TOBIAS PETRY

Next-Level Database Techniques For Developers

Learn secret and lesser-known SQL features and approaches to become a database wizard.

SQL For Devs

Table of Contents

In 2021 I started sharing database tips and tricks for developers that gained more and more interest every time I shared something. As of now, thousands of developers are following me to read my weekly database advice. This book was written to collect the best ones and make them easier to look up for future reference. As time passes, I hope to have all these and much more available in more detailed versions on SQLforDevs.com. Don't forget to subscribe to the newsletter to be notified about new content.

This book is designed as a cookbook with many small independent recipes. You can skip chapters or tips as you want to. In contrast to a whole book, the descriptions are all kept short by design. You should be able to read it thoroughly in a single evening and start using its ideas the next day.

I hope you learn a lot while reading it and level up your knowledge as planned. I really appreciate hearing any feedback from you.

Have fun reading it.
Tobias Petry

Data Manipulation

A database without any `INSERT`, `UPDATE` or `DELETE` query would be a barely valuable application. Although some applications exist with only static content, they are the exception and you will have to do data modifications all the time. While this seems to be the most uncomplicated functionality in SQL, it still has room for improvement for your applications. Always remember that the number of write operations your disk can do in a second is very limited. If you can reduce the operations per second, your application will be much more performant.

The data manipulation chapter will teach you tricks to update rows based on information in other tables, delete duplicate rows or make your application faster by removing lock contention. You should study the last tip closely, as I have often found this to be a performance problem.

Prevent Lock Contention For Updates On Hot Rows

```
-- MySQL
INSERT INTO tweet_statistics (
  tweet_id, fanout, likes_count
) VALUES (
  1475870220422107137, FLOOR(RAND() * 10), 1
) ON DUPLICATE KEY UPDATE likes_count =
likes_count + VALUES(likes_count);

-- PostgreSQL
INSERT INTO tweet_statistics (
  tweet_id, fanout, likes_count
) VALUES (
  1475870220422107137, FLOOR(RANDOM() * 10), 1
) ON CONFLICT (tweet_id, fanout) DO UPDATE SET likes_count =
tweet_statistics.likes_count + excluded.likes_count;
```

In some applications counters for e.g. likes of a tweet are constantly updated. During a traffic spike or for trendy content a counter may get updated countless times within a second. Due to the database's concurrency control the updates will start interfering with each other as a row can only be locked by one transaction (query) at a time. Every update will be executed one after another instead of parallel execution for independent rows.

Instead of updating a single row, the increments are fan outed to e.g. 100 different rows in a special counter table. The scaling factor now increases by the number of additional rows the counter is written to. Those values are later aggregated to a single value and saved in their original column that would have had lock contention.

Updates Based On A Select Query

```sql
-- MySQL
UPDATE products
JOIN categories USING(category_id)
SET price = price_base - price_base * categories.discount;

-- PostgreSQL
UPDATE products
SET price = price_base - price_base * categories.discount
FROM categories
WHERE products.category_id = categories.category_id;
```

Tables are often not updated in isolation, but the values are updated based on information stored in other tables. For e.g. discounting all products on Black Friday, a discount for every product category will be applied. Instead of the naive approach to execute an update query for every category, you can update the products by joining them to their categories. The manual join in the application is replaced by a more efficient one by the database.

> **Notice:** I have written a more extensive text about this topic on my database focused website SqlForDevs.com: UPDATE from a SELECT

Return The Values Of Modified Rows

```
-- PostgreSQL:
DELETE FROM sessions
WHERE ip = '127.0.0.1'
RETURNING id, user_agent, last_access;
```

Many maintenance operations are based on finding particular rows, processing them (e.g. sending an email or calculating some statistics) and marking them as processed. Typically a flag within the row is updated or deleted as it is not needed anymore. This workflow can be simplified by using the RETURNING feature and doing the data manipulation and selection of the data in one step.

This feature is available for `DELETE`, `INSERT` and `UPDATE` queries and will always return the data after the modification, e.g. the inserted or updated data with all triggers executed and generated values available.

> **Notice:** This feature is only available for PostgreSQL.

Delete Duplicate Rows

```sql
-- MySQL
WITH duplicates AS (
  SELECT id, ROW_NUMBER() OVER(
    PARTITION BY firstname, lastname, email
    ORDER BY age DESC
  ) AS rownum
  FROM contacts
)
DELETE contacts
FROM contacts
JOIN duplicates USING(id)
WHERE duplicates.rownum > 1;

-- PostgreSQL
WITH duplicates AS (
  SELECT id, ROW_NUMBER() OVER(
    PARTITION BY firstname, lastname, email
    ORDER BY age DESC
  ) AS rownum
  FROM contacts
)
DELETE FROM contacts
USING duplicates
WHERE contacts.id = duplicates.id AND duplicates.rownum > 1;
```

After some time, most applications will have duplicated rows resulting in a bad user experience, higher storage requirements and less database performance. The cleaning process is usually implemented in application code with complex chunking behavior as the data does not fit into memory entirely. By using a Common Table Expression (CTE) the duplicate rows can be identified and sorted by their importance to keep them. A single delete query can afterward delete all duplicates except a specific number of ones to keep. The former complex logic is done by one simple SQL query.

