LeoDB

Sid Premkumar & Emmanuel Amponsah

01

Introduction & Background

What motivated our research?

02

LeoDB

How does LeoDB address our problem?

03

Experiment

How we setup and conducted our experiments.

04

Results & Conclusion

What we found & next steps.

Introduction and Background

01

Motivation

What motivated our research?

02

Project Set-up

How does LeoDB address our problem?

Motivation

Lots of databases have to pick between either being read or write optimized, but what if you can get both? Can we create an on-demand database that switches between read and write optimized?

Project Set Up

Standard Library

Implementation

Google Tests

Unit and Integration Tests

Google benchmark

Experiments

Loguru

Error Tracking

LeoDB

01

Main Idea

Driving principles

02

API

Features we decided to implement

03

Optimizer

Trade off between reading and writing

04

Auxiliary Structures

Fence pointer and Bloom Filters

Main Idea

General Purpose

We wanted LeoDB to be a general purpose database that doesn't only store int

Read/Write

We wanted LeoDB to be able to optimize for our current workload without having to stop and tune the database

API Overview

Put/Get/Delete

Built an in-memory database that
supports basic operators

Max/Min

Basic metadata for
number based entries

Optimize

Configure
LSM-Hybrid database

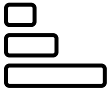
Scan

Performs a search
over a range

Avg/StdDev

Data for number based
entries

Optimizer



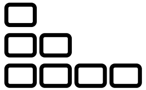
Leveling

Read Performance

Read favorable as we merge and sort each level.

LeoDB

A hybrid mix between Leveling and Tiering.



Tiering

Write Performance

Write favorable as no merging and sorting has to be done.

Auxiliary Structures

Fence Pointer

Lookup table

Improves read performance by allowing us to map values to the files they're stored in

Bloom Filter

Oracle

Prevents us from doing unnecessary searches when the value is not in the table

Experiment Setup

OS

Alpine Linux

CPU

One CPU

Memory

512Mi

State

Only LeoDB was running on machine

Results & Conclusion

01

Benchmarks

Graphs and Results from benchmarks

02

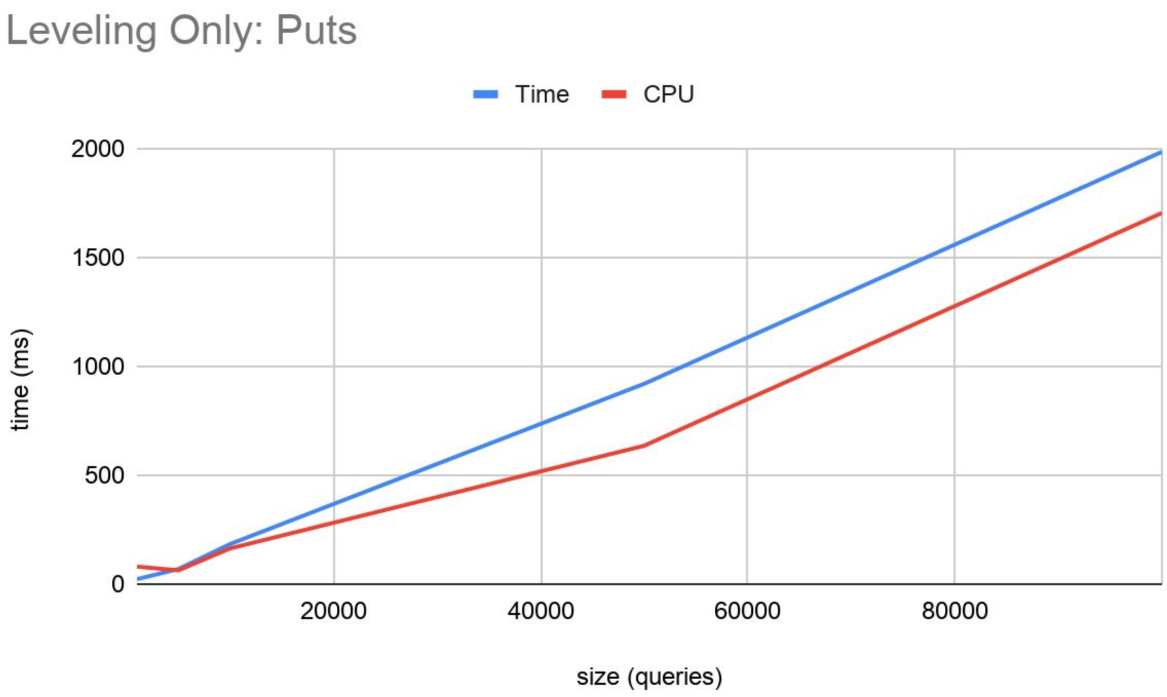
Learnings

What did we learn?

