Weaviate

Solving multi-tenancy in vector search requires a paradigm shift

The AI Conference, Sep 26th, 2023

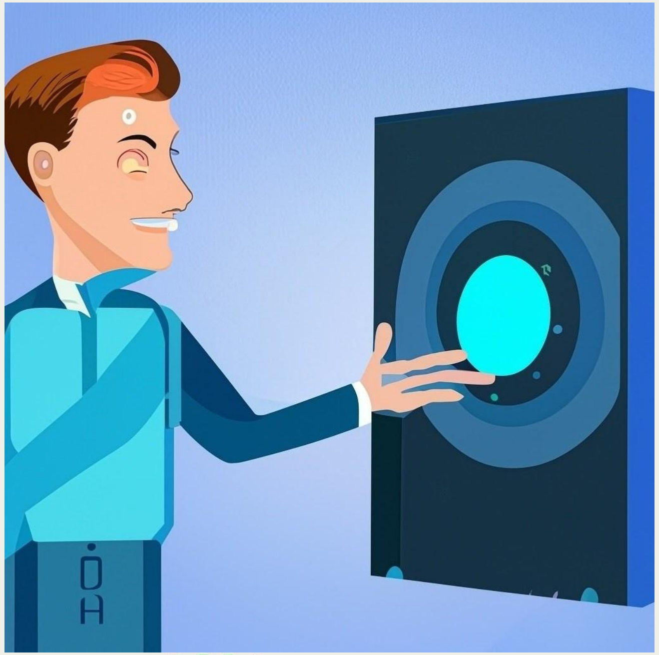
Etienne Dilocker
Co-Founder & CTO

Multi-Tenancy!?

Why should we care?

Imagine an app that lets you upload all the files on your hard drive and chat with it.

You wouldn’t want someone else to chat with *your* files.



As the vendor of this chatbot app:

Why should you pay for all your potential users if only 5% are active right now?

Previous multi-tenancy solutions made you pay for all, not for active tenants.

Prioritization is hard.

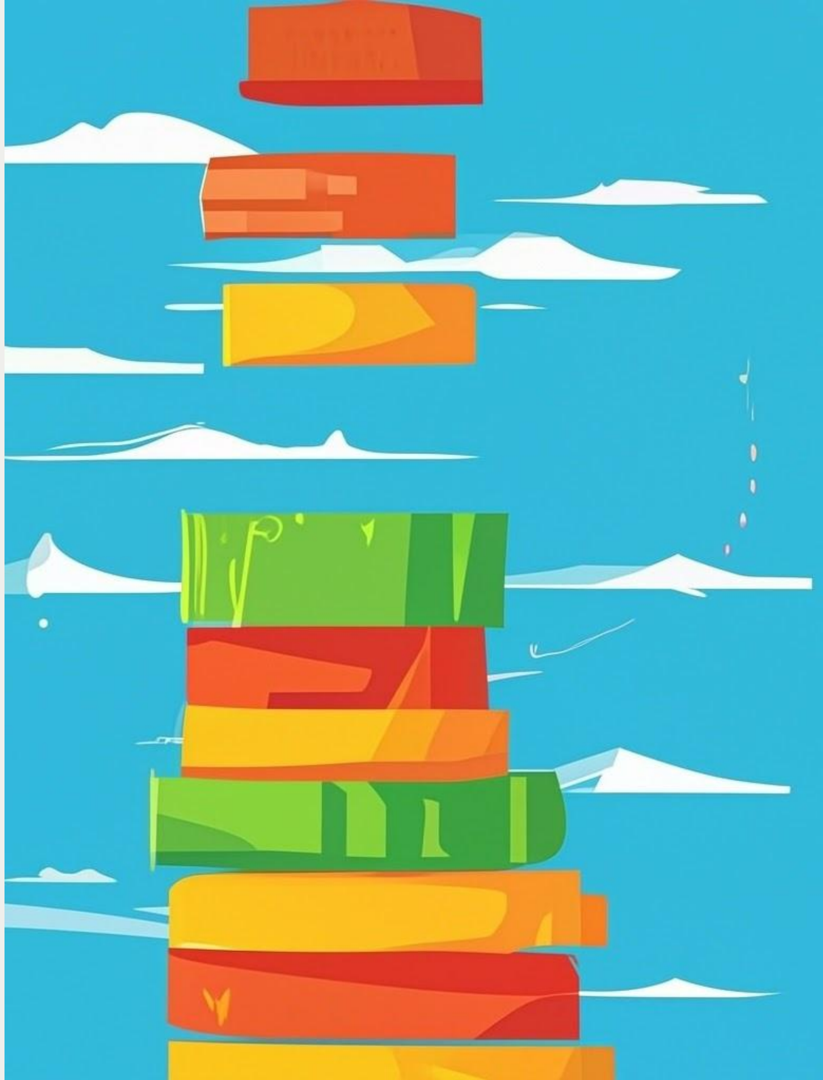
But sometimes it’s obvious what you need to do.

Commercial Customers:

- “We need **strong isolation** between tenants for security and compliance (GDPR, etc.)”
- “We need to **reduce cost**.”

Engineers:

- “We need to add **hundreds of thousands of tenants** and there is no solution that can do this right now.”



Open-Source Community:

"I'm using workarounds with Weaviate and other vector DBs and they have limitations.

Can you address the root cause?"

Why existing solutions failed

Attempt Number 1

The Filter

```
SELECT AWESOMENESS
FROM VECTORDB
WHERE TENANT_ID=12345
```

Why is single index with filters not ideal?

No isolation

Difficult to scale dynamically

Inefficient

Very expensive off-boarding
(tenant deletion)

Attempt Number 2

Create one collection per tenant

(The recommended way in Weaviate in the past)

Solves some of the previous shortcomings

- Strict isolation
- No filter necessary
- Easy off-boarding
- Good scalability

Solves some of the previous shortcomings

Strict isolation

No filter necessary

Easy off-boarding

(In theory) Good scalability

But isn’t the right solution either

Does not scale to millions of tenants

👍 data scheduling scales well

👎 schema does not

Lots of duplication

Terrible MTTR

How can we make this better?

Let’s make tenants first-class citizens!

