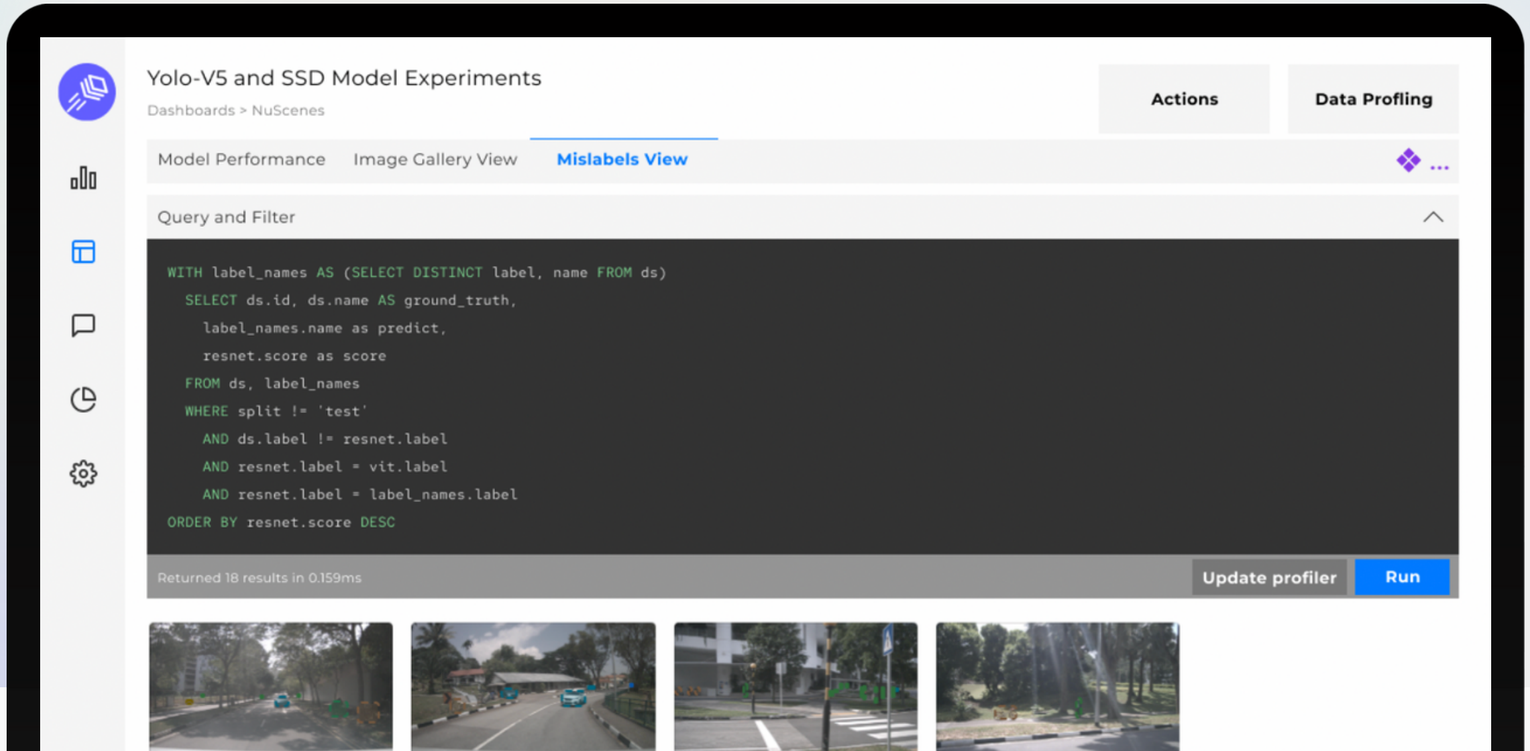
LanceDB

LanceDB: Writing a Vector Database in Rust



LanceDB.com

LanceDB

Open-Source In-process Vector Database

- Blazing Fast Vector Search, SQL, Full Text Search
- Multi-model data: Vector, Image, Text, Videos
- Written in Rust, with Python and Typescript SDKs
- Cloud-native. Data and Vector Index directly stored on cloud storage
- Backed by Lance columnar format, also written in Rust. Apache Arrow compatible

