FILE SYSTEMS IN ML SETTINGS

Professor Ken Birman
CS4414 Lecture 22

IDEA MAP FOR TODAY

We have seen that file systems come in many shapes, sizes, and run in many places!

Yet what file systems are doing is inherently high-latency: Fetching bytes from some random place on a storage unit that may be a rotating physical platter accessed by moving read heads.

File systems hide this with caching and prefetching, but depend on predictable or observed behavior of the application

AI/ML systems increasingly span across many machines and the file system itself might even be on separate machines than the servers running the AI. This forces the AI to do its own caching!

Goal of caching? Track the "working set"

PERFORMANCE OF APPLICATIONS THAT DO HEAVY ACCESS TO FILE SYSTEMS

Application must open the file

- Linux will need to access the directory
- … scan it to find the name and inode number
- … load the inode into memory
- … check access permissions

So, opening a file will always involve 2 or more disk reads (more if the directory is large), unless this data happens to already be in cache.

THE FUNDAMENTAL ISSUE?

Data transfers from a local disk are pretty fast

- They use a feature called direct memory access (DMA)
- DMA can match memory speeds, if the disk can send/receive data that rapidly (some devices can, many can't).

Data transfers from a remote disk add a network hop, even if the network itself is a fancy one supporting "remote DMA" directly to application memory.

Thus, **transfer speeds can be quite high**, yet **transfer delay (latency) is often a barrier**, especially on "unanticipated" requests where Linux didn't predict/prefetch

LET’S SEE HOW THIS PLAYS OUT FOR A MODERN LLM

Today we will look at a typical large language model scenario

It involves

- Training the LLM to teach it the language and grammar rules, domain-specific terminology, resources it can leverage
- Building a repository of indexed, quickly accessible documents
- Running queries on this pre-trained model + pre-indexed data

RAG LLM: THE WINNING STORY FOR GENERATIVE AI!

Generative AI is trained offline: today’s models learned from 2020 data and hence need help from “current events” data!

RAG LLMs form prompts by combining queries with relevant documents:

- Start with documents, chunks of text, or data of other kinds.
- Embed each object as a point in a high-dimensional space.
- Given a query, embed it too, then search for the nearest neighbors

Even more basic: How do they work?

WHY THIS TERMINOLOGY: “RAG”?

The term is part of “retrieval augmented generative language model”, hence RAG-LLM. The RAG database is a special kind of database that can do *approximate match*. It differs from the kind of “table lookup” seen in relational databases that use SQL for queries.

We call these *vector databases*. They center on a notion of match distance and a way of doing lookup that minimizes the distance.

So our goal is to extend the LLM with a way to look for “relevant documents”, namely ones “approximately matched” to the query

Even more basic: How do they work?

THERE ARE MANY STAGES IN A RAG LLM PIPELINE…

For example, updates and queries for IVF employs:

- Document chunking, using a “transformer”
- Document and query embedding, again via a transformer
- Formation of a fast lookup index, to assist in queries
- When doing queries, approximate nearest neighbor search
- Enlarged prompt formation
- Actual generative response (and sanity/toxicity check)

Even more basic: How do they work?

DOCUMENT CHUNKING: THE FIRST STAGE OF THE RAG LLM PIPELINE

Given a document, such as a medical office visit or the kind of "information about your sprained shoulder" document shared with patients…

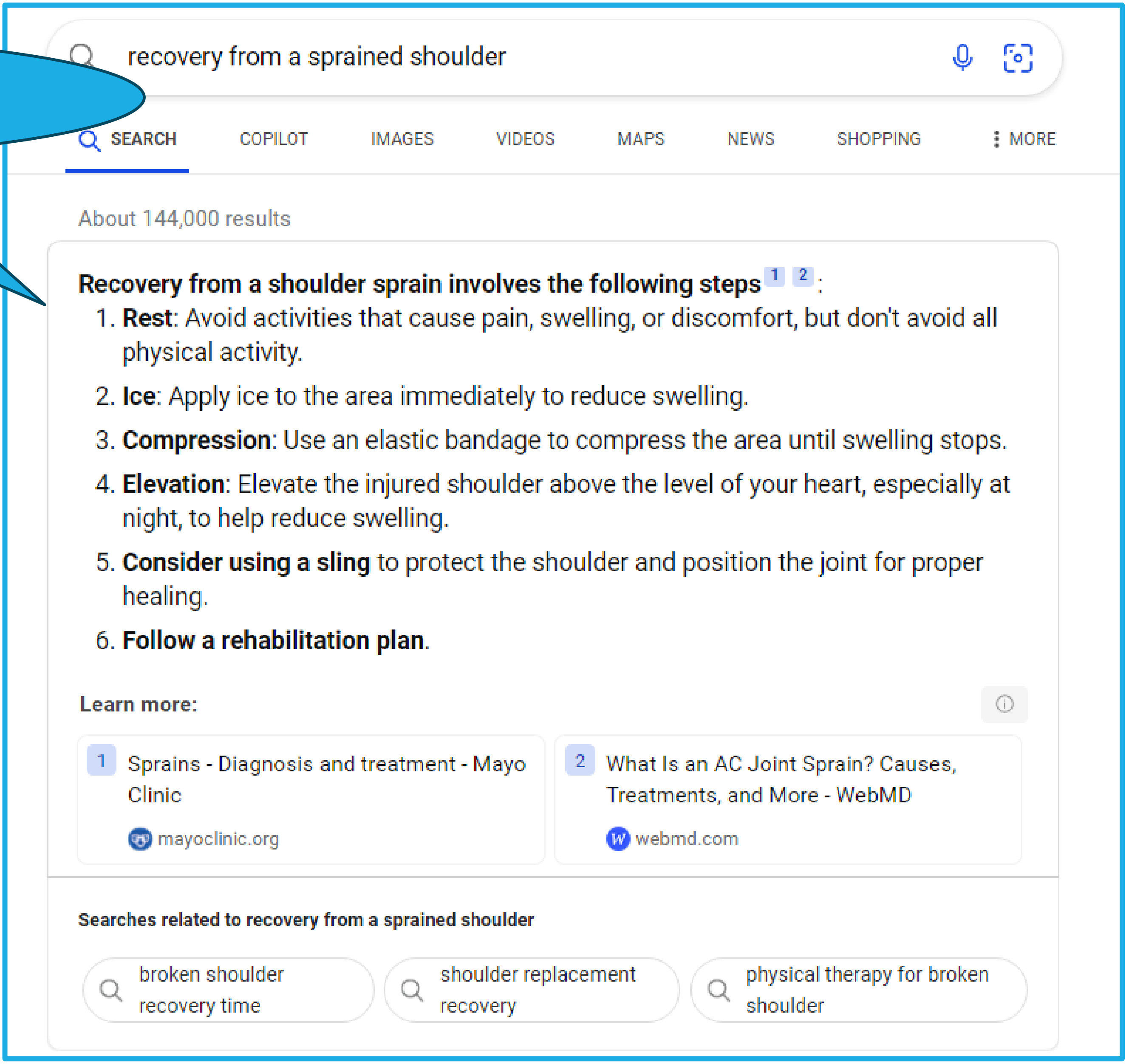
- Scan the document, creating a sequence of language tokens
- Use a "transformer" to identify key concepts, terms, etc.
- Form small chunks centered on these concepts in the context where they arose, associated with a hyperlink to the full doc.

