Sionic AI

Building a seamless **RAGOps Systems** Like A Pro

Sigrid Jin \ Sionic AI

Speaker

Sigrid Jin (Jin Hyung Park)

Forward Deployed Software Engineer

- github.com/sigridjineth
- sigridjin.medium.com
- https://instruct.kr

In this talk,

I am going to cover

- considerations for building/operating/choosing vector dbs.
- Operator. Kubernetes. DevOps.

I won't cover

- No ML things. finetuning LLMs or BERT-wise retriever models.
- Prompting strategy. lexical models. approximation (e.g. ANN)

RAG is more than just embedding search

RAG is more than just embedding search

RAG is more than just embedding search

RAG is more than just embedding search

RAG is more than just embedding search

RAG is more than just pip install faiss-gpu

Query:

"Why do T and *T have different method sets?"

Relevant Passage:

"The frequently asked questions about Go"

The emergence of new machine learning models creates an opportunity untapped by traditional databases, since searching by both meanings and keywords at scale is a big problem.

But nothing more changed for running infrastructure, but things getting more messed.

Considerations for Vector DBs.

Choosing a vector db for 100 million pages of text. Leaning towards Milvus, Qdrant or Weaviate. Am I missing anything, what would you choose?

For context my vector db research started today from 0 knowledge and I feel absolutely unqualified to be making this decision but here we are.

I have narrowed the search down to Milvus, Qdrant and potentially Weaviate.

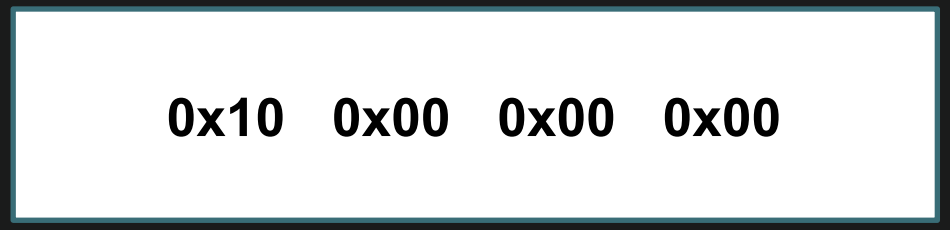
I am scoping out a project for a client where we need to store up to 100 million pages. The application is scientific so retrieval precision is a top priority as is search time latency and cost.

It seems:

- Milvus seems the most established and easiest to setup. also itis fast but takes up a lot of memory so can get quite expensive.
- Qdrant is fast and quite a bit cheaper than Milvus but lacks dynamic sharding
- I have seen two conflicting reports one saying Weaviate is incredibly quick with a benchmark of 0.12s for a particular query which took Milvus 0.9s to perform the same and then another where it says it is slow. and it is the cheapest.
- PG-vector is not as performant as the dedicated vector stores but are tried and tested part of the ecosystem and anecdotally great to work with
- Chroma is not the best for accurate retrieval and I haven't heard many recommending it as the best except for its usability and ease of integration.

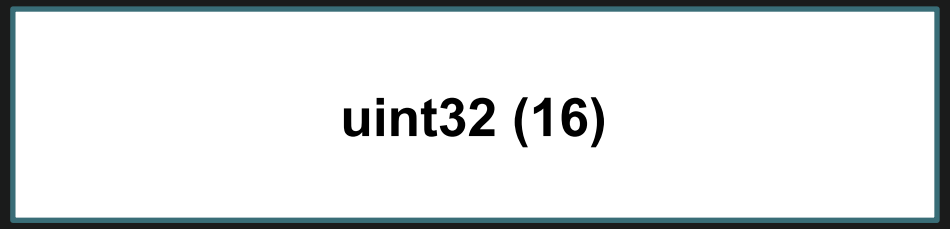
Think of Vector Embeddings
as arrays of floats

[0.762, 0.123, -0.882, ..., 0.993]



there are numbers

represented as bytes on disk

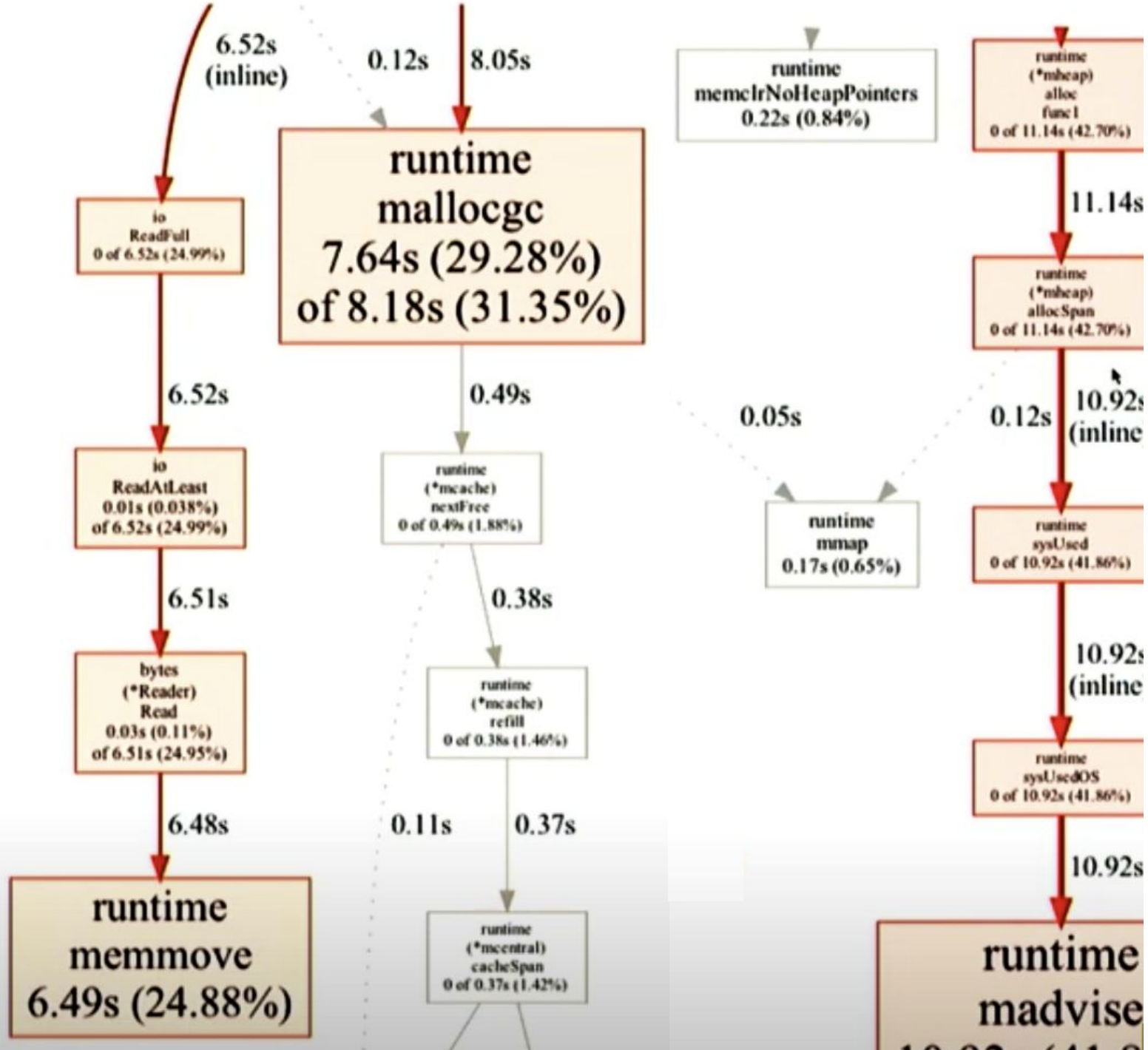


parsed into Go data structure

encoding/binary

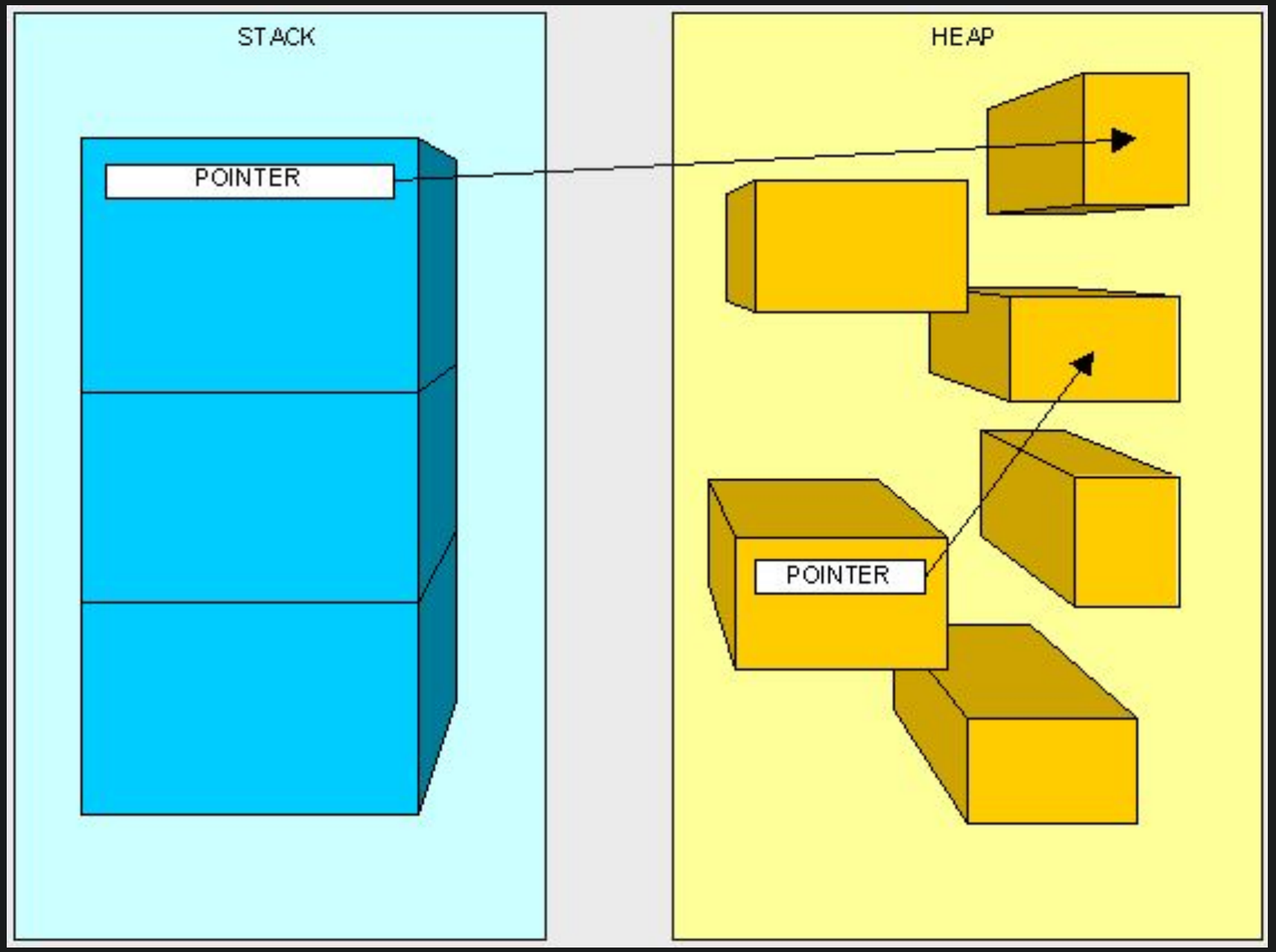
```
var r io.Reader
var num uint32

err := binary.Read(r, binary.LittleEndian, &num)
if err != nil {
    return err
}
```

https://www.youtube.com/watch?v=K1R7oK2piUM



parsing 1bn uint64s takes around 26s. memory allocation.