Notice: I have written a more extensive text about this topic on my database focused website SqlForDevs.com: Delete Duplicate Rows

Table Maintenance After Bulk Modifications

```sql
-- MySQL
ANALYZE TABLE users;

-- PostgreSQL
ANALYZE SKIP_LOCKED users;
```

The database needs up-to-date statistics about your tables like the approximate amount of rows, data distribution of values and more to calculate the most efficient way to execute your query. Contrary to indexes that are automatically altered whenever a row affecting its data is created, updated or deleted the statistics are not mutated on every change. A recalculation is only triggered when a threshold of changes to a table is crossed.

Whenever you change a big part of a table, the number of affected rows may still be below the statistics recalculation threshold but significant enough to make the statistics incorrect. Some queries may become very slow as the database predicts the best query plan based on the now incorrect information about the table. Therefore, you should analyze a table to trigger the statistics recalculation after every significant change to ensure fast queries.

Querying Data

Most of the SQL queries you write and execute will be the ones reading data from the database. It is the cornerstone of your application because not showing any data would make it useless. But it's also the best opportunity to remove a lot of application boilerplate by using more fancy querying approaches. In many use cases, those approaches also improve the performance as you do the data processing where the data is instead of transferring it all to your application.

The querying chapter will show you exceptional features like for-each loops within SQL, some null handling tricks, pagination mistakes you probably do and many more. You should read the tip about data refinement with common table expressions very closely; once you understand it, you will use it very often. Trust me.

Reduce The Amount Of Group By Columns

```sql
SELECT actors.firstname, actors.lastname, COUNT(*) as count
FROM actors
JOIN actors_movies USING(actor_id)
GROUP BY actors.id
```

You probably learned long ago that when grouping on some columns, you must add all `SELECT` columns to the `GROUP BY`. However, when you group on a primary key, all columns of the same table can be omitted because the database will add them for you automatically. Your query will be shorter and therefore easier to read and understand.

Fill Tables With Large Amounts Of Test Data

```sql
-- MySQL
SET cte_max_recursion_depth = 4294967295;
INSERT INTO contacts (firstname, lastname)
WITH RECURSIVE counter(n) AS(
  SELECT 1 AS n
  UNION ALL
  SELECT n + 1 FROM counter WHERE n < 100000
)
SELECT CONCAT('firstname-', counter.n), CONCAT('lastname-', counter.n)
FROM counter

-- PostgreSQL
INSERT INTO contacts (firstname, lastname)
SELECT CONCAT('firstname-', i), CONCAT('lastname-', i)
FROM generate_series(1, 100000) as i;
```

You might need to fill a database with many rows for some performance testing sometimes. This data is usually generated with fake data generation libraries and a lot of code to make the data look the most realistic. Instead of slowly inserting a lot of test data one by one, you let the database generate many dummy rows just to fill up the table for testing the efficiency of your indexes.

Warning: For meaningful benchmarks, the tables need realistic data and values distribution. But you can use this approach to fill the tables with more rows not affected by your queries.

Simplified Inequality Checks With Nullable Columns

```sql
-- MySQL
SELECT * FROM example WHERE NOT(column <> 'value');

-- PostgreSQL
SELECT * FROM example WHERE column IS DISTINCT FROM 'value';
```

Searching for rows with a nullable column not equal to a specific value is complicated. The standard approach `col != 'value'` does not work as a null isn't comparable to anything and therefore would not be included in the result. You always use the more complicated `(col IS NULL OR col != 'value')` approach to get a correct result. Luckily, both databases support a special feature to compare columns for inequality with included null handling.

Prevent Division By Zero Errors

```
SELECT visitors_today / NULLIF(visitors_yesterday, 0)
FROM logs_aggregated;
```

Calculating statistics within the database is easy and you've probably done it hundreds of times. But you may have received errors from those queries months later because the assumptions made when writing the query were no longer valid. Probably your website didn't have any visitors on a specific day because of downtime, or your online shop did not sell any product yesterday for the first time. As no row is available for that day, a division by `SUM(visitors_yesterday)` will trigger a division by error. You should always ensure you are not dividing by zero in case some data is missing. By transforming the divisor from a zero to a null value, you can eliminate that problem.

Sorting Order With Nullable Columns

```sql
-- MySQL: NULL values placed first (default)
SELECT * FROM customers ORDER BY country ASC;
SELECT * FROM customers ORDER BY country IS NOT NULL, country ASC;

-- MYSQL: NULL values placed last
SELECT * FROM customers ORDER BY country IS NULL, country ASC;

-- PostgreSQL: NULL values placed first
SELECT * FROM customers ORDER BY country ASC NULLS FIRST;

-- PostgreSQL_ NULL values placed last (default)
SELECT * FROM customers ORDER BY country ASC;
SELECT * FROM customers ORDER BY country ASC NULLS LAST;
```

With MySQL any NULL value will be placed before everything when sorting in ascending direction, but with PostgreSQL they will be at the end. As the SQL standard forgot to specify a sorting order for NULL values, they both had to invent their own rules. But where those values are ranked shouldn't be a decision of your database technology. The application should control this to improve the user experience: A user sorting e.g. a list of names in ascending or descending order will not be interested in seeing null values first when searching for a specific row. They should be at the end as they are the least important ones. But when searching for rows with missing information to fill, they should be the first.

