**Utility Operators, Keywords, and Functions(Postgres)**

GREATEST

Gives the greatest value from the values provided

**SELECT GREATEST(200, 10, 30);**

Example

Compute the cost to ship each item

Shipping is the maximum of (weight \* $2) or $30

**SELECT name, weight, GREATEST(30, 2 \* weight)**

**FROM products;**

LEAST

This will give the least value from the list given

**SELECT LEAST(1, 20, 50, 100)**

Example

All products are on sale and the rule to calculate the sale price of a product is

Price is the least of the products price \* 0.5 or $400

**SELECT name, price, LEAST(price \* 0.5, 400)**

**FROM products;**

**The case keyword**

Example

Print each product and its price. Also, print a description of the price

If price > 600 then ‘high’, if price > 300 then ‘medium’ else print ‘cheap’

**SELECT**

**name,**

**price,**

**CASE**

**WHEN price > 600 THEN 'high'**

**WHEN price > 300 THEN 'medium'**

**ELSE 'cheap'**

**END**

**FROM products;**

**Else keyword isn’t compulsory (but if there is a row that doesn’t satisfy the two conditions, then we will end up with a default value of 1**

**PGAdmin**

* Tool to manage and inspect a Postgres database
* Can connect to local or remote databases
* Can view/change just about anything in PG

Postgres Server

* A PG Server can contain multiple databases
* All data for a single app lives in a single DB
* Having multiple DB’s is more about working with more than one app on your machine

**Data types**

* Integer
* Date/time
* Boolean
* Currency
* Character
* Range
* Xml
* Binary
* JSON
* Arrays
* UUID
* Geometric

**Numeric types**

* Smallint, integer and bigint -> Numbers without any decimal points
* Smallserial, serial and bigserial -> No decimal point, auto increment
* Decimal, numeric, real, double precision and float -> Number with decimal points

|  |  |
| --- | --- |
| **Numeric types** | **Range** |
| Smallint | -32768 to 32767 |
| Integer | -2147583648 to 2174483647 |
| Bigint | -922337203685775808 to 9223372036854775807 |
| smallserial | 1 to 32767 |
| serial | 1 to 2147483647 |
| bigserial | 1 to 9223372036854775807 |
| decimal | 131072 digits before decimal point, 16383 after |
| Numeric | 131072 before decimal point, 16383 after |
| Real | 1E-37 to 1E37 with at least 6 places precision |
| Double precision | 1E-307 to 1E308 with at least 15 place precision |
| float | Same as real or double precision |

**Fast Rules on Numeric Data Types**

|  |  |  |
| --- | --- | --- |
|  | ‘id’ column of any table | Mark the column as **serial** |
|  | Need to store a number without a decimal | Mark the column as **integer** |
| Bank balance, grams of gold, scientific calculations | Need to store a number **with a decimal and this data needs to be very accurate** | Mark the column as **numeric** |
| Kilograms of trash in a landfill, litres of water in a lake, air pressure in a tire | Need to store a number **with a decimal and the decimal doesn’t make a big difference** | Mark the column as **double precision** |

Floats are notoriously known for being inaccurate

SCRIPTS

SELECT 2 + 2; // returns a column with value 4

SELECT (2.0::INTEGER); // converts the datatype from numeric to integer

**Character Types**

CHAR(5) – Store some characters, length will always be 5 even if PG has to insert spaces

VARCHAR – Store any length of string

VARCHAR(40) – Store a string up to 40 characters, automatically remove extra characters

TEXT – Store any length of string

*NB: There is no performance difference in these character types unlike other types of databases*

**Boolean Types**

true, yes, on, 1, t, y - TRUE

false, no, off, 0, f, n - FALSE

null - NULL

Example

SELECT ('y'::BOOLEAN); // returns TRUE

SELECT ('n'::BOOLEAN); // returns FALSE

SELECT (NULL::BOOLEAN); // returns [null]

**DATE**

1980-11-20 -> 20 November 1980

Nov-20-1980 -> 20 November 1980

20-Nov-1980 -> 20 November 1980

1980-Novemberr-20 -> 20 November 1980

November 20, 1980 -> 20 November 1980

***NB: PostgreSQL converts any date format into a fixed date value as above***

***SELECT ('NOV-20-1980'::DATE);***

***SELECT ('NOV 20 1980'::DATE);***

***SELECT ('NOV,20,1980'::DATE);***

***SELECT ('NOV 20,1980'::DATE);***

*Gives –> 1980-11-20*

**TIME, Time without time zone**

01:23 AM -> 01:23, no time zone

05:23 PM -> 17:23, no time zone

20:34 -> 20:34, no time zone

**SELECT ('01:23 AM'::TIME); SELECT ('01:23 PM'::TIME WITHOUT TIME ZONE);**

Returns 01:23:00

**SELECT ('01:23 PM'::TIME);**

Returns 13:23:00

**SELECT ('02:30'::TIME);**

Returns 02:30:00

**TIME WITH TIME ZONE**

01:23 AM EST -> 01:23-05:00 (The minus 5 indicates, 5 hrs behind UTC time)

05:23 PM PST -> 17:23-08:00

05:23 PM UTC -> 17:23+00:00

05:23 PM UTC -> 17:23+00:00

**SELECT ('01:23:23 AM EST'::TIME WITH TIME ZONE);**

**SELECT ('01:23:23 AM Z'::TIME WITH TIME ZONE); (z is utc)**

**SELECT ('01:23:23 AM UTC'::TIME WITH TIME ZONE); (utc)**

**TIMESTAMP WITH TIME ZONE**

Nov-20-1980 05:23 PM PST -> 1980-11-20 18:23:00-07

**SELECT ('NOV-20-1980 1:23 AM PST'::TIMESTAMP WITH TIME ZONE);**

Returns -> 1980-11-20 19:23:00+10

**INTERVAL**

1 day -> 1 day

1 D -> 1 day

1 D 1M 1S -> 1 day 1 minute 1 second

Think of an interval as a duration of time

**SELECT ('1 day'::INTERVAL);**

**SELECT ('1 D'::INTERVAL);**

**SELECT ('1 D 20 H'::INTERVAL);**

Returns -> 1 day 20:00:00

**SELECT ('1 D 20 H 30 M 40 S'::INTERVAL);**

Returns -> 1 day 20:30:40

Can perform operations like:

**SELECT ('1 D 20 H 30 M 40 S'::INTERVAL) - ('1 D'::INTERVAL);**

Returns -> 20:30:40

**SELECT ('NOV-20-1980 1:23 AM EST':: TIMESTAMP WITH TIME ZONE) - ('1 D'::INTERVAL);**

Returns -> 1980-11-19 16:23:00+10

**SELECT**

**('NOV-20-1980 1:23 AM EST':: TIMESTAMP WITH TIME ZONE)**

**-**

**('NOV-10-1980 1:23 AM EST':: TIMESTAMP WITH TIME ZONE);**

Returns -> 10 days

It can work with different time zones as well:

**SELECT**

**('NOV-20-1980 1:23 AM EST':: TIMESTAMP WITH TIME ZONE)**

**-**

**('NOV-10-1980 5:43 AM PST':: TIMESTAMP WITH TIME ZONE);**

Returns -> 9 days 16:40:00

**Database-Side Validation and Constraints**

**Applying a NULL Constraint**

**Row-Level Validation**

* Is a given value defined?
* Is a value unique in its column
* Is a value >, <, >=, <=, = some other value?

