Postgres VS Mongo: Performance Comparison for semi-structured data

Egor Tarasov · Follow
7 min read · Apr 28, 2024

Test	Mongo	Postgres
Index creation on an empty database	**152.9 ms**	402.0 ms
Batch insert	**53.50 ms**	219.15 ms
One by one insert	**319.3 ms**	641.9 ms
Read multiple records	**21.83 ms**	66.63 ms
Index creation on filled database	**1.563 s**	1.916 s
Aggregated query	174.51 ms	**60.43 ms**

The goal of this article is to compare the performance of MongoDB and PostgreSQL for data having a partially dynamic structure. We'll write data access code using C# and perform tests using Benchmarks.NET.

The Setup

Both databases will be deployed via a simple docker-compose file:

```
services:
  postgres:
    image: postgres
    environment:
      POSTGRES_PASSWORD: postgres
    ports:
      - "5432:5432"
  mongo:
    image: mongo
    ports:
      - "27017:27017"
```

Here's the test data model:

```
public record Expense(
    string Id,
    int Amount,
    Dictionary<string, string> Labels
);
```

Note that `Labels` can contain an arbitrary key. We'll generate test data using https://app.json-generator.com/. Here's the script:

```
JG.repeat(5000, {
  id: JG.objectId(),
  amount: JG.integer(1, 5000),
  labels: {
    userId: JG.integer(1, 1000).toString(),
    category: JG.random('food', 'transport', 'fun', 'home')
  }
});
```

Json Generator has a data size limit, but for our tests, it's better to have a much

greater table size. So for each insert iteration, we'll create a copy of each record with a modified ID to avoid duplication like that:

```
public static class TestData
{
    static readonly Expense[] raw = GetRaw();

    public static Expense[] GetRaw() {
        var dataJson = File.ReadAllText("data.json");
        return JsonSerializer.Deserialize<Expense[]>(dataJson, new JsonSerializ
            PropertyNameCaseInsensitive = true
        })!;
    }

    public static Expense[] ForIteration(int iteration, int? shift = 0, int? ta
        var shifted = raw.Select(d => d with { Id = d.Id + iteration + shift })
        var result = take.HasValue ? shifted.Take(take.Value) : shifted;
        return result.ToArray();
    }
}
```

We'll need to test querying capabilities using both the `userId` and `category` fields. Let's add code to `TestData` that provides us with random values for such queries:

```
static Random random = new(8899);
public static readonly string[] UserIds = RandomUserIds(20_000);
static readonly string[] categories = [ "food", "transport", "fun", "home" ];

public static string RandomCategory()
    => categories[random.Next(0, categories.Length)];

public static string[] RandomUserIds(int count)
    => Enumerable.Range(0, count).Select(i => random.Next(0, 1000).ToString()).
```

Now let's connect to the databases.

```
public static class MongoFactory {
    public static IMongoCollection<Expense> GetCollection() {
        var mongoClient = new MongoClient("mongodb://localhost:27017");
        var db = mongoClient.GetDatabase("performance-battle");
        return db.GetCollection<Expense>("expenses");
    }
}
```

The test should start clean, so the first thing we are going to do in `Program.cs` is preparation for a fresh start.

```
MongoFactory.GetCollection().Database.DropCollection("expenses");
```

PostgreSQL will need a little more setup. Although, EF Core is able to insert records with `Dictionary<string, string>` to Postgres as `hstore` it doesn't really provide any querying capabilities for those, therefore we'll need to use `jsonb` instead. Here's what it looks like:

```
public class Db : DbContext {
    public DbSet<ExpenseRecord> Expenses { get; set; } = null!;
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilde
        optionsBuilder.UseNpgsql("Host=localhost;Port=5432;Database=performance
    }
}

public class ExpenseRecord {
    public string Id { get; set; } = null!;
    public int Amount { get; set; }
    public JsonDocument Labels { get; set; } = null!;

    public static ExpenseRecord FromExpense(Expense expense) => new ExpenseReco
        Id = expense.Id,
        Amount = expense.Amount,
        Labels = JsonDocument.Parse(JsonSerializer.Serialize(expense.Labels))
```

```
    };
}
```

and for a fresh start in `Program.cs`:

```
new Db().Database.EnsureDeleted();
new Db().Database.EnsureCreated();
```