Notice: I have written a more extensive text about this topic on my database focused website SqlForDevs.com: Placement of NULL values for ORDER BY with nullable columns

Deterministic Ordering for Pagination

```sql
SELECT *
FROM users
ORDER BY firstname ASC, lastname ASC, user_id ASC
LIMIT 20 OFFSET 60;
```

Every tutorial and explanation of pagination is wrong! The most simplistic pagination approach with a `LIMIT` and `OFFSET` keyword to skip some rows has severe problems in real applications. When multiple rows have the same values, their ordering is not guaranteed, e.g. numerous contacts with the same first and last names. Due to concurrent modifications on a table between two SQL queries or different execution strategies the sorting results may differ both times. With some bad luck, a row will be displayed as the last result on the current page and the first on the next page. You should always make the ordering deterministic by adding more columns so every row is unique, adding the primary key is the most straightforward approach.

More Efficient Pagination Than LIMIT OFFSET

```sql
-- MySQL, PostgreSQL
SELECT *
FROM users
WHERE (firstname, lastname, id) > ('John', 'Doe', 3150)
ORDER BY firstname ASC, lastname ASC, user_id ASC
LIMIT 30
```

While the former tip is a step in the right direction, it is still wrong. When rows are inserted or deleted concurrently, the page offset will be wrong as it is not reflecting any changes to the table. Some contacts will be shown again or never when rows have been deleted and the offset is too big or too small. The solution to these problems is an approach named keyset pagination. The values of the last item on the current page are passed to get the following page results. It's also much faster than `LIMIT OFFSET` pagination for high page numbers but jumping to arbitrary pages is not supported.

Database-Backed Locks With Safety Guarantees

```sql
START TRANSACTION;

SELECT balance FROM account WHERE account_id = 7 FOR UPDATE;

-- Race condition free update after processing the data
UPDATE account SET balance = 540 WHERE account_id = 7;

COMMIT;
```

Almost any application I have seen is prone to race conditions: A value is selected from the database, the application processes some data and the value is updated with a new one. But the application is not protected against any updates happening during the calculation. Some critical parts are protected with locking solutions against race conditions. However, locking algorithms are hard to build as crashing application logic could miss removing a lock at the end. And automatic time-based lock releasing approaches to solve this problem will still hold the lock far too long.

When you run any data modification query (e.g. `UPDATE`), the database is on its own locking any affected row until the end of the transaction to guarantee data correctness. Instead of using a locking approach in your application, you can collaborate with the database by using `FOR UPDATE` on your `SELECT` queries to lock the read data against race conditions. These locks are automatically released when the transaction finishes or the client disconnects.

Warning: Any application logic updating a value has to use locking otherwise race conditions will end in unwanted results. And you have to do it everywhere, just at a tiny part is not solving the race condition problem.

Notice: I have written a more extensive text about this topic on my database focused website SqlForDevs.com: Transactional Locking to Prevent Race Conditions

Refinement Of Data With Common Table Expressions

```sql
WITH most_popular_products AS (
  SELECT products.*, COUNT(*) as sales
  FROM products
  JOIN users_orders_products USING(product_id)
  JOIN users_orders USING(order_id)
  WHERE users_orders.created_at BETWEEN '2022-01-01' AND '2022-06-30'
  GROUP BY products.product_id
  ORDER BY COUNT(*) DESC
  LIMIT 10
), applicable_users (
  SELECT DISTINCT users.*
  FROM users
  JOIN users_raffle USING(user_id)
  WHERE users_raffle.correct_answers > 8
), applicable_users_bought_most_popular_product AS (
  SELECT applicable_users.user_id, most_popular_products.product_id
  FROM applicable_users
  JOIN users_orders USING(order_id)
  JOIN users_orders_products USING(product_id)
  JOIN most_popular_products USING(product_id)
) raffle AS (
  SELECT product_id, user_id, RANK() OVER(
    PARTITION BY product_id
    ORDER BY RANDOM()
  ) AS winner_order
  FROM applicable_users_bought_most_popular_product
)
SELECT product_id, user_id FROM raffle WHERE winner_order = 1;
```

When you have to get rows from the database with complex rules that are hard to do in one sinle query, you can split it by using Common Table Expressions (CTE). With every step, you can refine the data and use it (even multiple times) in later refinements to finally get the desired results. Instead of traditional approaches like e.g. many nested sub-queries or dozens of joins, a CTE is easier to read and you can debug the iterative steps in isolation. For performance, both approaches are kind of the same. The database may transform it internally to a nested sub-query or find a more efficient approach by caching single steps used multiple times.

First Row Of Many Similar Ones

```sql
-- PostgreSQL
SELECT DISTINCT ON (customer_id) *
FROM orders
WHERE EXTRACT (YEAR FROM created_at) = 2022
ORDER BY customer_id ASC, price DESC;
```

Sometimes you have numerous rows, and you only want one for e.g. every customer. You can stick to a for-each-loop-like lateral join as described before or use PostgreSQL's `DISTINCT ON` invention. A standard `DISTINCT` query will filter rows with exact matches on all columns of a row. But with the showcased feature, you can specify a subset of columns to make distinct and only the first matching row after the sort will be kept.

> **Notice:** This feature is only available for PostgreSQL.