Even more basic: How do they work?

This probably came from a longer web page or document

This information is typical of data one might find in the RAG database. The LLM knows how to find it, but did not “memorize it”.

The content comes directly from trusted websites: Mayo clinic and WebMD



recovery from a sprained shoulder

About 144,000 results

Recovery from a shoulder sprain involves the following steps [1] [2]:

1. **Rest**: Avoid activities that cause pain, swelling, or discomfort, but don't avoid all physical activity.
2. **Ice**: Apply ice to the area immediately to reduce swelling.
3. **Compression**: Use an elastic bandage to compress the area until swelling stops.
4. **Elevation**: Elevate the injured shoulder above the level of your heart, especially at night, to help reduce swelling.
5. **Consider using a sling** to protect the shoulder and position the joint for proper healing.
6. **Follow a rehabilitation plan**.

Learn more:

1 Sprains - Diagnosis and treatment - Mayo Clinic
mayoclinic.org

2 What Is an AC Joint Sprain? Causes, Treatments, and More - WebMD
webmd.com

Searches related to recovery from a sprained shoulder

broken shoulder recovery time

shoulder replacement recovery

physical therapy for broken shoulder

Even more basic: How do they work?

EMBEDDING: THE NEXT STEP

There are many ways to do this, but they tend to use transformers

These focus on a tokenized chunk or query, and compute a “self-attention weighted” numerical vector that maps similar inputs to similar locations in a high-dimensional space.

Even more basic: How do they work?

THE WEIGHTS COME FROM TRAINING

We will revisit in a few minutes, but keep in mind that this step (and every step) centers on a very costly form of training.

For embedding, it centers on how those self-attention weights are computed: doing this involves training the model using a **huge** dataset of "tagged" data that shows the model what it should be doing. This data takes the form of a huge set of files.

The model is big (and it ends up stored in a file).

Even more basic: How do they work?

NEAREST NEIGHBORS

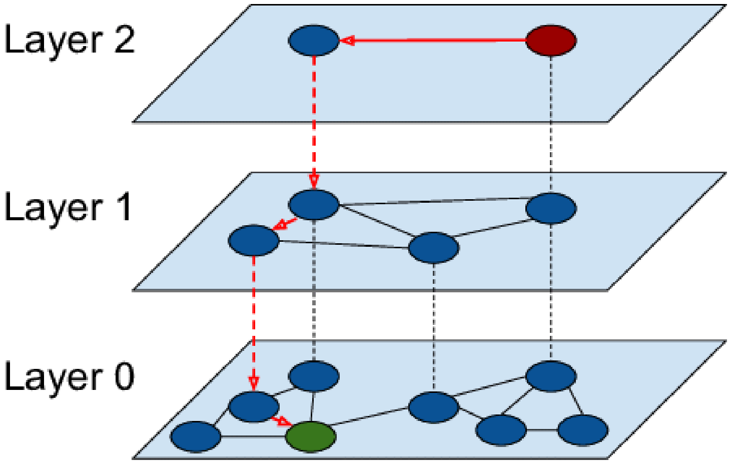
Arises in two ways

- When uploading documents, save them close to similar docs
- For a query, find the most relevant (nearest) documents

Each vector database product offers its own approximate match data structure ("index"). Different products have different performance and efficiency.

Even more basic: How do they work?

EXAMPLE 1: HNSW: ORGANIZES DOCS INTO A GRAPHICAL STRUCTURE, SEARCH VIA GRAPH-WALK.



Strengths:

- For individual queries, extremely fast
- Concept is simple and elegant

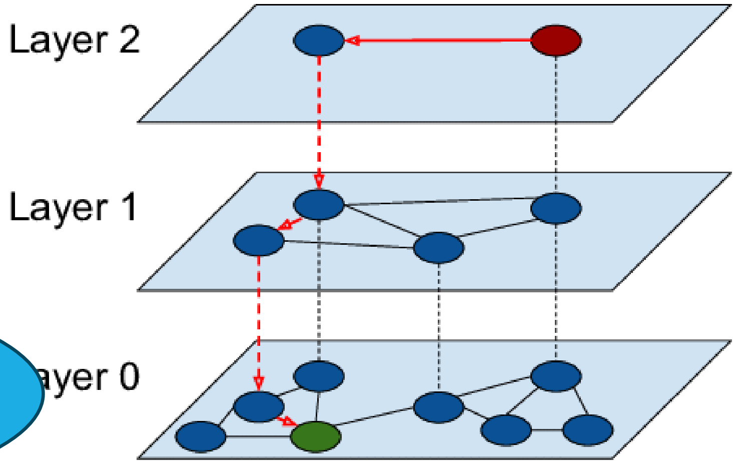
Limitations:

- Scales poorly if each server sees high volume query workloads
- Hard to distribute a single HNSW index over multiple servers if a document repository exceeds capacity of the individual servers.