Tests

To correctly estimate write performance it's better to have indexes in place. So, our first test will show how long it takes to create an index on an empty database. Although the test doesn't show much I still find it interesting and we'll need to create an index anyway.

```
[SimpleJob(RunStrategy.Monitoring, iterationCount: 1)]
public class CreateUserIdIndex
{
    [Benchmark]
    public async Task Mongo() {
        var collection = MongoFactory.GetCollection();
        await collection.Indexes.CreateOneAsync(new CreateIndexModel<Expense>("
    }

    [Benchmark]
    public async Task Postgres() {
        using var db = new Db();
        await db.Database.ExecuteSqlRawAsync("""CREATE INDEX IF NOT EXISTS idx_
    }
}
```

```
Results:

╔═══════════╦═══════════╦═══════╗
║ Method    ║ Mean      ║ Error ║
╚═══════════╩═══════════╩═══════╝
```

```
| Mongo    | 152.9 ms | NA |
| Postgres | 402.0 ms | NA |
```

Now let's insert a million records (5000 records * 200 times) into our database and measure batch insert performance at the same time

```
[SimpleJob(RunStrategy.Monitoring, iterationCount: 200, id: "WriteBatches")]
public class WriteBatches
{
    private int iteration = 0;

     [Benchmark]
    public async Task Mongo() {
        var collection = MongoFactory.GetCollection();
        var actualData = TestData.ForIteration(iteration++);
        await collection.InsertManyAsync(actualData);
    }

    [Benchmark]
    public async Task Postgres() {
        using var db = new Db();
        db.Expenses.AddRange(TestData.ForIteration(iteration++).Select(ExpenseR
        await db.SaveChangesAsync();
    }
}
```

Results:

Method	Mean	Error	StdDev	Median
Mongo	53.50 ms	8.449 ms	35.77 ms	42.32 ms
Postgres	219.15 ms	15.864 ms	67.17 ms	203.79 ms

MongoDB significantly outperforms Postgres in the test. But I assume that in real life in most cases records will be inserted one by one other than in batch. Let's see

how our database engines will compare in this scenario

```
[SimpleJob(RunStrategy.Monitoring, iterationCount: 10, id: "WriteOneByOne")]
public class WriteOneByOne
{
    private int iteration = 0;

    [Benchmark]
    public async Task Mongo() {
        var collection = MongoFactory.GetCollection();
        foreach (var item in TestData.ForIteration(iteration++, shift: 5_000, t
            await collection.InsertOneAsync(item);
        }
    }

    [Benchmark]
    public async Task Postgres() {
        await using var db = new Db();
        foreach (var item in TestData.ForIteration(iteration++, shift: 5_000, t
            db.Expenses.Add(ExpenseRecord.FromExpense(item));
            await db.SaveChangesAsync();
        }
    }
}
```

```
Results:

╔═══════════╦═══════════╦═══════════╦═══════════╦═══════════╗
║ Method    ║ Mean      ║ Error     ║ StdDev    ║ Median    ║
╠═══════════╬═══════════╬═══════════╬═══════════╬═══════════╣
║ Mongo     ║ 319.3 ms  ║ 156.3 ms  ║ 103.4 ms  ║ 306.2 ms  ║
║ Postgres  ║ 641.9 ms  ║ 726.1 ms  ║ 480.3 ms  ║ 491.0 ms  ║
╚═══════════╩═══════════╩═══════════╩═══════════╩═══════════╝
```

Mongo still beats Postgres in the tests although not as radically as in the previous scenario. Now let’s query some data:

```
[SimpleJob(RunStrategy.Monitoring, iterationCount: 100, id: "ReadByUserId")]
```

```
public class ReadByUserId
{
    int iteration = 0;

    [Benchmark]
    public async Task Mongo() {
        var userId = TestData.UserIds[iteration++];
        var collection = MongoFactory.GetCollection();
        await collection
            .Find(e => e.Labels["userId"] == userId)
            .ToListAsync();
    }

    [Benchmark]
    public async Task Postgres() {
        var userId = TestData.UserIds[iteration++];
        using var db = new Db();
        await db.Expenses
            .AsNoTracking()
            .Where(e => e.Labels.RootElement.GetProperty("userId").GetString()
            .ToArrayAsync();
    }
}
```

Results:

Method	Mean	Error	StdDev	Median
Mongo	21.83 ms	12.40 ms	36.57 ms	17.37 ms
Postgres	66.63 ms	25.53 ms	75.29 ms	54.84 ms

Before our next read test let's check how hard it would be to create an index when we already have one million records:

```
[SimpleJob(RunStrategy.Monitoring, iterationCount: 1)]
public class CreateCategoryIndex {
    [Benchmark]
    public async Task Mongo() {
        var collection = MongoFactory.GetCollection();
```

```
        await collection.Indexes.CreateOneAsync(new CreateIndexModel<Expense>("
    }

    [Benchmark]
    public async Task Postgres() {
        using var db = new Db();
        await db.Database.ExecuteSqlRawAsync("""CREATE INDEX IF NOT EXISTS idx_
    }
}
```

Results:

Method	Mean	Error
Mongo	1.563 s	NA
Postgres	1.916 s	NA

The results are not that different, hence not really interesting. Now let's get back to our final competition with a slightly more complicated query and then summarize the results.

```
[SimpleJob(RunStrategy.Monitoring, iterationCount: 100, id: "SumByCategory")]
public class SumByCategory
{
    [Benchmark]
    public async Task<int> Mongo() {
        var collection = MongoFactory.GetCollection();
        var aggregateRecord = await collection.Aggregate()
            .Match(e => e.Labels["category"] == TestData.RandomCategory())
            .Group(e => 1, g => new { Sum = g.Sum(e => e.Amount) })
            .FirstAsync();

        return aggregateRecord.Sum;
    }

    [Benchmark]
    public async Task<int> Postgres() {
        using var db = new Db();
```

```
            return await db.Expenses
                .Where(e => e.Labels.RootElement.GetProperty("category").GetString(
                .SumAsync(e => e.Amount);
        }
    }
```

Results:

Method	Mean	Error	StdDev
Mongo	174.51 ms	12.05 ms	35.53 ms
Postgres	60.43 ms	22.42 ms	66.11 ms

This is the test where Postgres finally took the lead. Not sure if it's the result of a better query composition for Postgres but anyway that's a 1 point for Postgres.

Conclusion

Here's the aggregated test results table:

Results:

Test	Mongo	Postgres
Index creation on an empty database	152.9 ms	402.0 ms
Batch insert	53.50 ms	219.15 ms
One by one insert	319.3 ms	641.9 ms
Read multiple records	21.83 ms	66.63 ms
Index creation on filled database	1.563 s	1.916 s
Aggregated query	174.51 ms	60.43 ms

As you may see Mongo did beat Postgres in almost all the metrics. It is worth noting that the tests were on semi-structured data which is the realm of Mongo. But Postgres does offer its own solution for the problem so comparing their

performance in the weight category shouldn't be considered unfair. And I did see quite a few similar tests where the results were in favor of Postgres.

Note that the test may be significantly influenced by the way the databases are deployed and accessed (EF Core may influence performance there). However, I would rather get a result that will probably reflect what I get in real life than try to provide arbitrarily fine-tuned scenarios.

I ran those tests in the following environment:

```
BenchmarkDotNet v0.13.12, macOS Sonoma 14.0 (23A344) [Darwin 23.0.0]
Apple M1, 1 CPU, 8 logical and 8 physical cores
.NET SDK 8.0.101
  [Host]             : .NET 8.0.1 (8.0.123.58001), Arm64 RyuJIT AdvSIMD
```

To run your own test just check out: https://github.com/astorDev/seege. Know how to improve the tests? Feel free to raise an issue, leave a comment, or create a pull request.

Mongodb Postgres Postgresql Performance Csharp

Follow

Written by Egor Tarasov

66 Followers · 3 Following

Full-Odd-Stack (.NET and Flutter) Software Dev. Building cutting-edge apps for 6+ years https://github.com/astorDev/ https://www.linkedin.com/in/astordev/

No responses yet

What are your thoughts?