Multiple Aggregates In One Query

```
-- MySQL
SELECT
  SUM(released_at = 2001) AS released_2001,
  SUM(released_at = 2002) AS released_2002,
  SUM(director = 'Steven Spielberg') AS director_stevenspielberg,
  SUM(director = 'James Cameron') AS director_jamescameron
FROM movies
WHERE streamingservice = 'Netflix';

-- PostgreSQL
SELECT
  COUNT(*) FILTER (WHERE released_at = 2001) AS released_2001,
  COUNT(*) FILTER (WHERE released_at = 2002) AS released_2002,
  COUNT(*) FILTER (WHERE director = 'Steven Spielberg') AS
director_stevenspielberg,
  COUNT(*) FILTER (WHERE director = 'James Cameron') AS
director_jamescameron
FROM movies
WHERE streamingservice = 'Netflix';
```

In some cases, you need to calculate multiple different statistics. Instead of executing numerous queries, you can write one which will collect all the information in one single pass through the data. Depending on your data and indexes this could speed up or slow down your execution time. You should definitely test it on your application.

Notice: I have written a more extensive text about this topic on my database focused website SqlForDevs.com: Multiple Aggregates in One Query

Limit Rows Also Including Ties

```
-- PostgreSQL
SELECT *
FROM teams
ORDER BY winning_games DESC
FETCH FIRST 3 ROWS WITH TIES;
```

Imagine you want to rank the teams of a sports league and show the top three ones. In rare cases, at least 2 teams will have the same amount of winning games at the end of the season. If they are both on 3rd place you may want to expand your limit to include both of them. The `WITH TIES` option is doing precisely that. Whenever some rows would be excluded despite having the same values as those included, they are included too although the limit is exceeded.

> **Notice:** This feature is only available for PostgreSQL.

Fast Row Count Estimates

```sql
-- MySQL
EXPLAIN FORMAT=TREE SELECT * FROM movies WHERE rating = 'NC-17' AND
price < 4.99;

-- PostgreSQL
EXPLAIN SELECT * FROM movies WHERE rating = 'NC-17' AND price < 4.99;
```

Showing the number of matching rows is a crucial feature for most applications, but it is sometimes hard to implement for large databases. The larger a database is, the slower counting the number of rows will be. The query will be very slow when no index exists to help calculate the count. But even an existing index will not make counting hundreds of thousands of index fast. However, an approximate count of rows may be good enough for some use cases. The database's query planner always calculates an approximate row count for a query that can be extracted by asking the database for the execution plan.

Date-Based Statistical Queries With Gap-Filling

```sql
-- MySQL
SET cte_max_recursion_depth = 4294967295;
WITH RECURSIVE dates_without_gaps(day) AS (
  SELECT DATE_SUB(CURRENT_DATE, INTERVAL 14 DAY) as day
  UNION ALL
  SELECT DATE_ADD(day, INTERVAL 1 DAY) as day
  FROM dates_without_gaps
  WHERE day < CURRENT_DATE
)
SELECT dates_without_gaps.day, COALESCE(SUM(statistics.count), 0)
FROM dates_without_gaps
LEFT JOIN statistics ON(statistics.day = dates_without_gaps.day)
GROUP BY dates_without_gaps.day;

-- PostgreSQL
SELECT dates_without_gaps.day, COALESCE(SUM(statistics.count), 0)
FROM generate_series(
  CURRENT_DATE - INTERVAL '14 days',
  CURRENT_DATE,
  '1 day'
) as dates_without_gaps(day)
LEFT JOIN statistics ON(statistics.day = dates_without_gaps.day)
GROUP BY dates_without_gaps.day;
```

The results for some statistical calculations will have gaps because no information was saved for specific days. But instead of back-filling these holes with application code, the database query can be restructured: A sequence of gapless values is created as source for joining to the statistical data. For PostgreSQL the generate_series function might be used to create the sequence, whereas for MySQL the same needs to be performed manually using a recursive common table expression (CTE).

Notice: I have written a more extensive text about this topic on my database focused website SqlForDevs.com: Fill Gaps in Statistical Time Series Results

Table Joins With A For-Each Loop

```
-- MySQL, PostgreSQL
SELECT customers.*, recent_sales.*
FROM customers
LEFT JOIN LATERAL (
  SELECT *
  FROM sales
  WHERE sales.customer_id = customers.id
  ORDER BY created_at DESC
  LIMIT 3
) AS recent_sales ON true;
```

When joining tables, the rows of both tables are linked together based on some conditions. However, the joining condition can only include all matching rows of the different table. It is impossible to control the number of rows for every iteration of the join to e.g. limit the bought products for every customer to just the last three ones.

The special lateral join type combines a join and a subquery. A subquery will be executed for every row of the join's source table. Within that subquery, you can e.g. select only the last three bought products of a customer. And as you already selected only matching sales for every customer, a special `true` join condition indicates that all rows will be used. You can now make for-each loops within your database. You've learned the holy grail of SQL!

Notice: I have written a more extensive text about this topic on my database focused website SqlForDevs.com: For each loops with LATERAL Joins

Schema

The schema is probably the most crucial part of your database. The more complex your schema is, the slower new developers will be able to work on your application. But it also provides the possibility to go new ways and make them more straightforward by using modern database features. Actually, many of those features can offload a lot of custom application logic to the database and make development faster.

The schema chapter will show you how e.g. JSON documents can replace many tables, data can be saved for faster querying or a simpler approach for storing trees.