A Bit Of History of LanceDB

- Built core Lance Columnar Format in C++

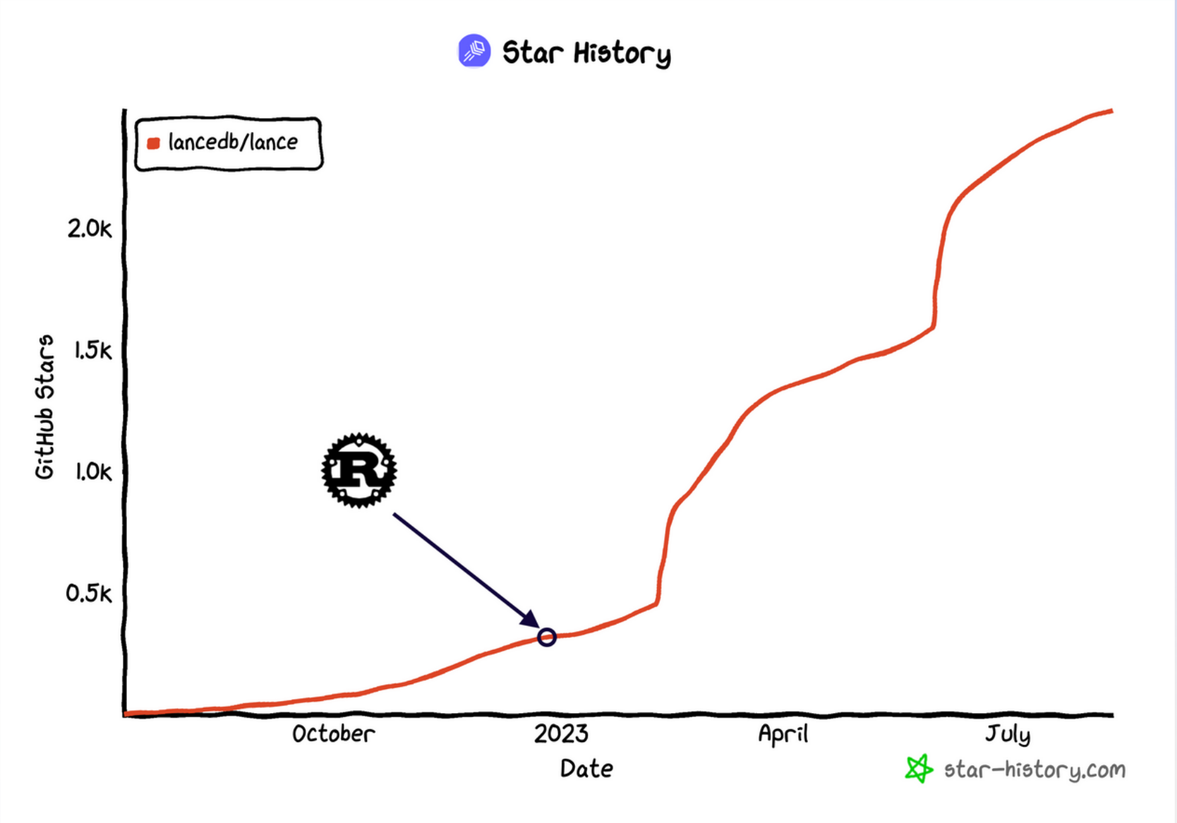
A Bit Of History of Lanc

- Built core Lance [illegible] Form[illegible]in

PAIN

A Bit Of History of LanceDB

- Let's do it again.
- Re-write in Rust in Jan 2023
  - Performance is **GREAT**
  - Community is **GREAT**
  - Productivity is **GREAT**
  - Ecosystem is **GREAT**