Golang Korea


STACK
HEAP
POINTER
POINTER

we know that stack is so much cheaper than heap

```
buf := make([] byte, 8, runtime.ON_STACK_PLEASE)
```

something went to heap, the clue of memory allocation issue?

```
go build -gcflags="-m"
```

```
./alloc_test.go:9:21: t does not escape
./alloc_test.go:15:7: moved to heap: num
./alloc_test.go:12:22: &bytes.Reader{...} escapes to heap
./alloc_test.go:16:14: binary.LittleEndian escapes to heap
./alloc_test.go:20:26: t does not escape
```

0x10 0x00 0x00 0x00

there are numbers

represented as bytes on disk

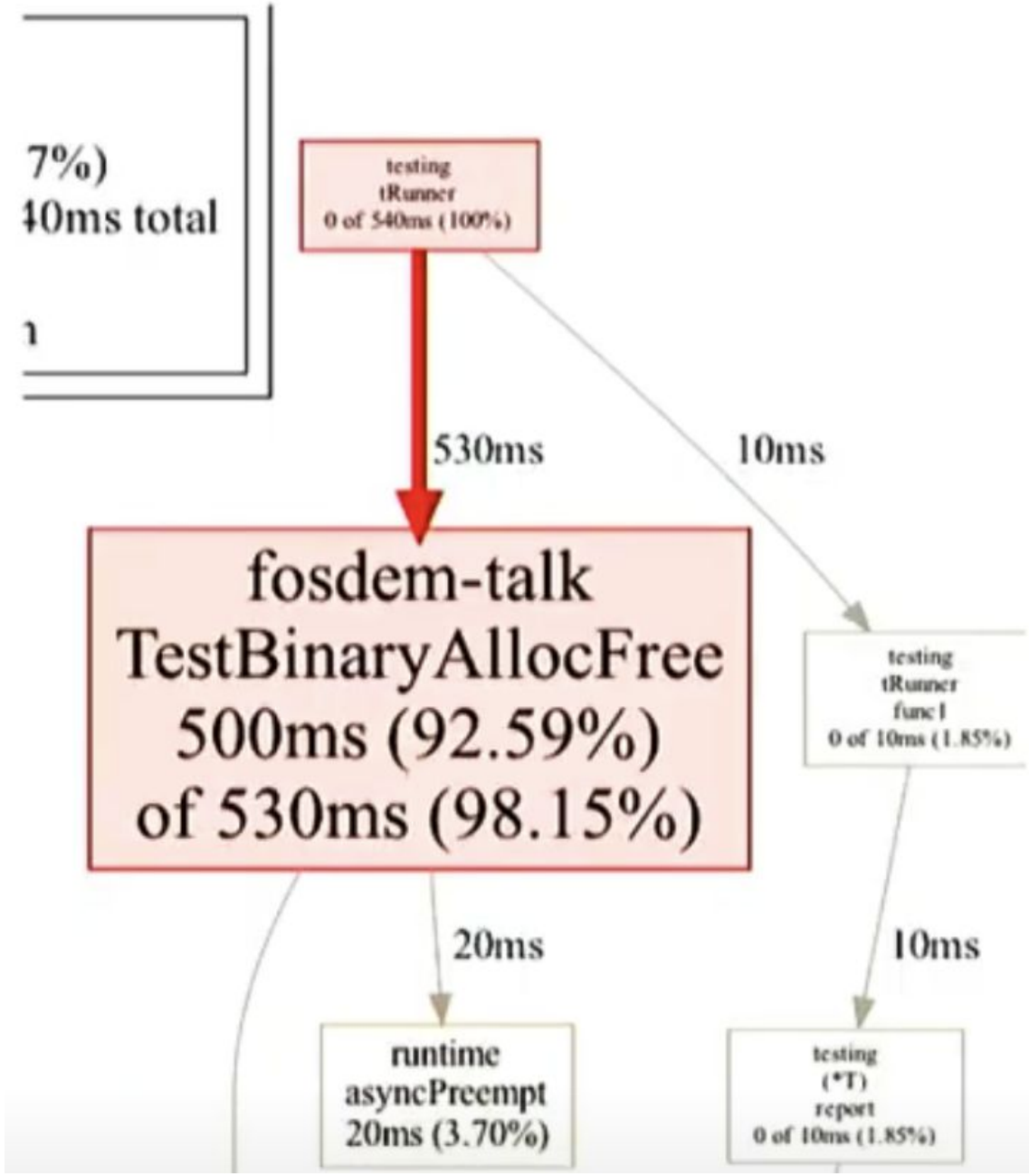
uint32 (16)

parsed into Go data structure

```
var buf bytes.Buffer
var offset int

num := binary.LittleEndian.Uint32(buf[offset:offset+4]
```

binary/LittleEndian

https://www.youtube.com/watch?v=K1R7oK2piUM



parsing 1bn uint64s takes around 600ms.

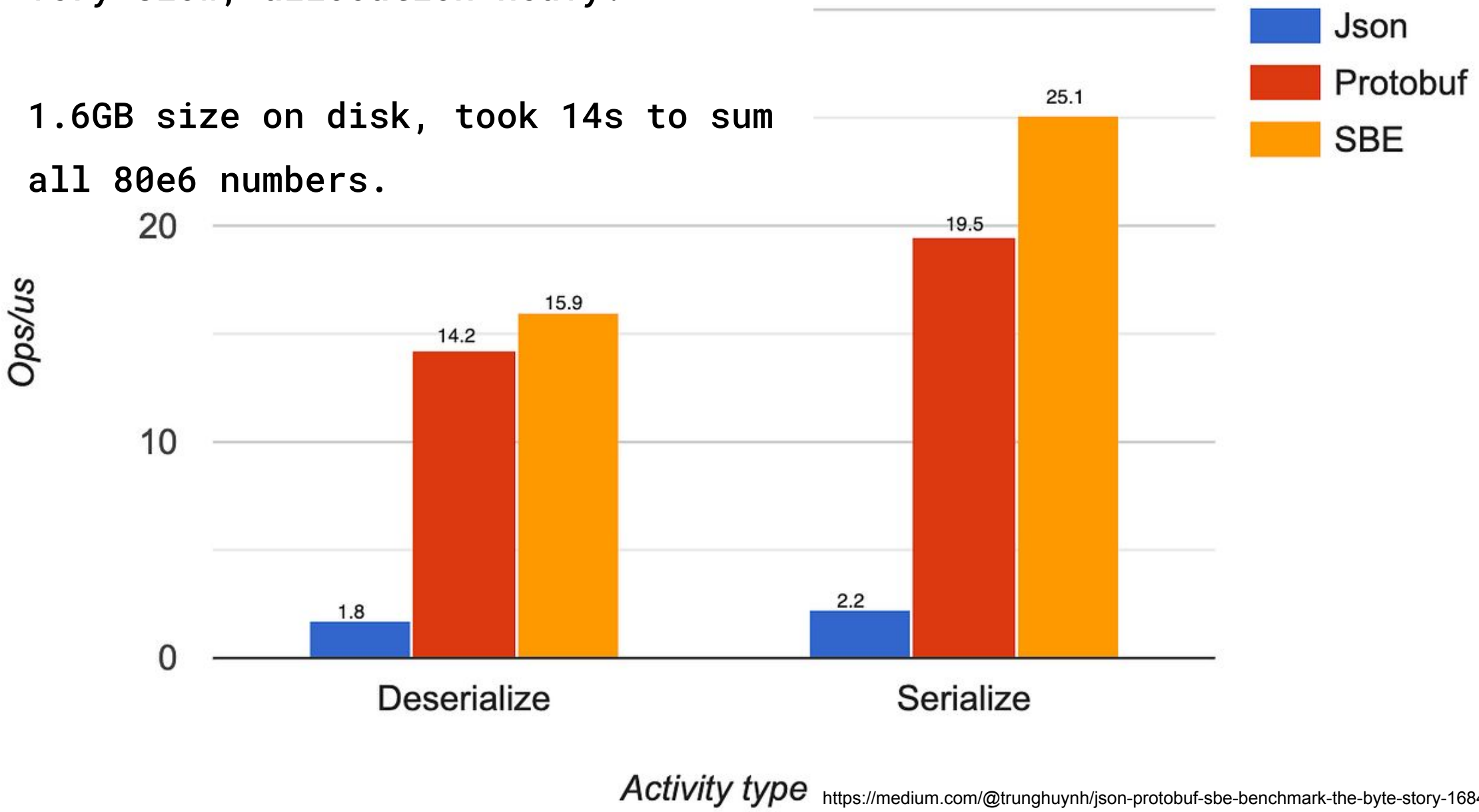
Sum all numbers in the nested array;
10e6 outer elements and each with 8 inner elements.

The data is stored on disk.

```
var data [][]uint64
```

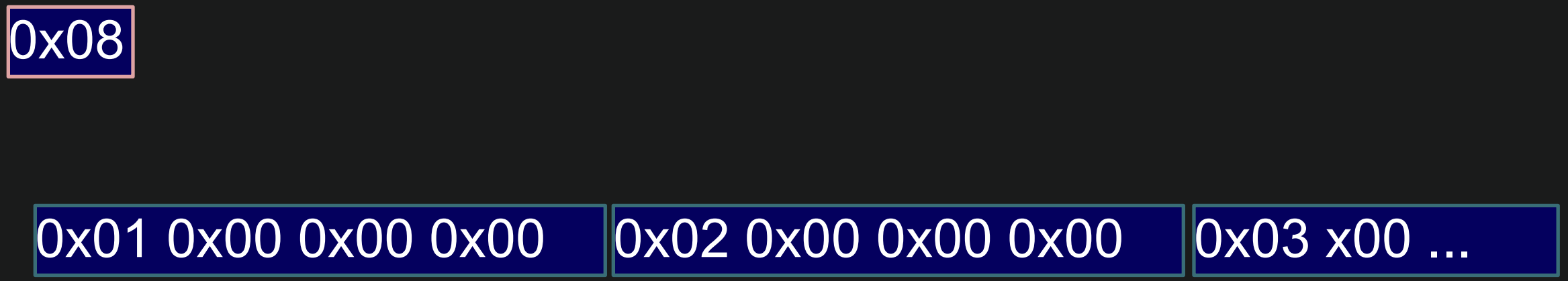
JSON: space inefficient, parsing is very slow, allocation heavy.