Respond

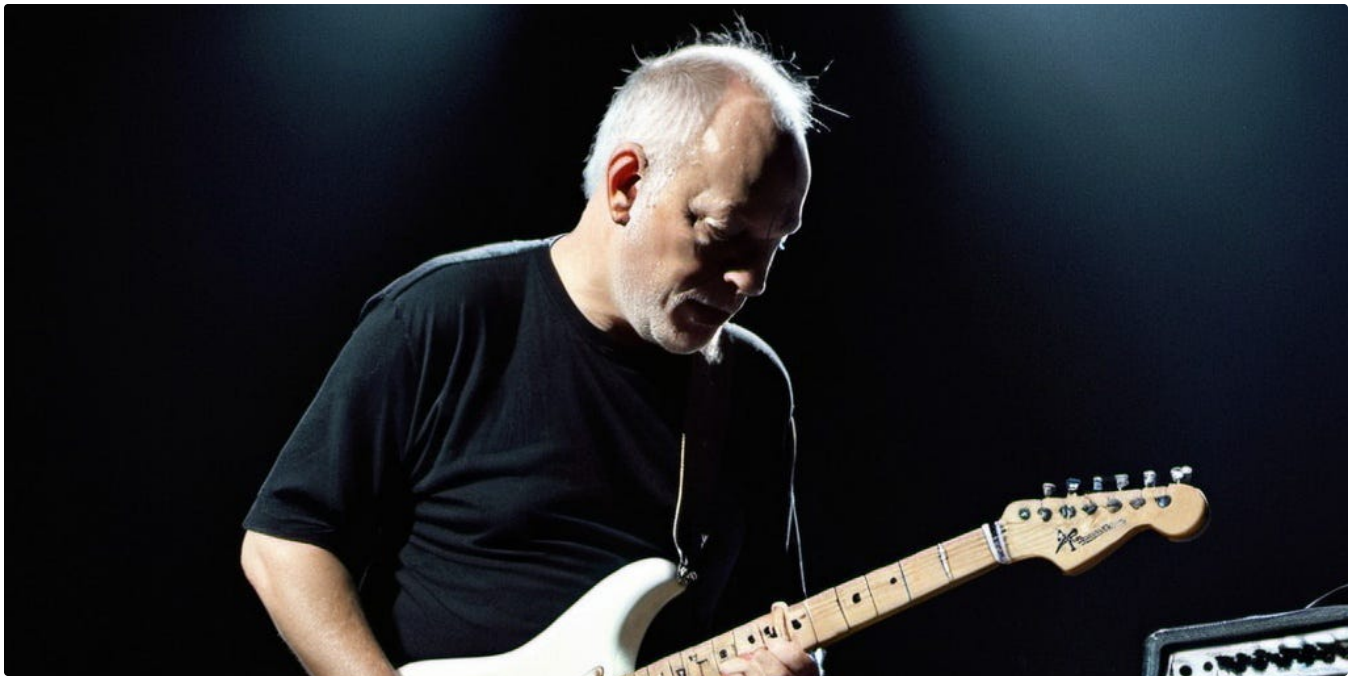
More from Egor Tarasov

Egor Tarasov

Making Your OpenAPI (Swagger) Docs UI Awesome in .NET 9

Since .NET 9 we no longer get a Swagger UI included in the default webapi template. Although the document is still included, now via the...

6d ago 38 1



Egor Tarasov

Environment Variables in ASP .NET Core

In this article, we are going to learn about environment variables in ASP .NET Core by organizing a small concert (in code, of course)...

May 5 20 2

Egor Tarasov

Request-Response Logging in ASP .NET Core

Logging ASP .NET Core http requests and responses is a common task almost every .NET developer faces sooner or later. For a long time, the...

Jun 5 66 1

Egor Tarasov

PostgreSQL Hostings Prices (some are free)

Recently, for my personal project, I've been looking for ways to move the burden of database management from my shoulders on an external...