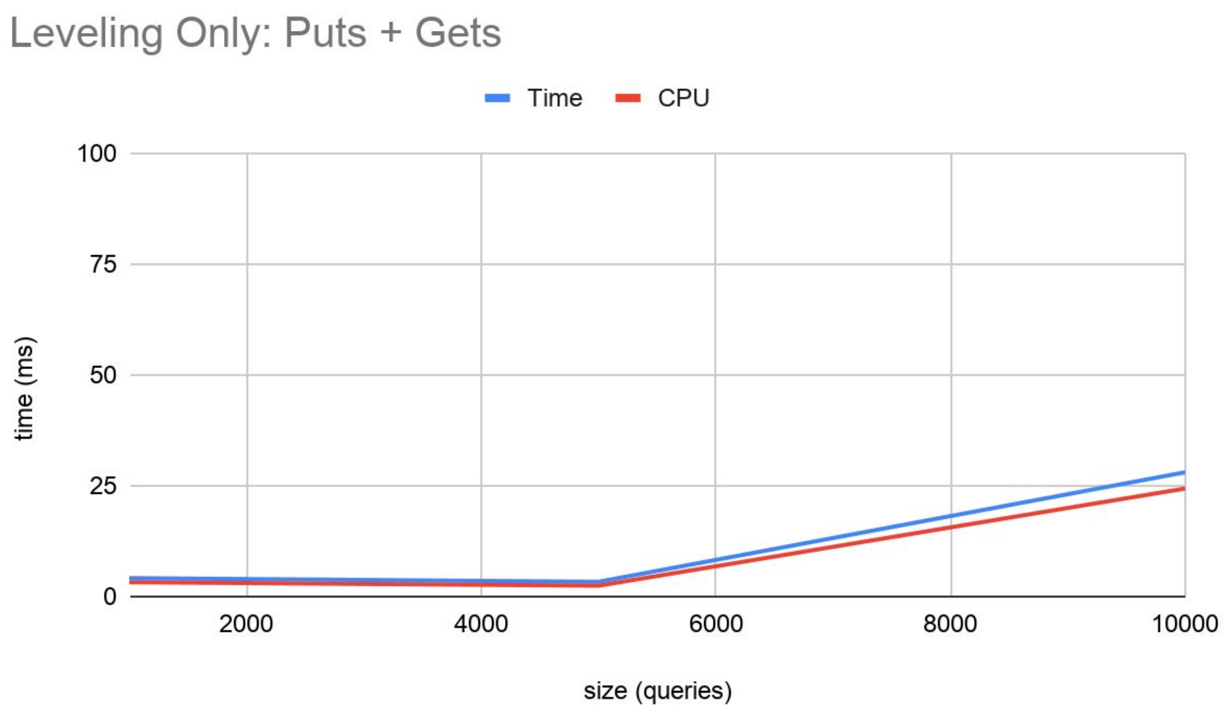
03

Conclusion

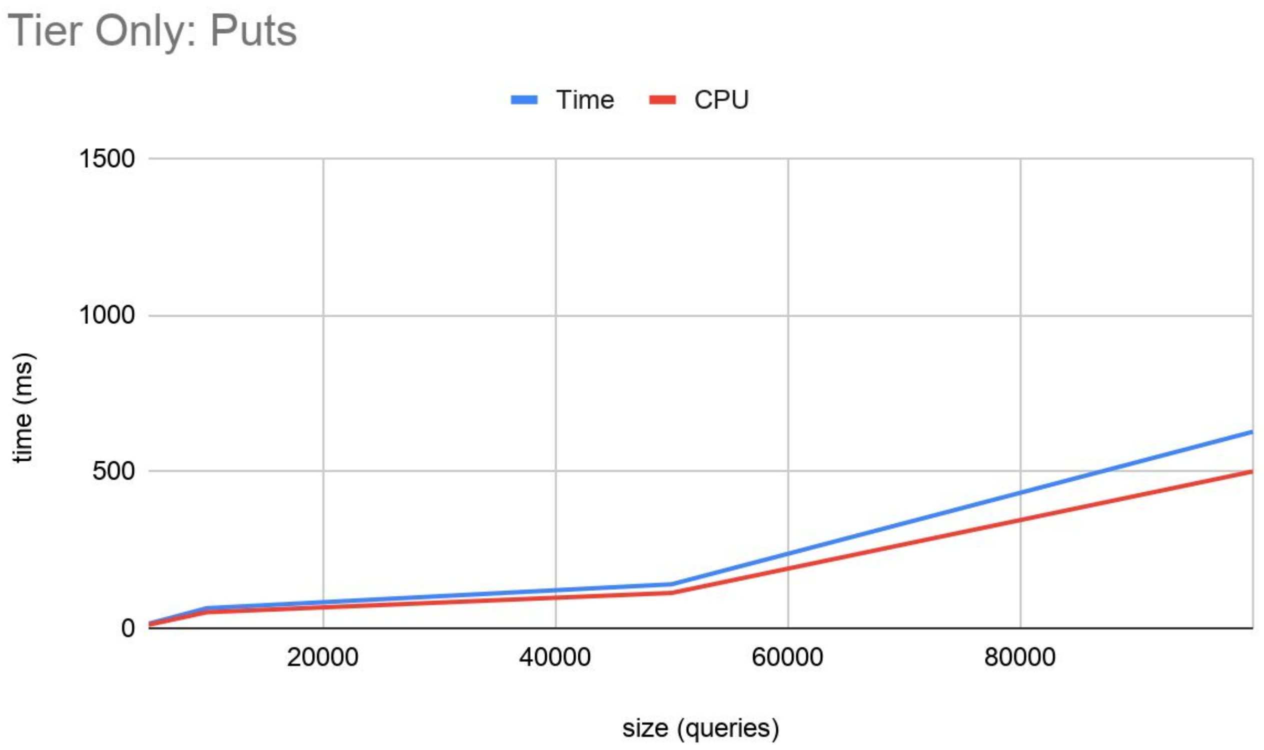