1.6GB size on disk, took 14s to sum all 80e6 numbers.



https://medium.com/@trunghuynh/json-protobuf-sbe-benchmark-the-byte-story-16840f763543

uint8 length.
the length indicator. we have 8 elements
coming up!



0x08 0x01 0x00 ...

convert them into slice of slice of slice w/ uint64.

just by using a more space efficient way to represent data much faster.
660MB size on disk, 260ms in total (250ms decoding; 10ms summing up)

```
var src []byte
data := make([][]uint64, size)
var o int // offset

for i := range data {
    length := binary.LittleEndian.Uint8(src[o:o+1])
    o += 1
    data[i] = decodeInner(src, length, o)
}
```

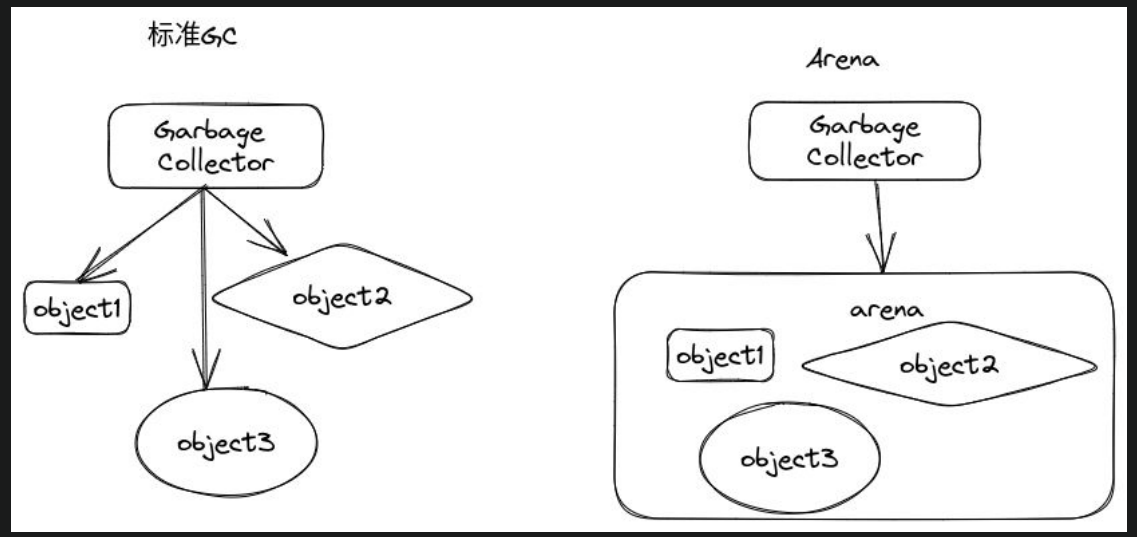
```
var src []byte
data := make([][]uint64, size)
var o int // offset

for i := range data {
    length := binary.LittleEndian.Uint8(src[o:o+1])
    o += 1
    data[i] = decodeInner(src, length, o)
}
```

allocate
massive slices
again.

```
var src []byte
var sum uint64
var o int // offset

for i := range data {
    ...
    sum += binary.LittleEndian.Uint64(src[o:o+8])
    ...
}
```

looping the data
without storing
it

much better
memory locality.



Golang 1.20 Memory Arenas.

```
// Memory pool to simulate arena-like memory allocation
var bufferPool = sync.Pool{
	New: func() interface{} {
		// Allocate a large buffer once and reuse it
		return make([]byte, 1024*1024) // 1MB buffer
	},
}

func processBinaryData(data []byte) uint64 {
	// Simulate decoding large binary data without excessive allocations
	var sum uint64
	var o int // offset

	// Assume we're processing fixed-width binary data
	for o < len(data) {
		// Read an uint64 from the byte slice
		val := binary.LittleEndian.Uint64(data[o : o+8])
		sum += val
		o += 8 // Move to the next 8-byte chunk
	}

	return sum
}
```

```
var a, b []float32
var c float32


for i := range a {
    c += a[i] * b[i]
}
```

two instructions are being involved.
multiplication and addition.

You can think of dot product is like multiplication and summing.

```go
package main

import (
    "fmt"
)

func dotProductUnrolled(a, b []float32) float32 {
    var c float32
    n := len(a)

    // Process 4 elements at a time
    for i := 0; i < n-(n%4); i += 4 {
        c += a[i] * b[i]
        c += a[i+1] * b[i+1]
        c += a[i+2] * b[i+2]
        c += a[i+3] * b[i+3]
    }

    // Process any remaining elements
    for i := n - (n % 4); i < n; i++ {
        c += a[i] * b[i]
    }

    return c
}

func main() {
    // Example vectors
    a := []float32{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0}
    b := []float32{5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0}

    // Calculate the dot product
    c := dotProductUnrolled(a, b)

    fmt.Printf(
}
```

Output... Filter... Libraries Overrides Add new... Add tool...

```
        TEXT    main.dotProductUnrolled(SB), NOSPLIT|ABIInternal, $24-48
        PUSHQ   BP
        MOVQ    SP, BP
        SUBQ    $16, SP
        MOVQ    AX, main.a+32(FP)
        MOVQ    DI, main.b+56(FP)
        FUNCDATA        $0, gclocals·cNGUyZq94N9QFR70tEjj5A==(SB)
        FUNCDATA        $1, gclocals·J5F+7Qw7O7ve2QcWC7DpeQ==(SB)
        FUNCDATA        $5, main.dotProductUnrolled.arginfo1(SB)
        FUNCDATA        $6, main.dotProductUnrolled.argliveinfo(SB)
        PCDATA  $3, $1
        XORL    CX, CX
        XORPS   X0, X0
        JMP     main_dotProductUnrolled_pc45
main_dotProductUnrolled_pc25:
        MULSS   12(DI)(CX*4), X1
        ADDQ    $4, CX
        ADDSS   X2, X1
        MOVQ    R8, BX
        MOVUPS  X1, X0
main_dotProductUnrolled_pc45:
        MOVQ    BX, DX
        ANDL    $3, BX
        MOVQ    DX, R8
        SUBQ    BX, DX
        NOP
        CMPQ    CX, DX
        JGE     main_dotProductUnrolled_pc228
        CMPQ    R8, CX
        JLS     main_dotProductUnrolled_pc357
        MOVSS   (AX)(CX*4), X1
        NOP
        CMPQ    SI, CX
        JLS     main_dotProductUnrolled_pc341
        MULSS   (DI)(CX*4), X1
        LEAQ    1(CX), DX
        ADDSS   X1, X0
        CMPQ    R8, DX
        JLS     main_dotProductUnrolled_pc330
```

Multiplies the low single-precision floating-point value from the source operand (second operand) by the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location.

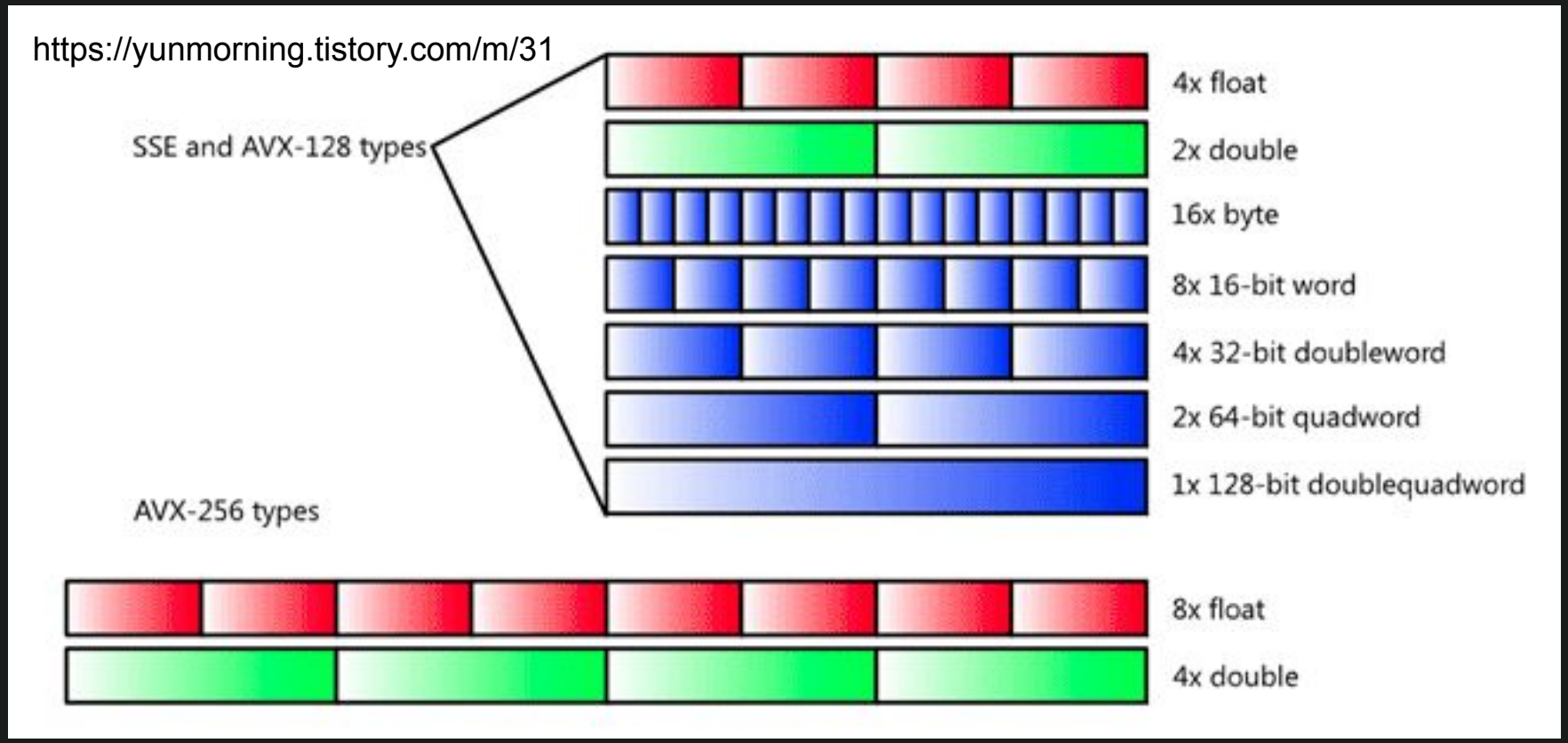
```
var a, b []float32
var c float32


for i := 0; i < len(a); i += 8 {
    c += a[i] * b[i]
    c += a[i+1] * b[i+1]
    c += a[i+2] * b[i+2]
    c += a[i+3] * b[i+3]
    c += a[i+4] * b[i+4]
    c += a[i+5] * b[i+5]
    c += a[i+6] * b[i+6]
    c += a[i+7] * b[i+7]
}
```

Weaviate's approaches: Unrolling?

8 multiplications + 8 additions.

SIMD. single instruction multiple data.

VADDDPS. VFMADD231SS. AVX Programming.



Go Wiki: AVX512

okay, Golang supports AVX512,

But How do you tell Golang to use AVX2/AVX512 instructions?

https://go.dev/wiki/AVX512

https://github.com/milvus-io/milvus/discussions/14942

Go 1.11 release introduces AVX-512 support.
This page describes how to use new features as well as some important encoder details.

