Introduction to Apache Beam RAG

Claude van der Merwe

Wednesday, July 09, 2025

BEAM SUMMIT
NYC 2025

Agenda

- Overview of Embeddings, Vector Search, Chunking and RAG
- Purpose of Apache Beam RAG module
- RAG Module - Chunking
- RAG Module - Ingestion
- RAG Module - Vector search
- Example use cases

Embeddings

What are Embeddings?

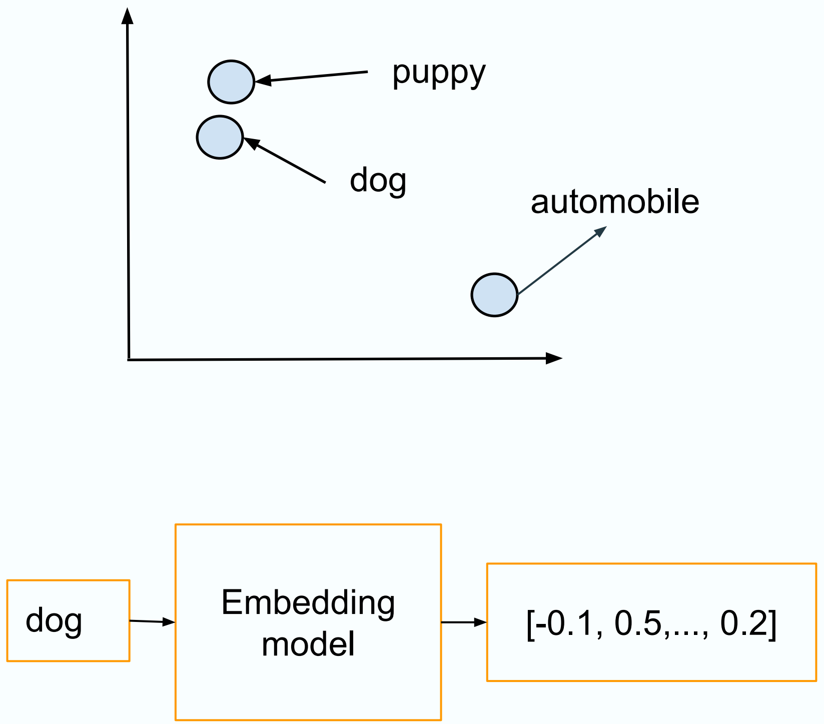
- Mathematical representation of meaning
- Dense vectors with hundreds of dimensions
- Convert text and images into numerical form

How Embeddings Work:

- Data/concepts with related meanings cluster together
- Mathematical distance = semantic distance

Embedding Models:

- Commercial: Vertex AI Gecko
- Open source: Hugging Face Sentence Transformers
- Custom: Fine-tuned for specific domains



Vector Search

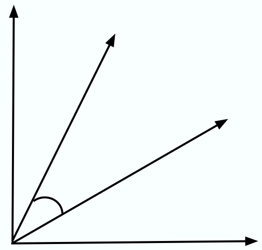
Goal: Find semantically similar data by calculating distance between vectors

```
VectorSearch(
  embedding=EmbeddingModel(text='dog'),
  distance_metric=euclidean_distance
)
```

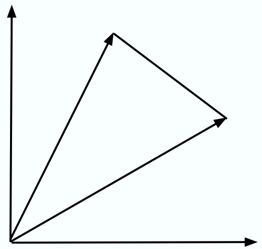
```
[
 {text='dog', distance=0.0},
 {text='puppy', distance=0.02},
 {text='automobile', distance=0.806}
]
```

Text	Vector Representation
dog	`[0.1, 0.9]`
puppy	`[0.1, 0.88]`
automobile	`[0.9, 0.1]`

Cosine



Euclidean



Chunking

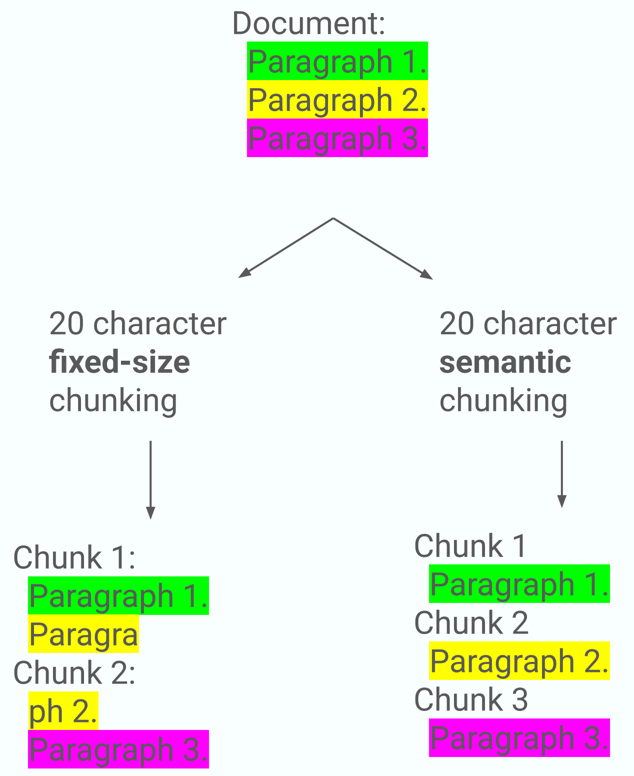
Chunking splits large documents into smaller units of text

Why use Chunking?