Design goals

Millions of tenants
in a single cluster

Resource isolation

Strict separation
for GDPR etc

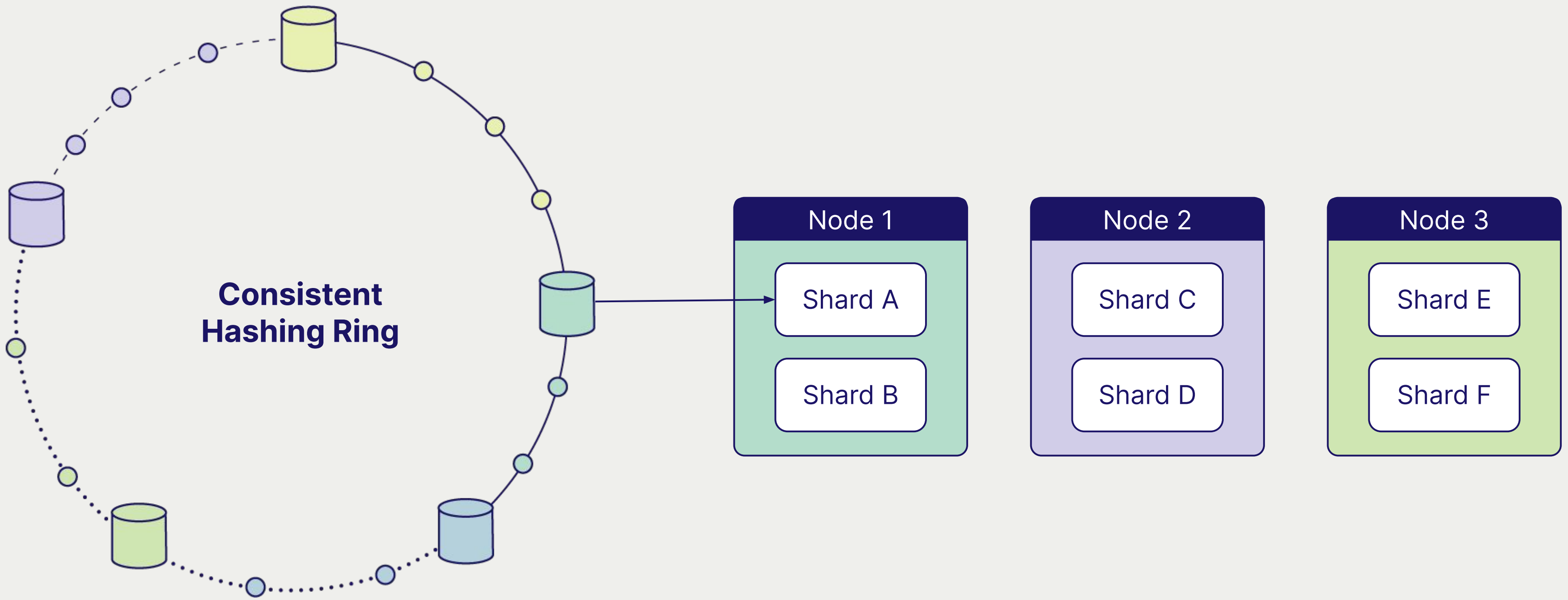
Simple offboarding

Linear scalability
Want more tenants?
Add more nodes!

Only pay for active tenants

Sharding would likely play a key role.

Sharding in a single-tenancy situation



What if we took this to the extreme and created a single shard per tenant?

Pros

Strict Isolation

separate storage, separate indexes, resource isolation

Schema is defined just once

All shards already belong to the same collection

Cheap and easy tenant offboardings

Deleting a tenant is deleting an entire shard (cheap)

Potential Cons

Ring-hashing no longer fits

It’s meant to distribute many keys to fewer shards

Potential for exploding cost

What is the cost of an empty shard?

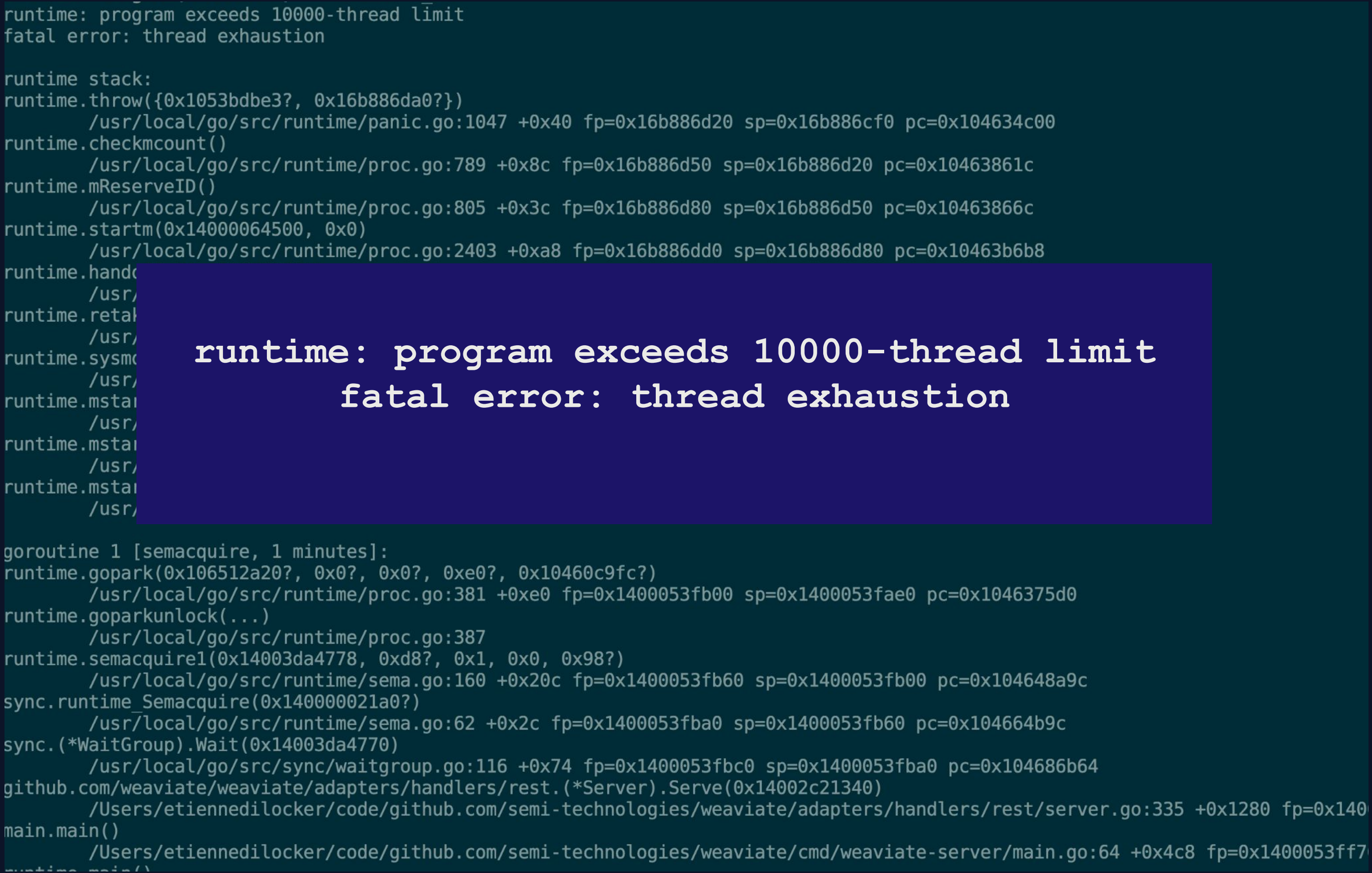
What if you have thousands or millions of them?

Performance Degradation

SURELY, IT WOULDN'T BE THAT BAD. RIGHT?
RIGHT?