Terminology

Most terminology comes from Intel Software Developer's manual.
Suffixes originate from Go assembler syntax, which is close to AT&T, which also uses size

Some terms are listed to avoid ambiguity (for example, opcode can have different meaning

Sum a slice of uint64s.

```
func main() {
        TEXT("Sum", NOSPLIT, "func(xs []uint64) uint64")
        Doc("Sum returns the sum of the elements in xs.")
        ptr := Load(Param("xs").Base(), GP64())
        n := Load(Param("xs").Len(), GP64())

        Comment("Initialize sum register to zero.")
        s := GP64()
        XORQ(s, s)

        Label("loop")
        Comment("Loop until zero bytes remain.")
        CMPQ(n, Imm(0))
        JE(LabelRef("done"))

        Comment("Load from pointer and add to running sum.")
        ADDQ(Mem{Base: ptr}, s)

        Comment("Advance pointer, decrement byte count.")
        ADDQ(Imm(8), ptr)
        DECQ(n)
        JMP(LabelRef("loop"))

        Label("done")
        Comment("Store sum to return value.")
        Store(s, ReturnIndex(0))
        RET()
        Generate()
}
```

```
// Code generated by command: go run asm.go -out sum.s -stubs stub.go. DO NOT EDIT.

#include "textflag.h"

// func Sum(xs []uint64) uint64
TEXT ·Sum(SB), NOSPLIT, $0-32
        MOVQ xs_base+0(FP), AX
        MOVQ xs_len+8(FP), CX

        // Initialize sum register to zero.
        XORQ DX, DX

loop:
        // Loop until zero bytes remain.
        CMPQ CX, $0x00
        JE   done

        // Load from pointer and add to running sum.
        ADDQ (AX), DX

        // Advance pointer, decrement byte count.
        ADDQ $0x08, AX
        DECQ CX
        JMP  loop

done:
        // Store sum to return value.
        MOVQ DX, ret+24(FP)
        RET
```

https://github.com/mmcloughlin/avo

Milvus uses Avo for building assembly L2 distance code.

```
func main() {
        TEXT("L2", NOSPLIT, "func(x, y []float32) float32")
        Doc("squared l2 between x and y")
        x := Mem{Base: Load(Param("x").Base(), GP64())}
        y := Mem{Base: Load(Param("y").Base(), GP64())}
        n := Load(Param("x").Len(), GP64())

        acc := make([]VecVirtual, unroll)
        diff := make([]VecVirtual, unroll)
        for i := 0; i < unroll; i++ {
                acc[i] = YMM()
                diff[i] = YMM()
        }

        for i := 0; i < unroll; i++ {
                VXORPS(acc[i], acc[i], acc[i])
                VXORPS(diff[i], diff[i], diff[i])
        }
```

```
        blockitems := 8 * unroll
        blocksize := 4 * blockitems
        Label("blockloop")
        CMPQ(n, U32(blockitems))
        JL(LabelRef("tail"))

        // Load x
        xs := make([]VecVirtual, unroll)
        for i := 0; i < unroll; i++ {
                xs[i] = YMM()
        }

        for i := 0; i < unroll; i++ {
                VMOVUPS(x.Offset(32*i), xs[i])
        }

        for i := 0; i < unroll; i++ {
                VSUBPS(y.Offset(32*i), xs[i], diff[i])
```

https://github.com/milvus-io/milvus/blob/3936d126614de6cefe52b9d061ef4007cd94ae3c/pkg/util/distance/asm/l2.go#L6

```rust
#[target_feature(enable = "avx")]
#[target_feature(enable = "fma")]
pub(crate) unsafe fn dot_similarity_avx(
    v1: &[VectorElementType],
    v2: &[VectorElementType],
) -> ScoreType {
    let n = v1.len();
    let m = n - (n % 32);
    let mut ptr1: *const f32 = v1.as_ptr();
    let mut ptr2: *const f32 = v2.as_ptr();
    let mut sum256_1: __m256 = _mm256_setzero_ps();
    let mut sum256_2: __m256 = _mm256_setzero_ps();
    let mut sum256_3: __m256 = _mm256_setzero_ps();
    let mut sum256_4: __m256 = _mm256_setzero_ps();
    let mut i: usize = 0;
    while i < m {
        sum256_1 = _mm256_fmadd_ps(_mm256_loadu_ps(ptr1), _mm256_loadu_ps(ptr2), sum256_1);
        sum256_2 = _mm256_fmadd_ps(
            _mm256_loadu_ps(ptr1.add(8)),
            _mm256_loadu_ps(ptr2.add(8)),
            sum256_2,
        );
        sum256_3 = _mm256_fmadd_ps(
            _mm256_loadu_ps(ptr1.add(16)),
            _mm256_loadu_ps(ptr2.add(16)),
            sum256_3,
        );
```

Note: In qdrant, product quantization is not SIMD-friendly while scalar quantization does. So if considering any options to use quantization, don't believe in benchmarks but do your own research.

Note: There is higher overhead to call other languages from Go to other Languages like Rust. That could be another wishlist.

https://github.com/qdrant/qdrant/blob/master/lib/segment/src/spaces/simple_avx.rs#L155

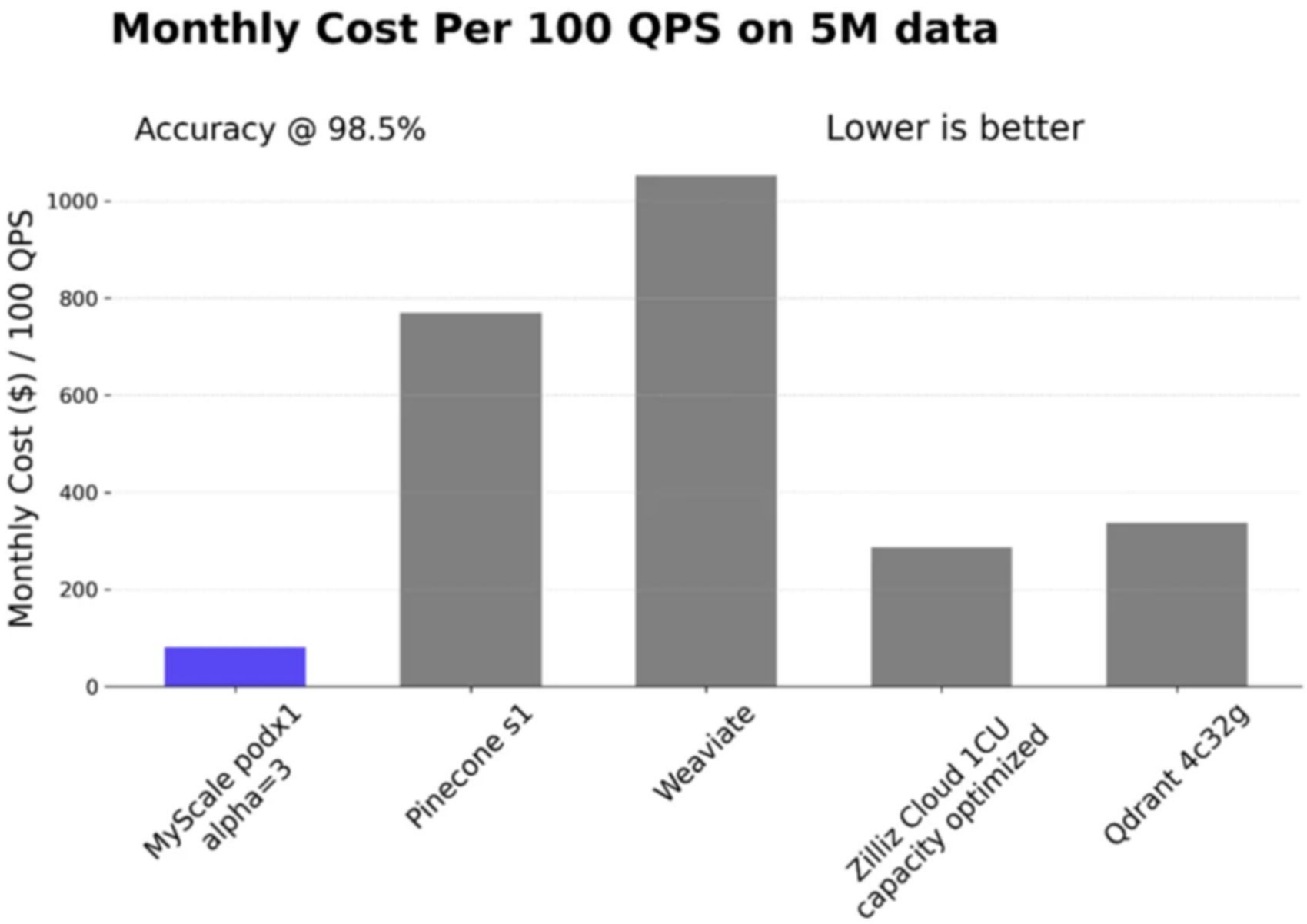
So the recap,

- Do your own research when choosing vector database since there's no silver bullet.
- I think that no generalized benchmarks are appropriate to your cases, but rather than performance, developer experience and maturity is far more important in a production stage IMO.

Disks and Costs.


"Vector DB"
RocksDB
KV Store

Reddit.


Monthly Cost Per 100 QPS on 5M data
Accuracy @ 98.5%
Lower is better
Monthly Cost ($) / 100 QPS
1000
800
600
400
200
0
MyScale podx1 alpha=3
Pinecone s1
Weaviate
Zilliz Cloud 1CU capacity optimized
Qdrant 4c32g

My requests are very slow or time out. What should I do?

