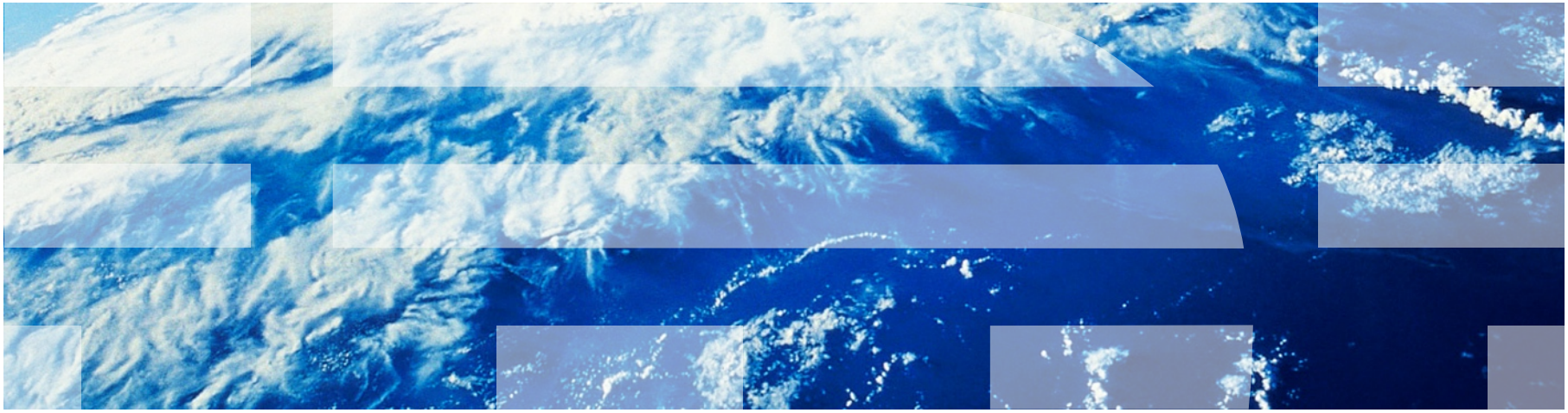
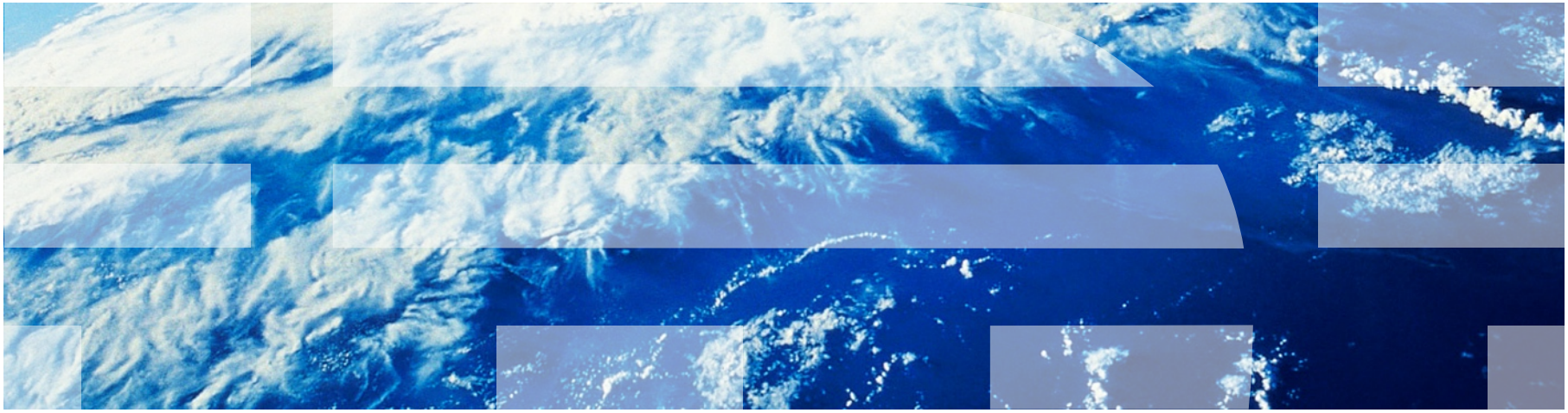
Computer Systems for Data Science

Distributed ML



Credit: Zhihao Jia

Distributed ML Training

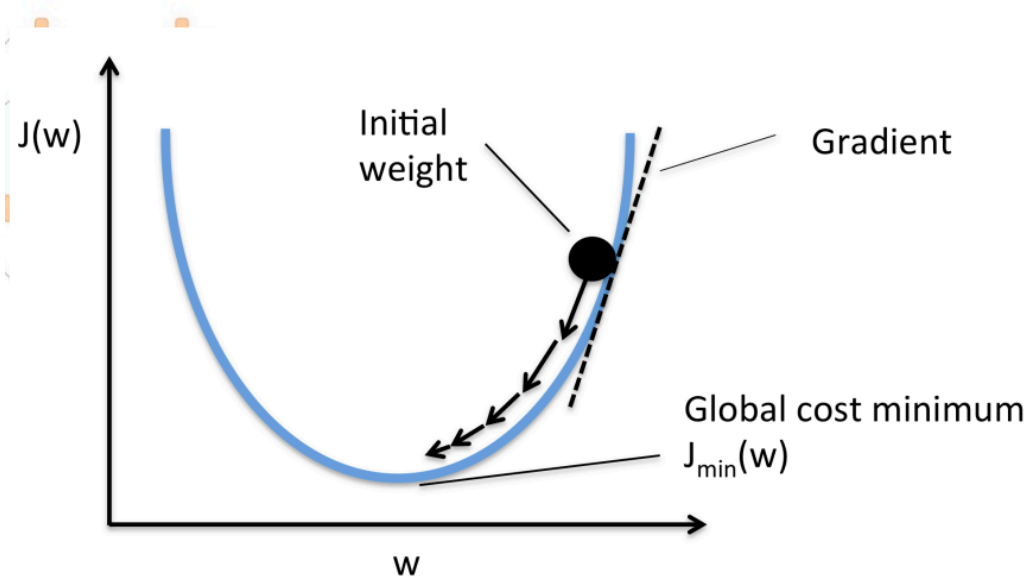


Today

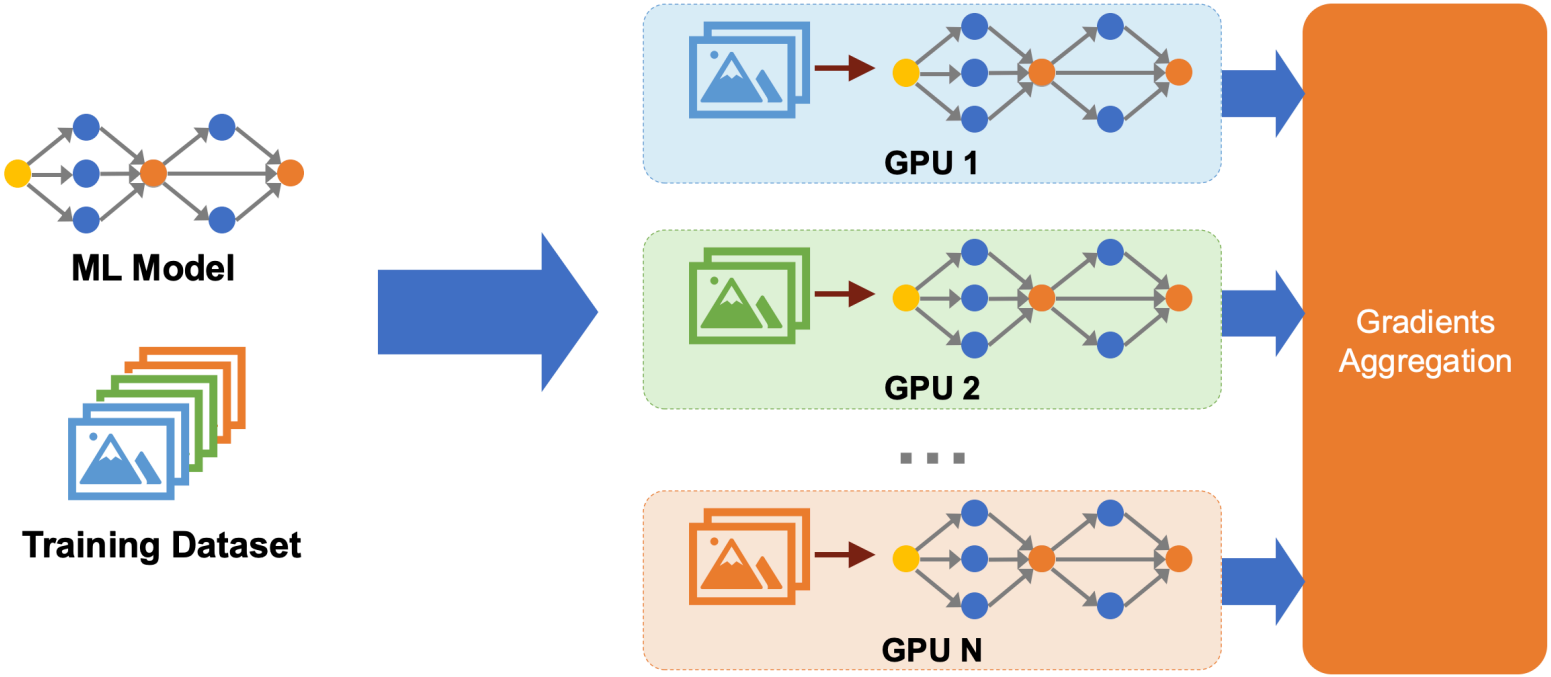
- Distributed training
  - Data parallelism – replicate model, partition data
  - Parameter server, naïve allreduce, ring allreduce
  - Model parallelism: partition model, replicate data
  - Pipelining on “micro-batches” solves parallelism problem
  - “State of the practice”: hybrids between data and model parallelism
- Inference
  - Becoming increasingly more important to optimize
  - Reasoning models: a single logical inference is composed of hundreds of inferences
  - Main challenge: memory efficiency
- RAG
  - Augment queries with information that the model was not trained on
  - Find the closest chunk of text to the query, and add it to the prompt
  - Implemented on a vector database
    - Goal: find K nearest neighbors
  - Two sample indices: clustering and graph