5000 shards

Does it run at all?



```
runtime: program exceeds 10000-thread limit
fatal error: thread exhaustion

runtime stack:
runtime.throw({0x1053bdbe3?, 0x16b886da0?})
        /usr/local/go/src/runtime/panic.go:1047 +0x40 fp=0x16b886d20 sp=0x16b886cf0 pc=0x104634c00
runtime.checkmcount()
        /usr/local/go/src/runtime/proc.go:789 +0x8c fp=0x16b886d50 sp=0x16b886d20 pc=0x10463861c
runtime.mReserveID()
        /usr/local/go/src/runtime/proc.go:805 +0x3c fp=0x16b886d80 sp=0x16b886d50 pc=0x10463866c
runtime.startm(0x14000064500, 0x0)
        /usr/local/go/src/runtime/proc.go:2403 +0xa8 fp=0x16b886dd0 sp=0x16b886d80 pc=0x10463b6b8
runtime.handc
        /usr/
runtime.retak
        /usr/
runtime.sysmc
        /usr/
runtime.mstar
        /usr/
runtime.mstar
        /usr/
runtime.mstar
        /usr/

goroutine 1 [semacquire, 1 minutes]:
runtime.gopark(0x106512a20?, 0x0?, 0x0?, 0xe0?, 0x10460c9fc?)
        /usr/local/go/src/runtime/proc.go:381 +0xe0 fp=0x1400053fb00 sp=0x1400053fae0 pc=0x1046375d0
runtime.goparkunlock(...)
        /usr/local/go/src/runtime/proc.go:387
runtime.semacquire1(0x14003da4778, 0xd8?, 0x1, 0x0, 0x98?)
        /usr/local/go/src/runtime/sema.go:160 +0x20c fp=0x1400053fb60 sp=0x1400053fb00 pc=0x104648a9c
sync.runtime_Semacquire(0x140000021a0?)
        /usr/local/go/src/runtime/sema.go:62 +0x2c fp=0x1400053fba0 sp=0x1400053fb60 pc=0x104664b9c
sync.(*WaitGroup).Wait(0x14003da4770)
        /usr/local/go/src/sync/waitgroup.go:116 +0x74 fp=0x1400053fbc0 sp=0x1400053fba0 pc=0x104686b64
github.com/weaviate/weaviate/adapters/handlers/rest.(*Server).Serve(0x14002c21340)
        /Users/etiennedilocker/code/github.com/semi-technologies/weaviate/adapters/handlers/rest/server.go:335 +0x1280 fp=0x140
main.main()
        /Users/etiennedilocker/code/github.com/semi-technologies/weaviate/cmd/weaviate-server/main.go:64 +0x4c8 fp=0x1400053ff7
```

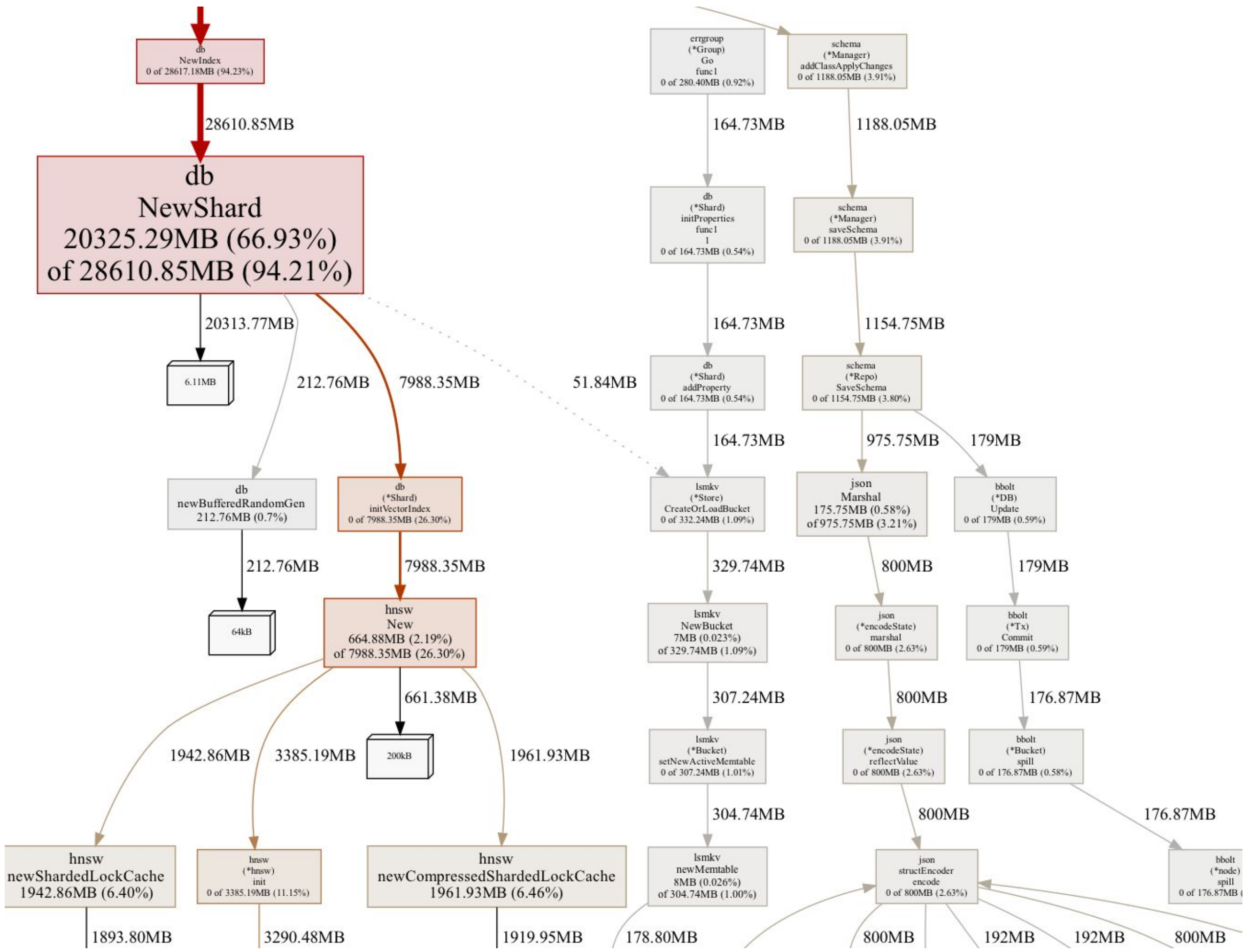
```
runtime: program exceeds 10000-thread limit
fatal error: thread exhaustion
```

How expensive would it be?

