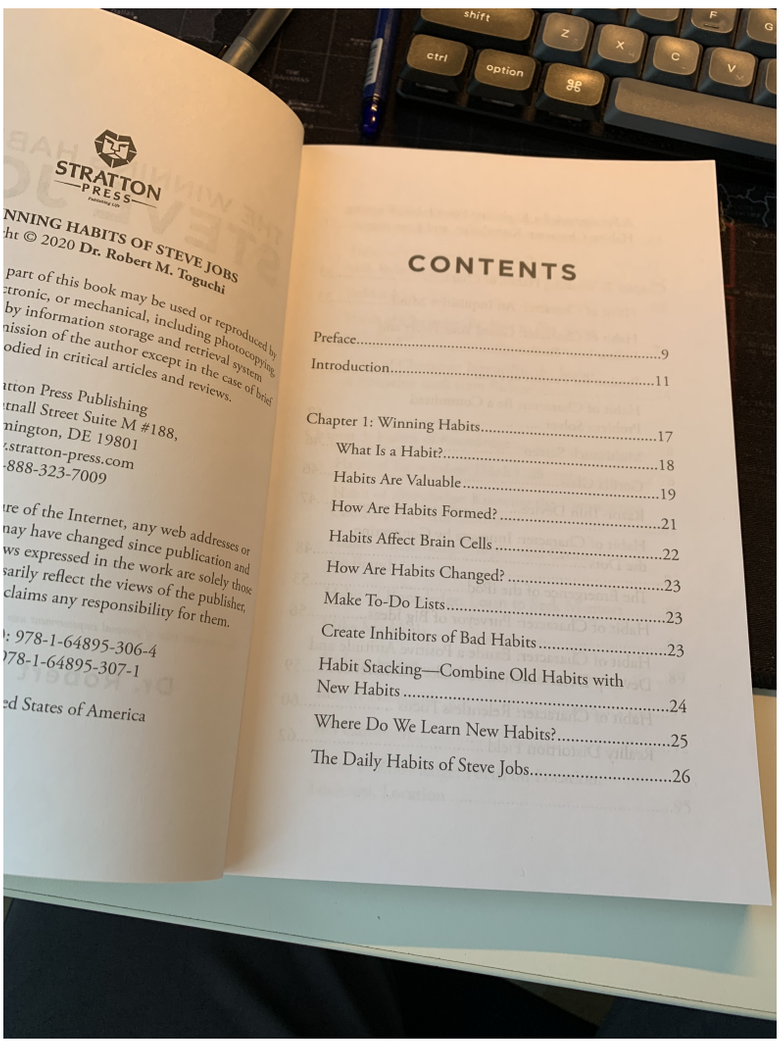
Database Optimization

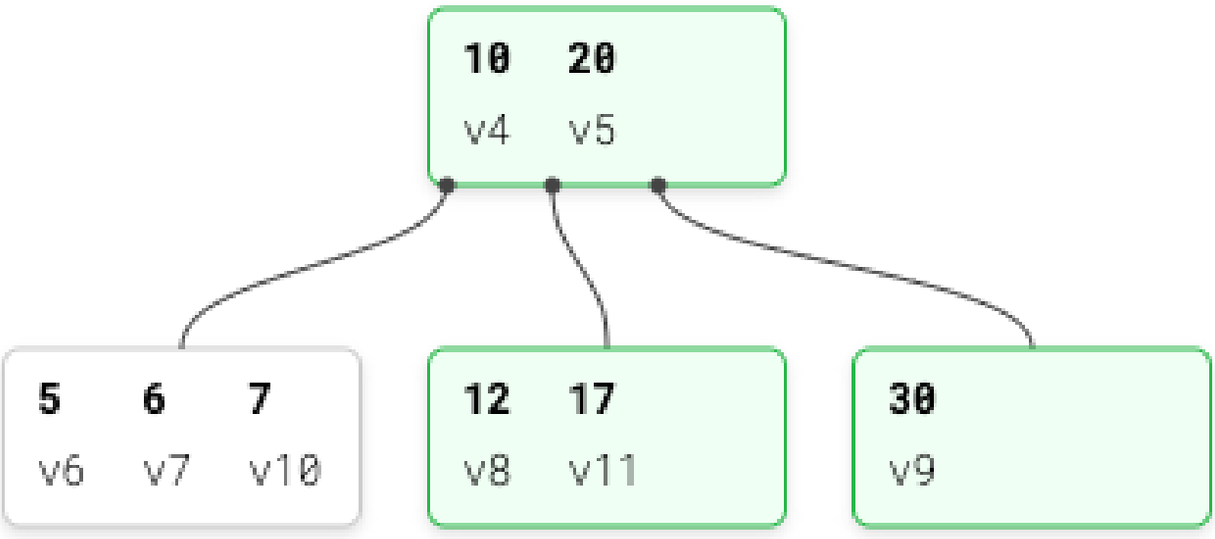
1. Use Indexes Strategically

Indexes allow MySQL to locate data without scanning entire tables, drastically speeding up queries



B-Tree

A B-tree is a self-balancing tree data structure that maintains sorted data and allows efficient insertion, deletion, and search operations.



https://btree.app

Let's visualize the B-Tree structure

https://btree.app

10, 20, 5, 6, 12, 30, 7, 17

How do we identify where to add indexes?

1. Look at WHERE filters

```
SELECT *
FROM orders
WHERE user_id = 1;
```

users

id	user_name	email
1	Alice	alice@example.com
2	Bob	bob@example.com
3	Charlie	charlie@example.com

orders

id	user_id	order_date
1	1	2024-02-15
2	2	2024-03-05
3	2	2024-12-25

How do we identify where to add indexes?

2. Look at JOIN conditions

```
SELECT
o.id,
o.order_date,
u.user_name,
u.email
FROM orders o
JOIN users u ON o.user_id = u.id;
```

users

id	user_name	email
1	Alice	alice@example.com
2	Bob	bob@example.com
3	Charlie	charlie@example.com

orders

id	user_id	order_date
1	1	2024-02-15
2	2	2024-03-05
3	2	2024-12-25

How do we identify where to add indexes?

3. Look at ORDER BY clauses

```
SELECT order_id, user_id, order_date
FROM orders
ORDER BY order_date DESC;
```

users

id	user_name	email
1	Alice	alice@example.com
2	Bob	bob@example.com
3	Charlie	charlie@example.com

orders

id	user_id	order_date
1	1	2024-02-15
2	2	2024-03-05
3	2	2024-12-25

Composite indexes