You can always add the constraint NOT NULL to prevent null values from being inserted into the table

When Creating the table, this is how you do it for price:

**CREATE TABLE products (**

**Id SERIAL PRIMARY KEY,**

**name VARCHAR(50),**

**department VARCHAR(50),**

**price INTEGER NOT NULL,**

**weight INTEGER**

**);**

When the Table has already been created:

**ALTER TABLE products**

**ALTER COLUMN price**

**SET NOT NULL;**

NB: The constraint cannot be set if there are already null values in the table. The null values have to be converted to an actual values for the constraint to be applied or the null value rows could be deleted

Example of converting a price column with null values

**UPDATE products**

**SET price = 9999**

**WHERE price IS NULL;**

Now that the price column has no null values in it, the constraint can be applied.

**ALTER TABLE products**

**ALTER COLUMN price**

**SET NOT NULL;**

**DEFAULT COLUMN VALUES**

When Creating The Table

CREATE TABLE products (

Id SERIAL PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

Department VARCHAR(50) NOT NULL,

Price INTEGER DEFAULT 999,

Weight INTEGER

);

After The Table Was Created

ALTER TABLE products

ALTER COLUMN price

SET DEFAULT 999;

**Applying a Unique Constraint to One Column**

A ***UNIQUE*** constraint ensures values in a particular column are not duplicated

In the case below, the UNIQUE constraint is used to ensure that the column name does not have products with the same name

When Creating The Table

CREATE TABLE products (

Id SERIAL PRIMARY KEY,

Name VARCHAR(50) NOT NULL UNIQUE,

Department VARCHAR(50) NOT NULL,

Price INTEGER DEFAULT 999,

Weight INTEGER

);

After The Table Was Created

ALTER TABLE products

ADD UNIQUE (name);

NB: You cannot add(alter) the UNIQUE constraint unless all values inside the table are already unique values

**Multi-Column Uniqueness**

When Creating The Table

CREATE TABLE products (

Id SERIAL PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

Department VARCHAR(50) NOT NULL,

Price INTEGER DEFAULT 999,

Weight INTEGER,

UNIQUE(name, department)

);

After The Table Was Created

ALTER TABLE products

ADD UNIQUE (name, department);

Dropping Constraints

ALTER TABLE products

DROP CONSTRAINT products\_name\_key;

ALTER TABLE products

DROP CONSTRAINT products\_name\_department\_key;

**Adding a Validation Check**

The CHECK below looks if the data being inserted or updated in the price column is greater than 0.

When Creating The Table

CREATE TABLE products (

Id SERIAL PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

Department VARCHAR(50) NOT NULL,

Price INTEGER CHECK (price > 0),

Weight INTEGER,

UNIQUE(name, department)

);

After The Table Was Created

ALTER TABLE products

ADD CHECK (price > 0);

NB: A check can only work on the row we are adding/updating. We cannot apply a check if all the rows inside our existing table don’t already satisfy the check.

**Checks Over Multiple Columns**

The CHECK below ensures that an order is delivered after it is created

CREATE TABLE orders (

id SERIAL PRIMARY KEY,

name VARCHAR(50) NOT NULL,

created\_at TIMESTAMP NOT NULL,

est\_delivery TIMESTAMP NOT NULL,

CHECK (created\_at < est\_delivery)

);

The check ensures only the colors red, green and blue are added into the database

CREATE TABLE cars (

id SERIAL PRIMARY KEY,

name VARCHAR(20),

color VARCHAR(20) CHECK (color IN ('red', 'green', 'blue'))

);

NB: We can only make use of columns inside the check statement inside the row that we are trying to insert

**Where to Apply Validation**

|  |  |
| --- | --- |
| **Web Server** | **Database** |
| Easier to express more complex validation | Validation still applied even if you connect with a different client |
| Fare easier to apply new validation rules | Guaranteed that validation is always applied |
| Many libraries to handle validation automatically | Can only apply new validation rules if all existing rows satisfy it |

NB: Better if you put the bulk of your validation at the web server level and only put some very critical ones in the database

**SQL Schema Designers**

1. Dbdiagram.io
2. Drawsql.app
3. Sqldbm.com
4. Quickdatabasediagrams.com
5. Ondras.zarovi.cz/sql/demo

Check on draw.io (can create good visual diagrams)

**A screenshot of a computer

Description automatically generated**

**How to build a like system**

Rules of likes

* Each user can like a specific post a single time
* A user should be able to ‘unlike’ a post
* Need to be able to figure out how many users like a post
* Need to be able to list which users like a post
* Something besides a post might need to be liked (comments)
* We might want to think about ‘dislikes’ or other kinds of reactions

How To Not Design A Like System

Do not add a ‘likes’ column to posts

* No way to make sure a user likes a post only once
* No way to make sure a user can only ‘unlike’ a post they have liked
* No way to figure out which users like a particular post
* No way to remove a like if a user gets deleted

Designing a Like system

Create a likes table with user\_id and post\_id

Assign a unique constraint to user\_id and post\_id such that a user cannot like a post twice -> UNIQUE(user\_id, post\_id)

Disadvantage of the system above – Can’t really track or know the exact reaction of the like of a post or track the dislikes

**Making a Reaction System Instead**

Make a reactions table with these columns: id, user\_id, post\_id, type

**Polymorphic Associations**

A like can be a ‘post like’ or a ‘comment like’. (polymorphic association)

Requires your app to figure out the meaning of each like

Can’t use foreign key columns – ‘liked\_id’ is a plain integer

Not recommended, but may still see it in use

Best Solution (use this)

Create a likes table with the following columns: id, user\_id, post\_id, comment\_id

|  |  |  |  |
| --- | --- | --- | --- |
| Likes | | | |
| Id | User\_id | Post\_id | Comment\_id |
| 1 | 3 | 1 | NULL |
| 2 | 1 | NULL | 3 |

The table above allows to associate a user with posts or comments and still get to use foreign key columns.

First row indicates a like for a post with an ID of one.

Second row indicates a like for a comment with an id of 3.

We need to make sure we don’t have a scenario where there are two values of post\_id and comment\_id defined or both the post\_id and comment\_id are null

Use the following Constraint:

**ADD CHECK of**

**(COALESCE((post\_id)::BOOLEAN::INTEGER,0)**

**+**

**COALESCE((comment\_id)::BOOLEAN::INTEGER, 0)) = 1**

*COALESCE returns the first value that is not null*

e.g.