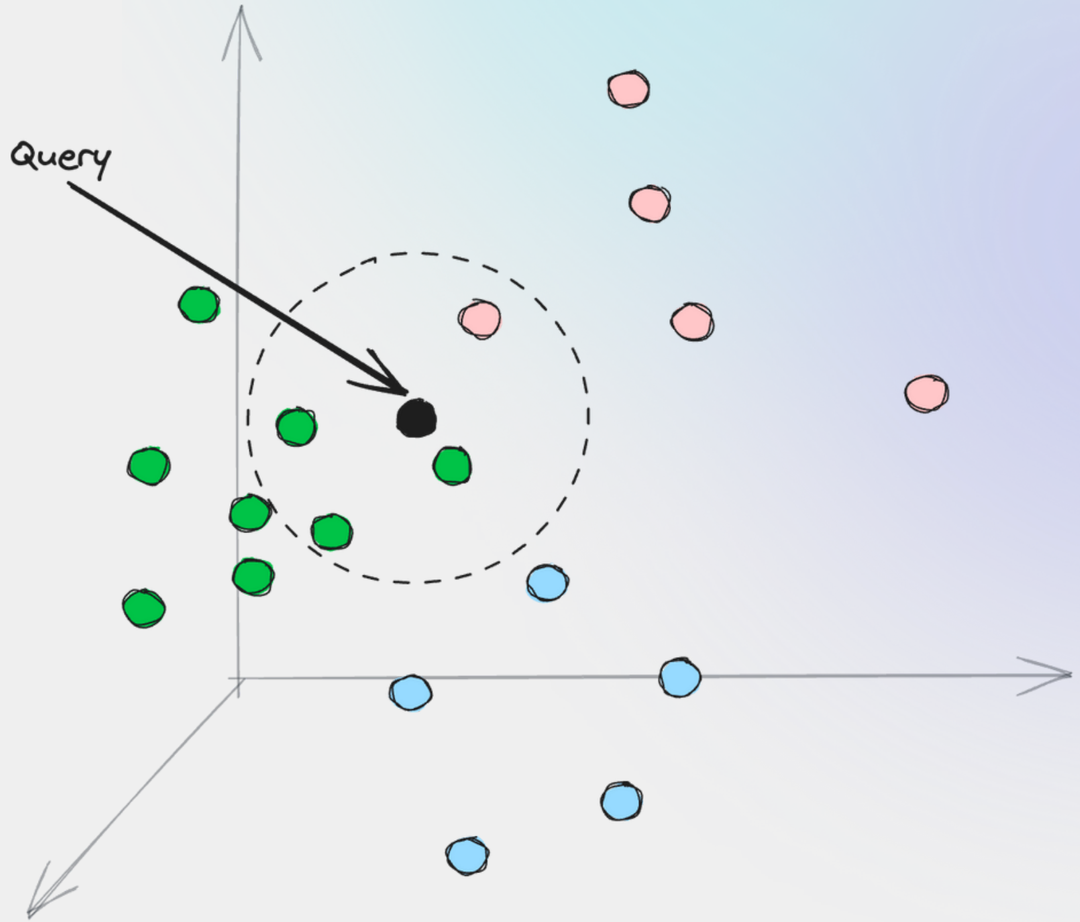
We love Rust!
Even w/ zero Rust experience

- Cargo >>> Cmake
  - Easy to link to high-quality libraries
- Beautiful Language: compiler error, modules, traits, functional programming, built-in test/bench/docs practice.
- Native language, easily embedded in other languages
- An extensive **std** library, especially **std::arch** for SIMD