- Documents often exceed embedding model token limits
- Chunks enable more precise retrieval

Common Chunking Strategies:

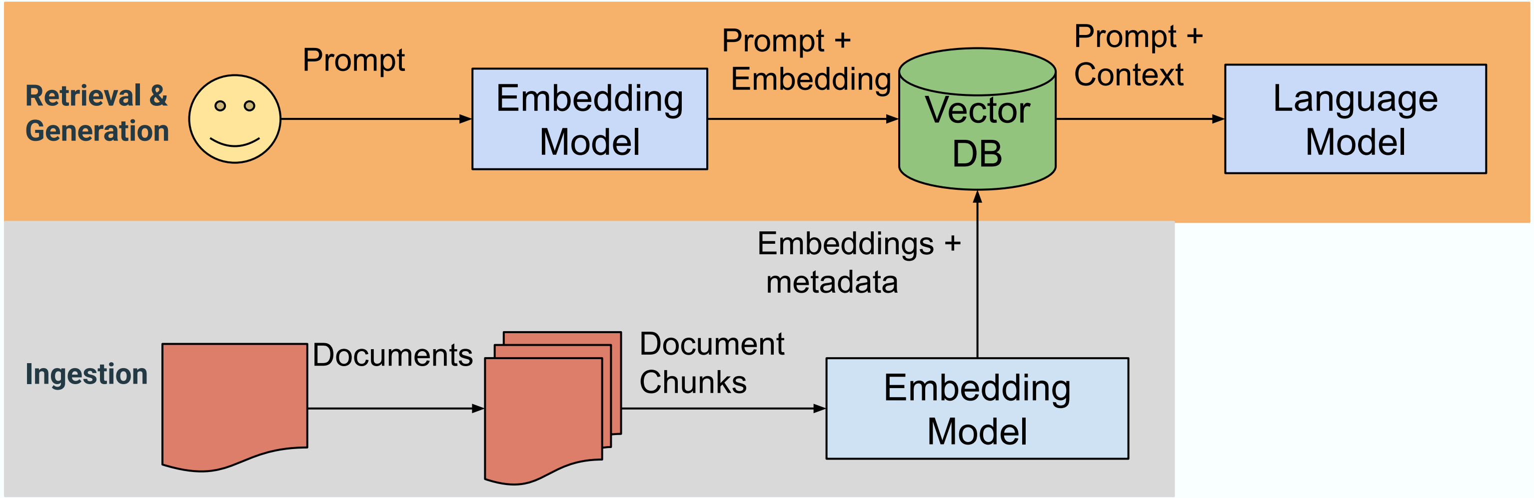
- Fixed-size chunking - Split every N tokens/characters)
- Semantic chunking -Split at natural boundaries (paragraphs, sections)



Overview of RAG

What is RAG?

- Uses vector search to enrich LLMs with external knowledge
- Creates grounded, accurate AI responses
- Bridges the gap between static LLM knowledge and fresh data



Apache Beam Rag Module: Why?

- Apache Beam enables large scale batch and stream data processing, ML inference and ingestion
- Complexities of dealing with embeddings, vector databases and vector search can be abstracted
- Goal is to make RAG components
    - Discoverable
    - Accessible
    - Extensible
    - Interchangeable

Rag Module Requirements

An end-to-end RAG module supports

- Chunking
- Embedding generation
- Embedding ingesting
- Vector search
- LLM Inference

Rag Module Types: Chunk

A simple ingestion pipeline:

- Reads raw documents from various data sources
- (Optional) Splits documents into smaller segments
- Embeds the chunks content into dense vectors with semantic value
- Ingests the embeddings along with metadata to a database that supports vector search

Embeddable data represented by

`apache_beam.ml.rag.types.Chunk`

`Chunk` contains

- `Content` - the data to be embedded
- `Embedding` - vector that captures semantic meaning of `Content`
- `Id, index` and `metadata`

```
@dataclass
class Chunk:
  content: Content
  id: str
  index: int = 0
  metadata: Dict[str, Any]
  embedding: Optional[Embedding]

@dataclass
class Embedding:
  dense_embedding: Optional[List[float]] = None
  sparse_embedding: Optional[Tuple[List[int],
List[float]]] = None
```