HNSW is one approach…

Hierarchical Navigable Small Worlds

EXAMPLE 1: HNSW: ORGANIZES DOCS INTO A GRAPHICAL STRUCTURE, SEARCH VIA GRAPH-WALK.



Strengths:

- For individual queries, extremely fast
- Concept is simple and elegant

No practical GPU acceleration options

Limitations:

- Scales poorly if each server sees high volume query workloads
- Hard to distribute a single HNSW index over multiple servers if a document repository exceeds capacity of the individual servers.

HNSW is one approach…

EXAMPLE 2: IVF FORMS POINT-CLOUD CLUSTERS, USES THEM FOR FAST SEARCH

Similar concept, but with a multi-stage search

- Map documents to points (vectors) in high-dimensional space
- Run a clustering algorithm, create a list of centroids
- Query is similarly mapped ("embedded"), then find closest centroids, then search those clusters for closest documents

IVF is a second approach…

Inverted Files

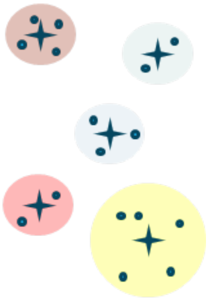
EXAMPLE 2: IVF FORMS POINT-CLOUD CLUSTERS, USES THEM FOR FAST SEARCH

Similar concept, but with a multi-stage search

- Map documents to points (vectors) in high-dimensional space
- Run a clustering algorithm, create a list of centroids
- Query is similarly mapped (“embedded”), then find closest centroids, then search those clusters for closest documents

IVF is a second approach…

LET'S ASSUME THAT OUR RAG LLM IS USING IVF. (THIS IS AN ARBITRARY CHOICE)



Why might the RAG LLM have picked IVF?

- Point-cloud representations "shard" nicely for scalable storage
- K-NN in this representation has a linear algebraic formulation.
- With a GPU accelerator, efficiently handles queries in parallel.

In fact, many RAG LLMs use other methods. But we want to understand **file system access patterns** and the **role of caching and prefetching**. Focusing on one example already illustrates the issues.

IVF is just one of many options…

TRAINING AN IVF MODEL

Centers on building a whole series of ML language models

IVF seems like a simple idea, but in fact requires multiple stages just like the way that C++ compilation runs in stages.

Each stage is a separate program running a separate task. Some of these programs might themselves be distributed tasks!

Deeper dive on IVF

ML TRAINING IS ITERATIVE AND VERY SLOW!

The generative step for a LLM like Llama3 from Meta might have 80B model parameters. The cutting edge OpenAI LLM has 175B. The chunking and embedding stages have (smaller) models, too!

Training is done using the same pattern we looked at a week ago, AllReduce or its cousin, MapReduce

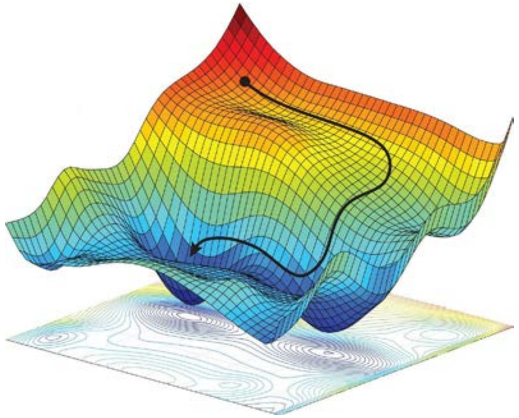
This occurs on a massive datacenter with thousands of machines

TRAINING OCCURS OFFLINE

The (many) RAG LLM stages that need a trained LLM all have an offline computation that occurs long before we “deploy” the RAG LLM.

But whereas LLMs as recently as two years ago “memorized everything”, with a RAG approach the LLM is trained for specific roles, like breaking a document into chunks.

TRAINING, MILE HIGH



OpenAI or Meta designs the model: heavy math, mostly GEMM (meaning: *expressed in terms of matrix-multiplication*).

They start with a random set of model parameters, then iteratively improve them using stochastic gradient descent. This can take **months**

The workers end up revisiting the same documents millions of times as the system gradually fine-tunes the model parameters

How training accesses files!

ALL THIS WORK YIELDS JUST ONE COMPONENT (FOR EXAMPLE, THE DOCUMENT CHUNKER)

A similar training process is used for each of the other smart components.

Training occurs in modern “big data” cloud compute infrastructures operated by Microsoft, Amazon, Google or others

File caching is a giant issue, and Linux doesn’t do so well…

Can file caching be effective here?

FILE CACHING DURING TRAINING IS DONE IN THE TRAINING SYSTEM ITSELF

During stochastic gradient descent various huge objects and files are created, plus the original documents (or other training data) is revisited again and again.

The developer team knows the odds that something will be reused, and how often. **But Linux doesn't know this.**

As a result we train on platforms like Spark/Databricks or Snowflake that allow the application to provide caching hints. They then take over the caching role to obtain better performance than pure Linux

Can file caching be effective here?

AFTER EVERYTHING IS TRAINED WE CAN TACKLE DEPLOYMENT

Recall that we are interested in RAG LLMs.

The idea here is to train the LLM on “general” content, but not have it memorize that content.

Instead we create specialist LLM components to chunk documents, “embed” them, etc.

Back to IVF and RAG LLMs

EMBEDDING CONCEPT

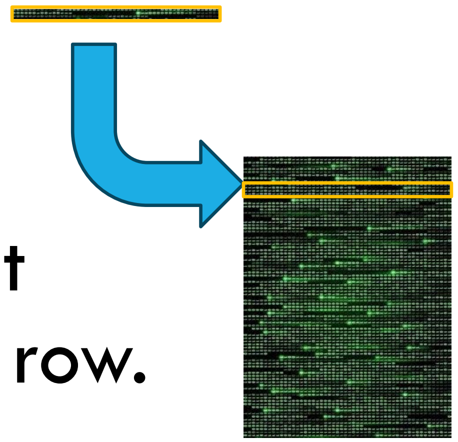
Matrix arithmetic is fast, so as much as possible AI and ML systems try to shift from working with text, images or other data to working with tensors: vectors, matrices, and higher dimensional objects too.