So, What is a Vector Database

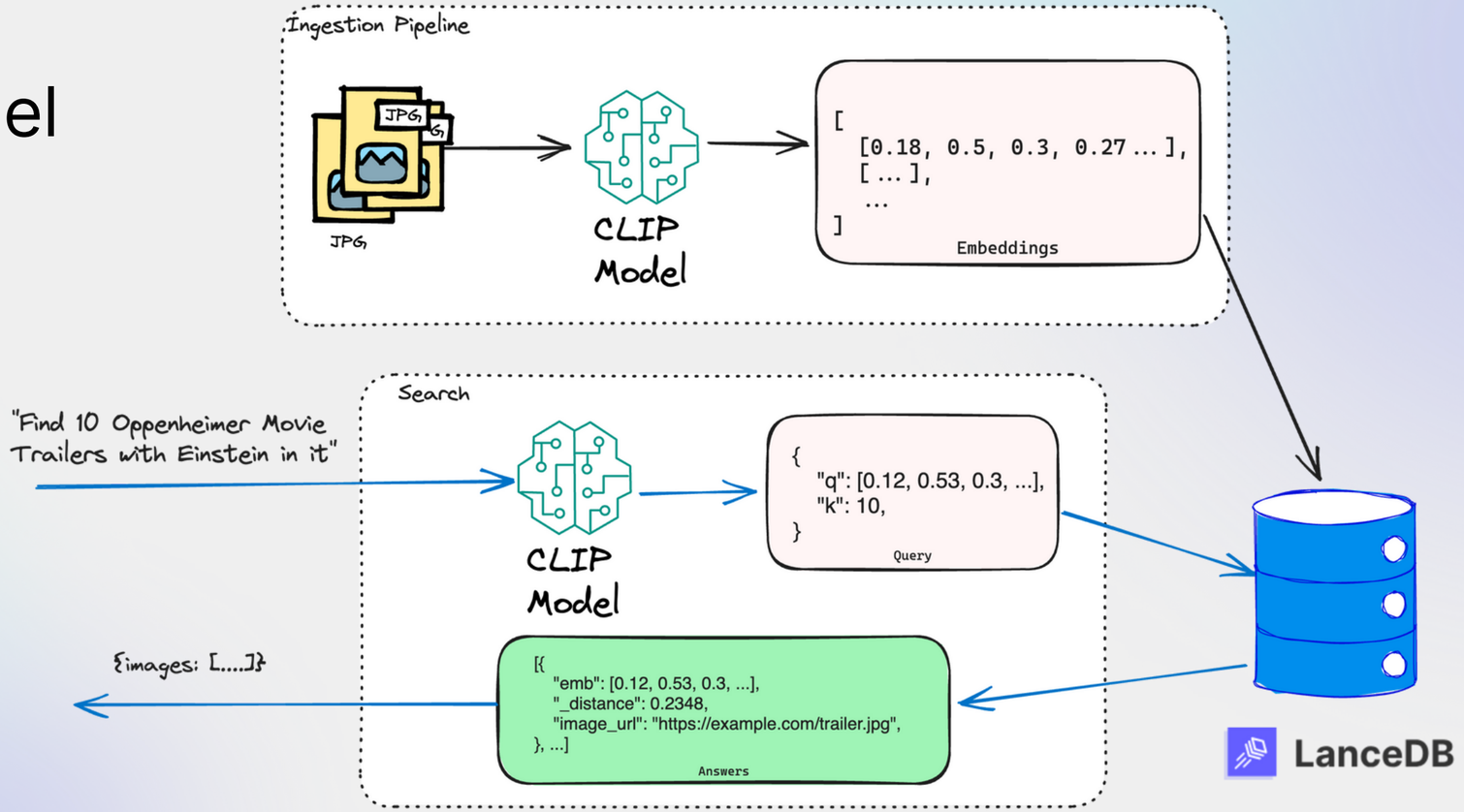
What is Vector Database

- Search K Nearest Neighbours in High-Dimensional Vector Space
  - 10^2 - 10^3 dimensions
- Diff to traditional DB
  - Linear (1D) space: b-tree or hash
- Applications:
  - ML Model Embeddings
  - LLM, Image Generation,