Training

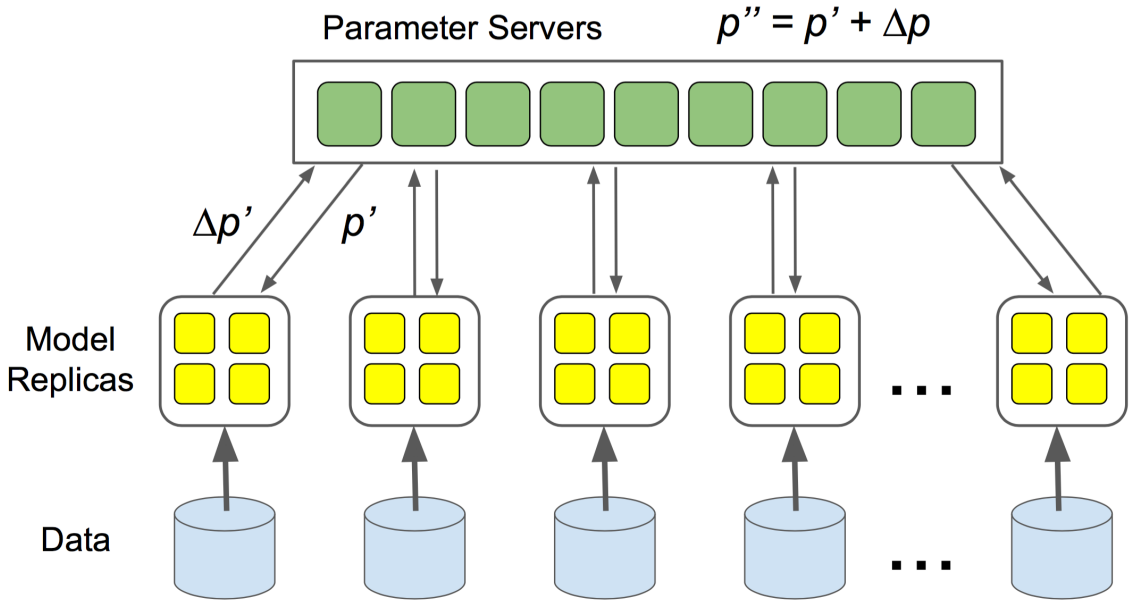
- Very high-level idea:
  - We have an objective function, we use a gradient descent algorithm that updates model weights iteratively in order to achieve a better score for the objective function
  - State-of-the-art models might have trillions of weights
- Gradient descent has three stages:
  - **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
  - **Backward propagation**: run model in reverse to produce error for each trainable weight
  - **Weight update:** use loss value (output of objective function) to update model weights



How do we parallelize this process?



Data Parallelism: Parameter Server

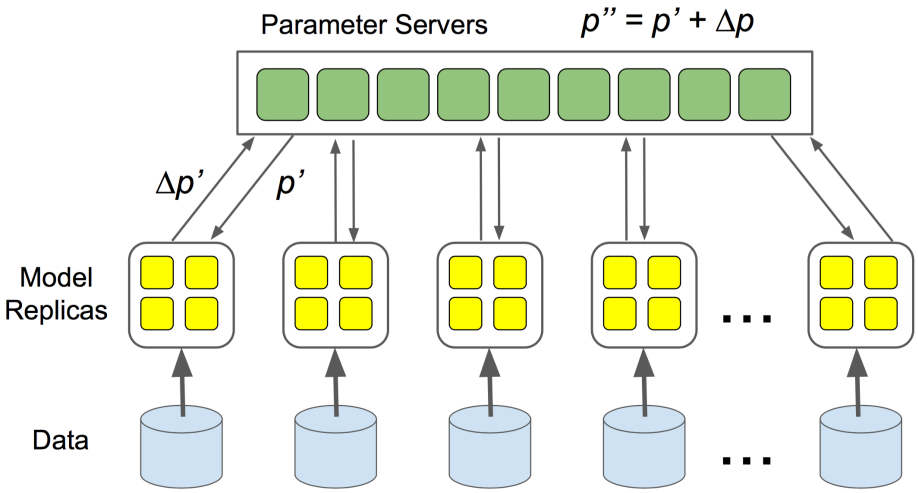


Workers push gradients to parameter servers and pull updated parameters back

Inefficiency of Parameter Server

- **Centralized communication**: all workers communicate with parameter servers for weights update; cannot scale to large numbers of workers

- How can we **decentralize** communication in DNN training?

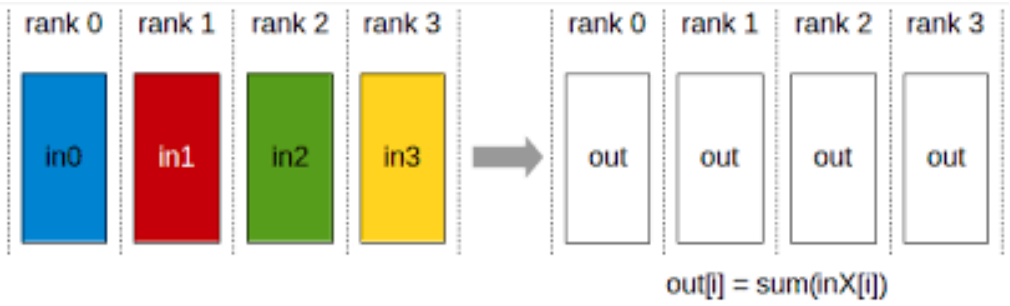


Inefficiency of Parameter Server

- **Centralized communication**: all workers communicate with parameter servers for weights update; cannot scale to large numbers of workers

- How can we decentralize communication in DNN training?

- **AllReduce**: perform element-wise reduction across multiple devices

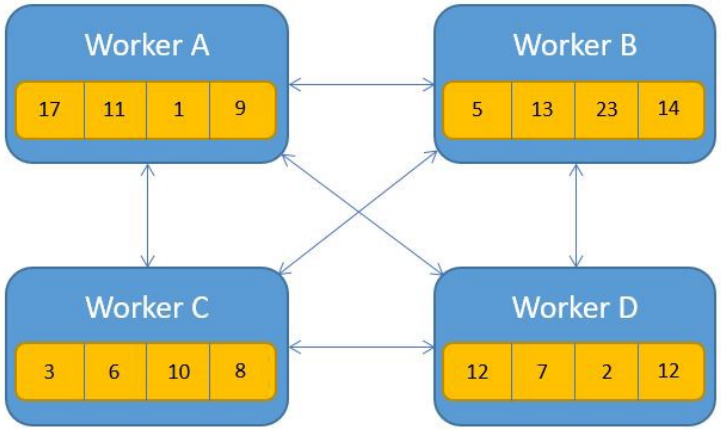


Different Ways to Perform AllReduce

- Naïve AllReduce
- Ring AllReduce
- Tree AllReduce
- Butterfly AllReduce

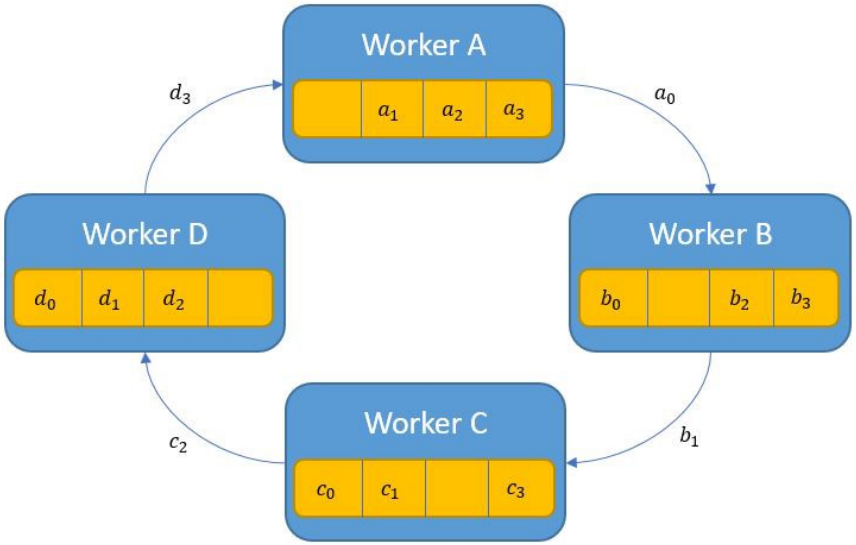
Naïve AllReduce

- Each worker can send its local gradients to all other workers
- If we have N workers and each worker contains M parameters
- Overall communication: N * (N-1) * M parameters
- **Issue**: each worker communicates with all other workers; have the same scalability issue as parameter server