We say that we are "embedding" a document or query when we map it from the original text (or image, etc) form into a vector in some space defined by the LLM design team

SINGLE DOCUMENT? MULTIPLE EMBEDDINGS

For the RAG LLM case, one document yields many chunks

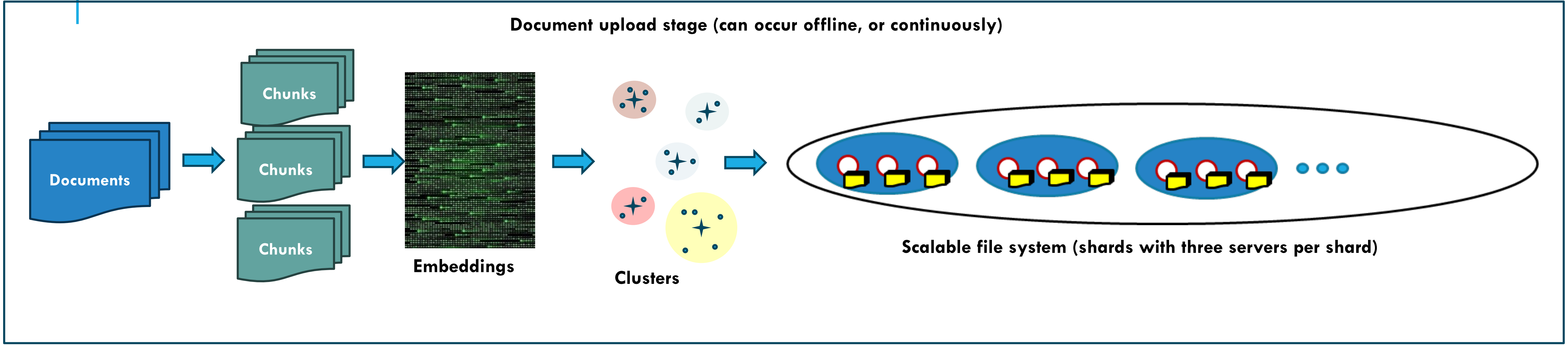
Each of these chunks probably gives one embedding vector, but it may be large (1024 * 32-bit Float = 4KB)



So one document could give us many embeddings: not a single vector but a matrix where each vector is one row.

Deeper dive on IVF

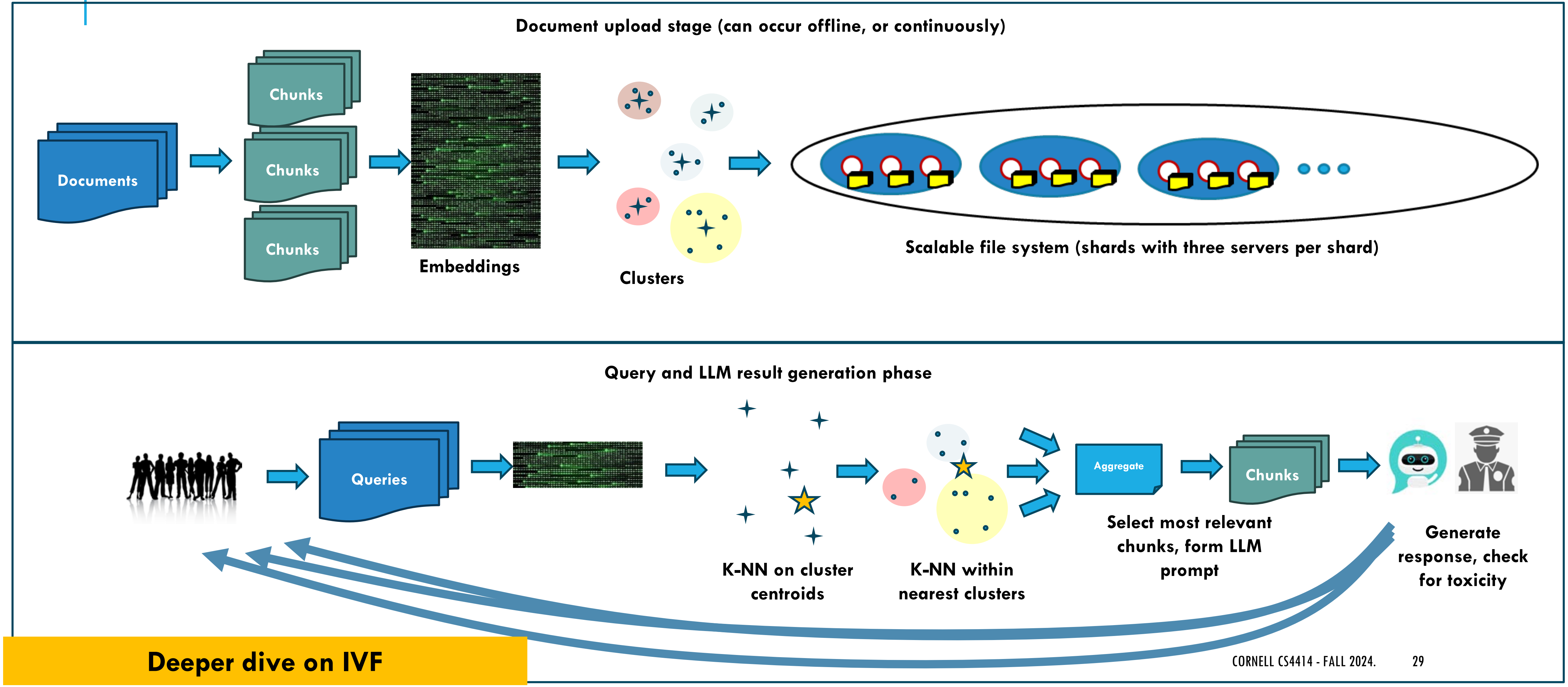
IVF IN ACTION… DOCUMENT UPLOAD WHILE BUILDING THE RAG DOCUMENT INDEX


Document upload stage (can occur offline, or continuously)
Documents
Chunks
Chunks
Chunks
Embeddings
Clusters
Scalable file system (shards with three servers per shard)