Wrap Up



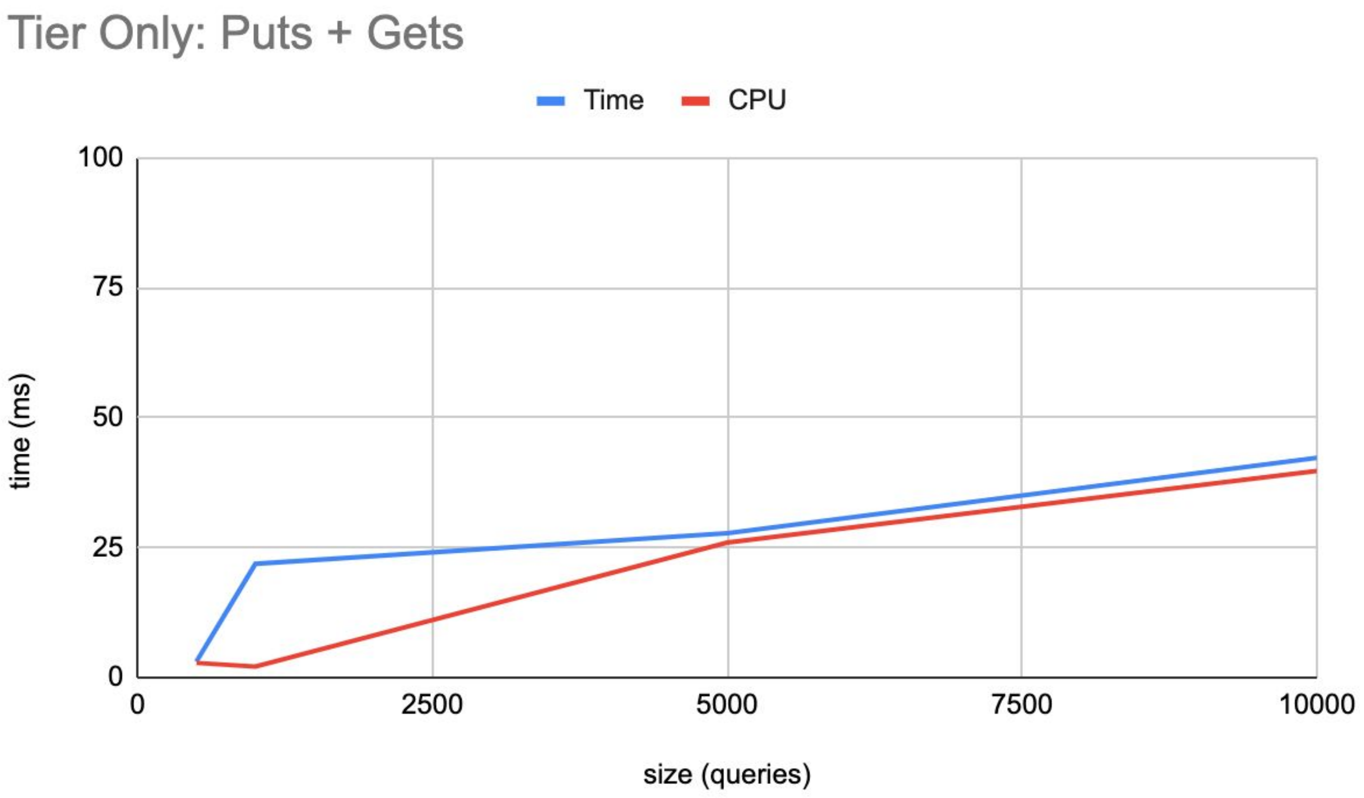
Leveling only results in bad write times



