned since they are not involved in other projects.

So by deleting an attribute we also remove certain valus of other attributes.

In short

When you have different but related **natural / logical entities** packed together in the same table, this causes anomalies and you need to normalize.

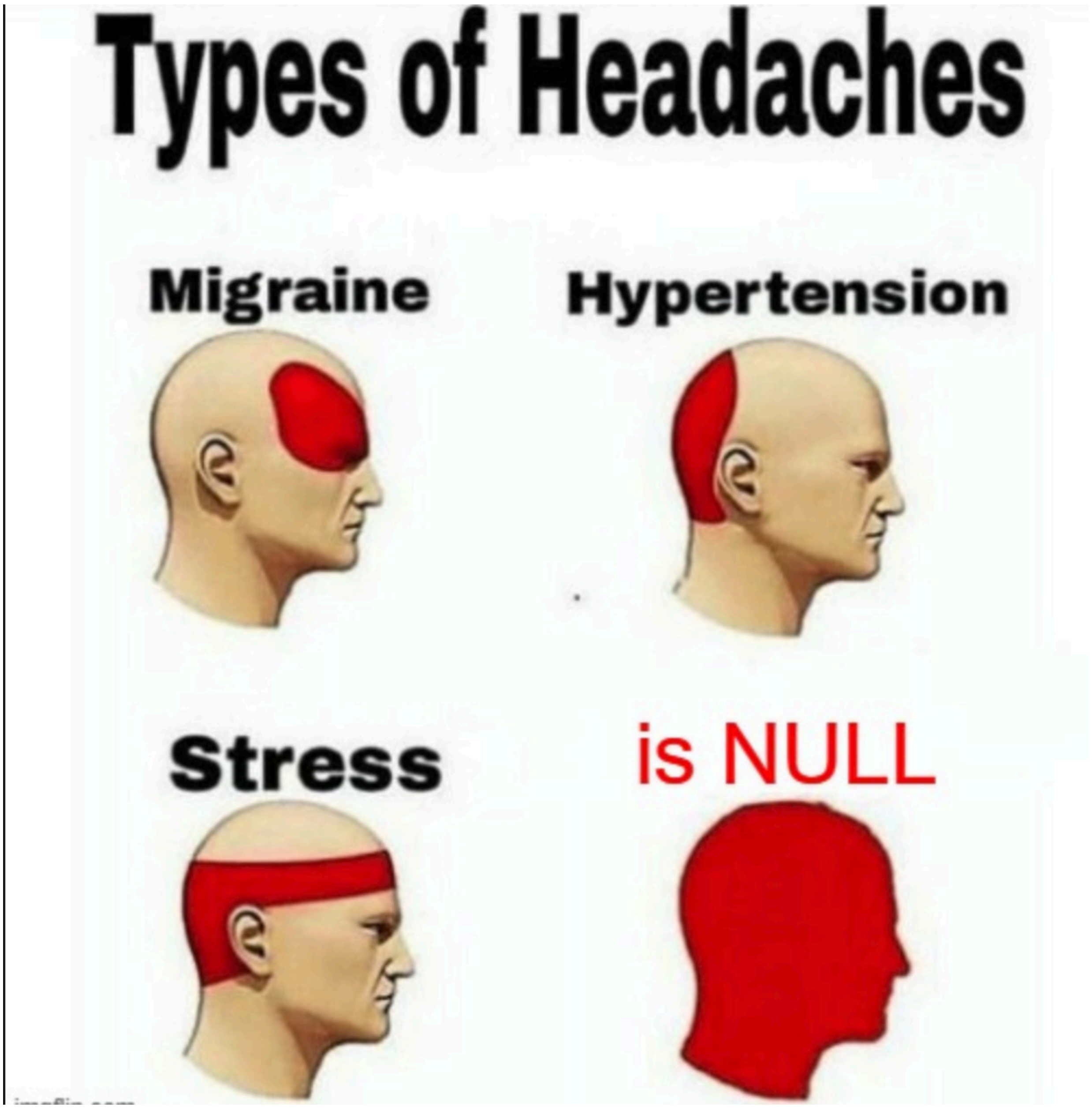
The problem with null values

Why is it a problem to have NULL values in a column?