Rows Without Overlapping Dates

```sql
CREATE TABLE bookings (
  room_number int,
  reservation tstzrange,
  EXCLUDE USING gist (room_number WITH =, reservation WITH &&)
);

INSERT INTO meeting_rooms (
    room_number, reservation
) VALUES (
  5, '[2022-08-20 16:00:00+00,2022-08-20 17:30:00+00]',
  5, '[2022-08-20 17:30:00+00,2022-08-20 19:00:00+00]',
);
```

Preventing e.g. multiple concurrent reservations for a meeting room is a complicated task because of race conditions. Without pessimistic locking by the application or careful planning, simultaneous requests can create room reservations for the exact timeframe or overlapping ones. The work can be offloaded to the database with an exclusion constraint that will prevent any overlapping ranges for the same room number. This safety feature is available for integer, numeric, date and timestamp ranges.

> **Notice:** This feature is only available for PostgreSQL.

Store Trees As Materialized Paths

```sql
-- MySQL
CREATE TABLE tree (path varchar(255));
INSERT INTO tree (path) VALUES ('Food');
INSERT INTO tree (path) VALUES ('Food.Fruit');
INSERT INTO tree (path) VALUES ('Food.Fruit.Cherry');
INSERT INTO tree (path) VALUES ('Food.Fruit.Banana');
INSERT INTO tree (path) VALUES ('Food.Meat');
INSERT INTO tree (path) VALUES ('Food.Meat.Beaf');
INSERT INTO tree (path) VALUES ('Food.Meat.Pork');
SELECT * FROM tree WHERE path like 'Food.Fruit.%';
SELECT * FROM tree WHERE path IN('Food', 'Food.Fruit');

-- PostgreSQL
CREATE EXTENSION ltree;
CREATE TABLE tree (path ltree);
INSERT INTO tree (path) VALUES ('Food');
INSERT INTO tree (path) VALUES ('Food.Fruit');
INSERT INTO tree (path) VALUES ('Food.Fruit.Cherry');
INSERT INTO tree (path) VALUES ('Food.Fruit.Banana');
INSERT INTO tree (path) VALUES ('Food.Meat');
INSERT INTO tree (path) VALUES ('Food.Meat.Beaf');
INSERT INTO tree (path) VALUES ('Food.Meat.Pork');
SELECT * FROM tree WHERE path ~ 'Food.Fruit.*{1,}';
SELECT * FROM tree WHERE path @> subpath('Food.Fruit.Banana', 0, -1);
```

You can use the lesser-known materialized path approach in addition to the widely known nested set and adjacency list approaches for storing trees. Every row stores the materialized path within the tree to itself, making queries for tree searching relatively easy. With PostgreSQL you'll get a wide range of querying and manipulation functionality provided by the label tree extension. While for MySQL you'll have to do use simple text searching functionalities.

JSON Columns to Combine NoSQL and Relational Databases

```sql
-- MySQL
CREATE TABLE books (
  id bigint PRIMARY KEY,
  author_id bigint NOT NULL,
  category_id bigint NOT NULL,
  name varchar(255) NOT NULL,
  price numeric(15, 2) NOT NULL,
  attributes json NOT NULL DEFAULT '{}'
);

-- PostgreSQL
CREATE TABLE books (
  id bigint PRIMARY KEY,
  author_id bigint NOT NULL,
  category_id bigint NOT NULL,
  name text NOT NULL,
  price numeric(15, 2) NOT NULL,
  attributes jsonb NOT NULL DEFAULT '{}'
);
```

Many database schemas can be simplified by copying ideas from NoSQL databases. The querying and data modification logic will be a lot easier by e.g. avoiding a lot of joins or complex architectures like the Entity–Attribute–Value approach (EAV). However, you should still continue storing most your data in a standard relational schema. You can use JSON columns to simply the schema following these rules:

- Move seldom-used data (e.g. joined from other tables) into JSON arrays and objects for easier querying.
- Think thoroughly whether you store references to other tables in JSON documents as you can't enforce foreign-key relationships. You need a good reason to do so.
- Never used deeply nested collections. Any modifications and queries to those documents will be a mess.

Notice: I have already written a more extensive text about this topic on SqlForDevs.com: JSON columns

Alternative Tag Storage With JSON Arrays

```
-- MySQL
CREATE TABLE products (
  id bigint,
  name varchar(255),
  tagids json
);
CREATE INDEX producttags ON products ((CAST(tagids as unsigned ARRAY)));

SELECT *
FROM products
WHERE JSON_ARRAY(3, 8) MEMBER OF(tagids) AND NOT(12 MEMBER OF(tagids));

-- PostgreSQL
CREATE TABLE products (
  id bigint,
  name text,
  tagids jsonb
);
CREATE INDEX producttags ON products (tags jsonb_path_ops);

SELECT *
FROM products
WHERE tagids @> '[3,8]' AND NOT(tagids @> '[12]');
```

Do you remember the last tip about storing references to other tables has to be a well-formed decision? It's a great way to simplify typical tagging schemas with m:n tables absolutely worth the missing foreign keys. If you store the primary keys of a tags table in a JSON array, you can do efficient lookups for products having and not having specific tags. A traditional approach would need six joins, three times to the m:n table and three times to the tags table, while the JSON approach is only a simple condition. It's very effective to skip doing many complex joins, especially for filter-based queries that use many tags.