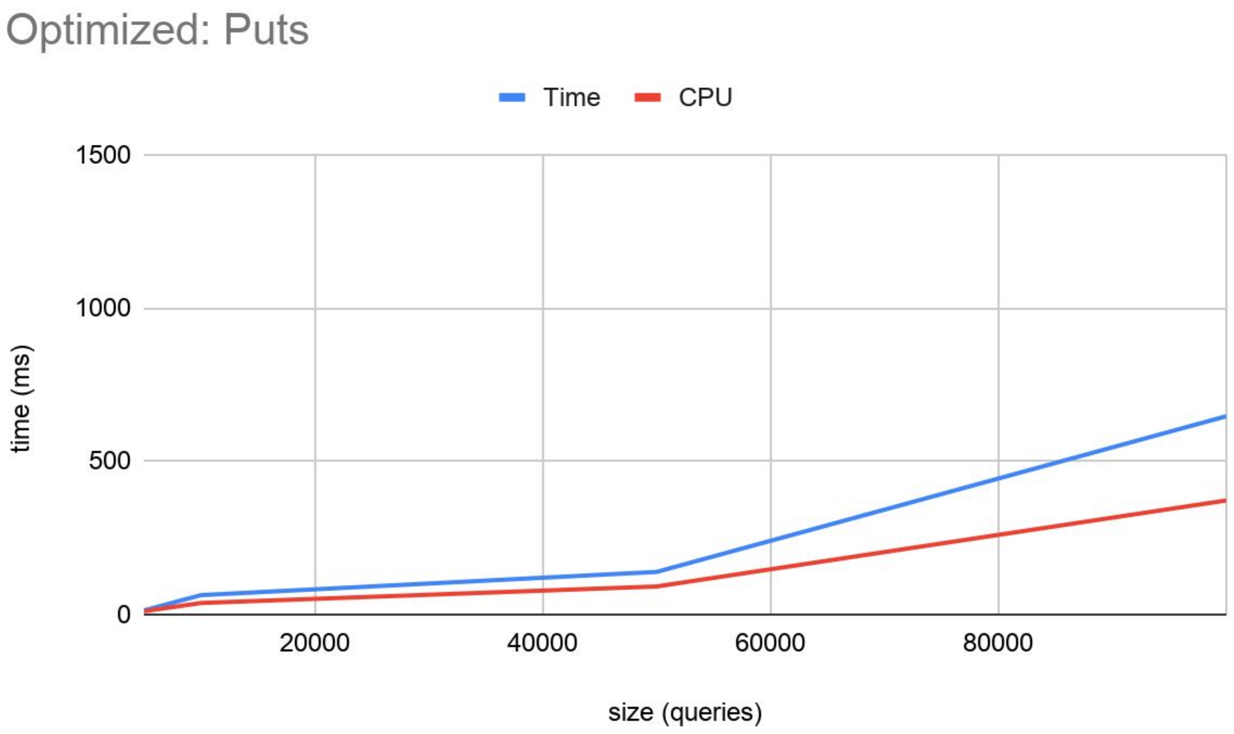
Leveling only results in good read times