SELECT COALESCE(NULL, 5); // returns 5

SELECT COALESCE(10, 5); // returns 10

SELECT (NULL)::BOOLEAN::INTEGER; // returns null(If put in a Coalesce function, coalesce is not going to return any value for that)

SELECT (534)::BOOLEAN::INTEGER; //returns 1

**The Simplest Alternative**

Each type of like gets its own table

Still want to write queries that will count up all likes? *You can use a Union or a View*

Creating a Posts\_likes and Comments\_likes table

The only downside is if a user has to like different kinds of things, we may end up creating many different kinds of tables

**How To Build A Mention System**

**Additional Features around Posts**

REAL datatype goes up to 6 decimals

Photo Mentions vs Caption Mentions

* Need to show a list of posts a user was mentioned in?
* Need to show a list of the most-often mentioned users?
* Need to notify a user when they’ve been mentioned?

Considerations on Photo Tags vs Caption Tags

Tag Solution 1

We could have a single table of tags with the following columns: id, user\_id, post\_id, x and y coordinates.

If the X and Y coordinates are NULL, then it means it’s a tag in a Caption.

Tag Solution 2

Create 2 tag tables: photo\_tags and caption tags.

Their difference being that the photo\_tags have the x and y coordinate and the caption tags have only the user\_id and post\_id

Which ones better?

Do you expect to query for caption\_tags and photo\_tags at different rates?

Will the meaning of a photo\_tag change at some point?

If you may expect any changes, solution 2 is the best, either way solution 2 is very flexible and convenient for any future changes.

Table users {

  id SERIAL [pk, increment]

  created\_at TIMESTAMP

  updated\_at TIMESTAMP

  username VARCHAR(30)

}

Table posts {

  id SERIAL [pk, increment]

  title VARCHAR

  url VARCHAR(200)

  user\_id INTEGER [ref: > users.id]

  caption VARCHAR(240)

  lat REAL

  lng REAL

}

Table comments {

  id SERIAL [pk, increment]

  contents VARCHAR(240)

  user\_id INTEGER [ref: > users.id]

  post\_id INTEGER [ref: > posts.id]

}

Table likes {

  id SERIAL [pk, increment]

  created\_at TIMESTAMP

  user\_id INTEGER [ref: > users.id]

  post\_id INTEGER [ref: > posts.id]

  comment\_id INTEGER [ref: > comments.id]

}

Table photo\_tags {

  id SERIAL [pk, increment]

  created\_at TIMESTAMP

  updated\_at TIMESTAMP

  post\_id INTEGER [ref: > posts.id]

  user\_id INTEGER [ref: > users.id]

  x iNTEGER

  y INTEGER

}

Table caption\_tags {

  id SERIAL [pk, increment]

  created\_at TIMESTAMP

  post\_id INTEGER [ref: > posts.id]

  user\_id INTEGER [ref: > users.id]

}

**How to Build a ‘Hashtag’ System**

Seeing hashtags used in posts, comments, and user bio’s might make you think we need some kind of relationship for each.

We only have to model resources in the DB if we expect to query for them at some point!

Do we expect to run a query to see what posts/comments/users contain a given hashtag?

Can search for posts that contain a hashtag – implies that hashtags in a post’s caption are modelled in the db!

We can’t search for comments or user with a hashtag – implies they are not modelled! (or that we don’t have to)

**Tables for hashtags**

Solution 1

A table of hashtags with columns: id, title and post\_id

This would work but for some performance reasons try using this below:

A table of hashtags with columns: id, title

A table of hashtag\_posts with columns: id, hashtag\_id, post\_id

The reason for the performance reasons is that there were a lot of words being duplicated and any time there is duplication of strings, there may be a concern of space and may be using up a little more storage than what we would want to use.

The second table stores and references the hashtags(hashtag\_id) through an integer, which is less storage consuming.

**A few more user columns**

 bio VARCHAR(400)

 avatar VARCHAR(200)

phone VARCHAR(25)

email VARCHAR(40)

password VARCHAR(50)

 status VARCHAR(15)

**Why No Number of Followers or Posts?**

# posts and # followers can be calculated by running a query on data that already exists in our DB. We call this **‘derived data’.** We generally don’t want to store derived data

**How to design a ‘Follower’ System**

Create a table of followers with columns: id, leader\_id, follower\_id

CHECK (user\_id<>follower\_id)

UNIQUE (leader\_id, follower\_id) # We can never have two rows with an identical leader\_id and follower\_id

**Implementing Database Design Patterns**

1. Create a new db using PGAdmin
2. Convert our design to a series of CREATE TABLE statements
3. Insert data into the database
4. Write some queries
5. Realize that a few things could have been designed better! Make some changes

NOT NULL (A value must be provided (empty strings are values!)

DEFAULT (Provide a default value if an INSERT statement doesn’t give one)

NOT NULL + DEFAULT (We always want a value, but it should be optional)

Latitude values range from -90 < lat < 90

Longitude values range from -180 < long < 180

**Restoring from Scratch**

1. Delete database
2. Recreate new one
3. Right click on new database and select restore
4. Under data options, enable: ***Do not save – owner, query – single transaction, disable – trigger, verbose messages***(usually on by default)

Adding Data to an already existing Database

Right Click on database and select restore, add your data from local path

Enable the following before restoring:

* ***Only data – yes***
* ***Owner – yes***
* ***Single transaction – yes***
* ***Trigger – yes***
* ***Verbose messages – yes***(default)

Query quizzes

-- Select three users with the highest ids from the users table

**SELECT id, username**

**FROM users**

**ORDER by id DESC**

**LIMIT 3;**

-- Join the users and posts table. Show the username of user ID 200 and the captions of all posts they have created (I’ve added username column)

**SELECT (SELECT username FROM users WHERE users.id = 200), posts.user\_id, posts.caption**

**FROM posts**

**JOIN users ON users.id = posts.user\_id**

**WHERE posts.user\_id = 200;**

-- Show each username and the number of 'likes' that they have created in descending order

**SELECT users.username, COUNT(\*) as num\_of\_likes**

**FROM users**

**JOIN likes ON likes.user\_id = users.id**

**-- WHERE Username = 'Desmond.Armstrong42'**

**GROUP BY username**

**ORDER BY num\_of\_likes DESC;**

**Thinking About Performance**

Performance with Postgres

* You can get away with quick tips and hints
* Much easier to understand performance if you understand the internals
* Look at how data is stored and accessed
* Investigate at how indexes are stored and used
* Put these together to understand how queries are executed

Where does Postgres Store Data

All databases are in the base folder (locally)

// shows the path where postgres stores its data

**SHOW data\_directory;**

// lists out the name of the different databases in our postgres installation and an internal identifier that is used for each one

**SELECT oid, datname**

**FROM pg\_database;**

// lists each file that represents individual objects(indexes, sequences, pks) inside our database (base/number/..)

**SELECT \* FROM pg\_class;**

The oid number represents the identifier of the actual raw file in the local database

**Heaps, blocks and Tuples**

* Heap – File that contains all the data (rows) of our table e.g 16386
* Tuple or item – Individual row from the table
* Block or Page – The heap file is divided into many different ‘block’ or ‘pages’. Each page/block stores some number of rows (always 8kb in size)

|  |  |  |
| --- | --- | --- |
| **Block 1** | | |
| Information about this block | | |
|  | Loc of item 1 | Loc of item 2 |
| Free Space | | |
| Data for Tuple 2 | | |
| Data For tuple 1 | | |

The free space is a collection of zeros and ones that are not actually being used by the block now. It is information or space that can eventually be used to assign some kind of actual user.