Here we see upload after the LLM components are already trained. For each document we chunk it, generate embeddings, then form clusters and identify their centroids. Those are then stored into our file system

Deeper dive on IVF

IVF WITH THE QUERY STAGE SHOWN, TOO



Document upload stage (can occur offline, or continuously)

Documents

Chunks

Chunks

Chunks

Embeddings

Clusters

Scalable file system (shards with three servers per shard)

Query and LLM result generation phase

Queries

Aggregate

Chunks

K-NN on cluster centroids

K-NN within nearest clusters

Select most relevant chunks, form LLM prompt

Generate response, check for toxicity

Deeper dive on IVF

UPLOAD IS DOMINATED BY WRITING TO THE FILE SYSTEM. QUERIES ARE MOSTLY READ-ONLY

For the query stage, the LLM pipeline (the one on the bottom):

- Read the matrix of centroid data (many gigabytes in size),
- For each cluster read the list of document embeddings held in that cluster (again, huge).
- Read the actual document chunks and the URLs pointing to the original document, to prompt the generative LLM

Once deployed, queries are very dominant in today's systems.

Deeper dive on IVF

DYNAMIC UPDATES? A FUTURE FEATURE

By and large, RAG LLM systems aren’t currently doing a great deal of dynamic updating of the RAG database.

In time, they will: the RAG database will start to hold “current context” for the user, or the world. Like minute-by-minute data about a football game, or an election.

But until that happens, each stage really runs separately: training, building the RAG data structure and querying.

Deeper dive on IVF

HOW WELL WILL LINUX BUILT-IN FILE SYSTEM CACHING PERFORM?

This question is the right one, but hard to answer

We really have a *lot* of programs running, each doing its own pattern of accesses.

Linux understands how to optimize for one process on one machine, but won't optimize across this big distributed system

Finally! Our file system topic!

… IMPLICATION?

Like we have seen for ML training systems, there will need to be more and more work on how to host inference (query) systems

These need to use the hardware efficiently and run at high speeds, and will not be able to "just trust Linux" cachine

Developing solutions for this space is a hot research topic today!

CACHING: THE CORE CHALLENGE IS TO HAVE THE WORKING SET IN THE CACHE

We use this term in several situations.

Linux sometimes does paging to reduce the pressure on memory. A process has the working set in memory if all the instructions and data it actually touches when running are resident.

Similarly, the disk buffer pool holds the working set if it already has a copy of the files the application is likely to access.

Insight: The real question is what to cache

A WORKING SET IS A COLLECTION OF CACHED DATA LARGE ENOUGH TO HOLD EVERYTHING

The concept is that as we execute, we periodically hit repetitious situations in which the same data is accessed again and again

With luck (and enough memory!) we can "discover" and hold the working set. Files won't need to be re-fetched over the network

Insight: The real question is what to cache

CACHING ALGORITHMS

Cache retention is decided by an algorithm built into the Linux file system itself. The algorithm is really the one used for eviction when we need more space.

For many years LRU was the most common (least recently used). LFU (least frequently used) was also explored.

Insight: The real question is what to cache

DENNING: WORKING SET (1968)

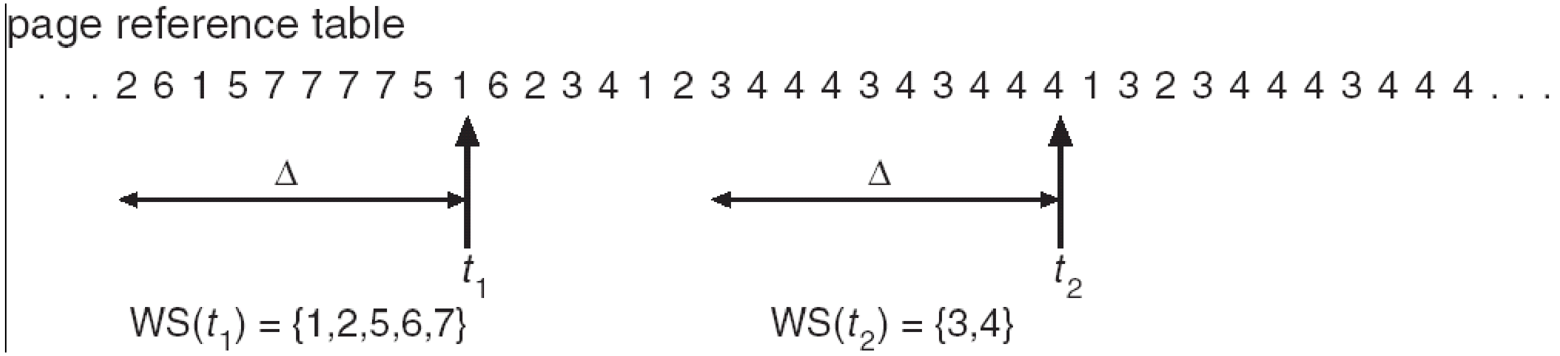
Peter Denning was the champion of working set algorithms

He showed that if we can estimate the length (in time) of these stable access periods, we can design an algorithm that will retain exactly the documents in current use: the working set.

Even a conservative estimate of the working set window works.

Insight: The real question is what to cache

DENNING FOCUSED ON VIRTUAL MEMORY AND PAGING



page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

$WS(t_1) = \{1,2,5,6,7\}$

$WS(t_2) = \{3,4\}$

Here we see a list of pages being referenced by a program. But if it was a file system cache, same idea: these could either be whole files, or pages within files.

Insight: The real question is what to cache

IS IT FAIR TO FOCUS ON PAGING ALGORITHMS?