A composite index is an index that includes more than one column from a table. It enables the database engine to efficiently execute queries that filter or sort based on multiple columns.

```
SELECT * FROM orders
WHERE user_id = 2
AND order_date >= '2023-04-10';
```

users

id	user_name	email
1	Alice	alice@example.com
2	Bob	bob@example.com
3	Charlie	charlie@example.com

orders

id	user_id	order_date
1	1	2024-02-15
2	2	2024-03-05
3	2	2024-12-25

Let's create few indexes...

2. Optimize Queries and Schema Design

Indexes allow MySQL to locate data without scanning entire tables, drastically speeding up queries

1. Avoid SELECT * (select only needed columns) to reduce I/O and network load .

2. Use proper sargable conditions.
Instead of WHERE YEAR(order_date)=2023 (which hides the indexed column in a function), rewrite the query to WHERE order_date BETWEEN '2023-01-01' AND '2023-12-31'

3. Leverage Caching.

4. Monitor and Tune Queries with EXPLAIN.

3. Use Limit and Batching

4. OFFSET-based pagination vs CURSOR-based pagination

OFFSET-based pagination

```
SELECT * FROM film
ORDER BY id
LIMIT 10 OFFSET 30;
```

CURSOR(Keyset)-based pagination

```
SELECT * FROM movies
WHERE id > 1000
ORDER BY id
LIMIT 10;
```

CURSOR(Keyset)-based pagination (Multi-column)

```
SELECT * FROM rental
ORDER BY rental_date DESC,
rental_id DESC
LIMIT 10;
```

```
SELECT * FROM rental
WHERE
(rental_date < '2005-07-17 18:33:22') OR
(rental_date = '2005-07-17 18:33:22' AND
rental_id < 12034)
ORDER BY rental_date DESC, rental_id DESC
LIMIT 10;
```