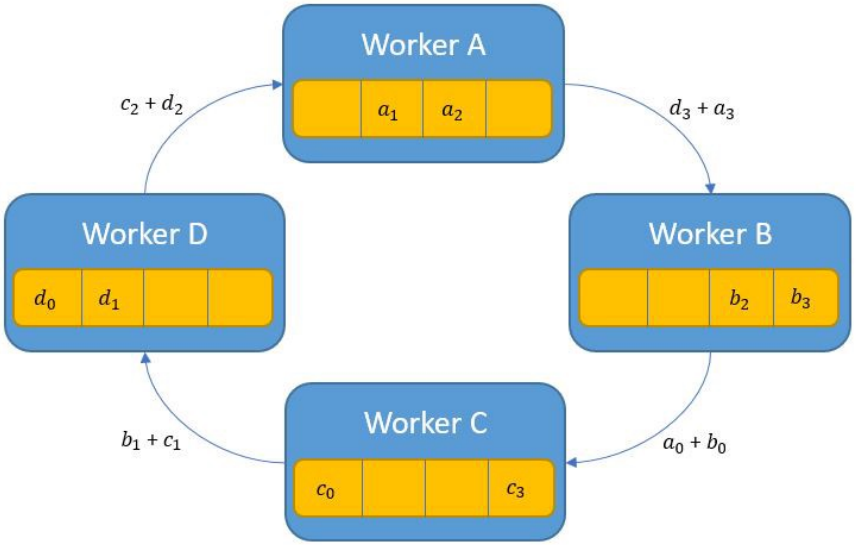
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times



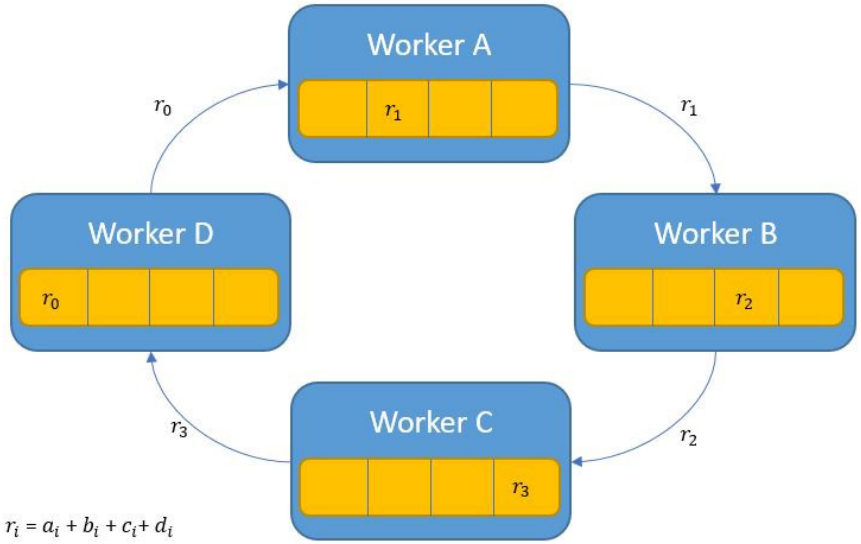
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times



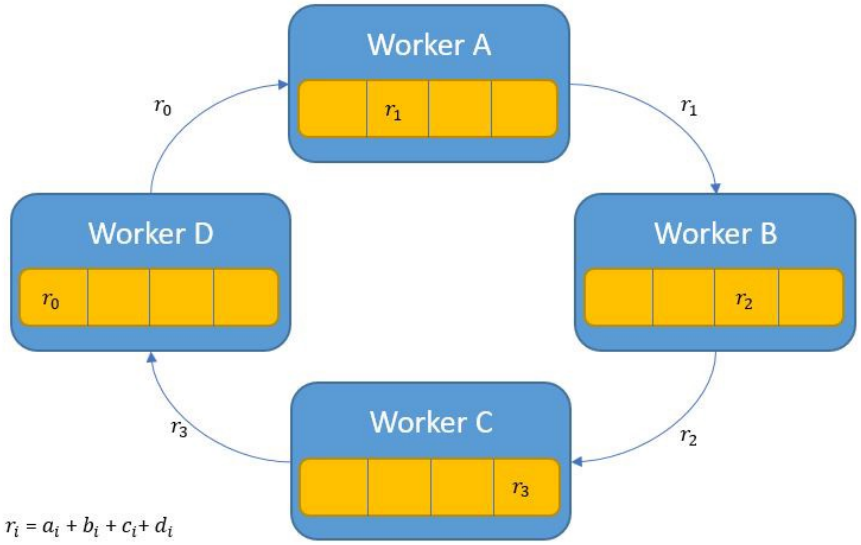
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- After step 1, each worker has the aggregated version of M/N parameters



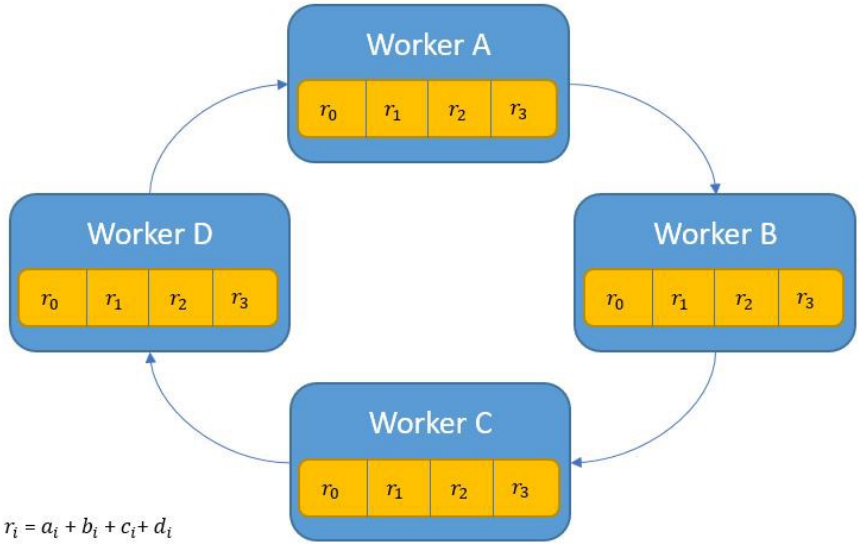
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times



Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times



Ring AllReduce

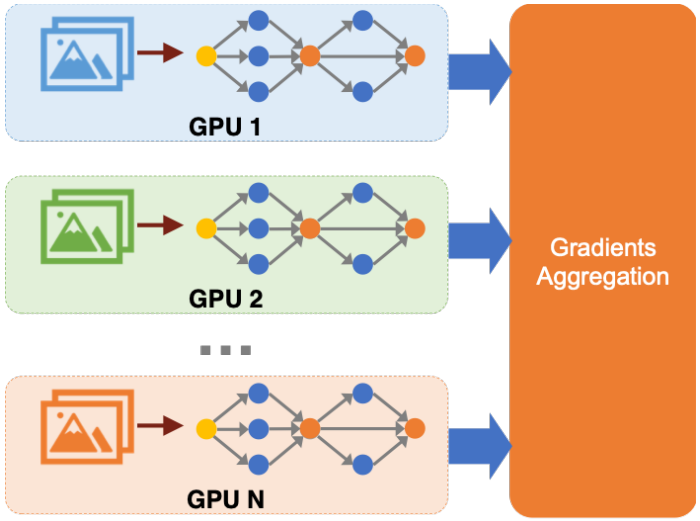
- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times
- Overall communication: 2 * M * N parameters
  - Aggregation: M * N parameters
  - Broadcast: M * N parameters

Other AllReduce methods

- Tree AllReduce
- Butterfly AllReduce
- Similar idea to Ring AllReduce, except use different topology to communicate across nodes
- Ring AllReduce is most popular

An Issue with Data Parallelism

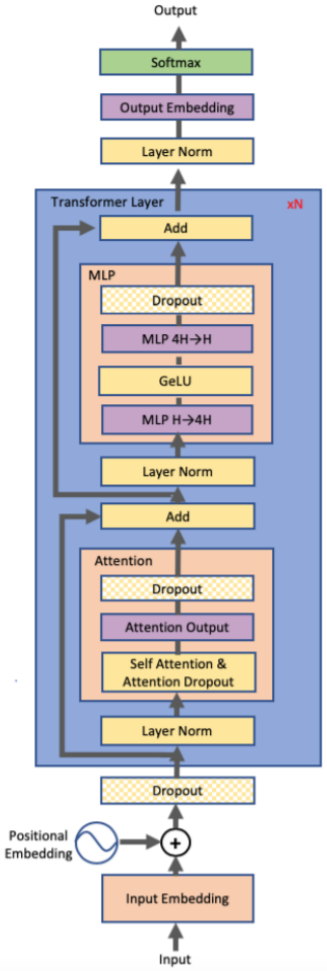
- Each GPU saves a replica of the entire model
- Cannot train large models that exceed GPU device memory