28GB for 5,000 shards 😱

5.6MB per shard

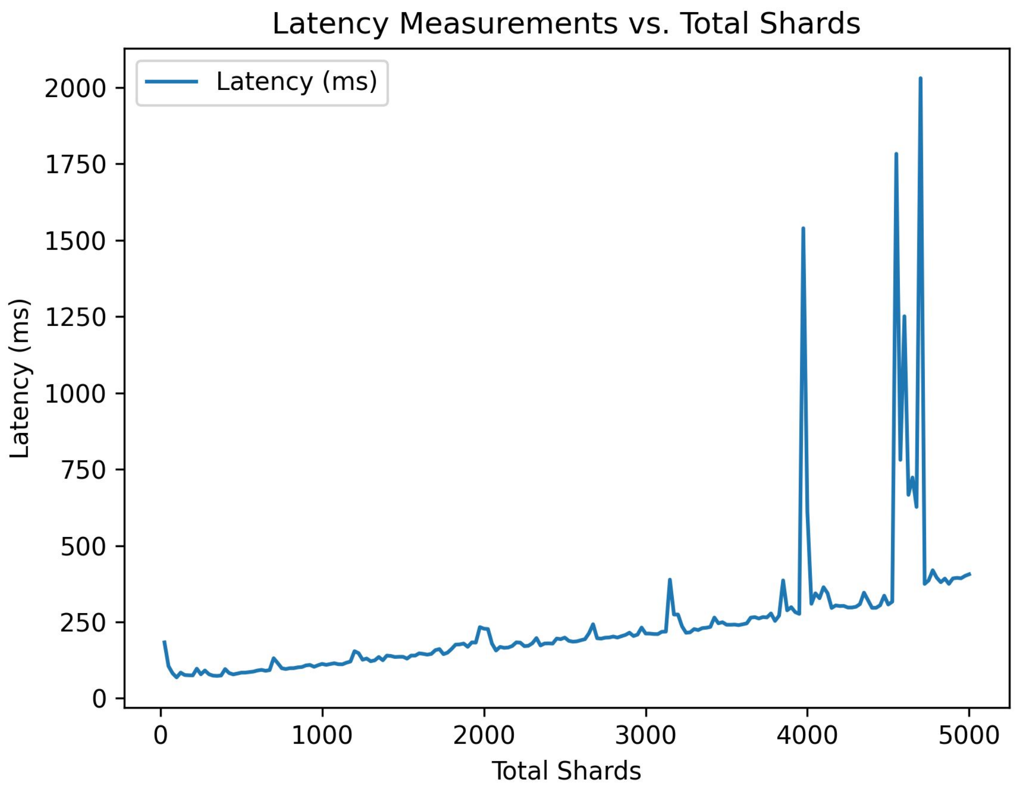
1M shards → 5.6TB 😅



db NewIndex 0 of 28617.18MB (94.23%)
28610.85MB
db NewShard 20325.29MB (66.93%) of 28610.85MB (94.21%)
20313.77MB
6.11MB
212.76MB
7988.35MB
51.84MB
db newBufferedRandomGen 212.76MB (0.7%)
212.76MB
64kB
db (*Shard) initVectorIndex 0 of 7988.35MB (26.30%)
7988.35MB
hnsw New 664.88MB (2.19%) of 7988.35MB (26.30%)
1942.86MB
3385.19MB
661.38MB
200kB
1961.93MB
hnsw newShardedLockCache 1942.86MB (6.40%)
1893.80MB
hnsw (*hnsw) init 0 of 3385.19MB (11.15%)
3290.48MB
hnsw newCompressedShardedLockCache 1961.93MB (6.46%)
1919.95MB
errgroup (*Group) Go func1 0 of 280.40MB (0.92%)
164.73MB
db (*Shard) initProperties func1 1 0 of 164.73MB (0.54%)
164.73MB
db (*Shard) addProperty 0 of 164.73MB (0.54%)
164.73MB
lsmkv (*Store) CreateOrLoadBucket 0 of 332.24MB (1.09%)
329.74MB
lsmkv NewBucket 7MB (0.023%) of 329.74MB (1.09%)
307.24MB
lsmkv (*Bucket) setNewActiveMemtable 0 of 307.24MB (1.01%)
304.74MB
lsmkv newMemtable 8MB (0.026%) of 304.74MB (1.00%)
178.80MB
schema (*Manager) addClassApplyChanges 0 of 1188.05MB (3.91%)
1188.05MB
schema (*Manager) saveSchema 0 of 1188.05MB (3.91%)
1154.75MB
schema (*Repo) SaveSchema 0 of 1154.75MB (3.80%)
975.75MB
179MB
json Marshal 175.75MB (0.58%) of 975.75MB (3.21%)
800MB
json (*encodeState) marshal 0 of 800MB (2.63%)
800MB
json (*encodeState) reflectValue 0 of 800MB (2.63%)
800MB
json structEncoder encode 0 of 800MB (2.63%)
800MB
192MB
192MB
800MB
bbolt (*DB) Update 0 of 179MB (0.59%)
179MB
bbolt (*Tx) Commit 0 of 179MB (0.59%)
176.87MB
bbolt (*Bucket) spill 0 of 176.87MB (0.58%)
176.87MB
bbolt (*node) spill 0 of 176.87MB

Would performance degrade?

Creating shards in batches of 25



Workarounds don’t scale.

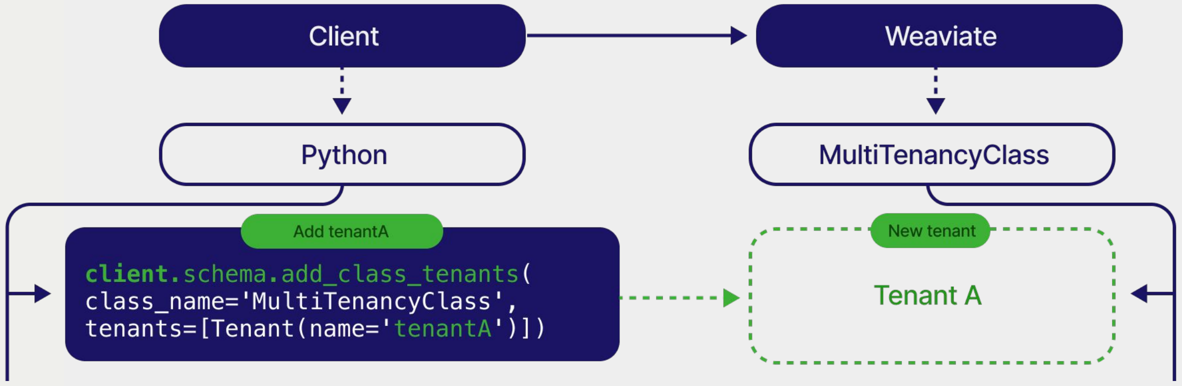
These are all addressable problems, but it shows you need an actual Multi-Tenancy solution.

Weaviate 1.19 introduces
Native Multi-Tenancy Support

Simple API

Specify the tenant with each request.

That's it.

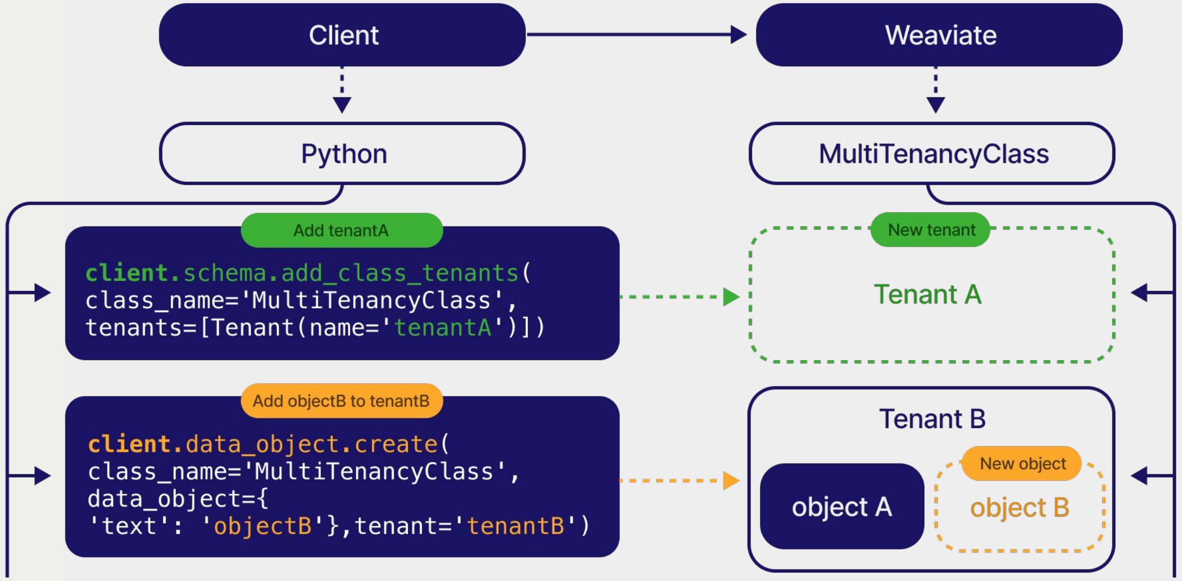


```
client.schema.add_class_tenants(
class_name='MultiTenancyClass',
tenants=[Tenant(name='tenantA')])
```

Simple API

Specify the tenant with each request.