The Data for Tuple 2 and 1 are the actual data themselves. (collections of zeros and ones)

Understanding how Postgres stores data at the binary level

[PostgreSQL: Documentation: 17: 65.6. Database Page Layout](https://www.postgresql.org/docs/current/storage-page-layout.html) (Outline how Postgres stores information in one individual page)

The first 24 bytes of each page consists of a page header (PageHeaderData) (first 24 cols)

**PageHeaderData layout**

| **Field** | **Type** | **Length** | **Description** |
| --- | --- | --- | --- |
| pd\_lsn | PageXLogRecPtr | 8 bytes | LSN: next byte after last byte of WAL record for last change to this page |
| pd\_checksum | uint16 | 2 bytes | Page checksum |
| pd\_flags | uint16 | 2 bytes | Flag bits |
| pd\_lower | LocationIndex | 2 bytes | Offset to start of free space |
| pd\_upper | LocationIndex | 2 bytes | Offset to end of free space |
| pd\_special | LocationIndex | 2 bytes | Offset to start of special space |
| pd\_pagesize\_version | uint16 | 2 bytes | Page size and layout version number information |
| pd\_prune\_xid | TransactionId | 4 bytes | Oldest unpruned XMAX on page, or zero if none |

**Page layout**

Check Figure 65.1 Page layout in [PostgreSQL: Documentation: 17: 65.6. Database Page Layout](https://www.postgresql.org/docs/current/storage-page-layout.html)

The item id points to the start of the item and the length in bytes

All table rows are structured in the same way. There is a fixed-size header (occupying 23 bytes on most machines), followed by an optional null bitmap, an optional object ID field, and the user data.

Meaning free data +23 bytes +filler info(around 8/9bits) then the actual data(the first byte is probably the Id of the row)

[Binary to Decimal Converter](https://www.rapidtables.com/convert/number/binary-to-decimal.html) (7 digits+ 8 digits)

**A Look at Indexes for Performance**

When loading a query like this:

SELECT \* FROM users WHERE username = ‘Riann’;

Postgres loads up all files locally (a heap file) into memory (RAM) then searches one by one until it finds user Riann

As a data engineer try as much as possible in minimizing the amount of data being moved from the Heap file on the hard drive to memory (performance)

**Full Table Scan** – PG has to load many (or all) rows from the heap file to memory (Frequently (but not always) poor performance

**Whats an Index?**

(SELECT \* FROM users WHERE username = ‘Riann’;)

Example – Riann is at Block 1 Index 2 (Riann is fetched directly from his block/page without having to look at any other block)

An index is a data structure that very efficiently tells us what block/index a record is stored at. (It will help in the above example)

**How an Index works**

Questions to ask?

(SELECT \* FROM users WHERE username = ‘Riann’;)

1. Which column do we want to have very fast lookups on? (username)
2. Extract only the property we want to do fast lookups by and the block/index for each
3. Sort in some meaningful way (alphabetical, ascending or descending)
4. Organize into a tree data structure (Evenly distribute values in the leaf nodes, in order left to right)
5. Add helpers to the root node

**Things to note from Screenshot:**

* We didn’t load up the vast majority of records inside the heap file i.e we did not have to access block 0 inside the heap file
* We never had to visit the left leaf node i.e with Alf and Jia – The evaluation at the root node made this possible

**Creating an index**

-- Creating an index without assigning a name

**CREATE INDEX ON users (username);**

-- Creating an index with a name (best to follow naming convention - users\_column\_idx)

**-- CREATE INDEX users\_username\_idx ON users (username);**

-- Delete index

**-- DROP INDEX users\_username\_idx;**

**Benchmarking Queries**

Using the ‘EXPLAIN ANALYZE’ keywords at the beginning of a query

*Example*

*-- Running the query with the index (0.05ms – execution time) 10 times faster*

**EXPLAIN ANALYZE SELECT \***

**FROM users**

**WHERE username = 'Emil30';**

*-- Running the query without the index (0.5ms)*

**EXPLAIN ANALYZE SELECT \***

**FROM users**

**WHERE username = 'Emil30';**

**Downsides of Indexes**

* Storage cost (a separate file is created in hard drive to store extra data) and can be expensive especially on cloud storage

**CREATE INDEX ON users (username);**

-- prints out the amount of space used by the users table (872kb)

**SELECT pg\_size\_pretty(pg\_relation\_size('users'));**

-- prints out the amount of space used by the username index file (184kb)

**SELECT pg\_size\_pretty(pg\_relation\_size('users\_username\_idx'));**

* Slows down insert/update/delete – the index has to be updated!
* Index might not actually used to speed up a particular query

**Index type**

* B-Tree -> General purpose index. 99% of the time you want this
* Hash -> Speeds up simple equality checks
* GiST – Geometry, full-text search
* SP-GiST -> Clustered data, such as dates -many rows might have the same year
* GIN -> For columns that contain arrays or JSON data
* BRIN -> Specialized for really large datasets

**Automatically generated Indexes**

* Postgres automatically creates an index for the primary key column of every table (e.g. hashtags\_pkey)
* Postgres automatically creates an index for any ‘unique’ constraint (e.g. hashtags\_title\_key)
* NB: They don’t get listed under ‘indexes’ in PGAdmin!

-- the pg\_class lists all the different objects in the database. You can see all the indexes listed as objects with the code below

**SELECT relname, relkind**

**FROM pg\_class**

**WHERE relkind = 'i';**

**Quiz**

You realize that you have to find phones based on their name very often, and you want queries involving the name to be as fast as possible.  What would you do to make these queries very fast?

Ans

I don’t need to create an index because there is already a uniqueness check on that column. Postgres will have already created an index for me!

**Behind the Scenes of Indexes**

When creating an index an actual file is created on the hard drive and is assigned a particular number

The structure of the file is:

* Meta page (information about the overall index)
* Leaf Block/Page
* Leaf Block/page
* Root Block/page
* Leaf Block/Page

Each file is 8kb

In memory:

Meta page -> Root Block – Leaf block(x3)

Parent nodes are created when Postgres has over certain number of records

Root – parent Node – Leaf Node

-- an extension gives us an additional functionality inside of Postgres

CREATE EXTENSION pageinspect;

-- bt - b tree, metap - metapage

-- root is the index of our root page inside of the index file

SELECT \*

FROM bt\_metap('users\_username\_idx');

-- ctid represents the index of the different leaf nodes

SELECT \*

FROM bt\_page\_items('users\_username\_idx', 3)

-- the ctid in a leaf node represents the page and the index in that page of where we can find a particular record in a heap file

SELECT \*

FROM bt\_page\_items('users\_username\_idx', 1);

-- shows ctid as (33,43)

SELECT ctid, \* FROM users WHERE username = 'Aaliyah.Hintz';

-- The first row in a node is a pointer to the first record in the next table

-- gives the name/identifier for the database itself

SELECT oid, datname from pg\_database;

SHOW data\_directory;

SELECT \*

FROM pg\_class

WHERE relkind = 'i';

**Basic Query Tuning**

The query processing timeline

Take a query like:

**SELECT \* FROM users WHERE username = ‘Alyson14’;**

Postgres takes the following steps to process the query:

1. **Parser** – Ensures what you wrote is valid sql (appropriate punctuation, keywords, spelling e.t.c.) A query tree is then built.
2. **Rewrite** – takes a look at the query tree and rewrites it if Postgres thinks some parts of the tree could be rewritten efficiently
3. **Planner** – Looks at the query tree, figures out what information am trying to fetch then comes with a series of different plans/strategies that could be used to get that information
4. **Execute**

Explain and Explain Analyze

Explain and Explain analyze help us understand how a query is actually being executed and helps us figure out how to improve the performance of that query as well

Explain – Build a query and display info about it (tells us what Postgres plans to do but doesn’t actually do it)

Explain analyze – Build a query plan, run it, and info about it (tells us what Postgres plans to do and actually runs it)

NB: They are for benchmarking + evaluating queries, not for use in real data fetching

e.g.