Constraints for Improved Data Strictness

```sql
ALTER TABLE reservations
ADD constraint start_before_end CHECK (checkin_at < checkout_at);

ALTER TABLE invoices
ADD constraint eu_vat CHECK (
  NOT(is_europeanunion) OR vatid IS NOT NULL
);
```

Some column values or their existence will depend on other columns. These dependencies are validated by your application but can also be checked by constraints in your database as the last frontier to ensure you always have valid data. You could argue that validating at the application is enough, but when you batch-update rows or modify the database manually your application checks won't be executed.

Validation Of JSON Colums Against A Schema

```sql
-- MySQL
ALTER TABLE products ADD CONSTRAINT CHECK(
  JSON_SCHEMA_VALID(
    '{
      "$schema": "http://json-schema.org/draft-04/schema#",
      "type": "object",
      "properties": {
        "tags": {
          "type": "array",
          "items": { "type": "string" }
        }
      },
      "additionalProperties": false
    }',
    attributes
  )
);

ALTER TABLE products ADD CONSTRAINT data_is_valid CHECK(
  validate_json_schema(
    '{
      "type": "object",
      "properties": {
        "tags": {
          "type": "array",
          "items": { "type": "string" }
        }
      },
      "additionalProperties": false
    }',
    attributes
  )
);
```

With JSON columns you are trading your data strictness guarantees for easier querying and modifications. Anything could be stored in that JSON documents and you have to document the expected values strictly for other developers. But when you use constraints to validate the data against a JSON schema you regain data strictness check and get automatic type documentation.

Notice: I have already written a more extensive text about this topic on SqlForDevs.com: JSON Schema validation for columns

UUID Keys Against Enumeration Attacks

```
-- MySQL
ALTER TABLE users ADD COLUMN uuid char(36);
UPDATE users SET uuid = (SELECT uuid_v4());
ALTER TABLE users CHANGE COLUMN uuid uuid char(36) NOT NULL;
CREATE UNIQUE INDEX users_uuid ON users (uuid);

-- PostgreSQL
ALTER TABLE users ADD COLUMN uuid uuid NOT NULL DEFAULT
gen_random_uuid();
CREATE UNIQUE INDEX users_uuid ON users (uuid);
```

Using your incrementing primary key in URLs exposes your data to misbehaving adversaries. They can increment your id one by one to crawl all your public data or by monitoring those values calculate your application's growth and popularity. A straightforward and effective countermeasure is adding a random UUID v4 to every row that will be used in URLs instead of exposing your real incrementing id. You can also easily add this UUID to an existing table, as shown.

Notice: I have already written a more extensive text about this topic on SqlForDevs.com: UUIDs to prevent Enumeration Attacks

Fast Delete Of Big Data With Partitions

```sql
ALTER TABLE logs DROP PARTITION logs_2022_january;
```

Deleting a big chunk of data from a table, e.g. historical rows, will take a lot of time. You can optimize it by executing delete queries in many small batches, nevertheless you are just stretching the amount of work over a longer timeframe. When you partition your big table into smaller parts, you can delete one or several partitions within seconds.

Pre-Sorted Tables For Faster Access

```sql
SELECT *
FROM product_comments
WHERE product_id = 2
ORDER BY comment_id ASC
LIMIT 10

-- MySQL
CREATE TABLE product_comments (
  product_id bigint,
  comment_id bigint auto_increment UNIQUE KEY,
  message text,
  PRIMARY KEY (product_id, comment_id)
);

-- PostgreSQL
CREATE TABLE product_comments (
  product_id bigint,
  comment_id bigint GENERATED ALWAYS AS IDENTITY,
  message text,
  PRIMARY KEY (product_id, comment_id)
);
CLUSTER product_comments USING product_comments_pkey;
```

Every data you insert into a table will be physically sorted in the database's file so that common tasks like selecting and updating rows are most efficient. But the database can't know how you will use those rows. When building an e-commerce application, you want to get some comments for a product. Typically, these comments are stored by an incrementing primary key and are distributed through the table by their insertion order. But you can enforce that these rows are stored physically in ascending order by the product (product_id) and date of the comment (comment_id). The database can now efficiently find the first comment and the next nine in the following rows instead of collecting them at 10 different locations. Whether you use an SSD or HDD, random access to multiple locations will always be slower than a single operation fetching multiple consecutive bytes.

This is the most exciting and most complex performance optimization I can teach you for big data. The article below shares much more information and implementation obstacles.

> **Notice:** I have already written a more extensive text about this topic on SqlForDevs.com: Sorted Tables for Faster Range-Scans

Pre-Aggregation of Values for Faster Queries

```sql
SELECT SUM(likes_count)
FROM articles
WHERE user_id = 1 and publish_year = 2022;
```

Even if your schema is very well-designed and your queries all use a perfect index, they may still be slow. When analytical queries, e.g. for a dashboard, have to aggregate tens or hundreds of thousands of rows the performance will suffer drastically. Such queries are constrained by computational limits about how fast data can be loaded and the required time to extract the information from the rows or indexes and aggregate them. This operation is very fast for small amounts of data, but the bigger it gets, the more you should look into storing pre-aggregated values. No intelligent indexing will beat the performance improvement of not having to aggregate tens of thousands of values.