Pages are not "whole files"

But in fact the working set idea is more general than Denning initially expected it to be.

His ideas for pages can be applied directly to whole-file caching

Insight: The real question is what to cache

HOW TO ASSESS EFFECTIVENESS OF WORKING SET AS A CACHING ALGORITHM

To evaluate WS, Denning defined a policy called WSOpt

- WSOpt has **perfect knowledge of the future.**
- For this algorithm (which cannot be implemented), the working set is computed over the next Δ references, not the last: R(t)..R(t+Δ-1)

He compared WS with WSOpt.

- WSOpt has knowledge of the future…
- …yet even though WS is a practical algorithm with no ability to see the future, the Hit and Miss ratios are identical for the two algorithms!

Insight: The real question is what to cache

KEY INSIGHT IN PROOF

Basic idea is to look at the paging decision made in WS at time t+Δ-1 and compare with the decision made by WSOpt at time t

Both look at the same references… hence make same decision

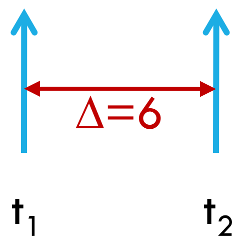
- Namely, WSOpt tosses out page R(t-1) if it isn't referenced "again" in time t..t+Δ-1
- WS running at time t+Δ-1 tosses out page R(t-1) if it wasn't referenced in times t...t+Δ-1
- … and these are the same references!

Insight: The real question is what to cache

KEY STEP IN PROOF

Page reference string

. . . 2 7 1 5 7 7 7 5 6 1 2 7 1 6 6 5 8 1 2 5 8 1 2 1 5 8 8 5 1 2 6 1 7 2 8 6 1 7 7 7 2 . . .



At time t_1 resident page set contains { 1,2,5,7 }

- WSOPT notices “5 will not be referenced in next 6 time units, and pages 5 out at time t_1
- WS will page 5 out too, but not until time t_2

Insight: The real question is what to cache

HOW DO WSOPT AND WS DIFFER?

WS maintains more pages in memory, because it needs Δ time "delay" to make a paging decision

- In effect, it makes the same decisions, but after a time lag
- Hence these pages hang around a bit longer

Insight: The real question is what to cache

WSOPT AND WS: SAME HIT RATIO!

WS is a little slower to remove pages from memory, but has the identical pattern of paging in and paging out, just time-lagged by Δ time units

Thus WS and WSOPT have the identical hit and miss ratio… a rare case of a “real” algorithm that achieves seemingly optimal behavior

Insight: The real question is what to cache

HOW DO WS AND LRU COMPARE?

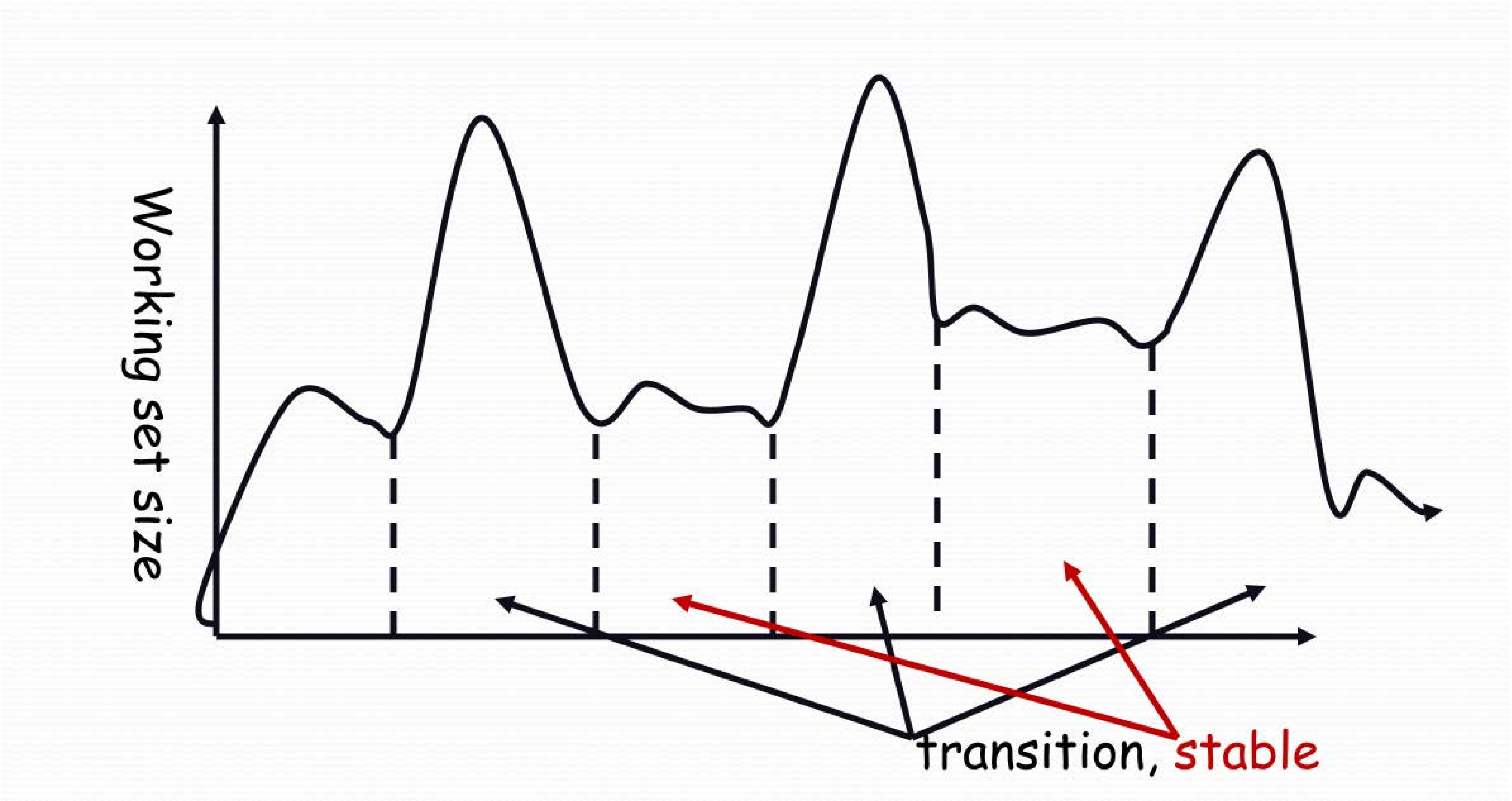
In contrast, WS can outperform LRU in an important way.
Suppose we use the same value of Δ