There are several possible reasons for that:

- **Using filters without payload index** — If you're performing a search with a filter but you don't have a payload index, Qdrant will have to load whole payload data from disk to check the filtering condition. Ensure yo
- **Usage of on-disk vector storage with slow disks** — If you're using on-d using local SSDs with at least 50k IOPS. Read more about the influence Consumption.
- **Large limit or non-optimal query parameters** — A large limit or offset m attention to the query/collection parameters that significantly diverge fro

- **Embedding Data**: 1M
- **512 chunk, 768-dimensional vector**

Storage Costs:

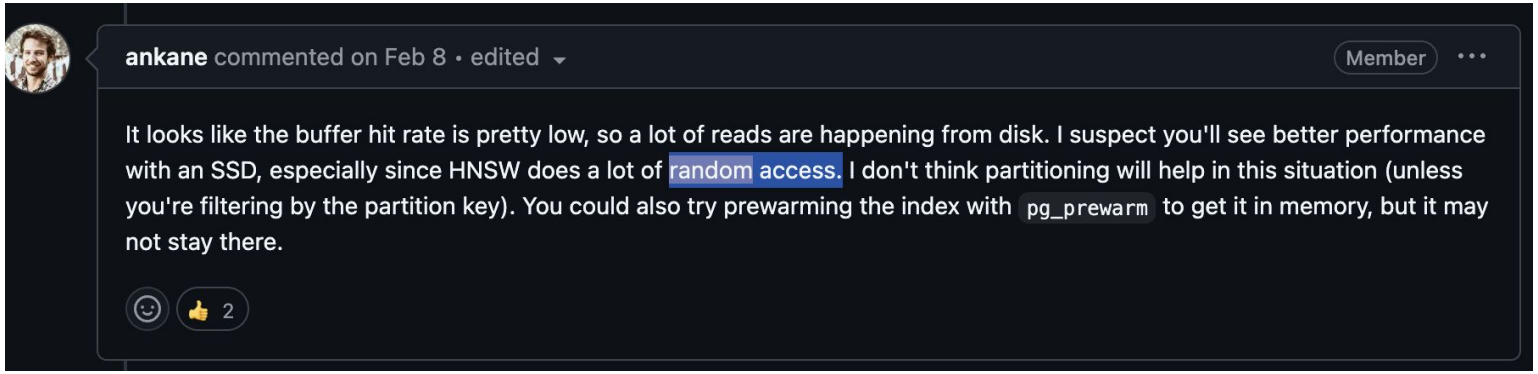
- **Storage size**: 5GB
- **Storage cost**: 5GB * $0.1278/GB/month = **$0.639/month**

IOPS Costs:

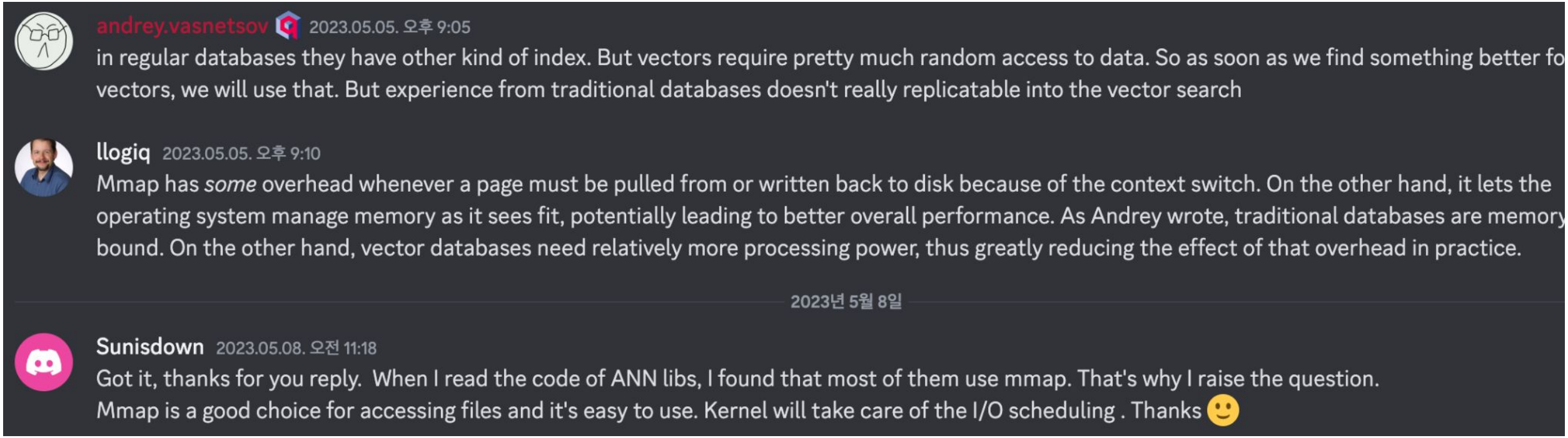
- **IOPS**: 50,000
- **IOPS cost**: 50,000 * $0.047/IOPS/month = **$2,350/month**

Total Monthly Costs:

- **Total cost**: $0.639 (storage) + $2,350 (IOPS) = **$2,350.639/month**



https://github.com/pgvector/pgvector/issues/455



▲ How Discord supercharges network disks for extreme low latency (discord.com)

551 points by techinvalley on Aug 15, 2022 | hide | past | favorite | 215 comments

Current Setup (AS-IS)

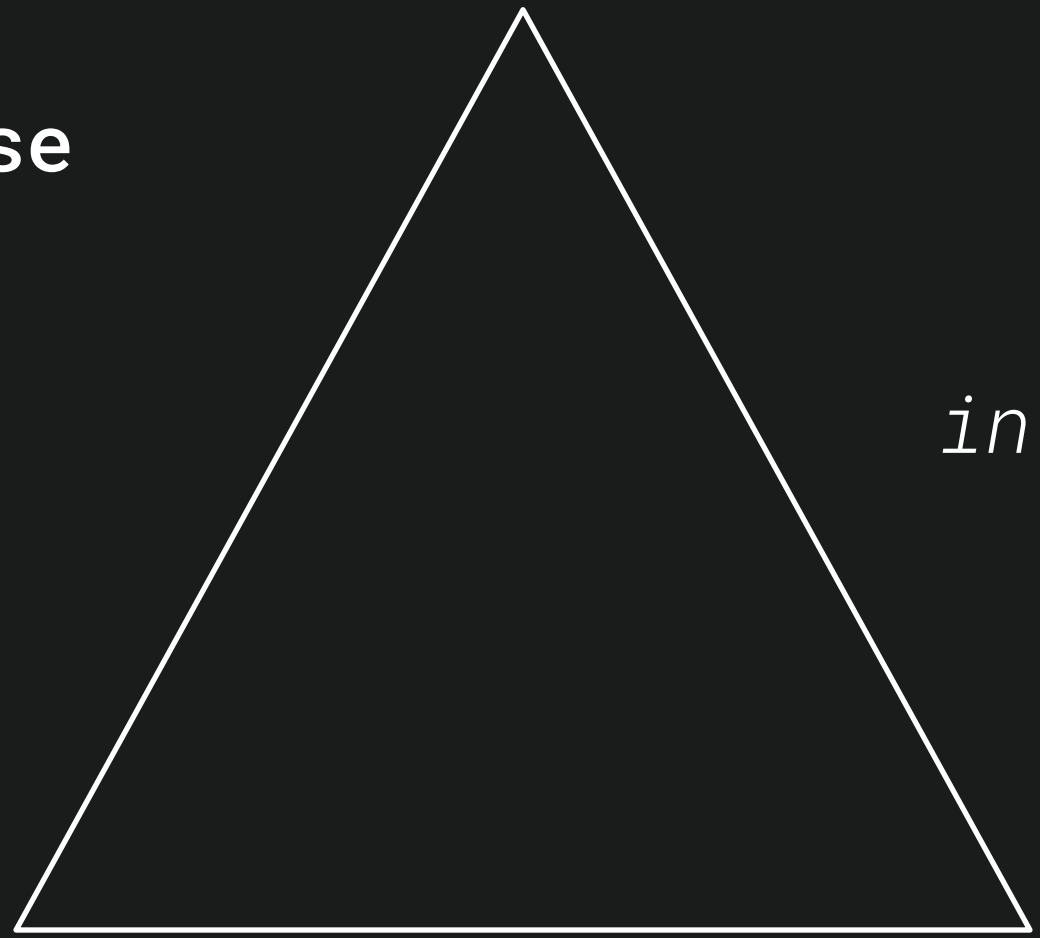
- **GCP Read-Write → Persistent Disk**
  - Offers high stability, but introduces **1-2ms latency** for disk operations, with **timeouts** occurring during disk reads.

Proposed Setup (TO-BE)

- Prioritize **Read performance** over **Write performance** for service speed.
- **Optimize Read speeds** by using **Local NVMe** (faster read performance but lower reliability).
- For **Write operations**, continue using **Persistent Disk** to ensure **data stability**.

Vector Database

Trilemma.



In-memory mode:
  Collection creation time: 0.1289 seconds
  Snapshot creation time: 0.7233 seconds

Disk mode:
  Collection creation time: 0.2587 seconds
  Snapshot creation time: 1.0139 seconds

Comparison:
  Collection speed-up (memory vs disk): 2.01x
  Snapshot speed-up (memory vs disk): 1.40x

In-memory mode:
  Collection creation time: 0.0835 seconds
  Snapshot creation time: 0.8281 seconds

Disk mode:
  Collection creation time: 0.2989 seconds
  Snapshot creation time: 1.1382 seconds

Comparison:
  Collection speed-up (memory vs disk): 3.58x
  Snapshot speed-up (memory vs disk): 1.37x

In-memory mode:
  Collection creation time: 0.0831 seconds
  Snapshot creation time: 0.7215 seconds

Disk mode:
  Collection creation time: 0.3550 seconds
  Snapshot creation time: 1.4362 seconds

Comparison:
  Collection speed-up (memory vs disk): 4.27x
  Snapshot speed-up (memory vs disk): 1.99x

Performance Metric	iSCSI Performance
Random Read 4KB, QD1	IOPS=4419, BW=17.3MiB/s
Random Read 4KB, QD32	IOPS=28.1k, BW=110MiB/s
Random Write 4KB, QD1	IOPS=4134, BW=16.1MiB/s
Random Write 4KB, QD32	IOPS=27.3k, BW=107MiB/s

https://lakefs.io/blog/tiers-in-the-cloud-how-lakefs-caches-immutable-data-on-local-disk/
https://github.com/seaweedfs/seaweedfs/wiki/Tiered-Storage
https://varunksaini.com/tiered-cache-in-go/

Tiered Storage

Glowry edited this page on Jul 18, 2023 · 20 revisions

All read and write operation for volume data is O(1) disk seek. However, there are fast, slow, and remote storages. Since data that are hot, warm, and cold, it would be cost-efficient to place data accordingly. SeaweedFS supports tiered storage, where you can place data to customizable disk types, and provides ways to move data to different tiers.