Large Model Training Challenges

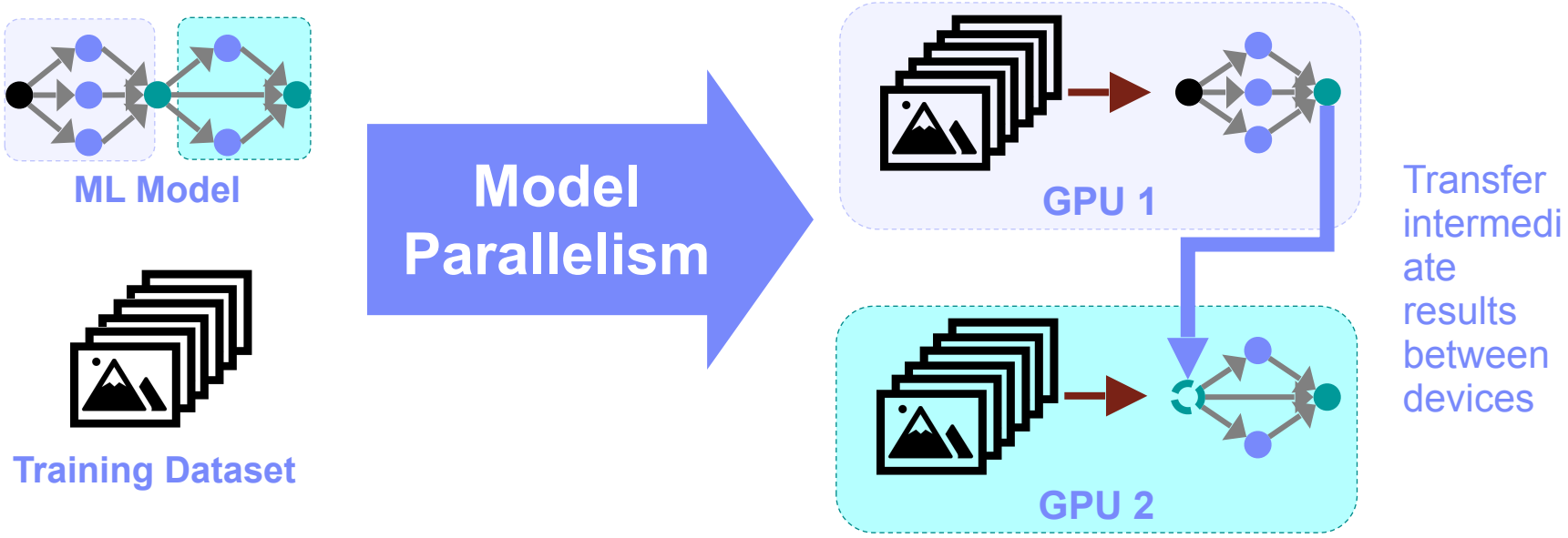
	Bert-Large	GPT-2	Turing 17.2 NLG	GPT-3
Parameters	0.32B	1.5B	17.2B	175B
Layers	24	48	78	96
Hidden Dimension	1024	1600	4256	12288
Relative Computation	1x	4.7x	**54x**	**547x**
Memory Footprint	5.12GB	24GB	**275GB**	**2800GB**

NVIDIA V100 GPU memory capacity: 16G/32G
NVIDIA A100 GPU memory capacity: 40G/80G



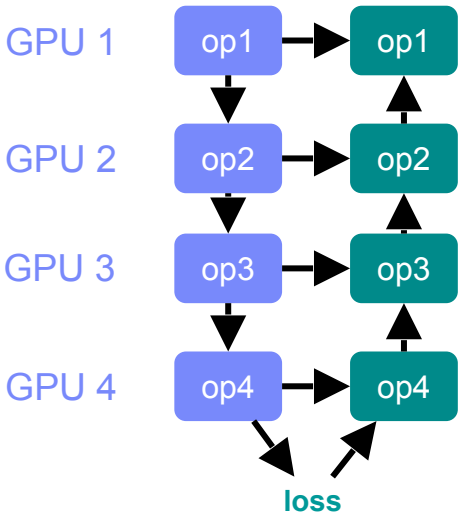
Model Parallelism

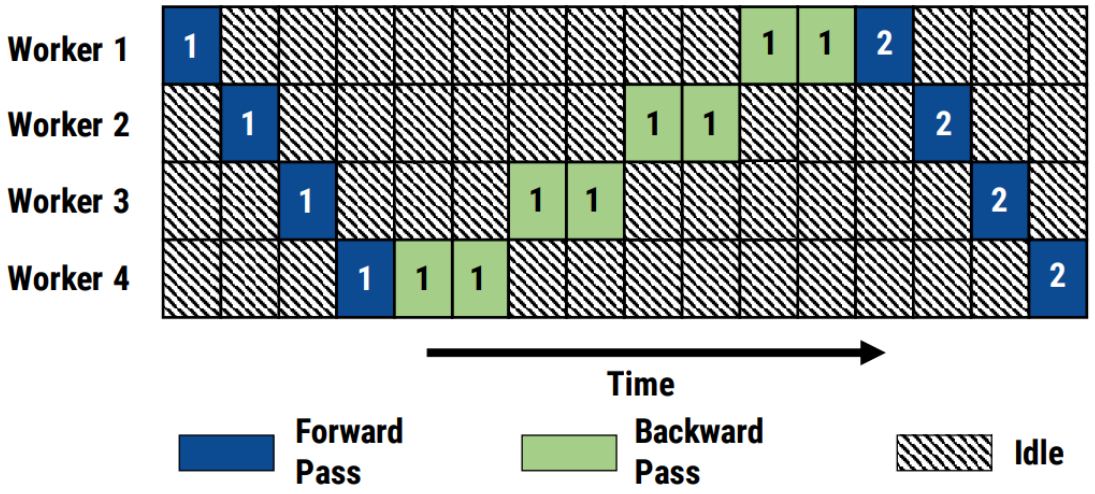
- Instead of partitioning the data across multiple nodes, partition the model across multiple nodes
- Split a model into multiple subgraphs and assign them to different devices



Model parallelism is hard to optimize

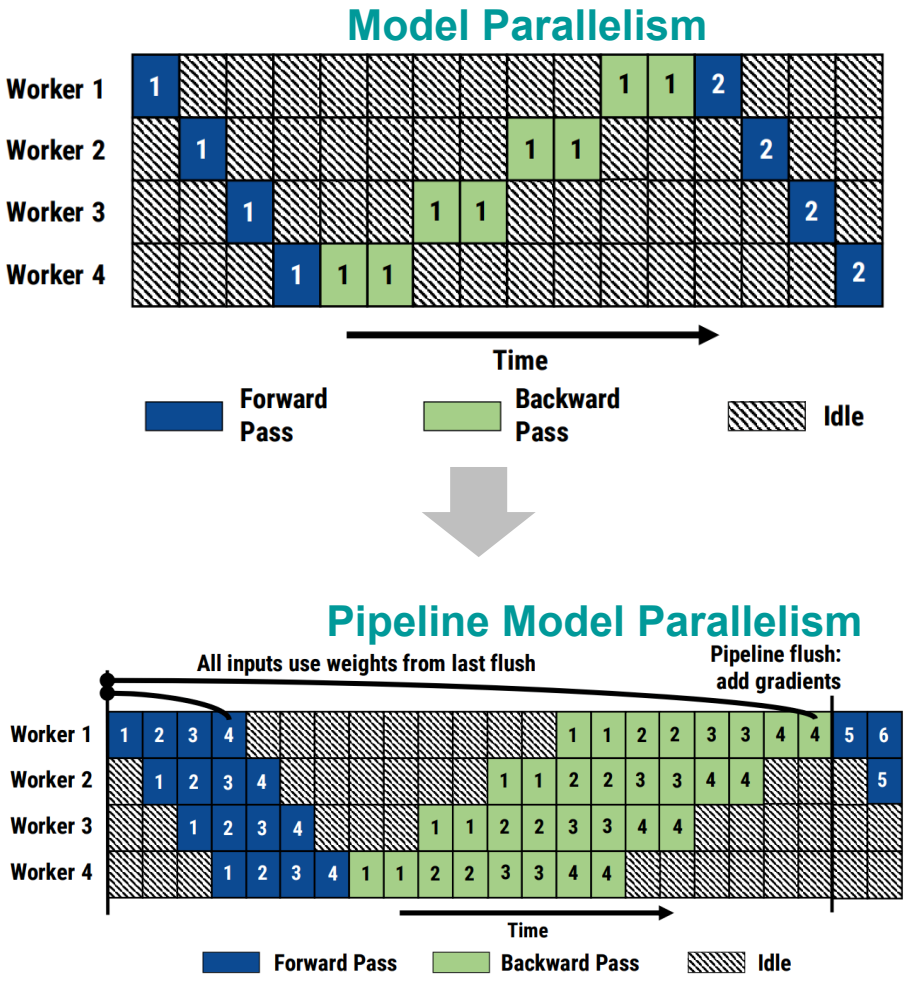
- Under-utilization of compute resources
- Low overall throughput due to resource utilization