That’s it.



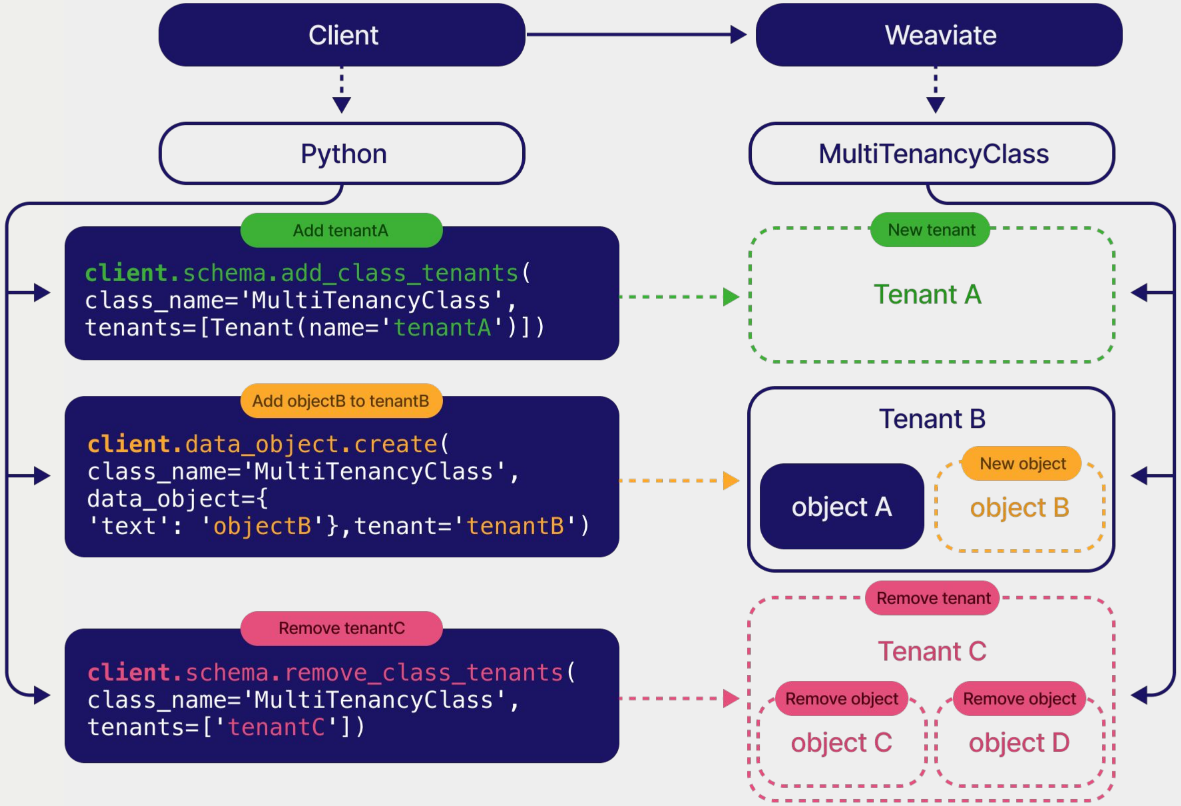
```
client.schema.add_class_tenants(
class_name='MultiTenancyClass',
tenants=[Tenant(name='tenantA')])
```

```
client.data_object.create(
class_name='MultiTenancyClass',
data_object={
'text': 'objectB'},tenant='tenantB')
```

Simple API

Specify the tenant with each request.

That’s it.

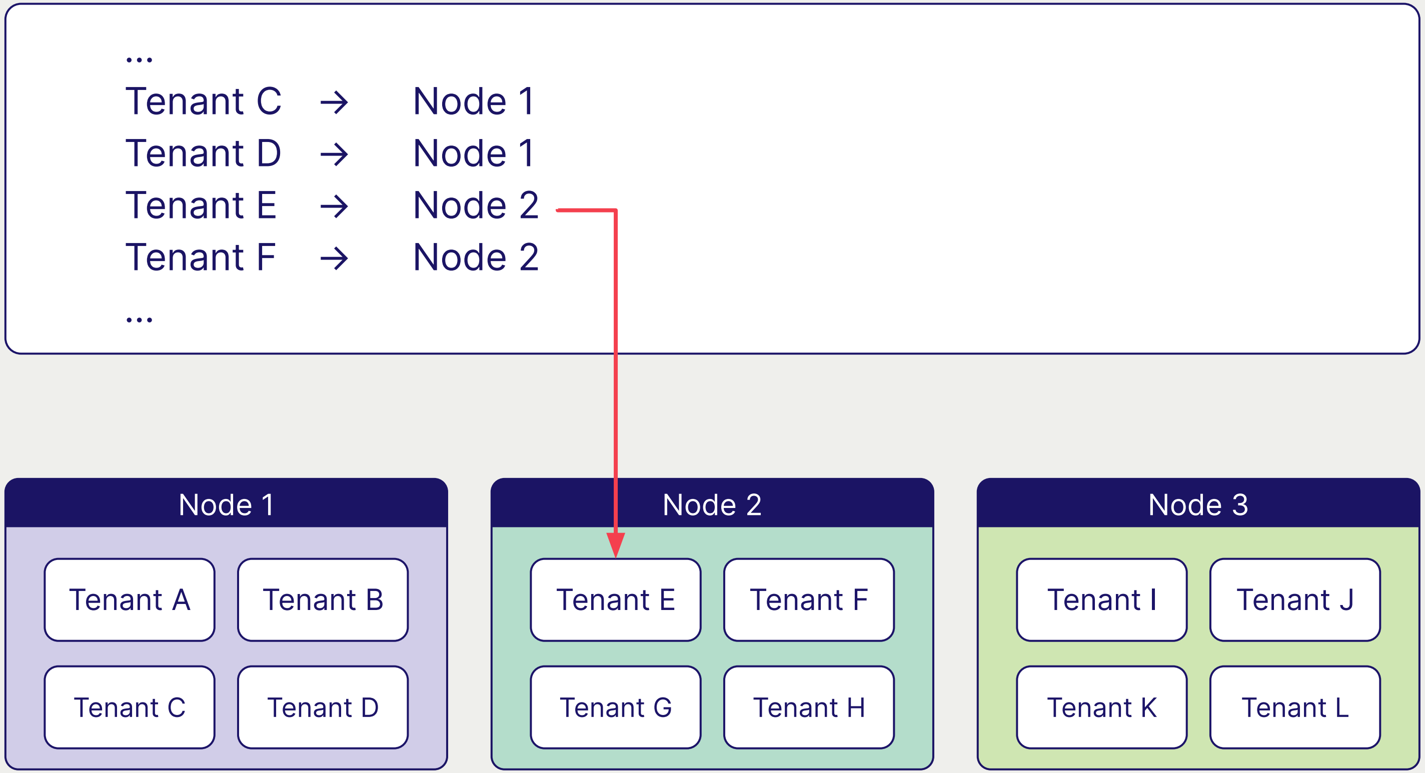


```
client.schema.add_class_tenants(
class_name='MultiTenancyClass',
tenants=[Tenant(name='tenantA')])
```

```
client.data_object.create(
class_name='MultiTenancyClass',
data_object={
'text': 'objectB'},tenant='tenantB')
```

```
client.schema.remove_class_tenants(
class_name='MultiTenancyClass',
tenants=['tenantC'])
```