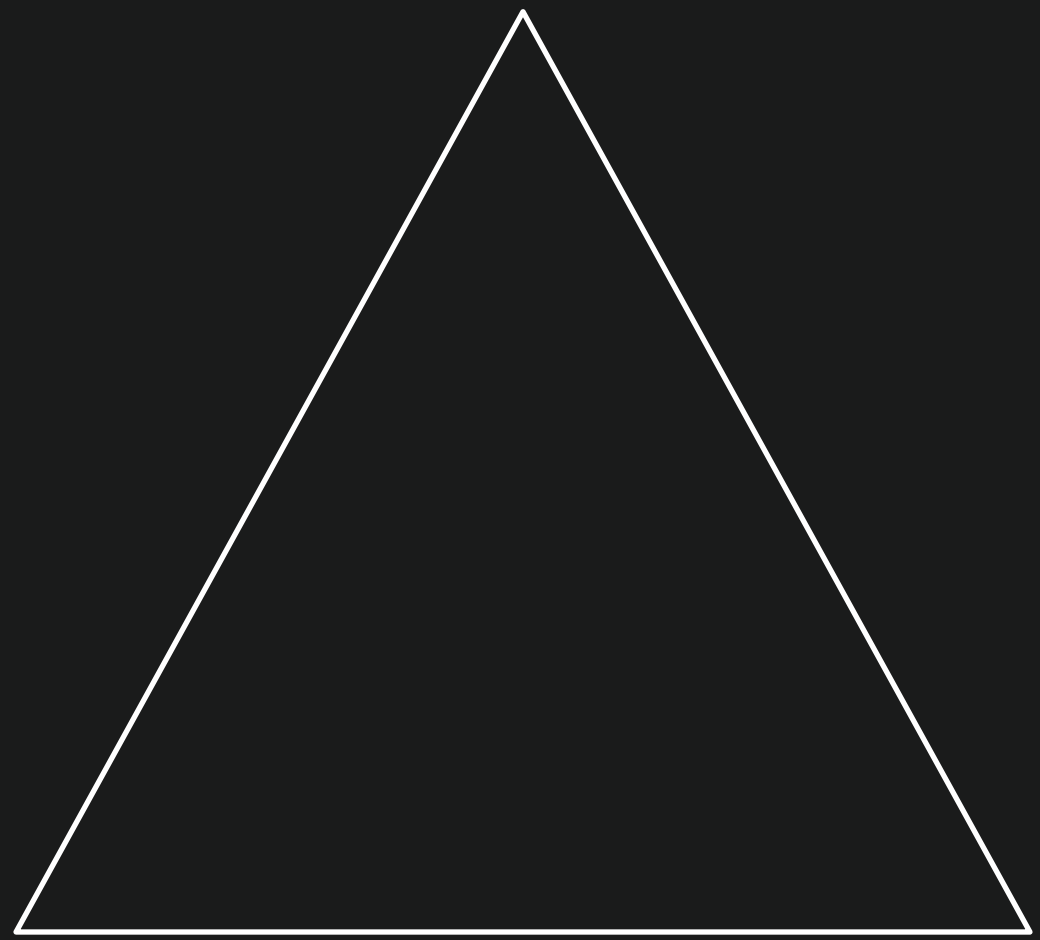
```
=> NVME      => SATA SSD => Fast HDD => Slow HDD => Cloud

=> Critical  => Hot      => Less Hot => Warm     => Cold
```

The service should be optimized using a tiered architecture for storage, considering its dynamics based on usage patterns and adjusts the type of storage depending on customer needs.

Tier 0: In-memory mode for high IOPS and cost efficiency.
Tier 1: Mid-level IOPS disks for balanced performance.
Tier 2: Low-level IOPS disks or cloud storage for cost-effective, long-term storage.

RAG Service

Trillemma.



- Cost Efficiency
  - Instead of providing one collection per customer, Qdrant minimizes memory usage by avoiding the creation of many small collections. Instead, data is segmented within a single large collection using metadata filters, which helps reduce the memory footprint required for embedding vectors.
- Performance
  - Creating multiple collections per customer can lead to performance degradation. By grouping data into a single collection and using metadata-based segmentation, Qdrant minimizes the number of collections, resulting in improved performance and faster search queries. This prevents the overhead associated with managing many small collections.
  - Instead of creating numerous small collections, Qdrant maintains performance by using one large collection with metadata filters. This reduces the overhead and enhances real-time search performance by avoiding the inefficiencies of handling multiple collections.
- Scalability
  - Using a single large collection with metadata filters improves scalability. As new customers or data are added, Qdrant can efficiently manage and search the data without the need to create additional collections, ensuring the system remains scalable even as data grows.

Operations

Building production RAG systems at scale (with 10s of millions users)

perplexity

Date	Time	Track	Room
Jun 26	10:45am - 11:05am	Expo Sessions	YBB Salon 13

Nikhil Thota

"- eng #2 at perplexity

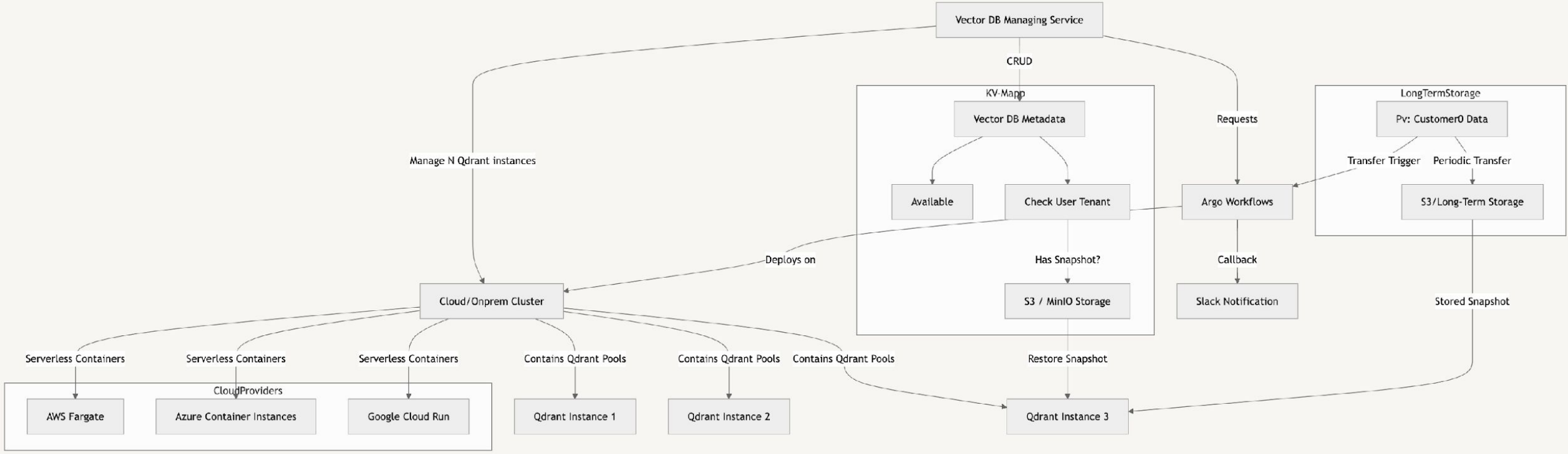
- ex-founder (personal search)
- senior SWE at whatsapp/meta"

Nikhil Thota, Member of Technical Staff

- Website
- LinkedIn
- Twitter

Argo Workflows is responsible for automating the management and deployment of pods across different stages and cloud platforms.

These workflows involve workers such as workflowWorkers, workflowTTLWorkers, and podCleanupWorkers that ensure efficient pod lifecycle management.



Examples

```go
package main

import (
    "fmt"
    "log"
    "sync"
    "time"

    "github.com/kelindar/event"
)

// QdrantEvent represents the event structure for Qdrant actions.
type QdrantEvent struct {
    EventType string
    AgentID   string
    Endpoint  string
}

// Event types
const (
    EventAllocateAgent = "AllocateAgent"
    EventPoolUpdated   = "PoolUpdated"
)
```

```go
var availablePool sync.Map
var allocatedAgents sync.Map

// QdrantService manages the in-memory pool and allocation of Qdrant instances.
type QdrantService struct {
    availablePool   *sync.Map
    allocatedAgents *sync.Map
    dispatcher      *event.Dispatcher
}

// NewQdrantService initializes a new service with an event bus.
func NewQdrantService() *QdrantService {
    return &QdrantService{
        availablePool:   &availablePool,
        allocatedAgents: &allocatedAgents,
        dispatcher:      event.NewDispatcher(),
    }
}

// AddToAvailablePool adds a new Qdrant endpoint to the available pool.
func (qs *QdrantService) AddToAvailablePool(endpointID string) {
    qs.availablePool.Store(endpointID, true)
    // Emit an event when the pool is updated
    event.Publish(qs.dispatcher, QdrantEvent{EventType: EventPoolUpdated, Endpoint
}
```

https://github.com/kelindar/event?tab=readme-ov-file#using-default-dispatcher

Examples

```go
// GetAvailablePool retrieves all available Qdrant instances.
func (qs *QdrantService) GetAvailablePool() []string {
    var pool []string
    qs.availablePool.Range(func(key, value interface{}) bool {
        pool = append(pool, key.(string))
        return true
    })
    return pool
}

// AllocateAgent assigns an available Qdrant instance to an agent.
func (qs *QdrantService) AllocateAgent(agentID string) error {
    var endpoint string
    allocated := false
    qs.availablePool.Range(func(key, value interface{}) bool {
        endpoint = key.(string)
        allocated = true
        return false // exit after finding the first available endpoint
    })

    if !allocated {
        return fmt.Errorf("no available Qdrant instances")
    }

    // Remove from available pool
    qs.availablePool.Delete(endpoint)
```

```go
    // Store the allocation details in the allocated agents map
    agentData := map[string]interface{}{
        "endpoint_id":    endpoint,
        "url":            fmt.Sprintf("http://%s-qdrant.[redacted], endpoint),
        "status":         "created",
        "created_at":     time.Now().Format(time.RFC3339),
        "last_checked_at": time.Now().Format(time.RFC3339),
        "key":            agentID,
    }
    qs.allocatedAgents.Store(agentID, agentData)

    // Emit an event when an agent is allocated
    event.Publish(qs.dispatcher, QdrantEvent{EventType: EventAllocateAgent, Agent
    return nil
}
```

```go
func main() {
    // Initialize the Qdrant service with event bus
    qdrantService := NewQdrantService()

    // Event subscribers
    event.Subscribe(qdrantService.dispatcher, func(ev QdrantEvent) {
        switch ev.EventType {
        case EventAllocateAgent:
            log.Printf("Allocated Qdrant instance %s to agent %s", ev.Endpoint, ev.
        case EventPoolUpdated:
            log.Printf("Qdrant instance %s added to the available pool", ev.Endpoir
        }
    })

    // Add Qdrant instances to the pool
    qdrantService.AddToAvailablePool("qdrant-1")
    qdrantService.AddToAvailablePool("qdrant-2")
    qdrantService.AddToAvailablePool("qdrant-3")

    // Fetch available Qdrant instances
    pool := qdrantService.GetAvailablePool()
    log.Printf("Available Qdrant endpoints: %v\n", pool)

    // Allocate an agent
    agentID := "agent-1234" // Example agent ID
    err := qdrantService.AllocateAgent(agentID)
    if err != nil {
        log.Fatalf("Failed to allocate agent: %v", err)
    }
}
```

The VectorDB Managing Service manages the pool of available Qdrant instances and the assignment of those instances to agents.

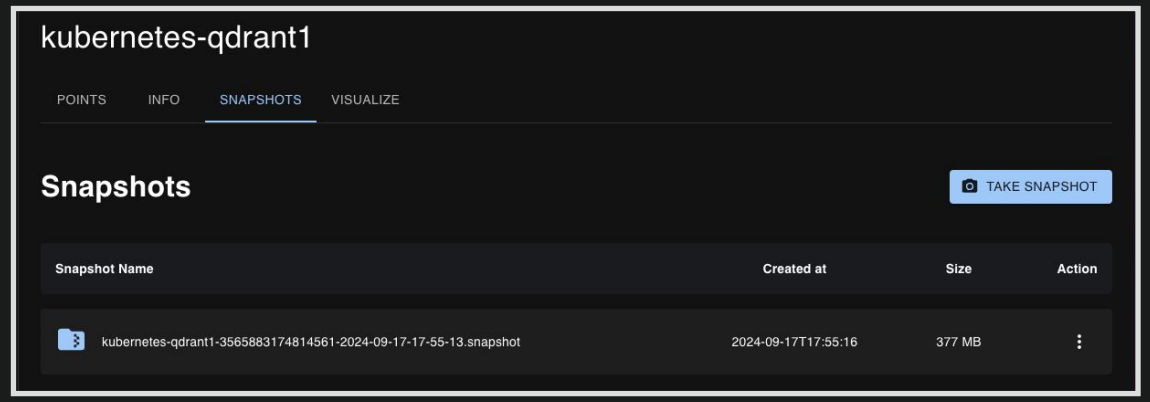
When a Qdrant instance is added to the pool or allocated to an agent, an event is emitted using the event bus - event.Publish().