May 8 5

See all from Egor Tarasov

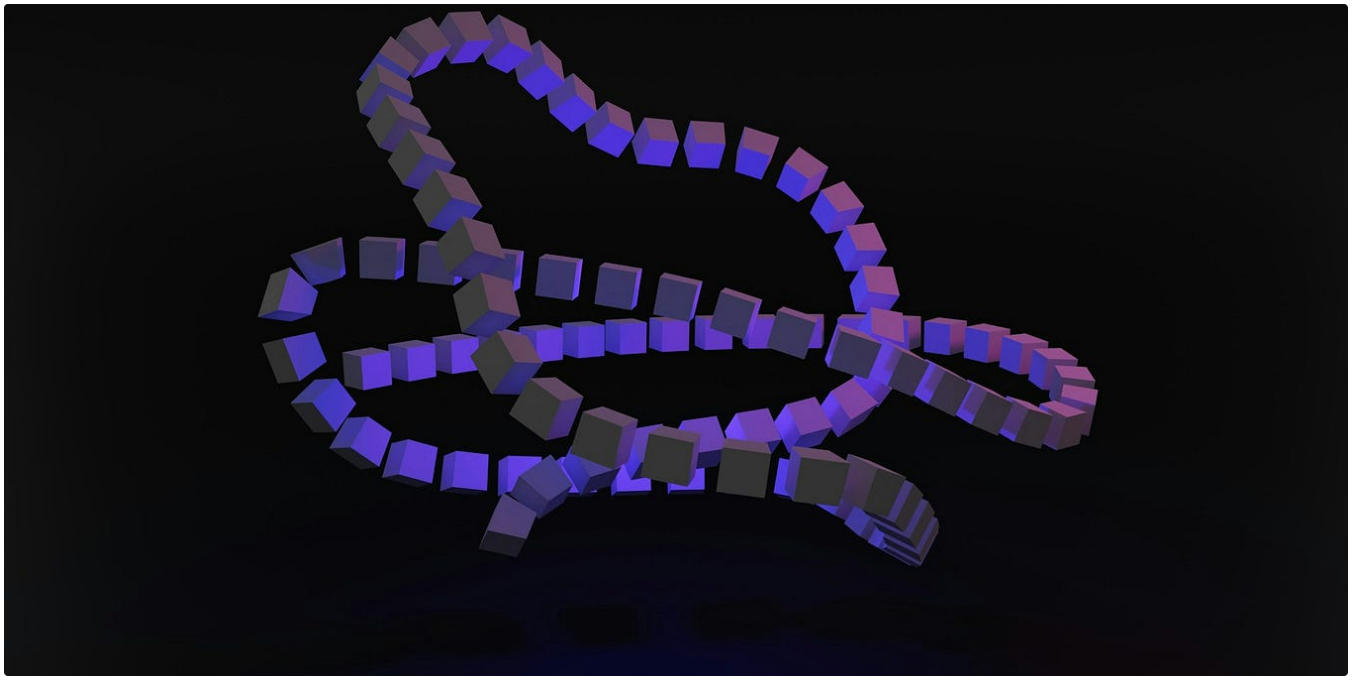
Recommended from Medium

Sheikh Wasiu Al Hasib

Log Sequence Number (LSN) Creation and Usage in PostgreSQL

The Log Sequence Number (LSN) in PostgreSQL is a critical part of its Write-Ahead Logging (WAL) mechanism, ensuring data integrity and...

Jul 13 1



In Xeynergy Blog by Niraj Ranasinghe

Achieving Scalability with Sharded Database Architectures

In the digital world we live in today, applications work with enormous amounts of data. To maintain performance and scalability, especially...