Application: Text-To-Image Recommendation

- Use OpenAI CLIP Model



Challenges

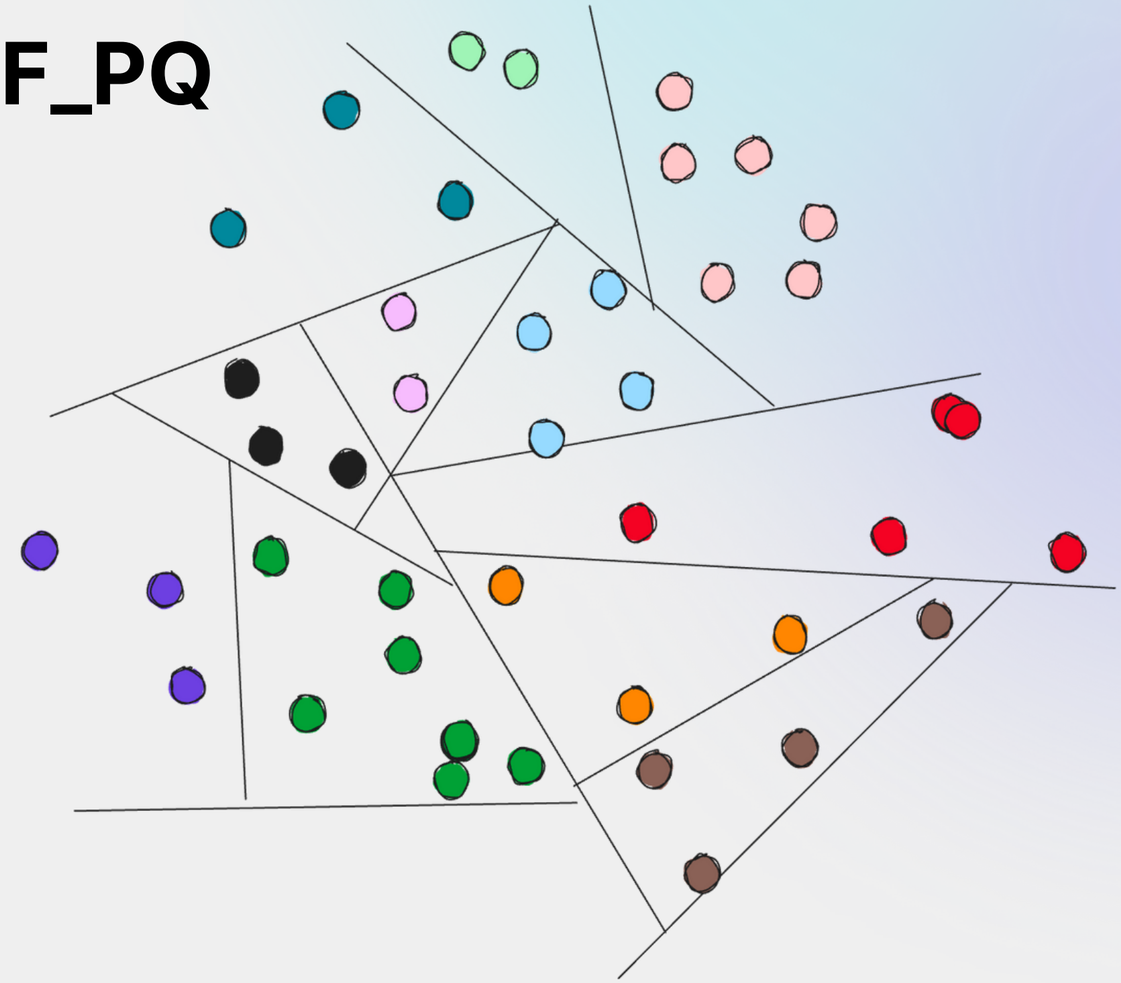
- Curse of dimensionality*
- ***Speed*** or ***Accuracy***: Pick one
- Especially difficult if everything is stored on S3*

Typical Dataset in LanceDB	
Dimension	768 ~ 1536
# of Vectors	500K ~ 1 Billion
Data Types	[float32] + metadata

* Curse of dimensionality, https://en.wikipedia.org/wiki/Curse_of_dimensionality
* Latency Numbers Every Programmer Should Know
https://colin-scott.github.io/personal_website/research/interactive_latency.html