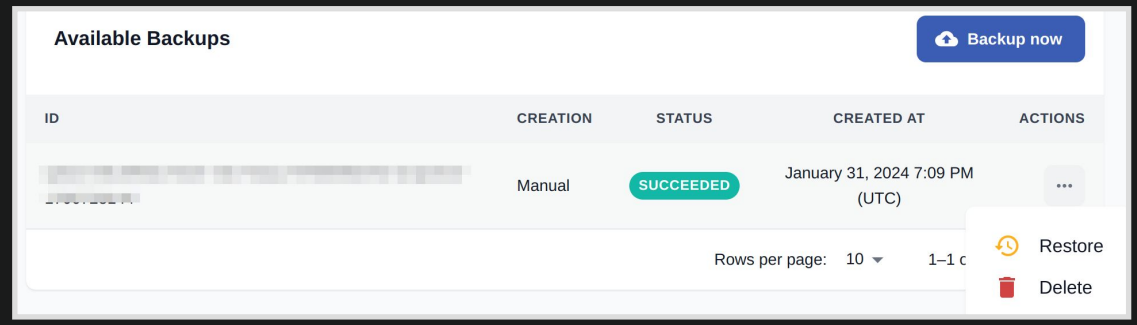
Event listeners - event.Subscribe() handle these events asynchronously, logging or performing additional actions.

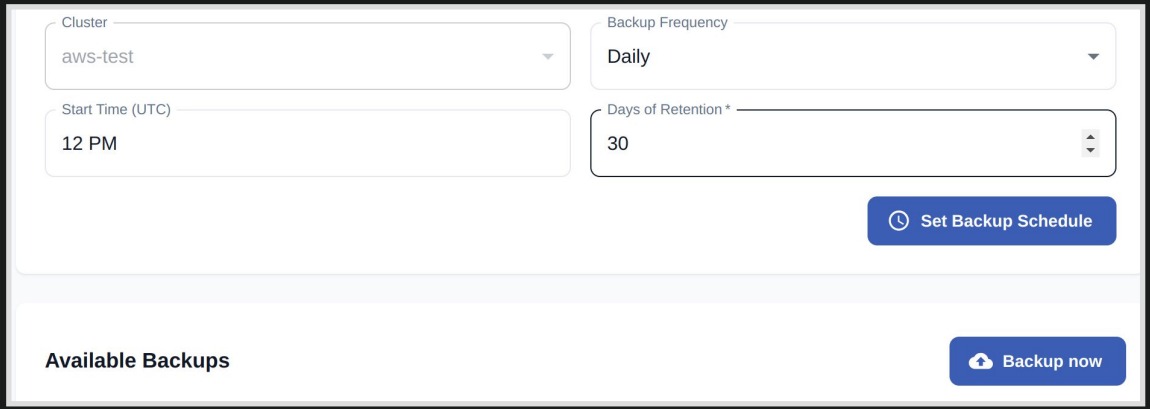
Backup Clusters & Restoring Snapshots

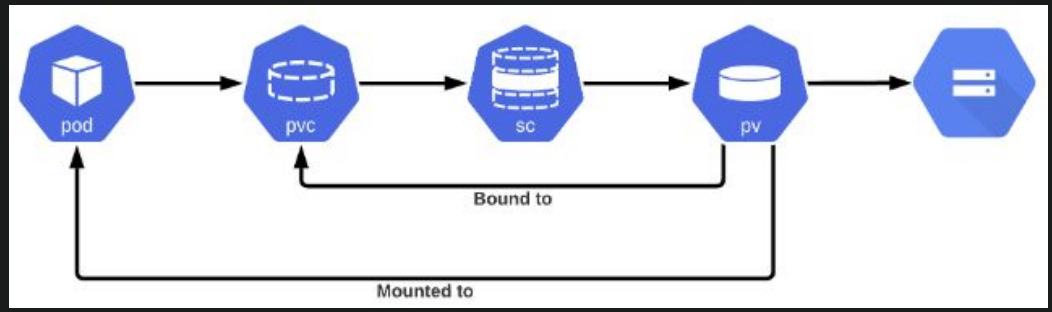
The vectors that customers actually use are all stored in memory.

When additional vectors are stored, a low-level snapshot is saved to disk, according to disk tiering.