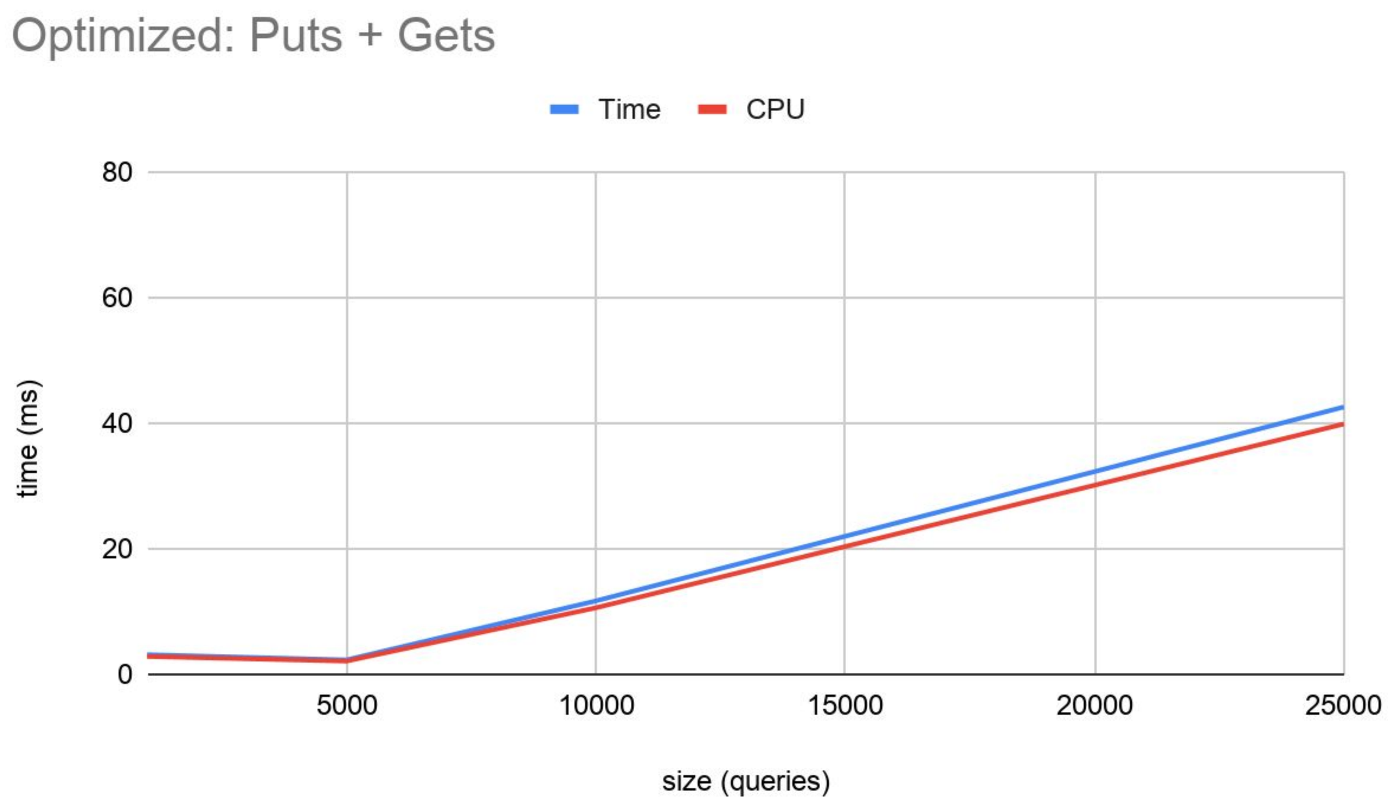
Tiering only database has relatively fast writes