**EXPLAIN ANALYZE SELECT username, contents**

**FROM users**

**Join comments ON comments.user\_id = users.id**

**WHERE username = 'Alyson14'**

Solving an Explain Mystery

A query node is represented by a row that has an arrow (->) on it in the query plan when executing explain analyze. A hash join can be taken as a query node too

Whenever we see an arrow in the query plan, the arrow indicates that it is trying to access some data in the database or inside of an index and tries to emit or pass that data to the nearest parent that has an arrow on it.

*Explain Analyze Output*

* Hash Join – How this node is generating data
* Cost = 7.77..1234.56 – amount of processing power required for this step
* rows = 11 – A guess at how many rows this step will produce
* width=81 – A guess at the average number of bytes of each row

Explain Output

* It does show the number of rows and average width of those rows without actually executing the query (How does Postgres know this?)
* Postgres keeps some very detailed statistics about what is going on inside of each of our different tables

Pg\_stats is a table that is maintained by Postgres that shows detailed statistics of all the different values and all the different columns of our different tables

The statistics shown of the tables is what helps Postgres have a guess of some of the different rows coming out of each of those steps of the query plan, the cost or the outcome of those different steps without having to do any processing

**SELECT \***

**FROM pg\_stats**

**WHERE tablename = 'users';**

**Advanced Query Tuning**

Developing an Intuitive Understanding of Cost

Working definition for cost – Amount of time to execute some part of our query plan (not super accurate but good enough)

On the planner step and assume this is the query that is to be executed:

**SELECT \* FROM users WHERE username = ‘Alyson14’;**

Postgres may launch either of the following steps:

1. Look at users\_username\_idx then get users?

* Find the IDs of users who have username of ‘Alyson14’
* Get root node
* Jump to some random child page
* Process the values in that node

*One page to execute above and below*

* Open users heapfile
* Jump to each block that has the users we are looking for
* Process the appropriate users from each block

1. Fetch all users and search through them?

* Open the users heap file

*Do below once for every page*

* Load all users from the first block
* Process each user, see if it contains the correct username
* Repeat the process for the next block (step 2)

NB: Loading data from random spots off a hard drive usually takes more time than loading data sequentially (one piece after another)

Assumption – Let’s assume that loading a random page takes 4 time stronger than loading up pages sequentially

Random calculation:

(2 pages loaded in random order) \* 4 -> 8

(110pages loaded sequentially) \* 1 -> 110

**Queries**

EXPLAIN SELECT username, contents

FROM users

Join comments ON comments.user\_id = users.id

WHERE username = 'Alyson14';

EXPLAIN ANALYZE SELECT username, contents

FROM users

Join comments ON comments.user\_id = users.id

WHERE username = 'Alyson14';

**A touch more on costs**

Formula for calculating the cost of a processing step in a query plan

Cost =

(Number of pages read sequentially) \* seq\_page\_cots

+ (Number of pages read at random) \* random\_page\_cost

+ (Number of rows scanned) \* cpu\_tuple\_cost

+ Number of index entries scanned) \* cpu\_index\_tuple\_cost

+ (Number of times function/operator evaluated) \* cpu\_operator cost

Quiz

The formula for calculating the cost of a processing step in a query plan is:

1. COST = (#pages read sequentially) \* seq\_page\_cost
2. + (# pages read at random) \* random\_page\_cost
3. + (# rows scanned) \* cpu\_tuple\_cost
4. + (# index entries scanned) \* cpu\_index\_tuple\_cost
5. + (# times function/operator evaluated) \* cpu\_operator\_cost

Where

1. seq\_page\_cost = 1.0
2. random\_page\_cost = 4.0
3. cpu\_tuple\_cost = 0.01
4. cpu\_index\_tuple\_cost = 0.005
5. cpu\_operator\_cost = 0.0025

What is the cost for a query node that has to open 5 pages of data sequentially and then process 100 rows total? **6**

**Quiz**

Question 2:

The formula for calculating the cost of a processing step in a query plan is:

1. COST = (#pages read sequentially) \* seq\_page\_cost
2. + (# pages read at random) \* random\_page\_cost
3. + (# rows scanned) \* cpu\_tuple\_cost
4. + (# index entries scanned) \* cpu\_index\_tuple\_cost
5. + (# times function/operator evaluated) \* cpu\_operator\_cost

Where

1. seq\_page\_cost = 1.0
2. random\_page\_cost = 4.0
3. cpu\_tuple\_cost = 0.01
4. cpu\_index\_tuple\_cost = 0.005
5. cpu\_operator\_cost = 0.0025

What is the cost for a query node that has to open 4 pages of an index (probably at random), process 75 tuples from the index, then open 20 different pages from a heap file (also at random) and process 214 tuples? **98.515**

**Startup vs Total Costs**

**A diagram of a graph

Description automatically generated with medium confidence**

Cost normally has two sections (e.g cost = 8.31..1756.11)

The 8.31 – represents the cost to produce the first row

The 1756.11 – represents the cost to produce all rows

NB: Its better if the startup cost is low as it indicates that some processing has already begun. If it’s a huge number, then it probably indicates that all rows may have been scanned first before any processing begins

***Costs flow up***

The hash join cost indicates the total cost of all the hash join children and the hash join itself (8.31).

**Use My Index**

Take a query like this:

EXPLAIN SELECT \*

FROM likes

WHERE created\_at > '2013-01-01';

CREATE INDEX likes\_created\_at\_idx ON likes (created\_at);

In as much as an index can be created at column ‘created\_at’ Postgres can decide to still use a sequential scan to produce results.

Indexes are best used for fetching a small amount of data and not too many. If the query like the one above is going to return results constituting almost 70% of the records, postgres automatically uses sequential scans.

Don’t need to force Postgres to do an index scan when it has already performed a sequential scan despite an index having been created at a particular column. Postgres has already done the math and knows that the sequential scan is the best method to be used for efficiency

**SIMPLE COMMON TABLE EXPRESSIONS**

Common Table Expressions

* These are techniques one can use to make a query easier to read
* They are defined with a ‘with’ before the main query
* Produces a table that we can refer to anywhere else
* It has two forms: A **Simple** and **Recursive form**
* A Simple form to make a query easier to understand
* A recursive form to write queries that are otherwise impossible to write!

e.g.