- **Ambiguity**: Null can mean "*unknown*", "*not applicable*" or "*missing*" leading to confusion.
- Null values require **special handling** and complicates
  - **queries**: (`IS NULL` vs `=`) .
  - **data analysis**: `NULL` values are ignored in aggregate functions (like `SUM` , `COUNT`).
- Impacts **indexing and performance**: since Nulls are excluded from indexes

- **Violates normalization**: indicates poor database design or incomplete relationships.

So avoid NULL values if you can!



What about anomalies in the trees table ?

What anomalies can you find for each type: insertion, update and deletion

Insertion Anomalies:

- To add a new tree species or genre without an associated tree, we'd have to insert a record with null values for most fields, which is not ideal.
- We can't add a new location without adding a tree, even if no trees exist there yet.

Update Anomalies:

- If we need to update information about a type of tree (e.g., its genre), we'd have to update multiple rows, risking inconsistency.
- Same for updating location information (e.g., arrondissement name) would require updating multiple rows, again risking inconsistency.

Deletion Anomalies:

- If we delete the last tree of a particular species, genre, variety, we lose all information about that species, genre or variety.

In short

There's a need for normalization when:

- the same data exists in multiple rows
- multiple columns with data of same nature, different types
- same data exists in multiple tables

or simply said:

> ## every non-key attribute must provide a fact about the key, the whole key, and nothing but the key

---

Normal forms

A normal form is a **rule** that defines a level of normalization.

- UNF: Unnormalized form
- 1NF: First normal form
- 2NF: Second normal form
- 3NF: Third normal form

There are multiple higher levels

- EKNF: Elementary key normal form
- BCNF: Boyce–Codd normal form

- 4NF: Fourth normal form
- ETNF: Essential tuple normal form
- 5NF: Fifth normal form
- DKNF: Domain-key normal form
- 6NF: Sixth normal form

In general a database is considered normalized if it meets 3NF level.

It gets very abstracts very quickly.

Normal forms are a gradual step by step process towards normalization. Each form is a guide that focuses on a single type of problem.

We can always choose to apply a normalization form or to ignore it

In the following, we mention:

- Relation, or an entity: table
- Attribute : column
- Primary key
- Composite key : key composed of multiple attributes

1NF

Each field contains a single value

> A relation is in first normal form
>
> if and only if
>
> no attribute domain has relations as elements.

which translates as no column as sets of values (relations) as elements

In short: columns cannot composite values : arrays, json, ...

More on 1NF:

- Wikipedia: Satisfying 1NF

Does the tree table follow 1NF ?

Wait, ... what ?

There's a contradiction between first normal form (1NF) and the existence of **ARRAY** and

JSONB data types (see also POINT, HRANGE, composite types, and many other) in SQL databases.

The rule of thumb is :

> When you frequently need to access, modify, handle the elements of the sets of values for a given record then apply 1NF

For instance a collaborative blog post where we start with 1 author (varchar), then a co-author comes along (array), then a third and a fourth. Then the 1st one give up and does not want to be in the list of authors etc etc etc. In that case using an Array to store the list of authors causes more trouble than it solves. and authors should have their own table.

In the end it's a balance between simplicity and ease of control.

For instance:

- Use normalization (1NF) when:
    - Storing a list of order items, where each item needs its own price, quantity, and product reference.
    - Managing a list of phone numbers for contacts, where you need to query or update individual numbers.
- Use multi-value types when:
    - Storing tags for a blog post, where the tags are simply a list of strings with no additional attributes.
    - ...

2NF

The table is in 2NF iff :

- The table is in 1NF,
- it has a single attribute unique identifier (UID),
- and every non key attribute is dependent on the entire UID

Some cases of non-compliance with 2NF

- Derived or calculated fields:
    - Relation: `Employee(EmployeeID, Name, BirthDate, Age)`
    - Here, `Age` is derived from `BirthDate`, causing a partial dependency.