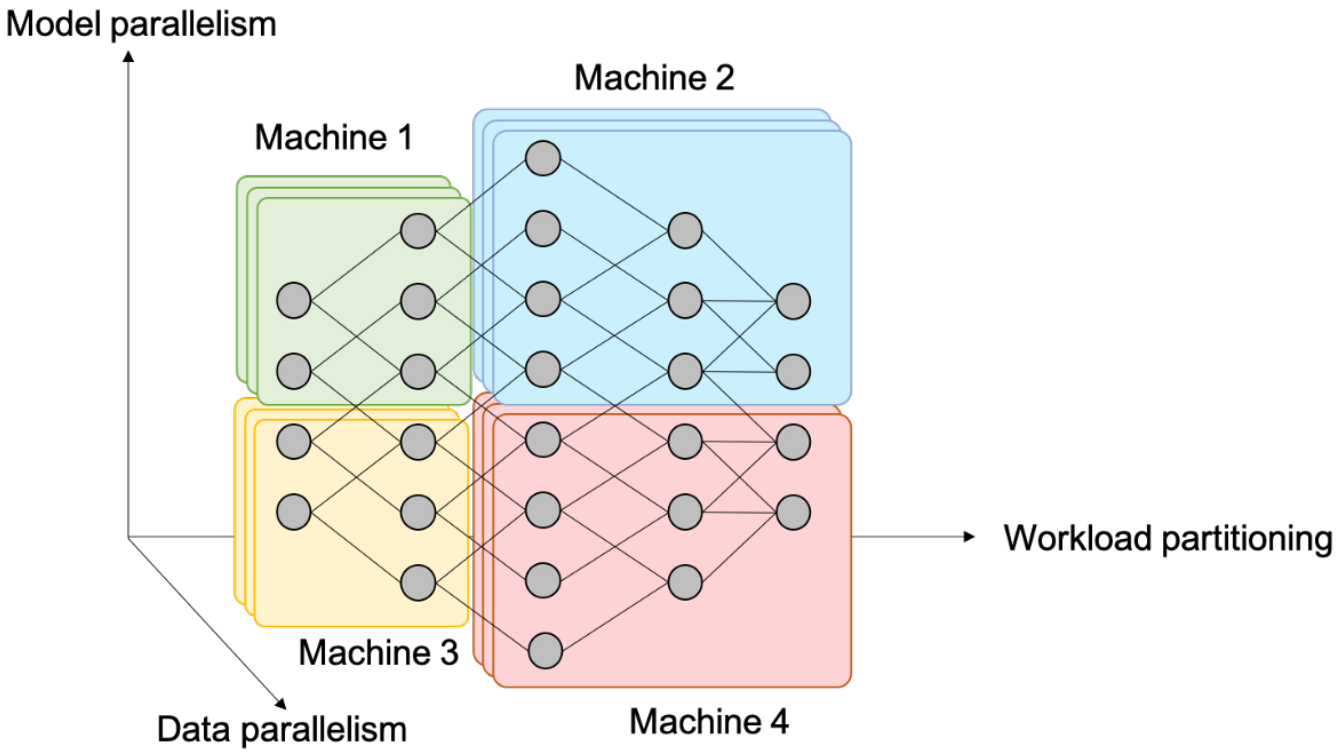


Pipeline Model Parallelism

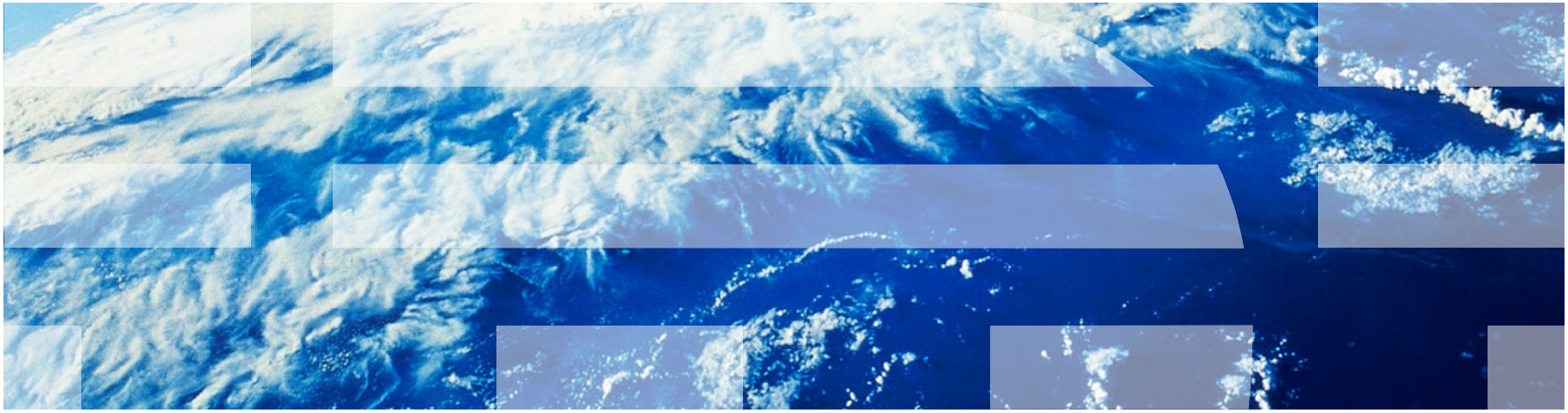
- **Mini-batch**: the number of samples processed in each iteration
- Divide a mini-batch into multiple **micro-batches**
- Pipeline the forward and backward computations across micro-batches