Show the username of users who were tagged in a caption or photo before January 7th, 2010. Also show the date they were tagged.

Solution without table expressions

**SELECT users.username, tags.created\_at**

**FROM users**

**JOIN (**

**SELECT user\_id, created\_at FROM caption\_tags**

**UNION ALL**

**SELECT user\_id, created\_at FROM photo\_tags**

**) AS tags ON tags.user\_id = users.id**

**WHERE tags.created\_at < '2010-01-07'**

Solution with table expressions (simple form)

**WITH tags AS (**

**SELECT user\_id, created\_at FROM caption\_tags**

**UNION ALL**

**SELECT user\_id, created\_at FROM photo\_tags**

**)**

**SELECT users.username, tags.created\_at**

**FROM users**

**JOIN tags ON tags.user\_id = users.id**

**WHERE tags.created\_at < '2010-01-07';**

**RECURSIVE COMMON TABLE EXPRESSION (CTES)**

* They are very different from simple CTE’s
* Useful anytime you have a tree or graph-type data structure
* Must use a ‘union’ keyword – simple CTE’s don’t have to use a union!
* One of the hardest concepts in SQL

Example

**WITH RECURSIVE countdown(val) AS (**

**SELECT 10 AS val** -- Initial, Non-recursive query

**UNION**

**SELECT val - 1 FROM countdown WHERE val > 1** -- Recursive query

**)**

**SELECT \***

**FROM countdown;**

Returns 3, 2, 1

**Recursive CTE step by step**

* Define the results and working tables
* Run the initial non-recursive statement, put the results into the results table and working table
* Run the recursive statement replacing the table name ‘countdown’ with a reference to the working table
* If recursive statement returns some rows, append them to the results and run recursion again
* If recursive statement returns no rows stop recursion

**Why Use Recursive CTE’s**

Take a look at Instagram suggestions

I follow The Rock and Kevin Hart. Both celebs then follow Justin Beiber, Jennifer Lopez and Snoop Dogg. Instagram will then suggest Justin Beiber, JLo and Snoop Dogg

To get a list of suggestions, we’ll need to write a query to take a look at some user is following find those people and find who they are following in turn

The scenario above is definitely a tree-graph like structure and a recursion would be needed

**Writing The Query**

We’ll use the users and followers table

Example

The query below list the suggestions (30) that user 1000 should try following based on the people the user already follows

**WITH RECURSIVE suggestions(leader\_id, follower\_id, depth) AS (**

**SELECT leader\_id, follower\_id, 1 AS depth**

**FROM followers**

**WHERE follower\_id = 1**

**UNION**

**SELECT followers.leader\_id, followers.follower\_id, depth + 1**

**FROM followers**

**JOIN suggestions ON suggestions.leader\_id = followers.follower\_id**

**WHERE depth < 3**

**)**

**SELECT DISTINCT users.id, users.username**

**FROM suggestions**

**JOIN users ON users.id = suggestions.leader\_id**

**WHERE depth > 1**

**LIMIT 30;**

**SIMPLIFYING QUERIES WITH VIEWS**

Most Popular Users

Example

Show the most popular users – the users who were tagged the most

Answer:

**WITH tags AS (**

**SELECT user\_id FROM caption\_tags**

**UNION ALL**

**SELECT user\_id FROM photo\_tags**

**)**

**SELECT users.username, COUNT(users.username)**

**FROM users**

**JOIN tags ON tags.user\_id = users.id**

**GROUP BY users.username**

**ORDER BY COUNT(users.username) DESC;**

A Possible Solution For Merging Tables

* We’ve had to find the union several times
* There’s been no benefit to keeping these records in separate tables!
* Guess we have a bad design!
* Two possibilities to fix this up

Solution 1

Merge the two tables, delete the original ones (Create a new table called tags which has similar columns to the photo and cation tags tables and copy all rows from photo\_tags and caption\_tags)

**Copying rows from photo\_tags** can be done this way (replace photo\_tags with caption\_tags for copying from caption\_tags)

**INSERT INTO tags (created\_at, updated\_at, user\_id, post\_id, x, y)**

**SELECT created\_at, updated\_at, user\_id, post\_id, x, y**

**FROM photo\_tags**

Downsides of this approach

* Can’t copy over the ID’s of photo\_tags and caption\_tags since they must be unique
* If we delete original tables, we break any existing queries that refer to them!

Creating A View

* Create a fake table that has rows from other tables
* These can be exact rows as they exist on another table, or a computed value
* Can reference the view in any place where we’d normally reference a table
* View doesn’t actually create a new table or move any data around
* Doesn’t have to be used for a union! Can compute absolutely any values
* NB: CTE’s can be referred to only in the query they are attached to
* Views are similar but can be referred to in different queries in the future – they are persistent

View Solution

**CREATE VIEW tags AS (**

**SELECT id, created\_at, user\_id, post\_id, 'photo\_tag' AS type FROM photo\_tags**

**UNION ALL**

**SELECT id, created\_at, user\_id, post\_id, 'caption\_tag' AS type FROM caption\_tags**

**);**

* Test using the select below

**SELECT users.username, COUNT(users.username)**

**FROM users**

**JOIN tags ON tags.user\_id = users.id**

**GROUP BY users.username**

**ORDER BY COUNT(users.username) DESC;**

When To Use A View

The 10 most recent posts are really important

* Show the users who created the 10 most recent posts
* Show the users who were tagged in the 10 most recent posts
* Show the average number of hashtags used in the 10 most recent posts
* Show the number of likes each of the 10 most recent posts received
* Show the hashtags used by the 10 most recent posts
* Show the total number of comments the 10 most recent posts received

Solution

Create a view of recent\_posts and refer to it to solve the challenges above

**CREATE VIEW recent\_posts AS (**

**SELECT \***

**FROM posts**

**ORDER BY created\_at DESC**

**LIMIT 10**

**);**

Show the users who created the 10 most recent posts:

**SELECT username**

**FROM recent\_posts**

**JOIN users ON users.id = recent\_posts.user\_id;**

Designing and Changing A View

Use **CREATE OR REPLACE VIEW** keyword to change or replace a view:

**CREATE OR REPLACE VIEW recent\_posts AS (**

**SELECT \***

**FROM posts**

**ORDER BY created\_at DESC**

**LIMIT 15**

**);**

To drop a view:

**DROP VIEW recent\_posts;**

**OPTIMIZING QUERIES WITH MATERIALIZED VIEWS**

A Materialized View – A Query that gets executed only at very specific times, but the results are saved and can be referenced without rerunning the query