Jun 20 16

Lists

A Guide to Choosing, Planning, and Achieving Personal Goals

13 stories · 2201 saves

Staff picks

778 stories · 1489 saves

Natural Language Processing

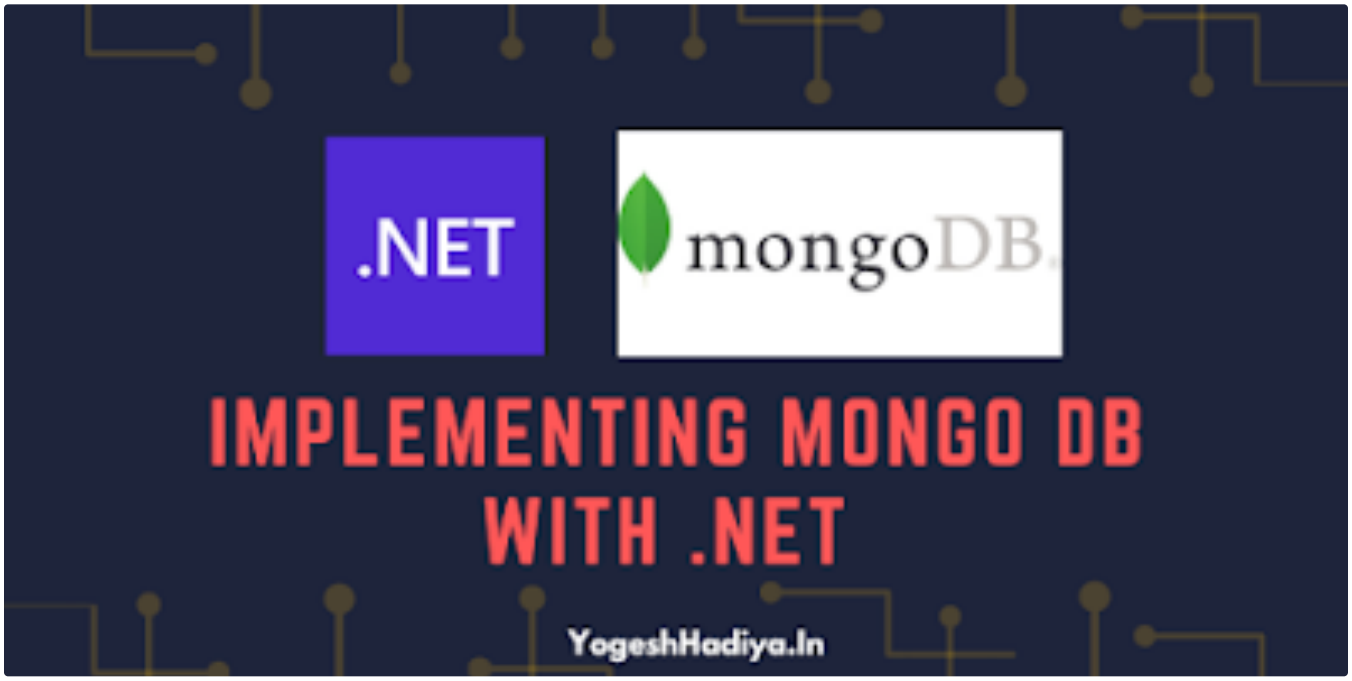
1842 stories · 1466 saves

Meghamishra

Mastering MongoDB Aggregation Pipelines: A Guide for E-Commerce Data Analysis

When working with databases, especially with large datasets, we often need to analyze or process data in more complex ways than simple...