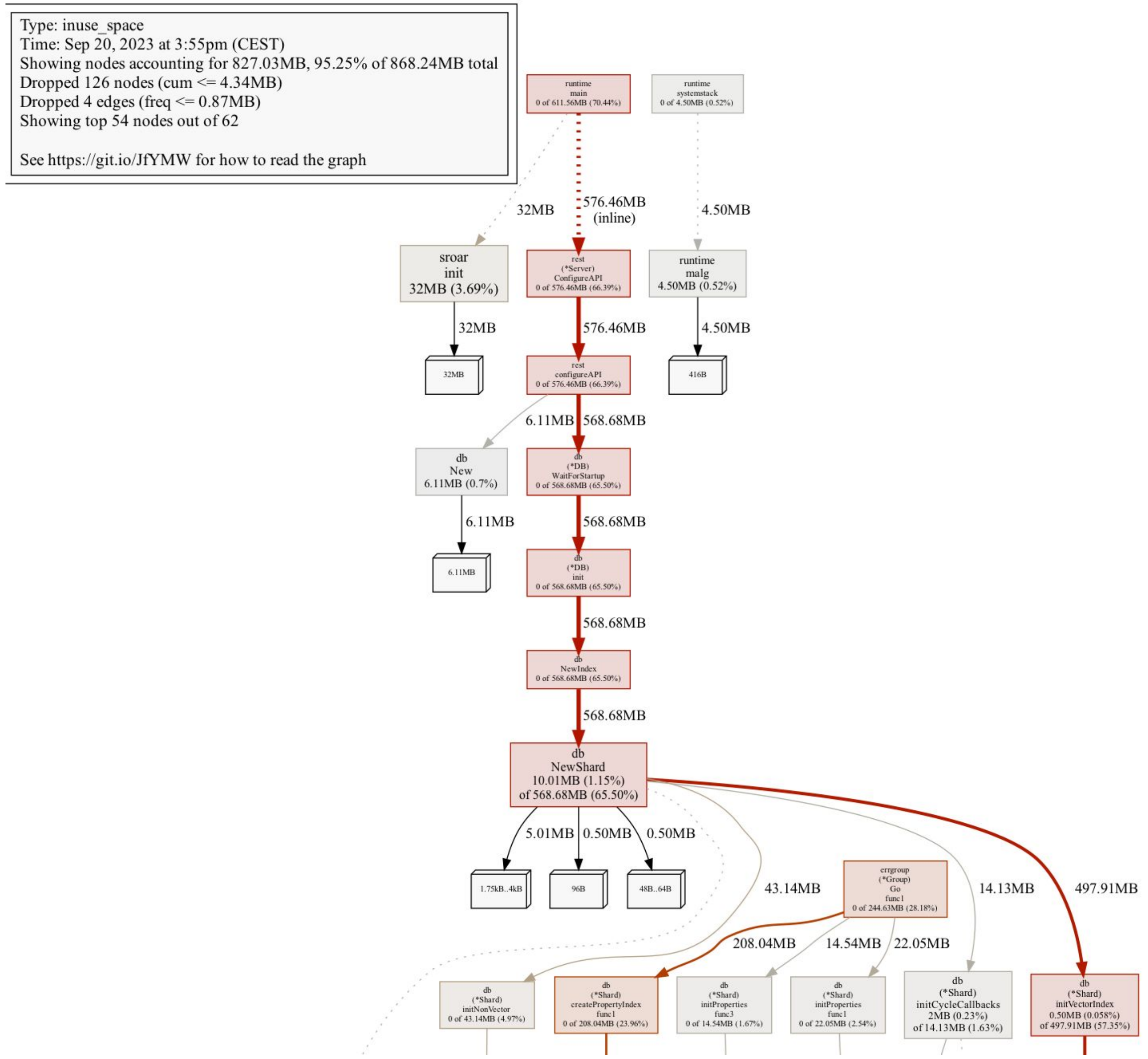
Shard routing is a simple (replicated) lookup list



870MB for 5,000 shards

174KB per shard

1M shards -> 174GB



```
Type: inuse_space
Time: Sep 20, 2023 at 3:55pm (CEST)
Showing nodes accounting for 827.03MB, 95.25% of 868.24MB total
Dropped 126 nodes (cum <= 4.34MB)
Dropped 4 edges (freq <= 0.87MB)
Showing top 54 nodes out of 62

See https://git.io/JfYMW for how to read the graph
```

runtime main 0 of 611.56MB (70.44%)
runtime systemstack 0 of 4.50MB (0.52%)
32MB
576.46MB (inline)
4.50MB
sroar init 32MB (3.69%)
rest (*Server) ConfigureAPI 0 of 576.46MB (66.39%)
runtime malg 4.50MB (0.52%)
32MB
576.46MB
4.50MB
32MB
rest configureAPI 0 of 576.46MB (66.39%)
416B
6.11MB
568.68MB
db New 6.11MB (0.7%)
db (*DB) WaitForStartup 0 of 568.68MB (65.50%)
6.11MB
568.68MB
6.11MB
db (*DB) init 0 of 568.68MB (65.50%)
568.68MB
db NewIndex 0 of 568.68MB (65.50%)
568.68MB
db NewShard 10.01MB (1.15%) of 568.68MB (65.50%)
5.01MB
0.50MB
0.50MB
1.75kB..4kB
96B
48B..64B
43.14MB
errgroup (*Group) Go func1 0 of 244.63MB (28.18%)
14.13MB
497.91MB
208.04MB
14.54MB
22.05MB
db (*Shard) initNonVector 0 of 43.14MB (4.97%)
db (*Shard) createPropertyIndex func1 0 of 208.04MB (23.96%)
db (*Shard) initProperties func3 0 of 14.54MB (1.67%)
db (*Shard) initProperties func1 0 of 22.05MB (2.54%)
db (*Shard) initCycleCallbacks 2MB (0.23%) of 14.13MB (1.63%)
db (*Shard) initVectorIndex 0.50MB (0.058%) of 497.91MB (57.35%)

Performance no longer degrades when adding tenants. Creating shards in batches of 25