State-of-the-art: Combine Data and Model Parallelism



Distributed ML Inference

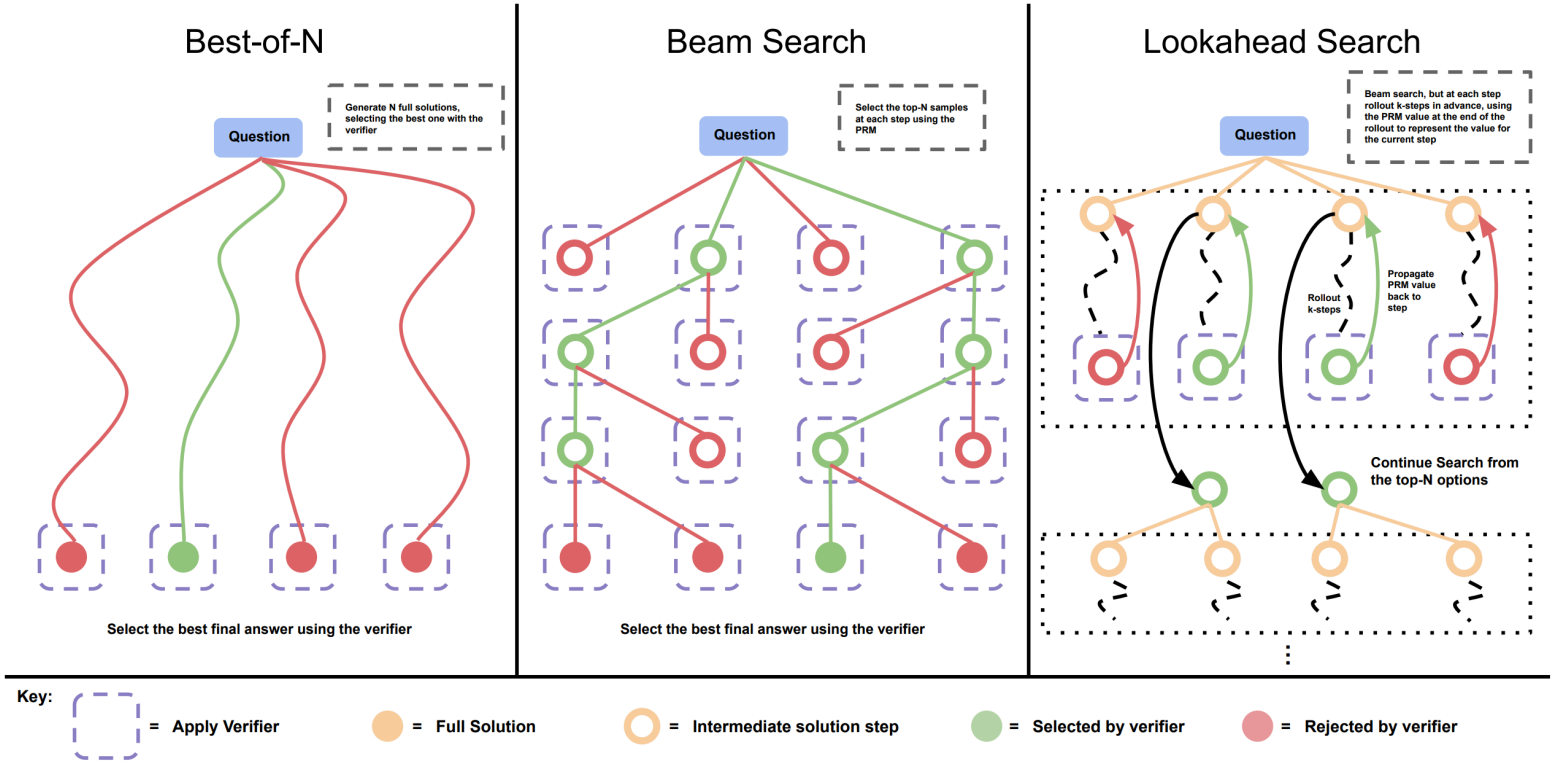


Inference: more complicated than it seems

- Historically inference was much simpler than training
  - Single forward pass, just one iteration
  - No backward pass or weight updates
- However, there are some complications
  - Companies may run many models at the same time
  - And conduct training and inference on the same GPU clusters
  - We can allow models to share parts of their memory (e.g., the KV cache in LLMs) to greatly reduce overall memory consumption
  - Goal: low inference latency, while maximizing total utilization/throughput
- Making things much more complicated: reasoning models / agents (more on that in a bit)
  - Examples: Open AI o1/o3, Deepseek R1
  - Inference is now becoming much more complex

Reasoning models: chain of thought

- Basic idea: instead of doing a “one pass” inference, create many “inference” paths composed of multiple sequences, apply a “verifier” model to select the best one



Credit: Snell, et al

What does this mean?

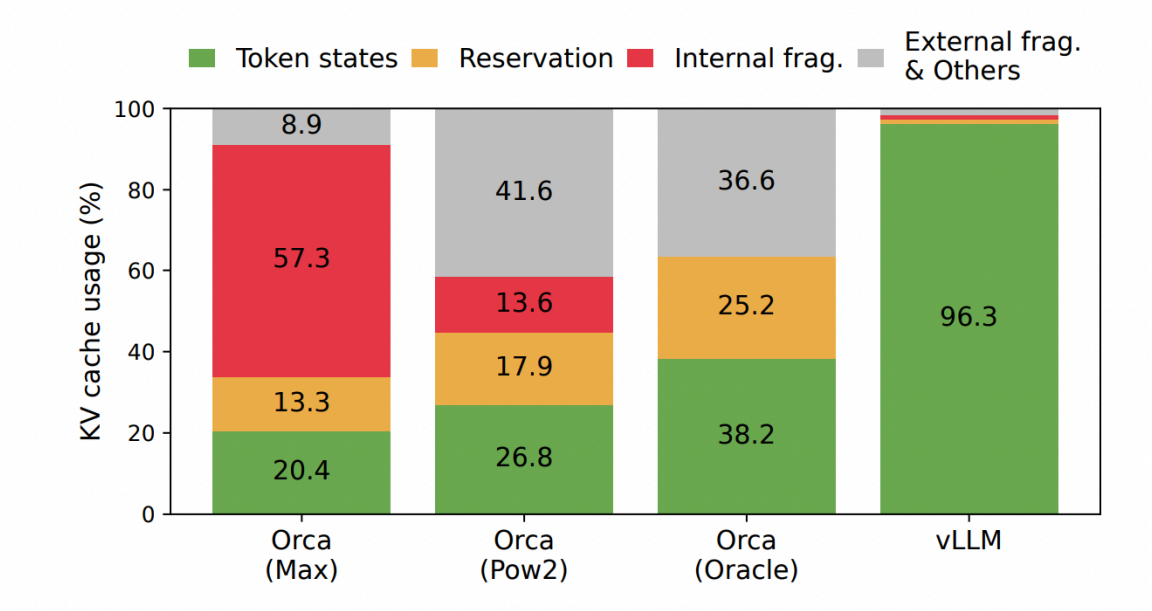
- A single “logical” inference may require thousands individual inferences in a chain of thought!
- Very useful in cases where the query result needs to be very accurate
  - Generate code
  - Generate an algorithm/blueprint
  - Interact with the real world
- Idea of agents: researchers are combining this chain of thought with allowing the model to interact with external sources
  - A vector database, search query, online websites
- Means inference becomes a lot more resource intensive and much harder to manage
  - Most of the new AI clusters being built are inference clusters

Inference/serving systems

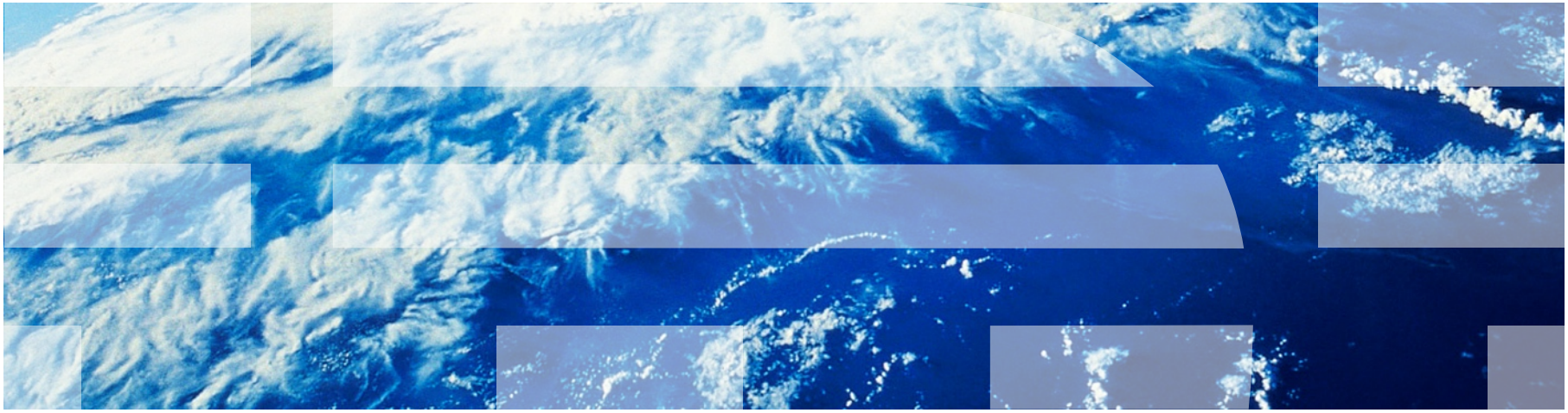
- Basic interface, complex systems:
  - Developer can deploy several models, and expose inference API
  - Serving system automatically distributes models across GPUs
  - Tries to “pack” models on GPU while maintaining latency
- Many options

Memory efficiency is essential

- Challenging because we don’t know in advance how much memory a model will need to allocate
- Try to reduce memory “wastage” within a model (e.g., a model allocates more memory than it actually needs)
- Try to share memory across models (e.g., KV cache) to minimize memory consumption



RAG and Vector Databases



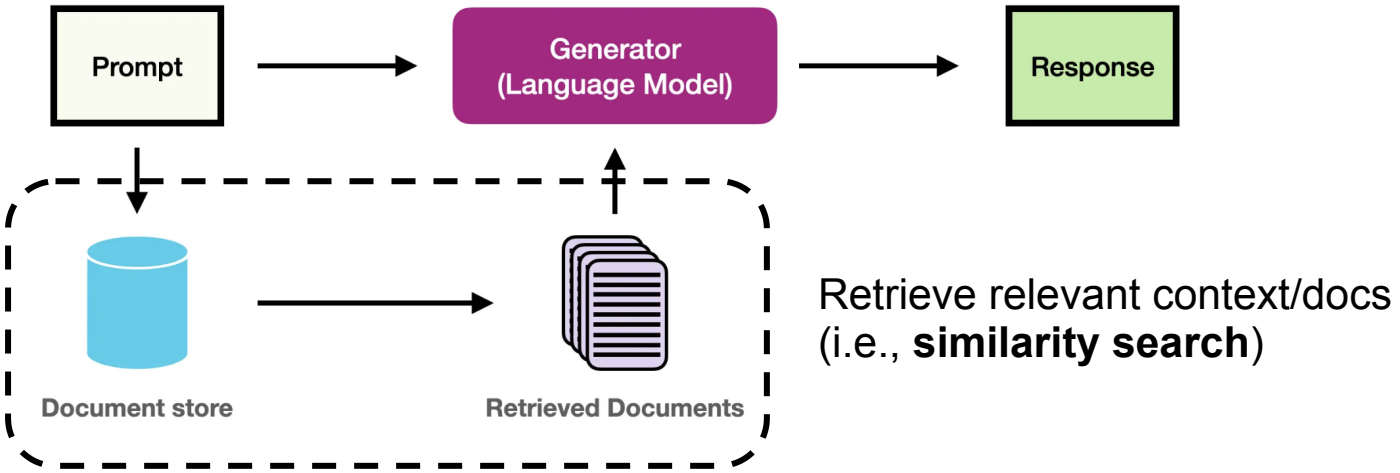
RAG: Motivation

- The context window of LLMs is limited
- Sometimes we want to “augment” the context with private information that the model was not trained on
  - E.g., prior product support chats, internal company documents
- RAG / vector databases to the rescue
  - Basic idea: augment the prompt with data from an external “database”
  - This database is somewhat different than a traditional database: instead of storing raw data, it stores “vectors” that represent the data
  - Instead of computing SQL queries, it tries to find the K nearest neighbors
- Analogy: context (stored in GPU memory) is like “memory” and RAG (typically stored on another CPU-based node) is like “disk”