- Redundant information:
    - Relation: `Order(OrderID, ProductID, ProductName, ProductCategory)`
    - `ProductCategory` and `ProductName` both depend on `ProductID`, not the full `OrderID`.
- general case: Inverted one to many
    - A has many Bs
    - Relation: `B(Bid, B.attribute, A.attribute)`
- Composite keys with partial dependencies:
    - Relation: `R(A, B, C, D)` where `(A, B)` is the composite primary key
    - If `C` depends only on `A`, we have a 2NF violation.

The 1 table version of the account phone table was not in 2NF.

see :

- Example of 2NF
- Satisfying 2NF

2NF violation in the trees table ?

The tree table has many 2NF violations, since all the categorical columns are into a one to many (category (A) has many trees (B)) relation.

2NF normalization tells us we should create a table for all the categorical columns.

But that may feel overkill and instead of simplifying the logical structure of the data it might add too much complexity it without clear gain. (but we will still do it anyway)

On the contrary, keeping the categories in the tree table is a form of denormalization.

When to apply 2NF to a categorical attribute ?

You might consider normalizing a categorical attribute (column) into a separate table if:

- The category data is large or complex (e.g., includes descriptions, hierarchies).
- Categories change frequently or need to be managed separately.
- There's a need to enforce referential integrity on categories.
- The application requires complex operations or reporting on categories.

In practice the design of the database can evolve:

- Start with categorical attributes.
- You can always normalize later.

3NF

A relation R is in 3NF if and only if both of the following conditions hold:

- The relation is in second normal form (2NF).
- No non-prime attribute of R is transitively dependent on the primary key.

> A **transitive dependency** occurs when a non-prime attribute (an attribute that is not part of any key) depends on another non-prime attribute, rather than depending directly on the primary key.

where:

- non-prime attribute: an attribute that is not part of any key

It's becoming a bit abstract 🥱 😴

For the statisticians in the room, it's a bit like **confonders**.

So basically, in a table you would have 2 columns that are not keys that sort of depends on each other.

Example

`Student(StudentID, Name, CourseID, CourseName, InstructorName)`

This relation violates 3NF because:

- `CourseID → CourseName` (CourseName depends on CourseID, not on StudentID)
- `CourseID → InstructorName` (InstructorName depends on CourseID, not on StudentID)

Here, we have transitive dependencies:

- `StudentID → CourseID → CourseName`
- `StudentID → CourseID → InstructorName`

[3NF on Wikipedia]

Difference between 2NF and 3NF

3NF is similar to 2NF

The key differences:

- Nature of the dependency:
  - 2NF addresses partial dependencies on a key.
  - 3NF addresses transitive dependencies through non-key attributes.
- Scope of the problem:
  - 2NF focuses on the relationship between non-key attributes and parts of the key.
  - 3NF looks at dependencies between non-key attributes.

Denormalize

Denormalization means to choose not to follow a normal form to simplify things.

So, when can we denormalize instead?

> *Once you have thoroughly normalized your database, some difficulties may appear, either because the fully normalized schema is too heavy to deal with at the application level without any benefits, or because having a highly normalized schema involves performances penalties that you've measured and cannot tolerate.*
>
> *Fully normalized schemas often have a high number of tables and references in between them. That means lots of foreign key constraints and lots of join operations in all your application queries.*
>
> *When there's no way to speed-up your application another way, then it is time to denormalize the schema, i.e. make a decision to put your data quality at risk in order to be able to serve your users and business.*