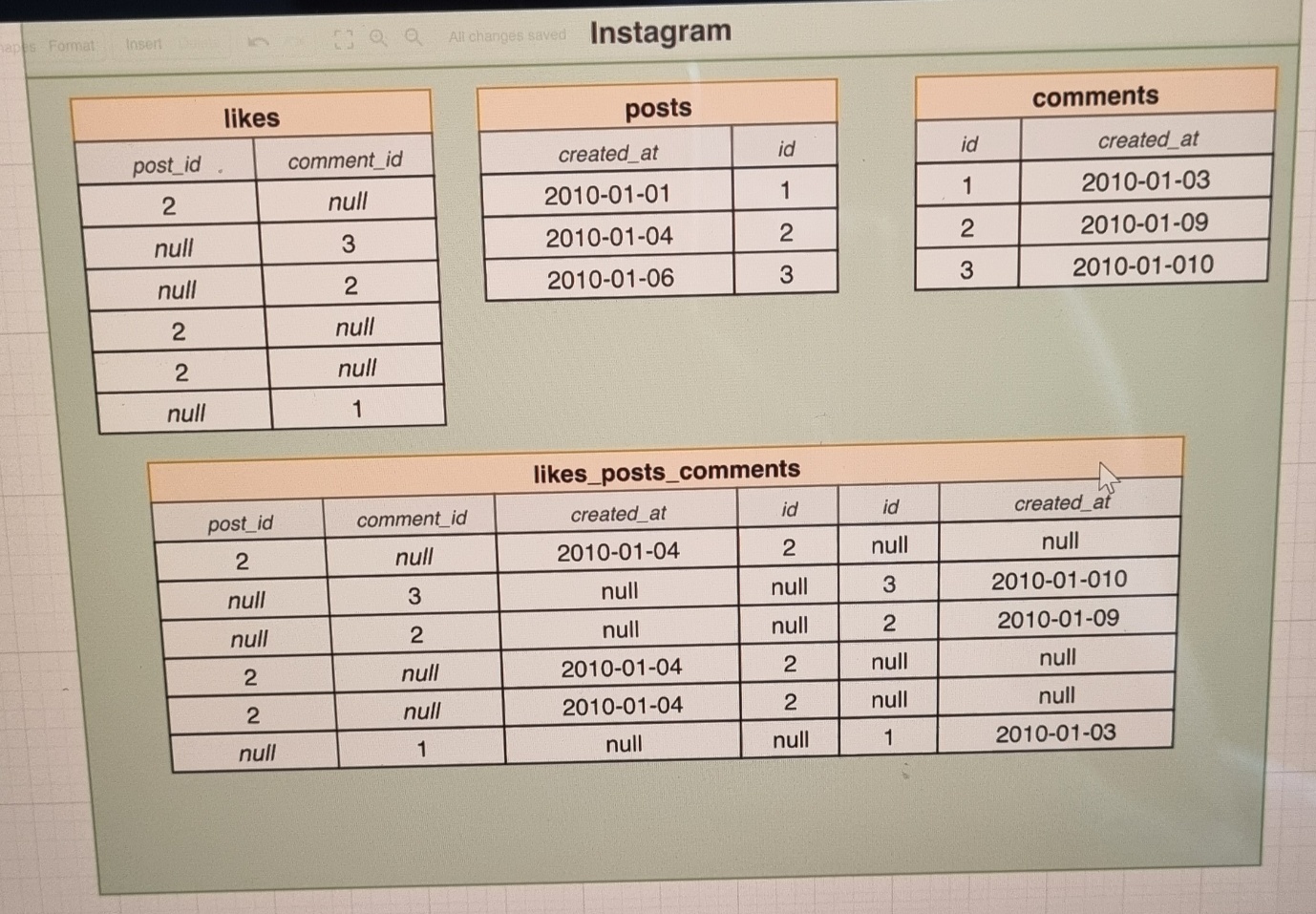
Views – Query that gets executed every time you refer to it

Example

For each week, show the number of likes that posts and comments received. Use the post and comment created\_at date, not when the like was received

Reminder on left joins

The picture below shows why we need to use a left join in this scenario. Using an inner join will remove all rows cause of the null values



Writing A Slow Query

The date trunc function allows one to pull one piece of information out of a timestamp

In this case we’ll pull out the week from all the different created at values that we have.

The date trunc rounds down the created\_at values to the nearest week

Solution:

**SELECT**

**date\_trunc('week', COALESCE(posts.created\_at, comments.created\_at)) AS week,**

**COUNT(posts.id) AS num\_likes\_for\_posts,**

**COUNT(comments.id) AS num\_likes\_for\_comments**

**FROM likes**

**LEFT JOIN posts ON posts.id = likes.post\_id**

**LEFT JOIN comments ON comments.id = likes.comment\_id**

**GROUP BY week**

**ORDER BY week;**

Creating and Refreshing Materialized Views

With data ensures that when we first create the materialized view, Postgres automatically runs the query one time and holds on to the results.

Solution

**CREATE MATERIALIZED VIEW weekly\_likes AS (**

**SELECT**

**date\_trunc('week', COALESCE(posts.created\_at, comments.created\_at)) AS week,**

**COUNT(posts.id) AS num\_likes\_for\_posts,**

**COUNT(comments.id) AS num\_likes\_for\_comments**

**FROM likes**

**LEFT JOIN posts ON posts.id = likes.post\_id**

**LEFT JOIN comments ON comments.id = likes.comment\_id**

**GROUP BY week**

**ORDER BY week**

**) WITH DATA;**

The speed was greatly increased in this scenario.

One downside to this is that, if we modify any of the underlying data e,g posts or comments, the cached results are not going to be modified.

We have to manually tell Postgres that it needs to go back and update the materialized view as below:

**REFRESH MATERIALIZED VIEW weekly\_likes;**

Quiz

**What is the primary difference between a view and a materialized view?**

Both Views and materialized views wrap up a query. When you refer to a view, the query is executed. When you refer to a materialized view, you get back the results from when the materialized view was created or when it was last refreshed.

**You are writing a query that takes twenty seconds to execute.  Even though the query gets executed many times per day, the results only change once per month**

**Would this query be a good candidate for a materialized view?**

Yes

**HANDLING CONCURRENCY AND REVERSIBILITY WITH TRANSACTIONS**

What are Transactions used for?

Example

Transfer $50 from Alyson to Gia

* Withdraw $50 from Alyson’s account:

UPDATE accounts

SET balance = balance -50

WHERE name = ‘Alyson’;

* Add $50 to Gia’s account

UPDATE accounts

SET balance = balance + 50

WHERE name =’Gia’;

If there is a crash in the server and only half of the code above is ran, then there is no way we’ll know that we need to add $50 to Gia’s account.

Transactions are therefore useful since they ensure that all the different updates are always executed or none of them.

The BEGIN keyword begins a transaction.

Changes made in a transaction are in a separate environment and need to be committed for the merging to occur into the main data pool.

The code below took place in a transaction environment.

**BEGIN;**

**UPDATE accounts**

**SET balance = balance - 50**

**WHERE name = 'Alyson';**

**UPDATE accounts**

**SET balance = balance + 50**

**WHERE name = 'Gia';**

Run ‘COMMIT’ to merge changes back into main data pool

**COMMIT;**

Run ‘ROLLBACK’ to dump all pending changes and delete the separate workspace(transaction environment) and making the current connection to continue looking at the main data pool

*NB: Running a bad command will put the transaction in an ‘aborted’ state and you must rollback*

A bad command is something like:

SELECT \* FROM asdafadfad

NB: Losing the connection(crashing) will automatically rollback the transaction

Transaction cleanup on Crash

Remember to run a rollback if your transaction is in an aborted state.