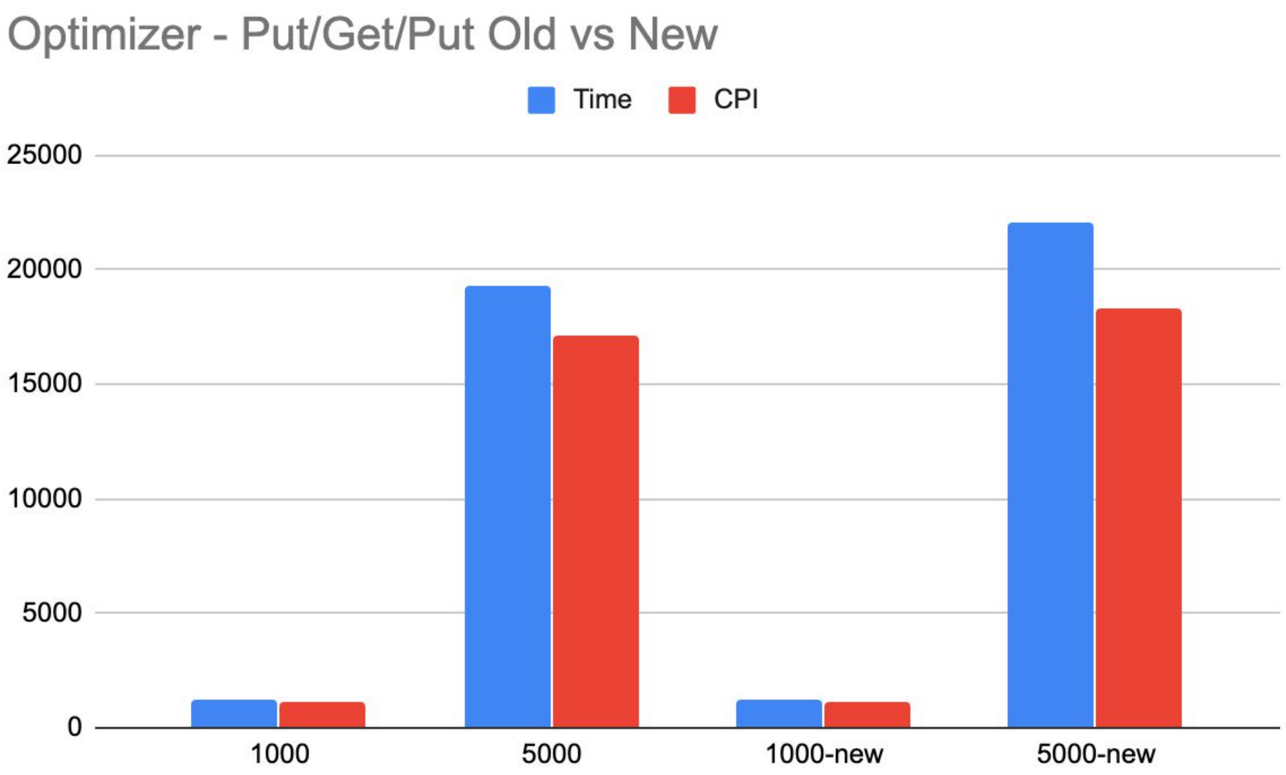
Tiering only database has relatively good reads



Optimizer ensures that read/writes are optimized based on workload



Optimized: Puts + Gets
Time
CPU
80
60
40
20
0
time (ms)
5000
10000
15000
20000
25000
size (queries)



Optimizer was able to determine out which values were being read frequently

Lesson Learned – Implementation

On demand tuning is expensive

Buffers are often too big to fit in memory

Real world solutions are hard to implement

Conclusions

01

General Purpose is Hard

Building a database for any use case is a messy process

02

Optimizer Overload

So many ways to decide how your optimizer should be implemented and it's hard to test all these methods

03

More structures, more complexity

Auxiliary structures like bloom filters and fence pointers improve the design of your database but add a lot of overhead in complexity