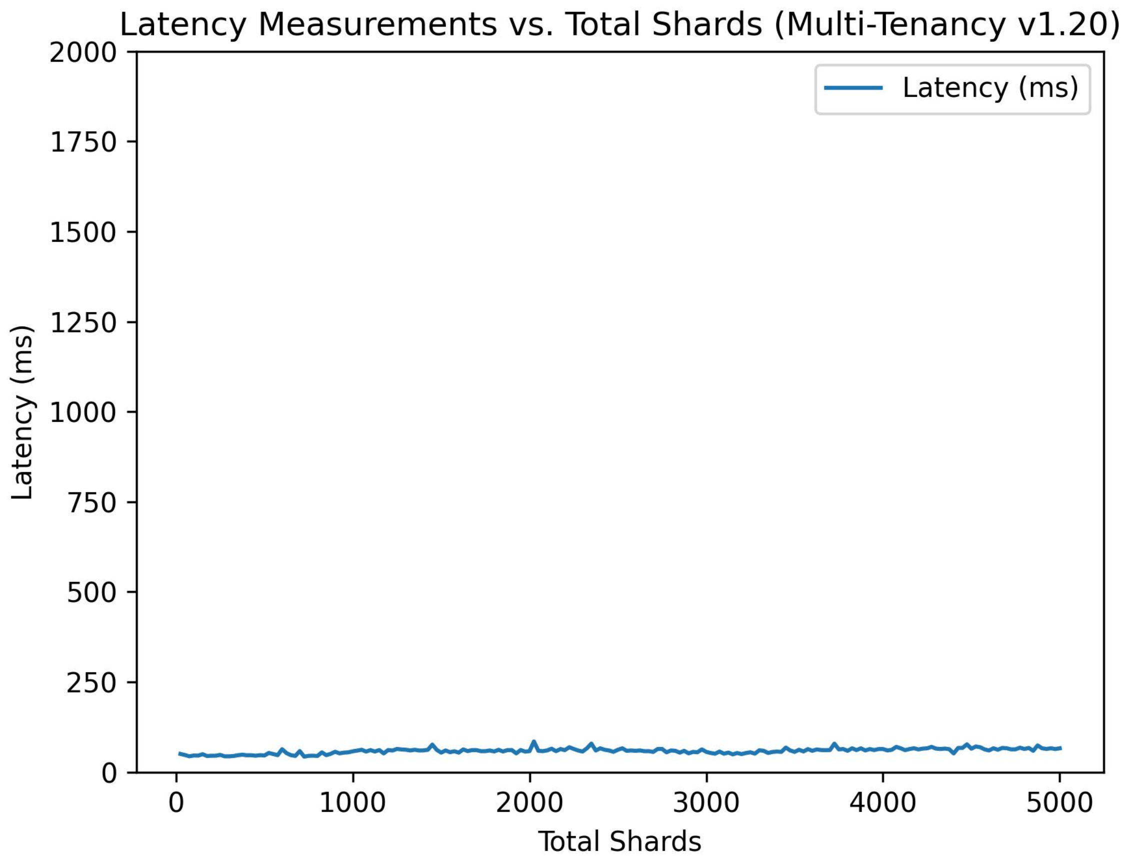
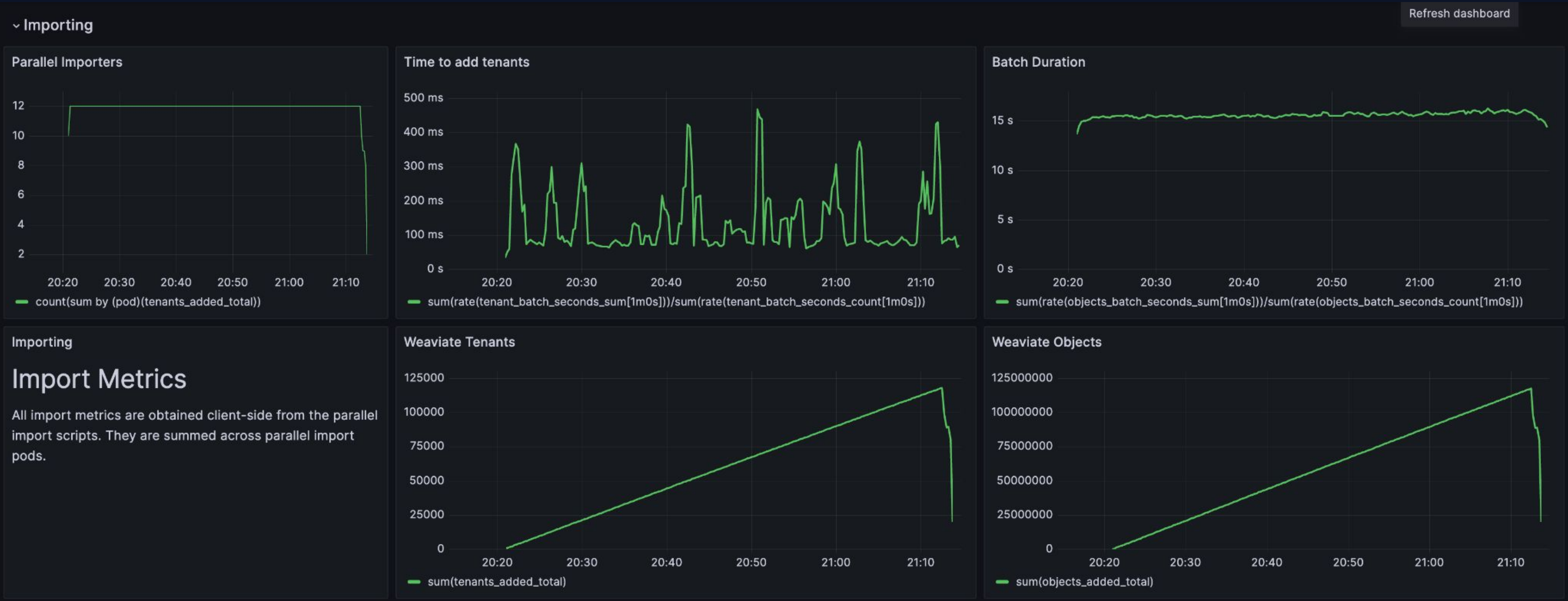


Chart uses an identical scale to the previous one (0<x<2000)