Oct 29

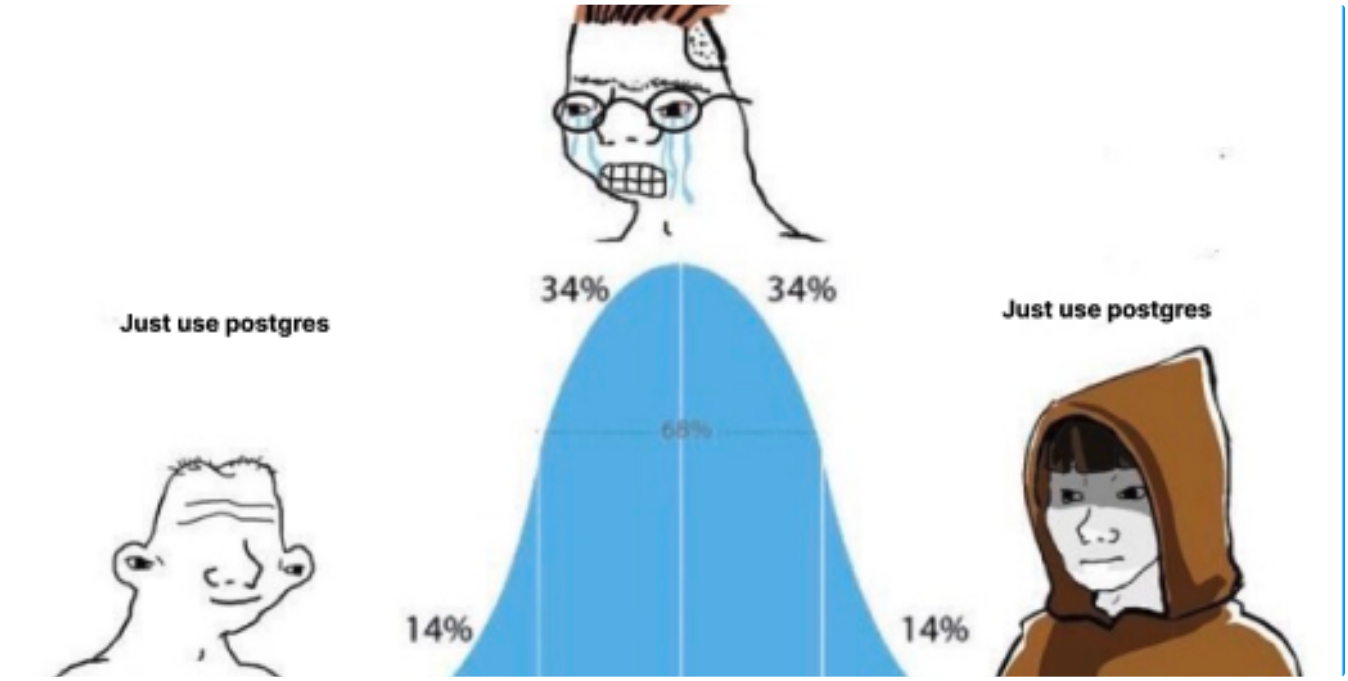


In .Net Programming by YogeshKumar Hadiya

Implementing MongoDB with .NET

In this article, we will implement basic operations in a MongoDB database using .NET and C#. We will use the MongoDB.Driver package to...

Jun 30 24

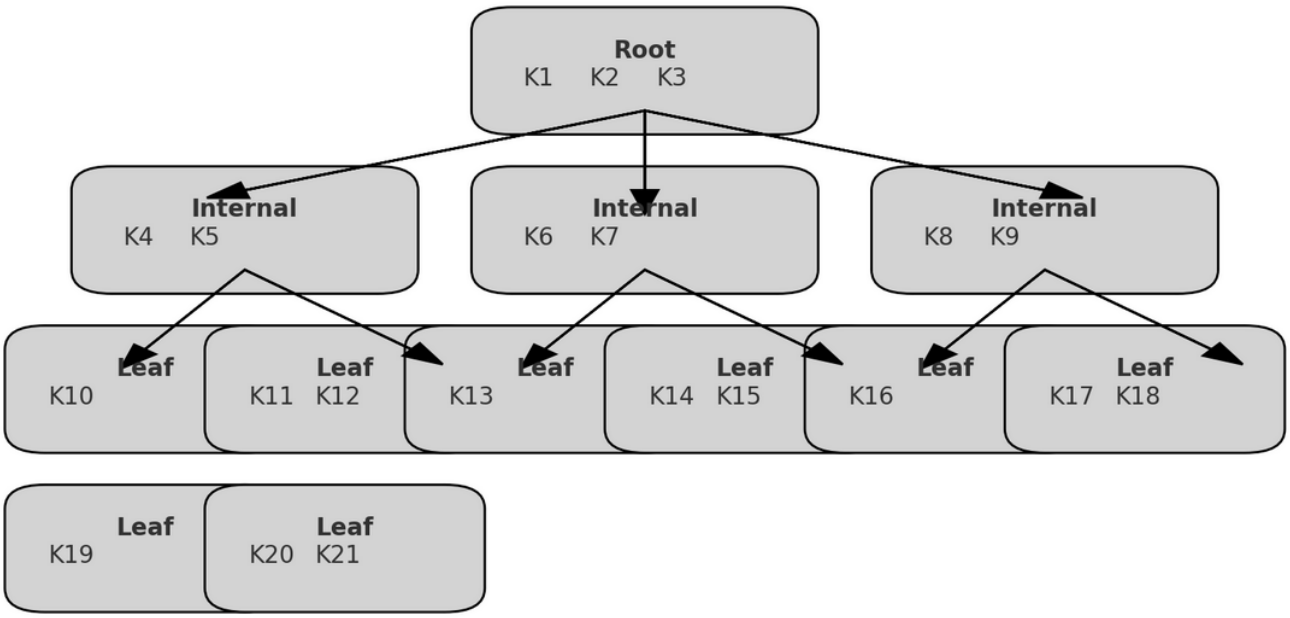


Sebastian Petrus

Pinecone? Milvus? PgVector Is 70% Faster and Cheaper and Open Source

Isn't it strange that, proprietary vector databases peform much worse that Free, Open Source alternatives?

Sep 4 167 1



Aditi Mishra

Use of B-Trees and LSM (Log-Structured Merge) trees in different database architectures

1. B-Trees in Relational Databases

Aug 5 23

See more recommendations