Build Vector Index in Rust: IVF_PQ

- Vector Index to **Speed Up**
  - But less accurate!
- Divide Space into Voronoi Cells
  - K-means
- Use **Product Quantization (PQ)** to compress vectors



Voronoi Cells

Yet Another KMean in Rust! (1/2)

- It is not a joke!
- We manually tuned KMean with **std::arch** SIMD on X86_64 and aarch64
  - L1/L2 cache friendly, loop unrolling
- Adaptive Sampling
- Use Apache Arrow (**arrow-rs**) in memory
- Faster than Numpy, Arrow, LLVM-auto-vectorization, and other benchmarks

```rust
impl L2 for [f32] {
    type Output = f32;

    #[inline]
    fn l2(&self, other: &[f32]) -> f32 {
        #[cfg(target_arch = "x86_64")]
        {
            if is_x86_feature_detected!("avx2") {
                use x86_64::avx::l2_f32;
                return l2_f32(self, other);
            }
        }

        #[cfg(target_arch = "aarch64")]
        {
            use aarch64::neon::l2_f32;
            l2_f32(self, other)
        }

        #[cfg(not(target_arch = "aarch64"))]
        l2_scalar(self, other)
    }
}
```

LanceDB.com

Yet Another KMean in Rust! (2/2)

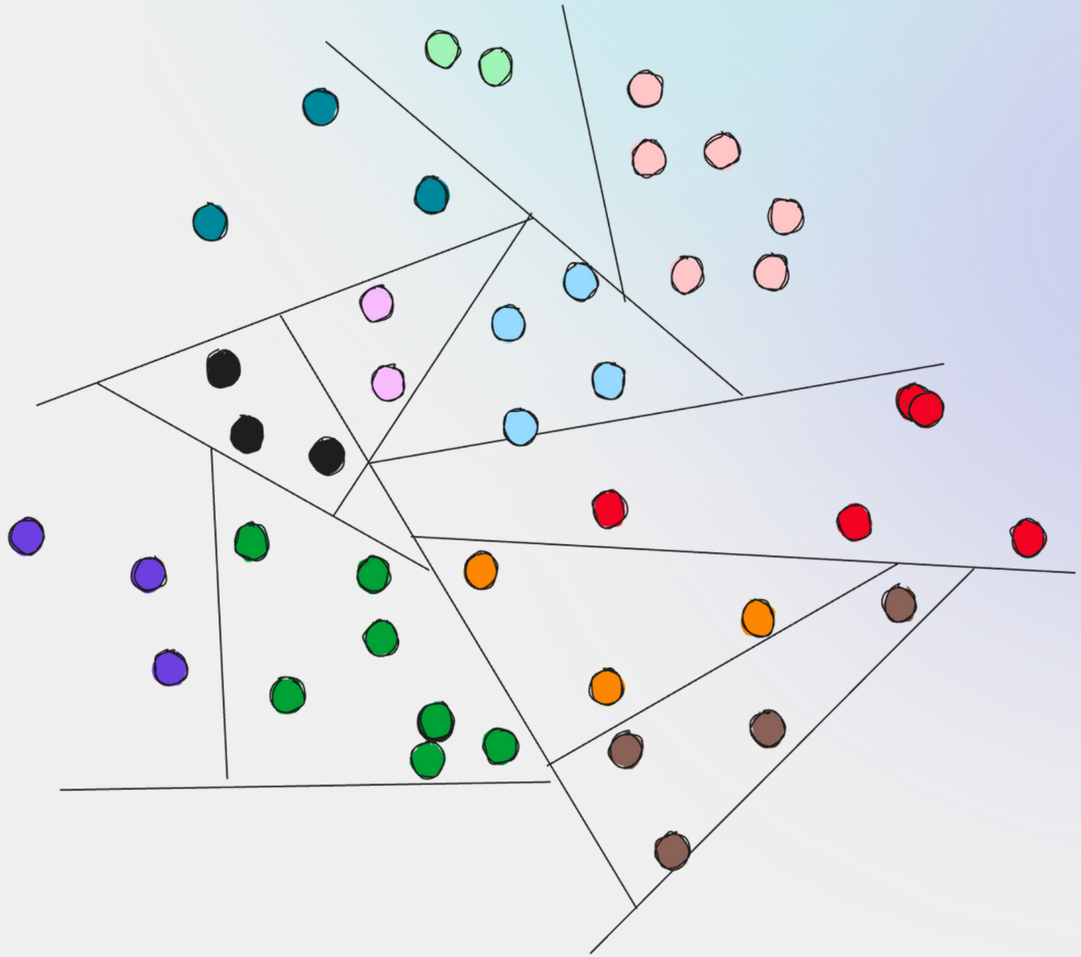
- What we LOVE about Rust:
  - Feature flag (**#[cfg(...)**) and **#[inline]**
  - Rich instruction sets in **std::arch**
  - Module for multi-arch code organization
  - **cargo bench**
  - **cargo flamegraph**
  - **rust.godbolt.org**
- What we wish that Rust (stable) has:
  - Generic specification

```rust
#[cfg(target_arch = "x86_64")]
mod x86_64 {
    pub mod avx {