Indexes

Without indexes your application would be slow as every operation would have to scan the whole table. Consequently, indexes are the most interesting topic for developers but also the most complicated one. A lot of content has been written about database indexing that I don't want to repeat. Therefore, I am only sharing more extraordinary approaches and features you may not have seen before.

The indexing chapter will show you a lot of exceptional indexing approaches like uniqueness constraints for soft-deleted tables, simple rules for multi-column indexing, ways to find and delete unused indexes and much more.

Indexes On Functions And Expressions

```sql
SELECT * FROM users WHERE lower(email) = 'test@example.com';

-- MySQL
CREATE INDEX users_email_lower ON users ((lower(email)));

-- PostgreSQL
CREATE INDEX users_email_lower ON users (lower(email));
```

Most developers are puzzled that their index on a column is not used when it is transformed by a function or expression. A Google search results in countless StackOverflow articles stating that you can't use an index in these cases, but this information is wrong! You can create specialized indexes on a function or expression that are used whenever the exact same transformation is applied in your `WHERE`.

> **Notice:** I have written a more extensive text about this topic on my database focused website SqlForDevs.com: Function-Based Indexes

Find Unused Indexes

```sql
-- MySQL
SELECT
  object_schema AS `database`,
  object_name AS `table`,
  index_name AS `index`,
  count_star as `io_operations`
FROM performance_schema.table_io_waits_summary_by_index_usage
WHERE object_schema NOT IN('mysql', 'performance_schema') AND index_name
IS NOT NULL AND index_name != 'PRIMARY'
ORDER BY object_schema, object_name, index_name;

-- PostgreSQL
SELECT
  pg_tables.schemaname AS schema,
  pg_tables.tablename AS table,
  pg_stat_all_indexes.indexrelname AS index,
  pg_stat_all_indexes.idx_scan AS number_of_scans,
  pg_stat_all_indexes.idx_tup_read AS tuples_read,
  pg_stat_all_indexes.idx_tup_fetch AS tuples_fetched
FROM pg_tables
LEFT JOIN pg_class ON(pg_tables.tablename = pg_class.relname)
LEFT JOIN pg_index ON(pg_class.oid = pg_index.indrelid)
LEFT JOIN pg_stat_all_indexes USING(indexrelid)
WHERE pg_tables.schemaname NOT IN ('pg_catalog', 'information_schema')
ORDER BY pg_tables.schemaname, pg_tables.tablename;
```

After several changes to a schema based on functional or business requirements and changed queries the former indexes may not fit anymore. You probably have added many new missing ones but never deleted an existing index. As every insert needs to update all the indexes you are wasting time updating no longer needed ones. The database keeps statistics about index usage that you can compare over time to get help finding unused indexes you can then delete.

Warning: Some index accesses may not update the statistics. You should evaluate these results carefully and not blindly trust them.

Safely Deleting Unused Indexes

```
-- MySQL
ALTER TABLE website_visits ALTER INDEX twitter_referrals INVISIBLE;
ALTER TABLE website_visits ALTER INDEX twitter_referrals VISIBLE;
```

Deleting an unused index in the schema is always a nerve-stretching task. If you are wrong, some queries will be very slow until you've been notified about a slowdown at 3am and the index has been recreated. And for huge tables creating the index can take more than 30 minutes or many hours. In the end, you choose the safe approach and keep the index as no one wants to be paged at night. But instead of deleting it, you can make it invisible so it's not used anymore. If it is not needed anymore you can safely delete it or make it visible again within a second. It's a much safer approach. Although you may still get woken up at 3am when your guess about its importance was wrong, but it will be much easier to resolve the issue now...

> **Notice:** This feature is only available for MySQL.

Index-Only Operations By Including More Columns

```sql
SELECT SUM(price) FROM invoices WHERE customer_id = 42 AND year = 2022;

-- MySQL
CREATE INDEX ON invoices (customer_id, year, price);

-- PostgreSQL
CREATE INDEX ON invoices (customer_id, year) INCLUDE (price);
```

The functionality of an index extends from fast lookups of rows to more performance optimization the database can do. When an index contains all columns for the row filtering conditions and selected columns of the query, it doesn't have to look up any row anymore. The complete query is executed just by information from the index. It's called an index-only query and is the most performant query you can get. With PostgreSQL you can use the INCLUDES option to include additional columns, while on MySQL you have to add them to the index. Therefore, with MySQL you can't use this tip for unique indexes as the uniqueness condition would be changed by the additional columns.

Partial Indexes To Reduce Index Size

```sql
SELECT * FROM invoices WHERE specialcase_law3421 = TRUE;

-- PostgreSQL
CREATE INDEX invoices_specialcase_law3421
ON invoices (specialcase_law3421)
WHERE specialcase_law3421 = TRUE;
```

A barely known feature is partial indexes that constrain the index's rows because many databases do not support it. Instead of a standard index with an entry for every row, you can specify which ones should be included. It is noteworthy for use cases like the invoicing example: Only a handful of invoices need to be handled differently because law #3421 demands a new formula to calculate taxes. A standard index on the column would have to include thousands of entries for invoices not affected by the law, compared to just the few hundred affected ones. Although the index size is much smaller, it is still used for queries using the partial index condition in their `WHERE`. Furthermore, unaffected rows don't have to update the partial index which decreases insertion time.

> **Notice:** This feature is only available for PostgreSQL.