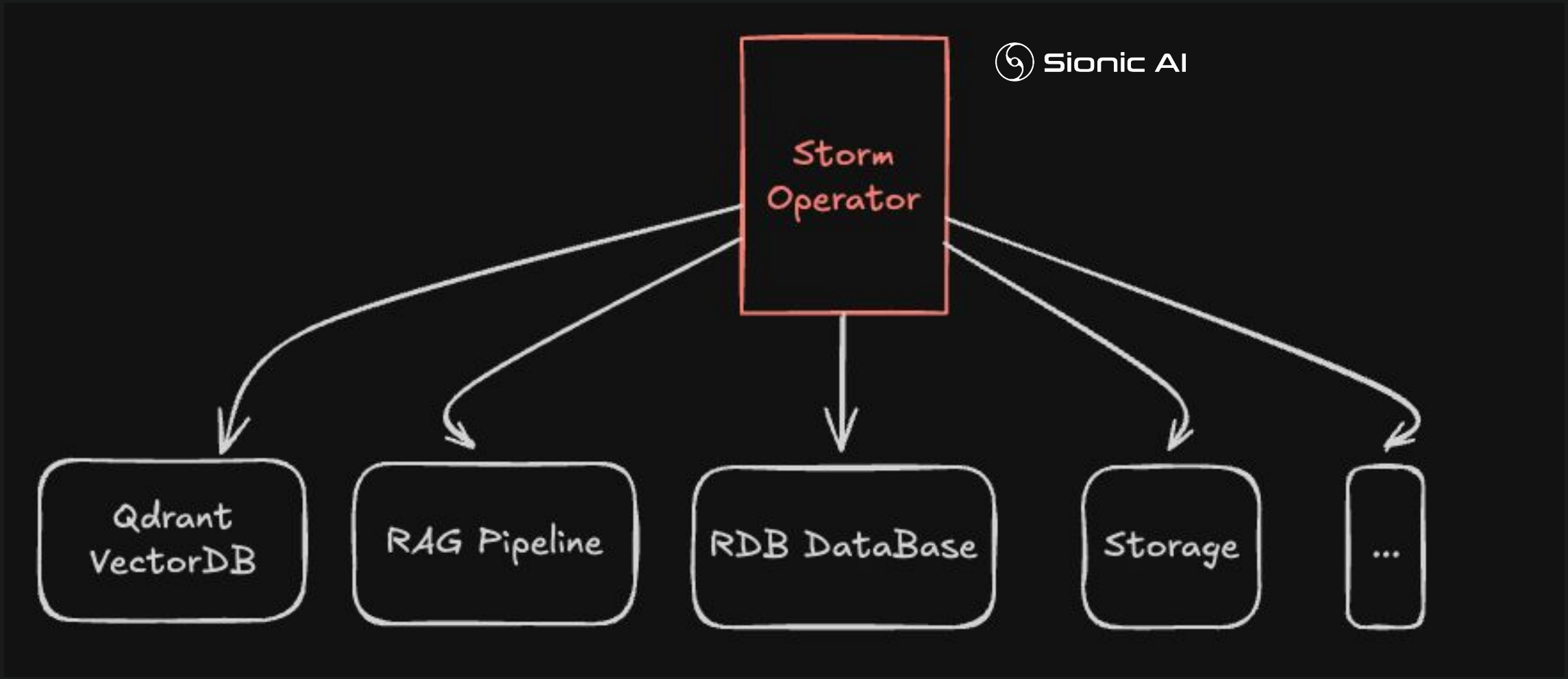




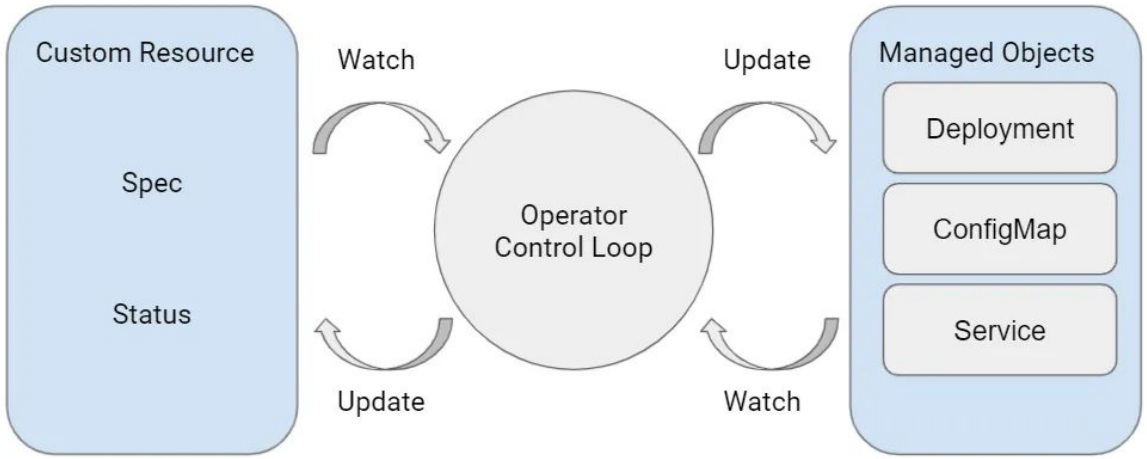


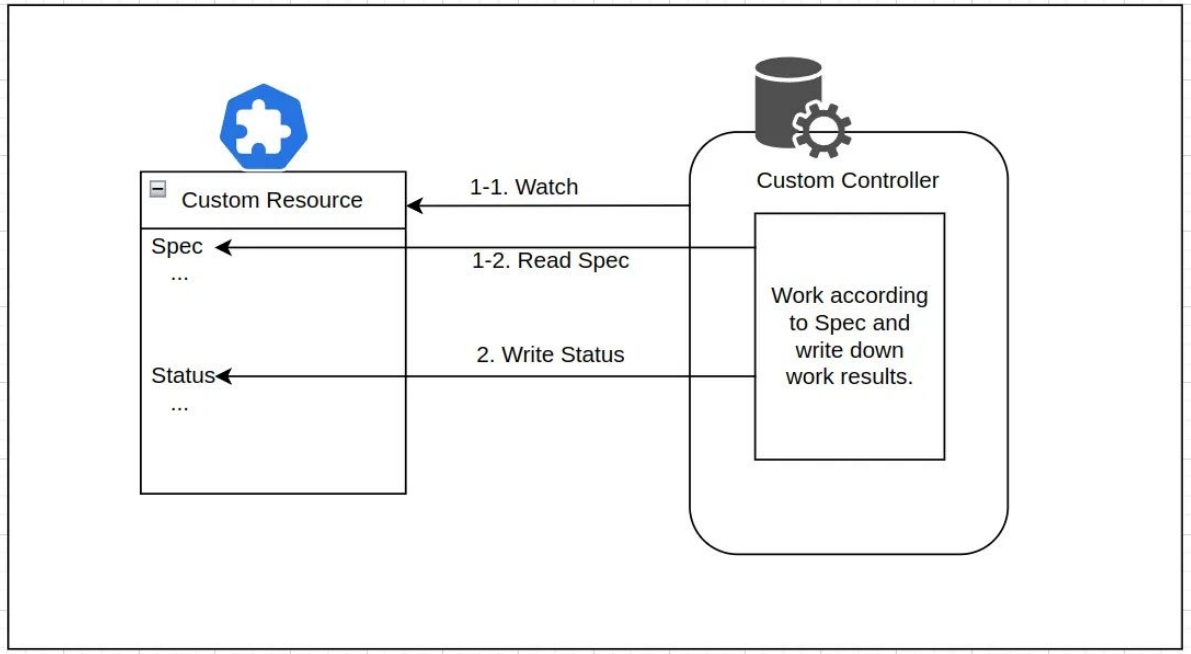


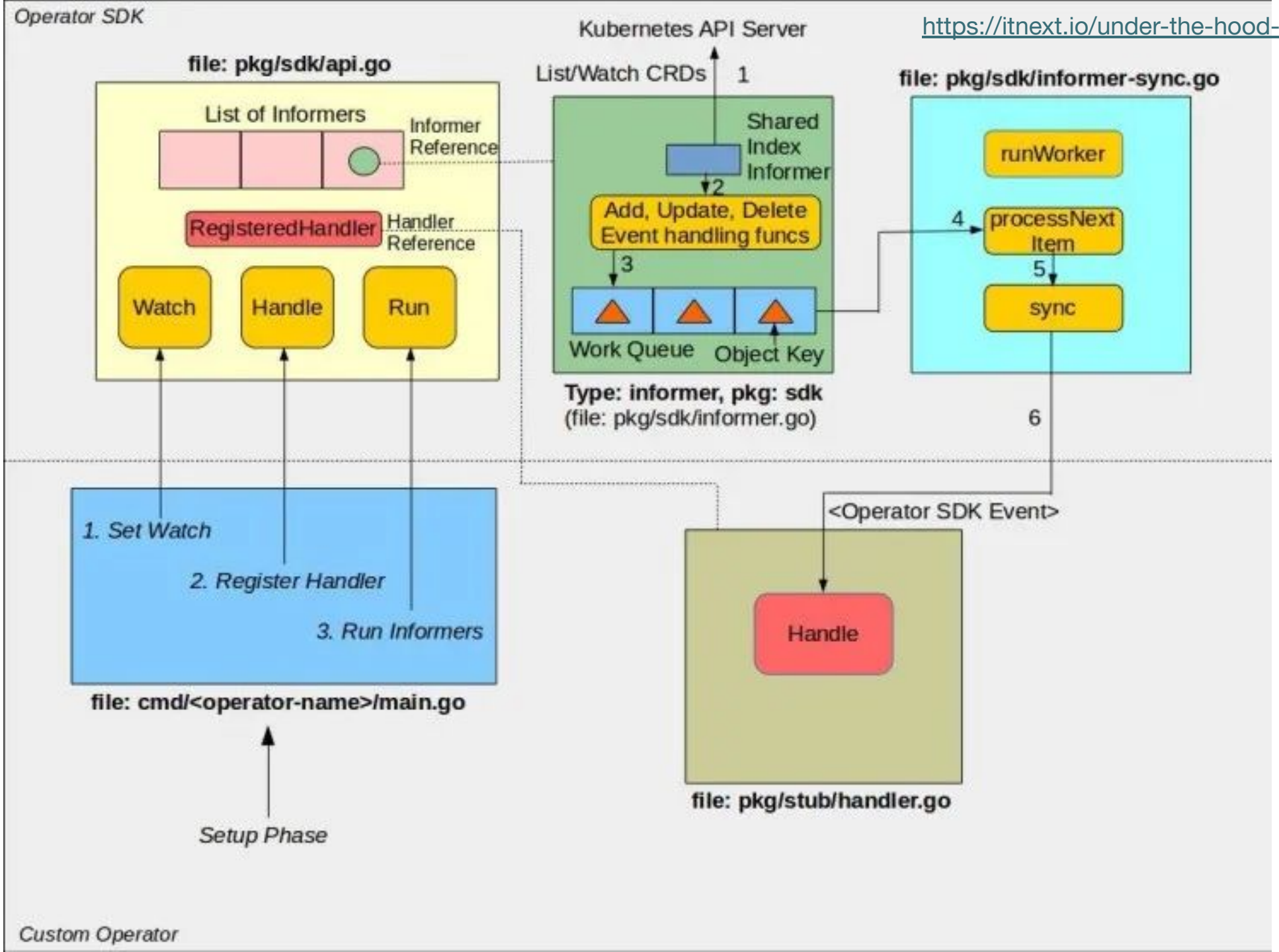
Kubernetes Custom Controller & Operator



What are Kubernetes Operators?






Operator SDK
Kubernetes API Server
https://itnext.io/under-the-hood-of-the-operator-sdk-eebc8fdeebbf
file: pkg/sdk/api.go
List/Watch CRDs
1
file: pkg/sdk/informer-sync.go
List of Informers
Informer Reference
Shared Index Informer
runWorker
2
Add, Update, Delete Event handling funcs
4
processNext Item
RegisteredHandler
Handler Reference
3
5
Watch
Handle
Run
sync
Work Queue
Object Key
Type: informer, pkg: sdk
(file: pkg/sdk/informer.go)
6
<Operator SDK Event>
1. Set Watch
2. Register Handler
3. Run Informers
Handle
file: cmd/<operator-name>/main.go
file: pkg/stub/handler.go
Setup Phase
Custom Operator

```
mkdir qdrant-operator
cd qdrant-operator
operator-sdk init --domain example.com --repo github.com/example/qdrant-operator
```

```
operator-sdk create api --group db --version v1alpha1 --kind Qdrant --resource --controller
```

```go
// QdrantSpec defines the desired state of Qdrant
type QdrantSpec struct {
    // Replicas specifies the number of Qdrant instances to run
    Replicas int `json:"replicas,omitempty"`

    // Image is the container image to use for the Qdrant deployment
    Image string `json:"image,omitempty"`
}

// QdrantStatus defines the observed state of Qdrant
type QdrantStatus struct {
    // Nodes are the names of the Qdrant pods
    Nodes []string `json:"nodes,omitempty"`
}
```

api/v1alpha1/qdrant_types.go

```go
package controllers

import (
    "context"
    "fmt"

    appsv1 "k8s.io/api/apps/v1"
    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "sigs.k8s.io/controller-runtime/pkg/controller/controllerutil"
    "sigs.k8s.io/controller-runtime/pkg/reconcile"
    ctrl "sigs.k8s.io/controller-runtime"

    dbv1alpha1 "github.com/example/qdrant-operator/api/v1alpha1"
)

// QdrantReconciler reconciles a Qdrant object
type QdrantReconciler struct {
    client.Client
    Scheme *runtime.Scheme
}
```

```go
// Reconcile is part of the main kubernetes reconciliation loop
func (r *QdrantReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    qdrant := &dbv1alpha1.Qdrant{}
    if err := r.Get(ctx, req.NamespacedName, qdrant); err != nil {
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    // Define the desired Deployment
    deployment := &appsv1.Deployment{
        ObjectMeta: metav1.ObjectMeta{
            Name:      qdrant.Name,
            Namespace: qdrant.Namespace,
        },
        Spec: appsv1.DeploymentSpec{
            Replicas: &qdrant.Spec.Replicas,
            Selector: &metav1.LabelSelector{
                MatchLabels: map[string]string{"app": qdrant.Name},
            },
            Template: corev1.PodTemplateSpec{
                ObjectMeta: metav1.ObjectMeta{
                    Labels: map[string]string{"app": qdrant.Name},
                },
                Spec: corev1.PodSpec{
                    Containers: []corev1.Container{
                        {
                            Name:  "qdrant",
                            Image: qdrant.Spec.Image,
                            Ports: []corev1.ContainerPort{{ContainerPort: 6333}},
                        },
                    },
                },
            },
        },
    }
```

controllers/qdrant_controller.go

```
    // Set the owner reference to Qdrant CR
    if err := controllerutil.SetControllerReference(qdrant, deployment, r.Scheme); err != nil {
        return ctrl.Result{}, err
    }

    // Check if the Deployment exists, if not create it
    found := &appsv1.Deployment{}
    err := r.Get(ctx, req.NamespacedName, found)
    if err != nil && client.IgnoreNotFound(err) == nil {
        return ctrl.Result{}, err
    } else if err != nil {
        fmt.Println("Creating Deployment", "Namespace", deployment.Namespace, "Name", deployment.Name)
        if err := r.Create(ctx, deployment); err != nil {
            return ctrl.Result{}, err
        }
    }

    return ctrl.Result{}, nil
}
```

```
// SetupWithManager sets up the controller with the Manager
func (r *QdrantReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&dbv1alpha1.Qdrant{}).
        Complete(r)
}
```

controllers/qdrant_controller.go

```
make generate
make manifests
```

```
kubectl apply -f config/crd/bases/db.example.com_qdrants.yaml
```

Generate and Apply your CRD to clusters.

```yaml
apiVersion: db.example.com/v1alpha1
kind: Qdrant
metadata:
  name: qdrant-sample
spec:
  replicas: 3
  image: "qdrant/qdrant:v1.11.0"
```

Create a qdrant-sample.yaml manifest file before applying to your clusters.

```
qdrant-operator/
├── Makefile
├── PROJECT
├── config
│   ├── crd
│   │   ├── bases
│   │   │   └── db.example.com_qdrants.yaml
│   ├── default
│   │   ├── kustomization.yaml
│   ├── manager
│   │   ├── kustomization.yaml
│   │   └── manager.yaml
│   ├── rbac
│   │   ├── qdrant_editor_role.yaml
│   │   ├── qdrant_viewer_role.yaml
│   │   ├── role_binding.yaml
│   │   └── role.yaml
│   ├── samples
│   │   └── qdrant_v1alpha1_qdrant.yaml
│   ├── testing
│   │   └── kustomization.yaml
│   └── webhook
│       ├── kustomization.yaml
│       └── service.yaml
├── controllers
│   └── qdrant_controller.go
├── api
│   └── v1alpha1
│       ├── groupversion_info.go
│       ├── qdrant_types.go
│       └── zz_generated.deepcopy.go
├── bin
│   └── controller-gen
├── hack
│   └── boilerplate.go.txt
├── go.mod
├── go.sum
├── Dockerfile
└── README.md
```

https://sdk.operatorframework.io/docs/building-operators/golang/

https://careers.sionic.ai https://sionic.ai

Thank
You!