- WS removes pages if they aren't referenced and hence keeps less pages in memory
- When it does page things out, it is using an LRU policy!
- LRU will keep all Δ pages in memory, referenced or not

Thus LRU often has a lower miss rate, but needs more memory.
On platforms shared by multiple processes, WS really pays off

Insight: The real question is what to cache

WORKING SETS IN THE REAL WORLD



Working set: A winning caching algorithm!

DOES WS REMAIN THE SAME IF WE ARE MANAGING A CACHE OF FILES?

In fact, yes!

We do need a concept of how long the window should be (Δ)

And we would only favor WS if the hardware is shared by multiple applications or users, who "contend" for cache space. But in that situation, it would dbe a good choice!

Insight: Linux prefetching may not be useful

WHAT ABOUT PREFETCHING?

In the word-count problem, prefetching was a huge win

It allowed us to

- Open files before we would access them, to get that work out of the way as a concurrent background task
- Access files sequentially so that block by block, they would be available as needed.

Insight: Linux prefetching may not be useful

WOULD PREFETCHING HELP FOR RAG LLM?

… not entirely clear!

Many AI systems prefer to have all the security checks done at one time, and then to load or store "whole files"

This leads to a key-value (KV) model: the key is the name of the file, and the value is the serialized byte vector with contents

Insight: Linux prefetching may not be useful

KEY VALUE CONCEPT

The idea is pretty trivial.

Instead of a file named "RAG-LLM/file0270" we just have a KV tuple. The file name is the key and the contents are the value.

Access via **v = get(k)**, or **put(k, v)**. Fewer system calls than with open/read/write/close. Enables "whole document" caching.

In fact, RAG LLMs don't even use files!

SWITCHING TO A KV MODEL ELIMINATES THAT ISSUE OF PRE-OPENING FILES

When the application first connects to the KV store, the permissions are checked. After that, it can access *any KV tuple*

We call this binding. It removes a big source of overhead!

The reason for whole-file access is that if we will be reading the whole file or writing it, why do it as a series of reads/writes?

… but everything we said about files applies

BUT KV STORES CAN DO CACHING TOO

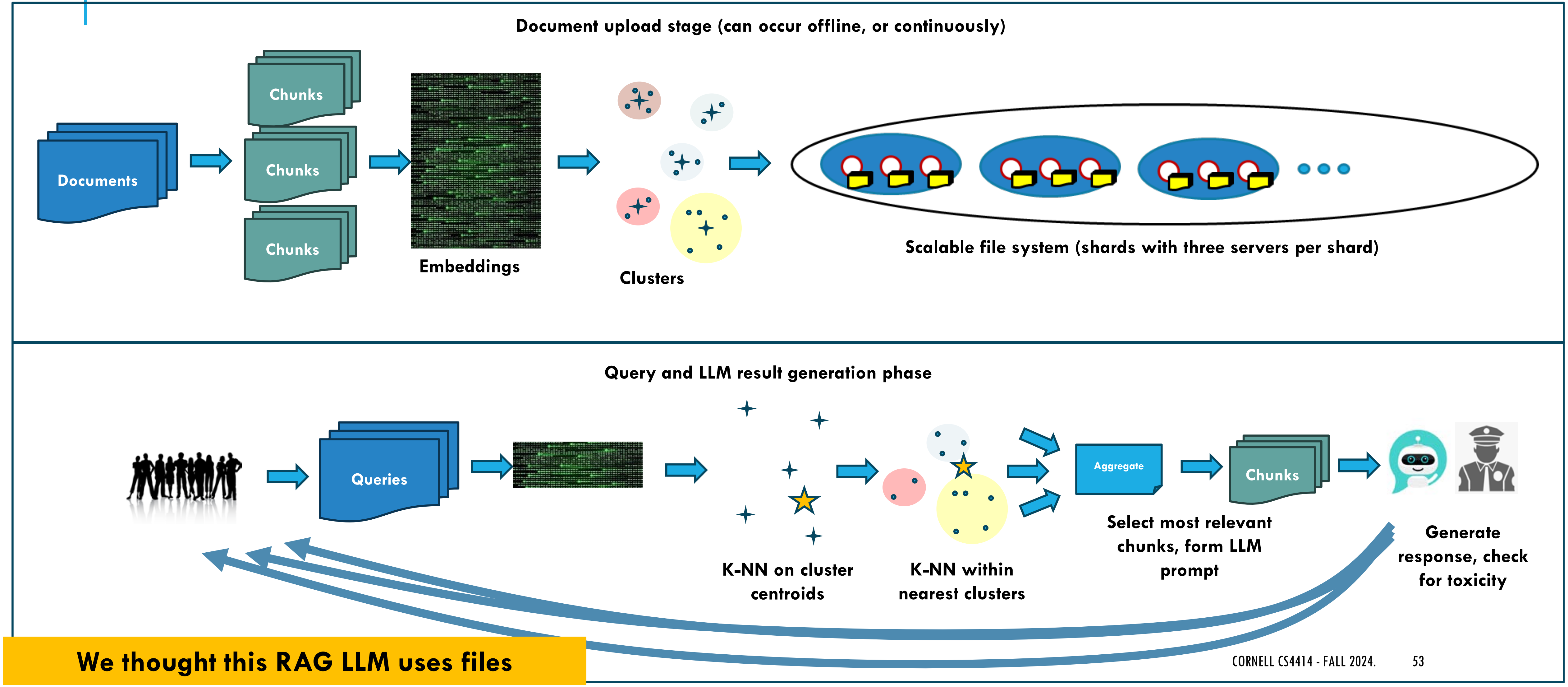
Any high quality KV store will manage its own cache, probably right in the address space of the process using it.