Partial Indexes For Uniqueness Constraints

```sql
-- MySQL
CREATE UNIQUE INDEX email_unique
ON users (email, (IF(deleted_at, NULL, 1)));

-- PostgreSQL
CREATE UNIQUE INDEX email_unique
ON users (email)
WHERE deleted_at IS NULL;
```

In some cases, you want e.g. the email address of users to be unique, but you are also using soft-deletes. Any email address used by a deleted user (deleted_at is no longer null) would result in an error as the email address exists multiple times. With partial indexes, you can constrain the rows of the unique index to just the non-deleted users. There aren't any duplicates anymore.

For MySQL, which doesn't support partial indexes, you can emulate the behavior. A unique index can have many entries with the same information when at least one null value is present. Therefore the values need to be transformed for the index:

- When a contact is deleted, its deleted_at timestamp in the index is replaced by a NULL value to allow the same email address multiple times.
- When a contact is not deleted, a static value is used instead of the NULL value to enforce unique violations on e.g. the value `(info@example.com, 1)` for multiple email addresses.

> **Notice:** I have written a more extensive text about this topic on my database focused website SqlForDevs.com: Unique Indexes With Some Rows Excluded

Index Support For Wildcard Searches

```sql
SELECT * FROM speakers WHERE name LIKE '%Tobias%';

-- PostgreSQL
CREATE EXTENSION pg_trgm;
CREATE INDEX trgm_idx ON speakers USING GIN (name gin_trgm_ops);
```

Every database can use an index for trailing wildcard searches. But when you use PostgreSQL, you can also do leading wildcard searches while still utilizing an index: Your column's text is split into many three characters long sequences (trigrams) that will be used when searching. A wildcard term like %Tobias% will search for values having the trigrams Tob, obi, bia and ias with full index support. All found rows are then filtered in a second step to check whether they really contain the substring Tobias because those trigrams could also be used in a different combination to form another name.

> **Notice:** This feature is only available for PostgreSQL.

Rules For Multi-Column Indexes

```sql
SELECT *
FROM shop_articles
WHERE tenant_id = 6382 AND category = 'books' AND price < 49.99;

CREATE INDEX shop_articles_key ON shop_articles (
  tenant_id, -- type = equality, different_values = 7293
  category, -- type = equality, different_values = 628
  price -- type = range, different_values = 142
);
```

Simplifying the complex rules for multi-column index ordering into a simple tip is complicated. I could write a complete book on this topic. Would you be interested? You can cover 80% of the multi-column indexing requirements with three simple rules. The basic idea is to reduce the number of possible table rows with every added column the most. This is called selectivity.

- It's best to start with a column that is compared with equality checks and has the most distinct values. This way, you reduce the number of remaining rows very fast.
- You then follow with more equality check columns in decreasing number of different values.
- Range columns, like dates or numbers, are often used best at the end.

Warning: This is a simple but still very accurate set of rules for multi-column indexes. They won't work for every use case, but I believe they work 80% of the time. Don't pin me down on these rules; skip them whenever they don't fit your query.

Hash Indexes To Descrease Index Size

```
-- PostgreSQL
CREATE INDEX invoices_uniqid ON invoices USING HASH (uniqid)
```

When you plan to search for a column only with equality checks like UUIDs or strings, you can optimize your indexes by using hash indexes. Compared to a standard b-tree index, a hash index will be a tiny bit faster for inserting and querying data. But the index will be much smaller, which is a significant improvement. However, unique hash indexes are not yet supported so you can not use them to enforce uniqueness constraints.

Warning: Hash indexes had been discouraged before PostgreSQL 10. If you are still using an outdated version, you shouldn't use them or upgrade your database.

Notice: This feature is only available for PostgreSQL.

Descending Indexes For Order By

```sql
SELECT *
FROM highscores
ORDER BY score DESC, created_at ASC
LIMIT 10;

CREATE INDEX highscores_correct ON highscores (
  score DESC, created_at ASC
);
```

Creating multi-column indexes to speed up queries with sorting is complicated. In most cases, your index will be used and you don't have to do anything special. Whether you use ascending or descending order in your query won't make any difference. But mixed orders in a query are more complicated because the data will still be sorted although your index has been used. To skip this unnecessary operation, you must create the index with the same sorting order as your query.

> **Notice:** I have written a more extensive text about this topic on my database focused website SqlForDevs.com: Descending Indexes

Ghost Conditions Against Unindexed Columns

```sql
-- Before
SELECT *
FROM shipments
WHERE status = 'open' AND transportinsurance = 1;

-- After
SELECT *
FROM shipments
WHERE status = 'open' AND transportinsurance = 1 AND type = IN(3, 6,
11);
```

Creating the correct indexes for e.g. user-defined data filtering or seldom used conditions is one of the most challenging tasks. You can't make an index for every column and must guess the most important ones. But you will still miss some resulting in slow queries. An excellent approach is adding "ghost conditions" to a query that are index-supported and don't change the results. These conditions are only added to help the database find the most efficient way of querying the data.

In the example, we are searching for open shipments with transport insurance. This exact query has no matching multi-column index but you know of some business rules the database couldn't know. Only packages of a specific type are allowed to be shipped with transport insurance, so these types are added to the query. The multi-column index `(status, type)` can now be used to filter the data more than the single index `(status)` could have done.