        use super::super::l2_scalar;

        #[inline]
        pub fn l2_f32(from: &[f32], to: &[f32]) -> f32 {
            unsafe {
                use std::arch::x86_64::*;
                debug_assert_eq!(from.len(), to.len());

                // Get the potion of the vector that is aligned to 32 bytes.
                let len = from.len() / 8 * 8;
                let mut sums = _mm256_setzero_ps();
                for i in (0..len).step_by(8) {
                    let left = _mm256_loadu_ps(from.as_ptr().add(i));
                    let right = _mm256_loadu_ps(to.as_ptr().add(i));
                    let sub = _mm256_sub_ps(left, right);
                    // sum = sub * sub + sum
                    sums = _mm256_fmadd_ps(sub, sub, sums);
                }
                // Shift and add vector, until only 1 value left.
                // sums = [x0-x7], shift = [x4-x7]
                let mut shift = _mm256_permute2f128_ps(sums, sums, 1);
                // [x0+x4, x1+x5, ..]
                sums = _mm256_add_ps(sums, shift);
                shift = _mm256_permute_ps(sums, 14);
                sums = _mm256_add_ps(sums, shift);
                sums = _mm256_hadd_ps(sums, sums);
                let mut results: [f32; 8] = [0f32; 8];
                _mm256_storeu_ps(results.as_mut_ptr(), sums);

                // Remaining
                results[0] += l2_scalar(&from[len..], &to[len..]);
                results[0]
            }
        }
    }
}
```

I/O is tricky too!

No linear indexing = Scan!