Writing a bad query e.g. a query with a typo will abort the transaction and you ought to do a rollback.

If there is a transaction crash, Postgres automatically aborts the transaction

**MANAGING DATBASE DESIGN WITH SCHEMA MIGRATIONS**

A Story On Migrations

Lessons

* Anytime we make a change to our database structure, we have to synchronize that and deploy it at the same time that we change our clients as well.

Changes to the DB structure and changes to clients need to be made at precisely the same time

Pic in docs

A diagram of a software application

Description automatically generated

* When working with other engineers, we need a really easy way to tie the structure of our database to our code

Migration Files

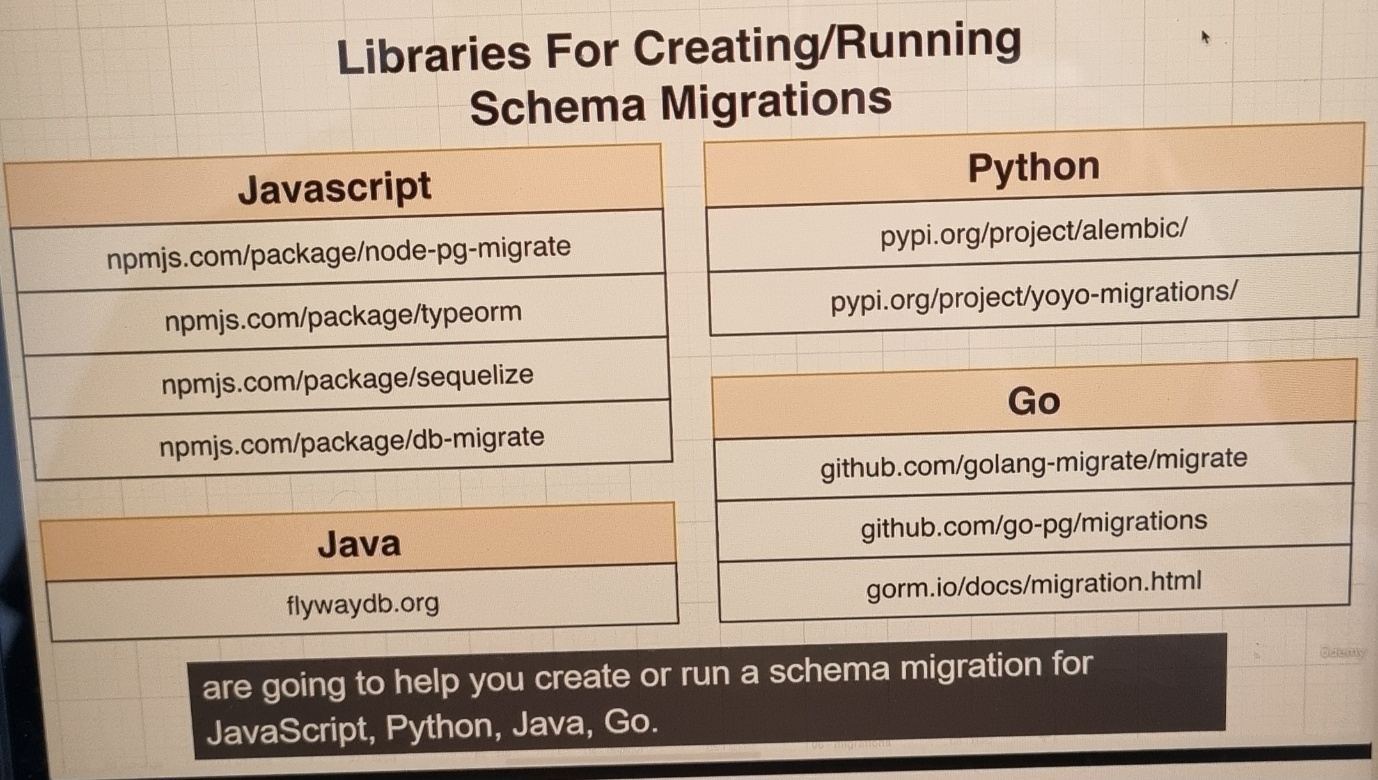
A Schema Migration File is a code that describes a precise change to make to the database.

Apply is the term used to describe taking a migration file and making a change to the database.

Revert is used to describe undoing that change.

A Few Notes On Migration Libraries

Pic in docs



Many migration tools can automatically generate migrations for you

It is highly recommended that you write all migrations manually using plain SQL

Project Creation

Create a folder called ig. Cd to the ig folder and run the below commands

**Npm init -y** – generates a package.json file which allows us to install some different modules into this project.

**npm install node-pg-migrate pg** – installs two modules(node-pg-migrate and Postgres module)

Generating And Writing Migrations

Go into the package.json file and change the scripts content to

“Scipts”: {

“migrate”: “node-pg-migrate”

},

The above allows us to access the node pg migrate from our terminal

**npm run migrate create table comments –** This creates a migration and in the migration file, you will see exports.up and exports.down functions

The exports.up contains or produces some amount of SQL or run some command or do whatever to advance the structure of our database in some way.

The exports.down contains some SQL or runs some command or do whatever to revert what we did with exports.up

Example

exports.up = (pgm) => {

    pgm.sql(`

        CREATE TABLE comments (

            id SERIAL PRIMARY KEY,

            created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

            updated\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

            contents VARCHAR(240) NOT NULL

        );

    `);

};

/\*\*

 \* @param pgm {import('node-pg-migrate').MigrationBuilder}

 \* @param run {() => void | undefined}

 \* @returns {Promise<void> | void}

 \*/

exports.down = (pgm) => {

    pgm.sql(`

        DROP TABLE comments;

    `);

};

Applying And Reverting Migrations

Firstly, we set up an environment variable called DATABASE\_URL that tells the Node PG migrate module that we’re using exactly how to connect to our database running on our local machine

The password is the initial password you set when installing Postgres

Username is **postgres**

General format is:

Postgres://USERNAME:PASSWORD@localhost:5432/socialnetwork

Once we have the string put together , we will run a specific command:

Windows with CMD

Set DATABASE\_URL=postgres://USERNAME:PASSWORD@localhost:5432/socialnetwork&&npm run migrate up

Windows with Powershell

$env:DATABASE\_URL=”postgres://USERNAME:PASSWORD@localhost:5432/socialnetwork”; npm run migrate up

$env:DATABASE\_URL=”postgres://postgres:**PASSWORD** @localhost:5432/socialnetwork”; npm run migrate up

Windows with Git Bash

DATABASE\_URL=postgres://USERNAME:PASSWORD@localhost:5432/socialnetwork npm run migrate up

To revert the migrations, replace the word ‘up’ with the word ‘down’

Generating And Applying A Second Migration

Changing the contents column to the name of ‘body’

**npm run migrate create rename contents to body**

**$env:DATABASE\_URL=”postgres://postgres:PASSWORD@localhost:5432/socialnetwork”; npm run migrate up**

Finish up by checking the table in pgadmin

SCHEMA VS DATA MIGRATIONS