RAG and Similarity Search

Retrieval-Augmented Generation (RAG) has many popular use cases:

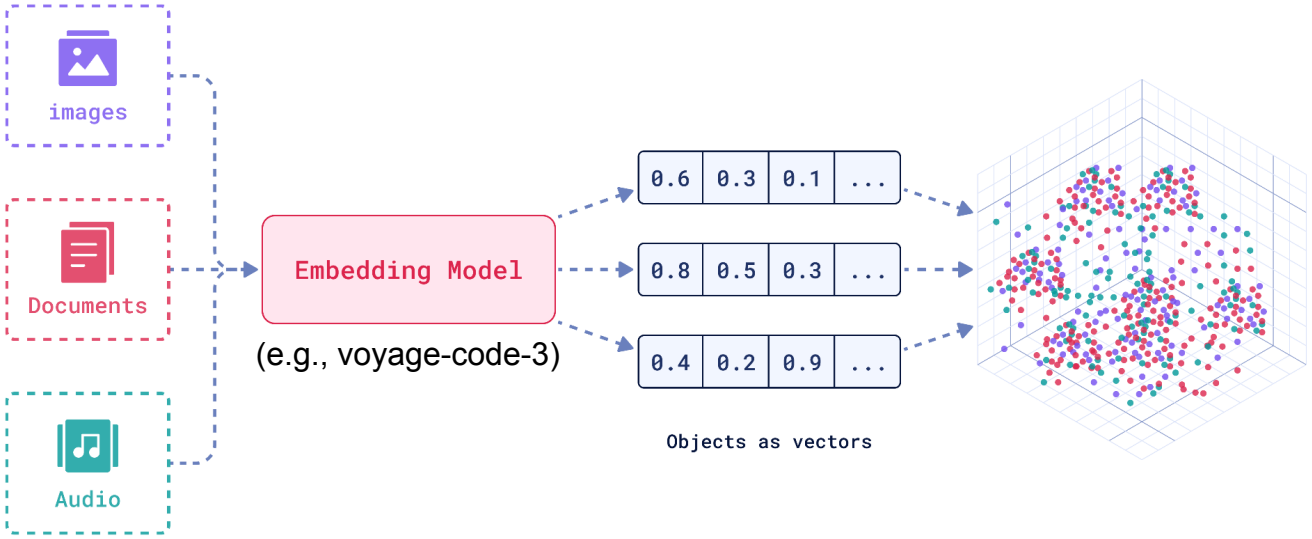
- LLM-backed chatbot
- Context-aware code generation/completion



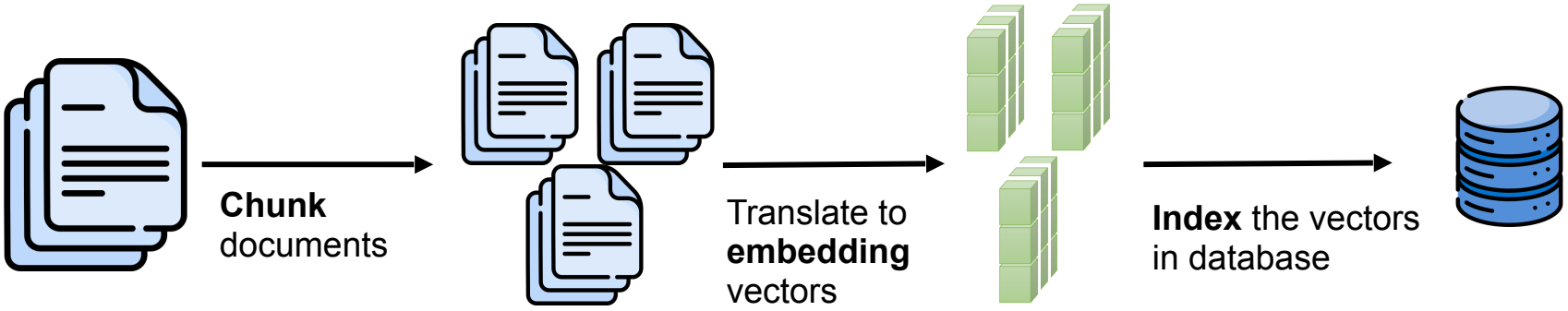
Similarity Search: Embedding Vectors

Embedding models can map a text chunk into an **embedding vector**

- Embedding vector encodes the **semantic info** of the text chunk
- Relevant texts have **similar** embedding vectors



Vector Databases for Similarity Search



Challenge: **How to find similar vectors in the database efficiently?**

- **Index** the vectors to facilitate similarity search

Vector Indices

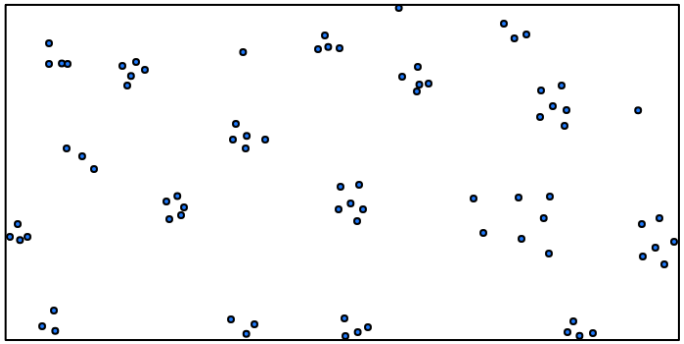
Problem definition: For a set of vectors, how to index them so that when given a query vector, we can find the **top-k *approximate* nearest neighbor** vectors efficiently?

Two types of vector indices:

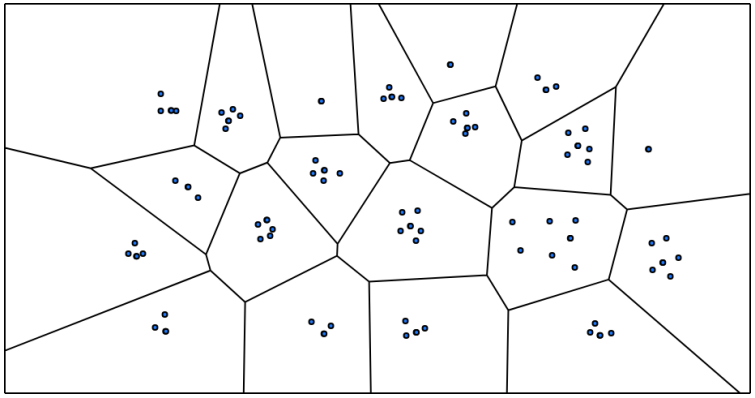
- Clustering indices
- Graph indices

Clustering Vector Indices

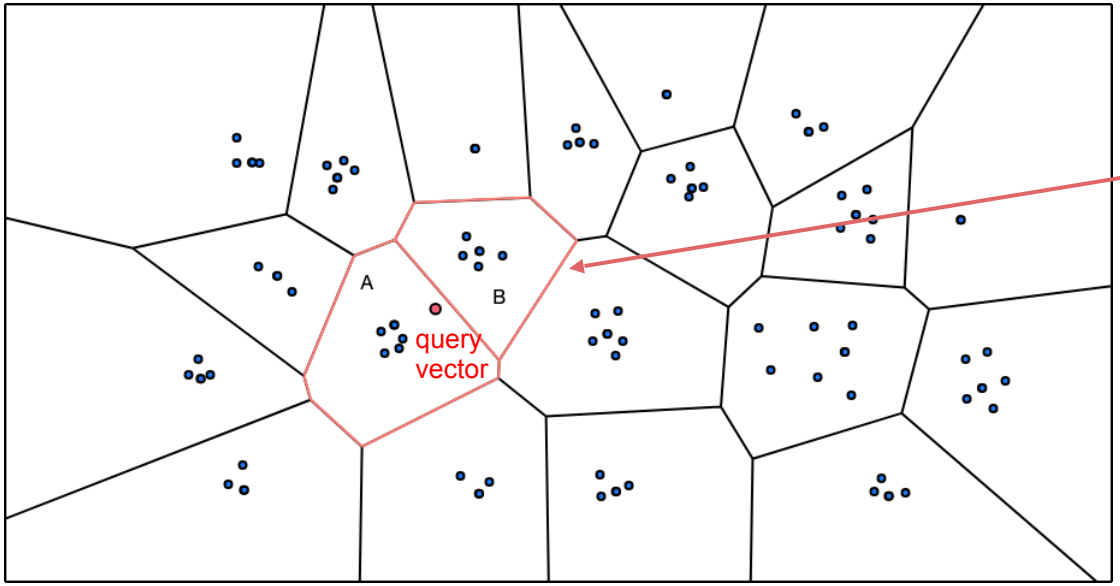
Given a set of vectors:



Run clustering algo (e.g., k-means)



Clustering Vector Indices

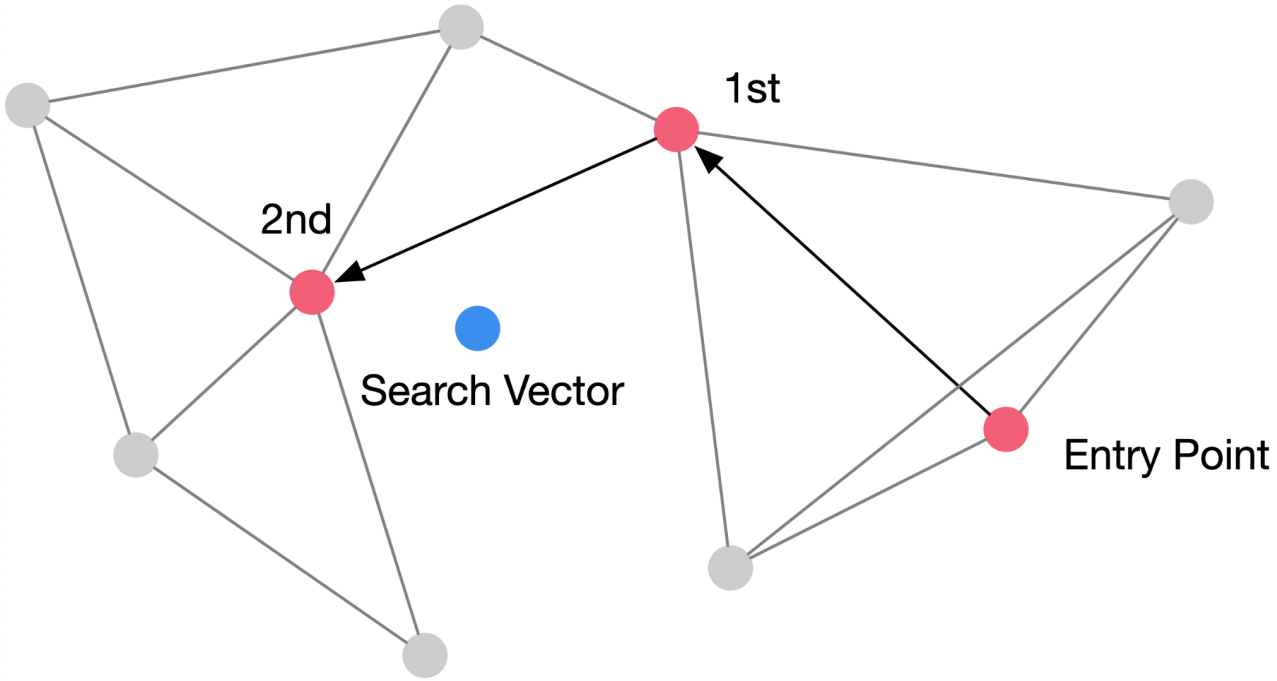


Only search in the clusters that are close to the query vector

Example cluster indices: IVFF, SPANN

Challenge: Many vectors could be **at the boundary of** different clusters

Graph Vector Indices



Example graph indices: HNSW, NSW, DiskANN

Challenges of Vector Indices

Vector databases can be HUGE:

- Search engines that need to index billions of web pages
- Code completion RAGs that encompass the entire codebase in Microsoft

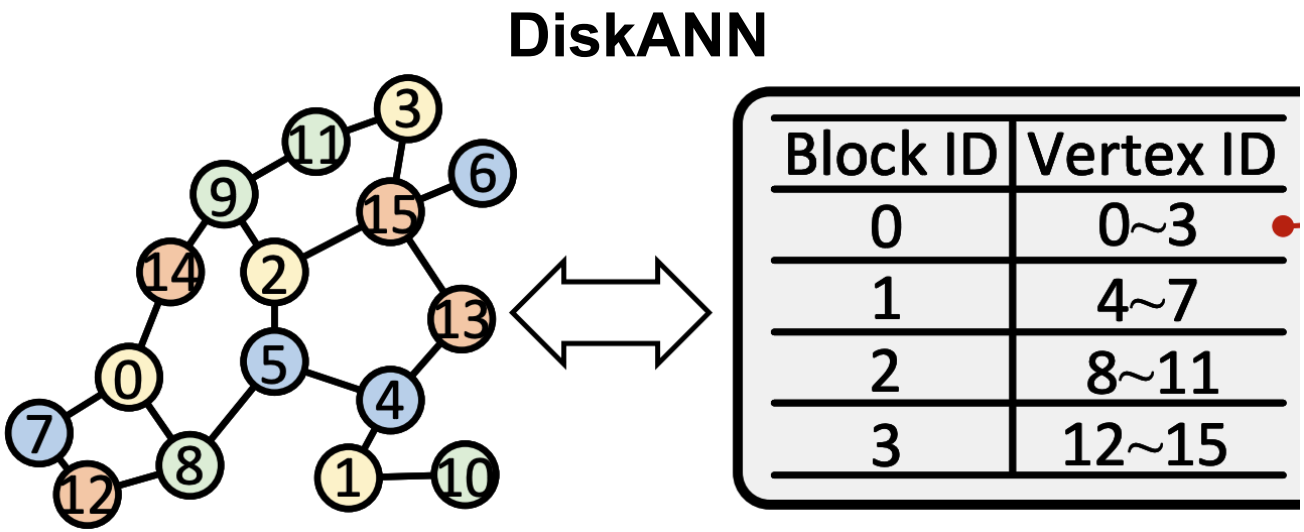
For example, SPACEV1B dataset (a search dataset) has 1.4 billion vectors:

- Vector data: 1.4 * 10^9 * 100 dimension * 1B per dimension = **130 GB**
- Neighbor list: 1.4 * 10^9 * 64 neighbors per node * 4B per neighbor = **334 GB**

Challenge: Infeasible to store the entire vector index **in the memory of a single server**

On-Disk Vector Indices

- On-disk graph index: DiskANN, Starling
- On-disk cluster index: SPANN, SPFresh



Caching Is Important!

Disks have **limited bandwidth**, which limits the max throughput of vectorDB

- Samsung 990 EVO 5.0 NVMe SSD 2TB: $130, **5 GB/s**
- Seagate FireCuda 540 SSD 2TB: $270, **10 GB/s**

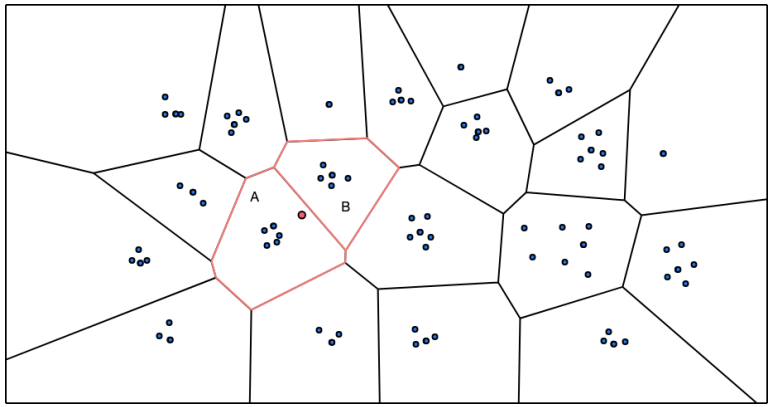
Disks also have **bandwidth amplification** issues since the block size is >= 4KB, but each node in a graph could be much smaller

- E.g., 100B + 64 * 4B = 356B, assuming 64 neighbors

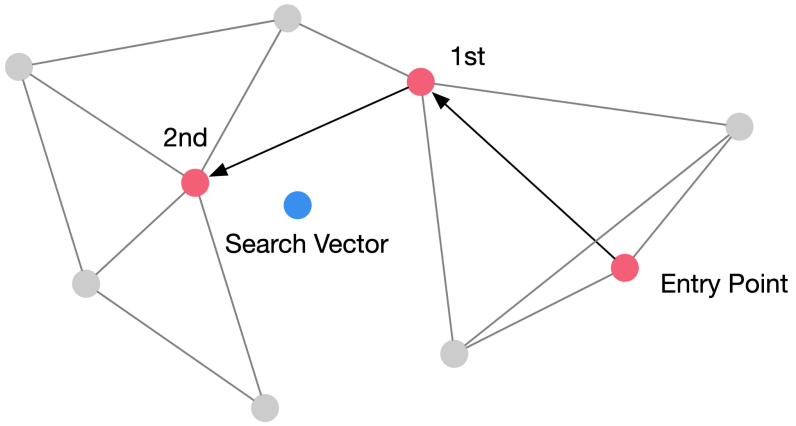
Also, the access distribution is always **skewed**; only few vectors are popular

How to cache? Clustering Indices vs. Graph Indices

clustering



graph



Many open challenges in vector databases

- How to efficiently update/delete vectors
  - Today most vector databases are read-only
- How to efficiently cache them
- How to partition vector databases across multiple servers