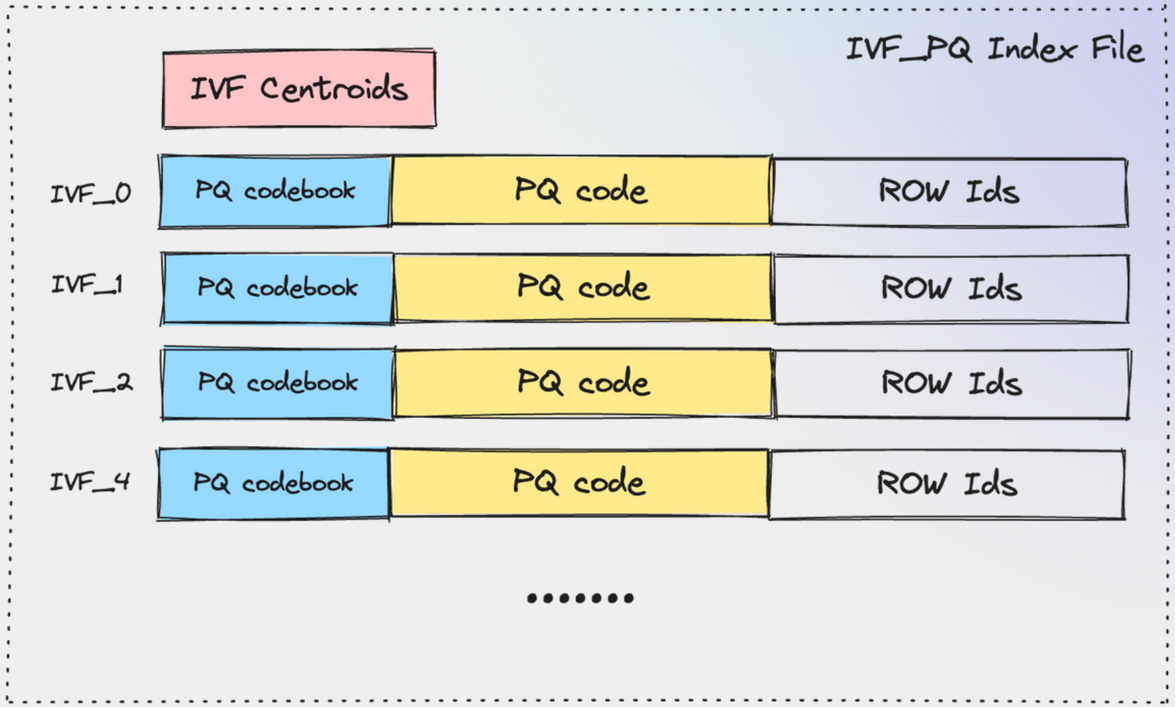
- Disk space is linear, which can not present multi-dimensional distance **statically** and **efficiently**.
- Vector distance depends ***dynamically*** on the Query Vector
  - Scan a lot from the disk for every different query
- Much random I/O to accommodate PQ distortion



Voronoi Cells

IVF PQ Index On-Disk Layout

- Optimize for scan and SIMD
  - Each block is an arrow-rs array
- Use IVF centroids to decide which partitions to scan
- Work nicely on local SSD and cloud object store
  - Different cache strategies
- Rust is much easier to work with multi-clouds than C++



How about SQL and Full Text Search?

SQL and Full Text Search

- Built on Lance, fastest growning columnar format
  - 2000x faster point query than Parquet
- SQL engine
  - **sqlparser-rs** and **datafusion**
- Full Text Search
  - **tantivy**, w/ customizations
- Async-io:
  - **tokio** + **futures + object_store**

But, How Can I Use it

Did we mention that LanceDB is In-Process DB?

- **No server, No K8S**
- **Disk-based index, no huge server to load everything in memory**
- Python and Typescript native SDK
  - **PyO3** and **Neon**
- **cargo install vectordb**

```
pip install lancedb
```

```
npm install vectordb
```

LanceDB is In-Process DB

```python
# pip install lancedb
import lancedb

uri = "data/sample-lancedb"
db = lancedb.connect(uri)
table = db.create_table(
  "my_table",
  data=[{"vector": [3.1, 4.1], "item": "foo", "price": 10.0},
        {"vector": [5.9, 26.5], "item": "bar", "price": 20.0}]
)
result = table.search([100, 100]).limit(2).to_df()
```

LanceDB is In-Process DB

- Realistically, only three languages can be used to build a multi-language in-process database
  - C
  - C++
  - Rust
- The choice is obvious :)

LanceDB Cloud

- Just change the URL to "**db://...**"
- **Pay-per-query**
- Fully managed

```python
# pip install lancedb
import lancedb

db = lancedb.connect("db://my_db", api_key="sk_a13bc3d...")
table = db.create_table(
  "my_table",
  data=[{"vector": [3.1, 4.1], "item": "foo", "price": 10.0},
        {"vector": [5.9, 26.5], "item": "bar", "price": 20.0}]
)
result = table.search([100, 100]).limit(2).to_df()
```

LanceDB

Thank You

Your feedback is important to us!

https://github.com/lancedb/lancedb (please give us a ⭐)

contact@lancedb.com