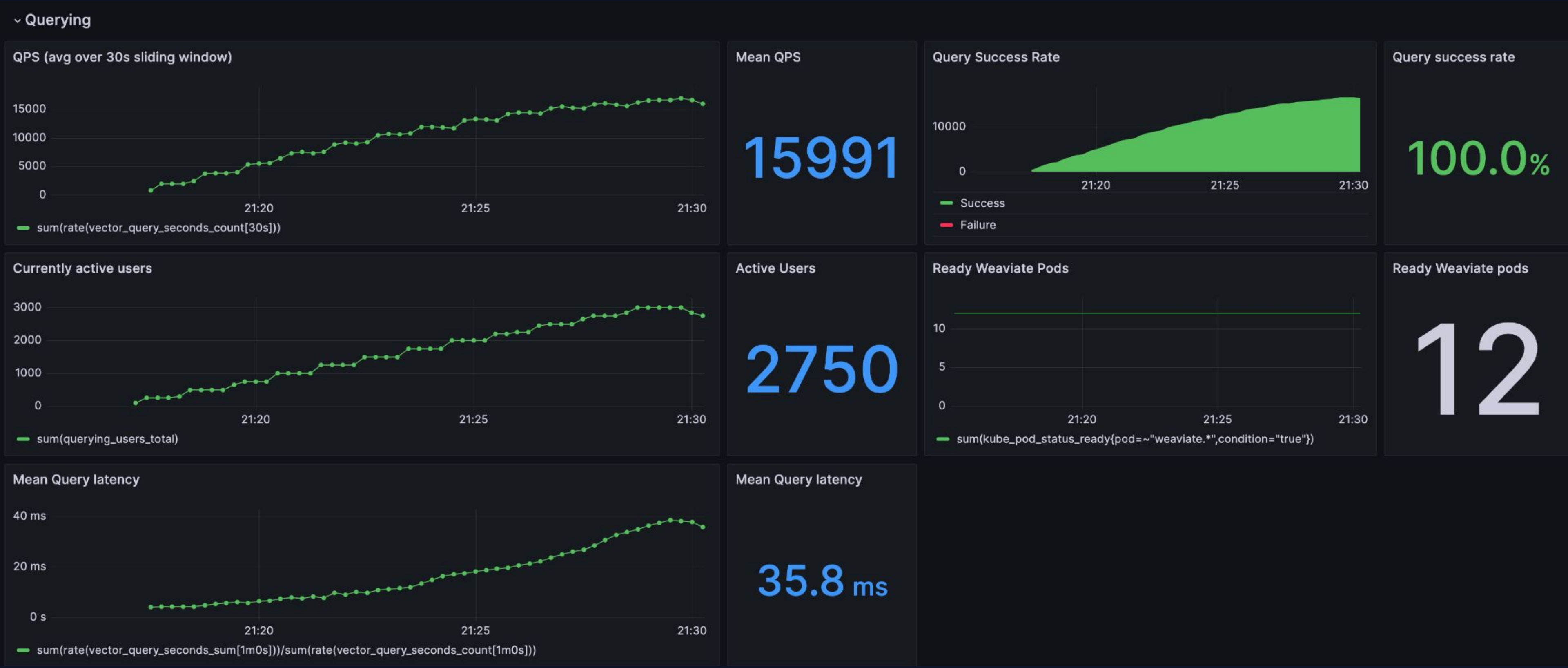


Importing

Import Metrics

All import metrics are obtained client-side from the parallel import scripts. They are summed across parallel import pods.

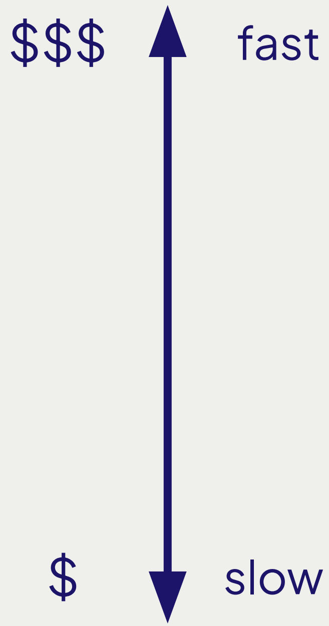
Automatic Multi-Tenancy Load test (Importing)



Automatic Multi-Tenancy Load test (Querying)

How can Multi-Tenancy reduce cost?

Storage Tiers and their cost



Memory
very fast, but very expensive

SSD Drives
medium speed, medium cost

Cloud Storage
cheap, but slow

Activating and Deactivating Tenants

Active Tenants
are already in memory
and **ready to go**

Inactive Tenants
are on disk, but can be loaded
in **a few hundred ms**

If 20% of tenants are active at any given time…

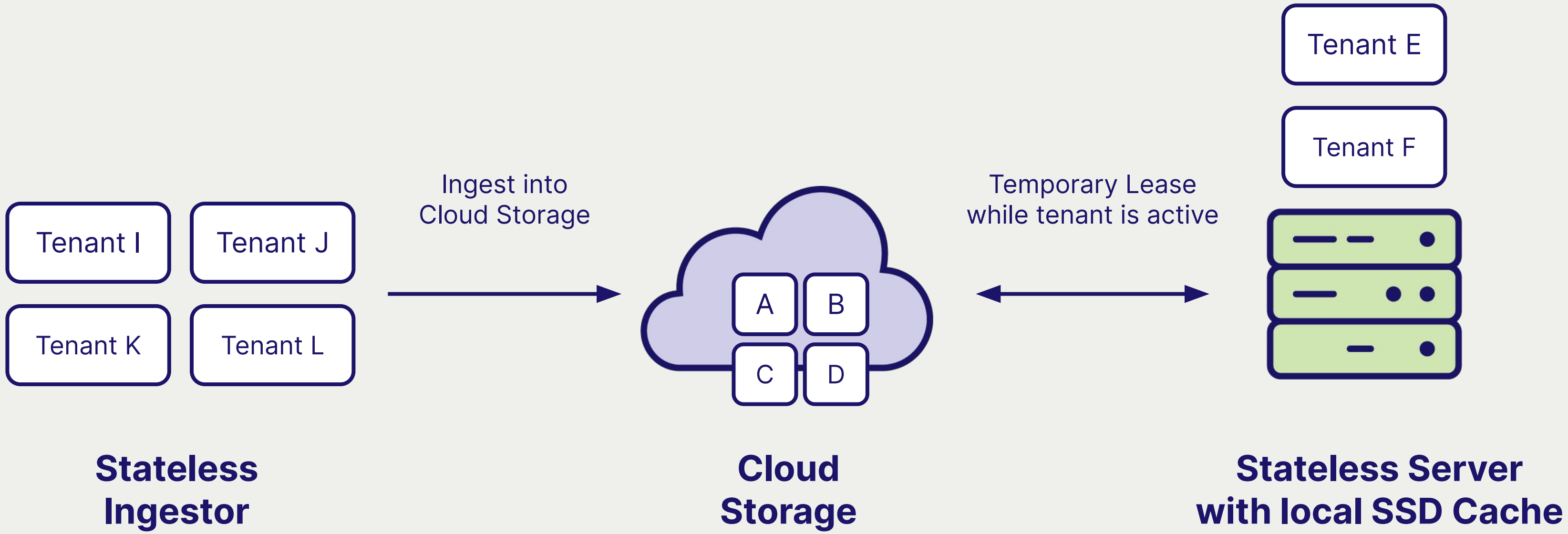
If 20% of tenants are active at any given time…

That's an 80% memory reduction.

$

Separation of Storage & Compute

Cloud-Storage-based Architecture


Tenant I
Tenant J
Tenant K
Tenant L
Ingest into Cloud Storage
A
B
C
D
Temporary Lease while tenant is active
Tenant E
Tenant F
Stateless Ingestor
Cloud Storage
Stateless Server with local SSD Cache

Recap

Most RAG and search apps require multi-tenancy.

Multi-Tenancy is hard and relying on workaround fails at scale.

With a dedicated Multi-Tenancy solution we can handle the scale and serve tenants efficiently.

There are a lot of new opportunities that couldn’t be solved before: Cost reduction, stateless vector dbs, etc.

Thank you

Connect with us!

weaviate.io

weaviate/weaviate

@etiennedi

@weaviate_io