This totally bypasses the Linux file system cache, yet uses the exact same ideas we have discussed!

Prefetching would now be harder: it would come down to anticipating what files (keys) will be accessed "next", and having an API to tell the KV service: "now would be a smart time to fetch xxx"

… but everything we said about files applies

REMEMBER IVF? IT REALLY WANTS "WHOLE FILE" STORAGE/RETRIEVAL...



Document upload stage (can occur offline, or continuously)

Documents

Chunks

Chunks

Chunks

Embeddings

Clusters

Scalable file system (shards with three servers per shard)

Query and LLM result generation phase

Queries

Aggregate

Chunks

K-NN on cluster centroids

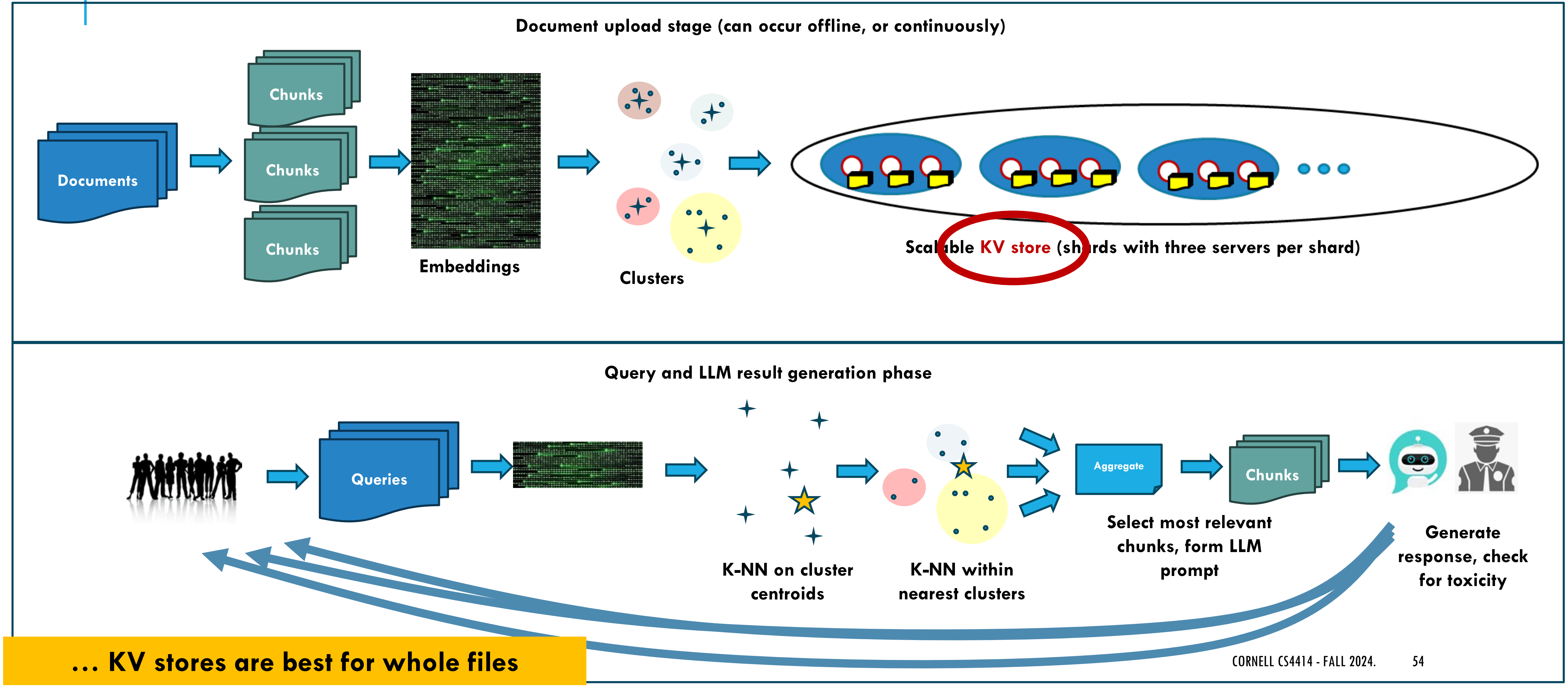
K-NN within nearest clusters

Select most relevant chunks, form LLM prompt

Generate response, check for toxicity

We thought this RAG LLM uses files

REMEMBER IVF? IT REALLY WANTS "WHOLE FILE" STORAGE/RETRIEVAL: A KV STORE


Document upload stage (can occur offline, or continuously)
Documents
Chunks
Chunks
Chunks
Embeddings
Clusters
Scalable KV store (shards with three servers per shard)
Query and LLM result generation phase
Queries
K-NN on cluster centroids
K-NN within nearest clusters
Aggregate
Chunks
Select most relevant chunks, form LLM prompt
Generate response, check for toxicity

… KV stores are best for whole files

WHAT IS ALL OF THIS TELLING US?

Given the huge importance of RAG LLM, these systems are receiving enormous attention.

They will need special-purpose “runtime hosting environments” and those systems will need their own caching solutions

- Ken and Alicia are developing Vortex as a research project. It focuses on these last two aspects, but especially queries.

RAG LLMs are a big topic for research too!

SUMMARY

Hi! I rule the world…

Our first 21 lectures focused mostly on the single machine case.

We understand that pretty well… but the world has shifted to a kind of big-data, big-application model in which everything is distributed.

Many things we are used to in Linux need to be reexamined for this new world. Otherwise, our systems run inefficiently!

SUMMARY

Hey Vortex, can you help me out?

Today we saw a RAG LLM design concept, and discussed training, RAG document upload and query pipelines.

Some ideas we learned about in prior lectures are relevant!

But we can also see that the world has changed, and we can’t simply assume that the old approaches are optimal.