Dimitri Fontaine - The Art of PostgreSQL (2022) The Art of PostgreSQL p 283

In short: **Keep it simple**

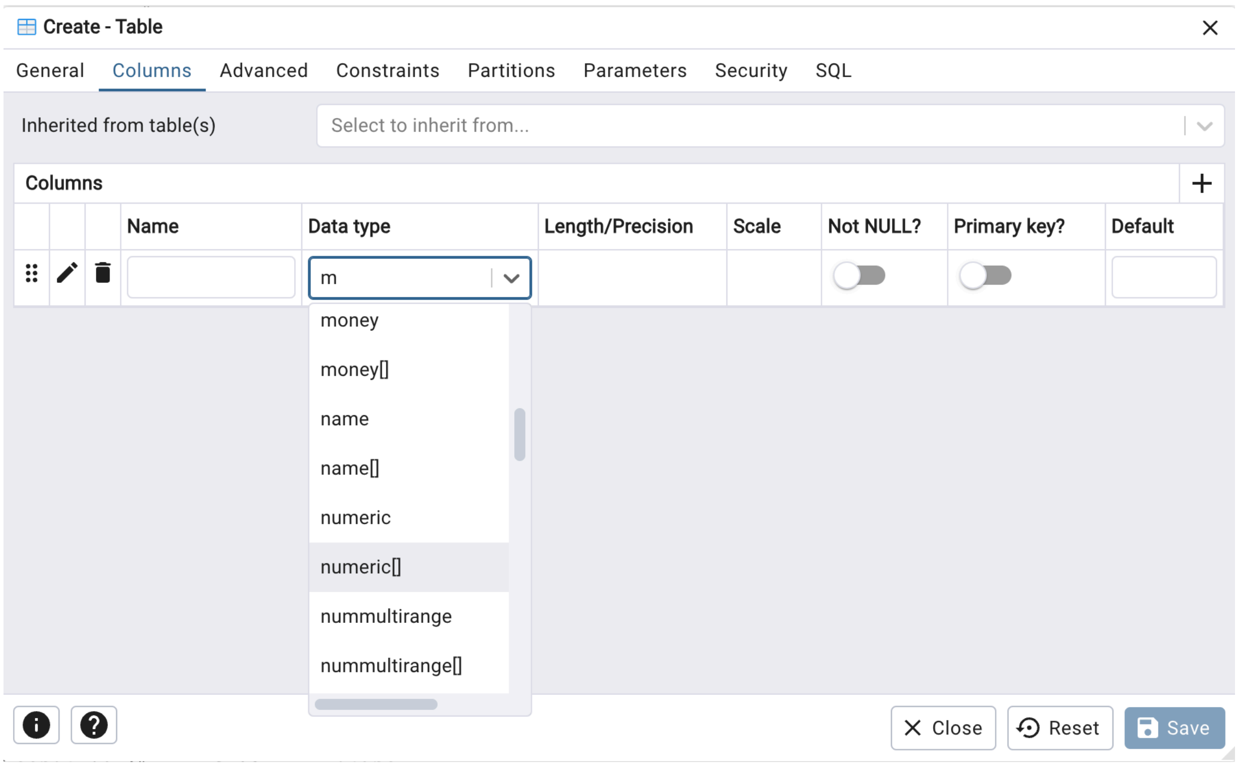
PostgreSQL multi-valued data types

PostgreSQL offers several data types that can store multiple values in a single column, which breaks the First Normal Form (1NF).

These types are often used for performance optimization, improved query efficiency, or when the data naturally fits a more complex structure.

There are many data types in postgreSQL

see for instance the long list of availabe data types when adding a column in pgAdmin



and the [documentation on data types]

Array Types:

Allows storing multiple values of the same type in a single column.

Example: `INTEGER[]`, `TEXT[]`

When to use: * When you need to store a fixed or variable number of elements of the same type. * For implementing tags, categories, or any list-like data where order matters. * When you frequently need to query or update the entire list as a unit.

JSONB (and JSON):

Stores JSON data with indexing and querying capabilities

When to use: * For semi-structured data that doesn't fit well into a relational model. * When your application needs to store flexible, schema-less data. * For data that is frequently read but less frequently updated. * When you need to query or index JSON data efficiently.

also : HSTORE, RANGE …

Composite types

User-defined type that combines multiple fields into a single column.

Example: CREATE TYPE address AS (street TEXT, city TEXT, zip TEXT);

When to use:

- When you have a logical grouping of fields that are always used together.
- For improving query performance by reducing joins.
- When you want to enforce a structure on a group of related fields.

Next

Let's apply normalization on the trees database.

- github > SkatAI > epitadb > docs > 03 > S4-normalize-treesdb.pdf