Rag Module: apache_beam.ml.rag.chunking

ChunkingTransformProvider provides interface for implementing chunking strategies

```
class ChunkingTransformProvider(MLTransformProvider):
  def __init__(self, chunk_id_fn: Optional[ChunkIdFn] = None):
    ...

  @abc.abstractmethod
  def get_splitter_transform(
      self
  ) -> beam.PTransform[beam.PCollection[Dict[str, Any]],
                       beam.PCollection[Chunk]]:
    """Creates transforms that emits splits for given content."""
    raise NotImplementedError(
        "Subclasses must implement get_splitter_transform")
```

Rag Module: LangchainChunker

Input:

```
{
  'content': 'This is a simple test document. It has multiple sentences.',
  'source': 'simple.txt',
  'language': 'en'
}
```

Output:

```
Chunk(
 content='This is a simple test document',
 index=0,
 metadata={'source': 'simple.txt', 'language': 'en'},
 id='simple.txt_0'
)
Chunk(
 content='It has multiple sentences',
 index=1,
 metadata={'source': 'simple.txt', 'language': 'en'}
 id='simple.txt_1'
)
```

Code snippet:

```python
from apache_beam.ml.transforms.base import MLTransform
from apache_beam.ml.rag.chunking.langchain import LangChainChunker
from langchain.text_splitter import RecursiveCharacterTextSplitter


# ... pipeline code
"Chunk document" >> MLTransform().with_transform(
  LangChainChunker(
    text_splitter=RecursiveCharacterTextSplitter(
      chunk_size=50,
      chunk_overlap=0,
      separators=["."]
    ),
    document_field="content",
    metadata_fields=["source", "language"],
    chunk_id_fn=lambda x: f"{x.metadata['source']}_{x.index}"
  )
)
# ... pipeline code
```

Rag Module: apache_beam.ml.rag.embeddings

- Namespace for embedding model handlers that process `Chunks`
- Input: `Chunk` => Output: `Chunk` with `Embedding`
- Includes
  - Local HuggingFace sentence-transformers
  - Remote Vertex AI embedding models

Rag Module: apache_beam.ml.rag.embeddings

Input:

```
Chunk(
 content='This is a simple test document',
 index=0,
 metadata={'source': 'simple.txt', 'language': 'en'},
 id='simple.txt_0'
)
Chunk(
 content='It has multiple sentences',
 index=1,
 metadata={'source': 'simple.txt', 'language': 'en'},
 id='simple.txt_1'
)
```

Pipeline snippet:

```python
from apache_beam.ml.transforms.base import MLTransform
from apache_beam.ml.rag.embeddings.huggingface import HuggingfaceTextEmbeddings


# ... pipeline code
'Generate Embeddings' >> MLTransform()
  .with_transform(
  HuggingfaceTextEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2")
)
# ... pipeline code
```

Output:

```
Chunk(
 content='This is a simple test document',
 …
 id='simple.txt_0',
 embedding=[0.5, 0.6, 0.7]
)
Chunk(
 content='It has multiple sentences',
 …
 id='simple.txt_1',
 embedding=[0.1, 0.2, 0.3]
)
```

Agenda

- What is the problem?
- What we were able to do?
- Cost calculation
- Our stack
- Lessons learned

Rag Module: apache_beam.ml.rag.ingestion

- `VectorDatabaseWriteConfig` provides interface for implementing vector database ingestion
- Write embedded `Chunks` to vector store
- Provide reasonable defaults with ability to utilize database specific features e.g. updating existing data, authentication and schema mapping

```
class VectorDatabaseWriteConfig(ABC):
  """Abstract base class for vector database configurations in RAG pipelines.
  @abstractmethod
  def create_write_transform(self) -> beam.PTransform[Chunk, Any]:
    """
    Creates a PTransform that writes embeddings to the vector database.
    """
```

Rag Module: apache_beam.ml.rag.ingestion

```
Input:
Chunk(
 content='This is a simple test document',
 index=0,
 metadata={'source': 'simple.txt', 'language': 'en'},
 id='simple.txt_0',
 embedding=[0.5, 0.6, 0.7]
)
Chunk(
 content='It has multiple sentences',
 index=1,
 metadata={'source': 'simple.txt', 'language': 'en'}
 id='simple.txt_1'
 embedding=[0.1, 0.2, 0.3]
)
```

BigQuery table: document_embeddings

content	embedding	id	metadata
This is a simple test document	[0.5,0.2...]	simple.txt_0	{language:en...}
It has multiple sentences	[0.2,0.3...]	simple.txt_1	{language:en...}

Pipeline snippet:

```
from apache_beam.ml.rag.ingestion.bigquery import BigQueryVectorWriterConfig
from apache_beam.ml.rag.ingestion.bigquery import SchemaConfig

BigQueryVectorWriterConfig(
  write_config={
      'table': 'document_embeddings',
      'create_disposition': 'CREATE_IF_NEEDED',
      'write_disposition': 'WRITE_TRUNCATE'
  },
  # Optional
  schema_config=SchemaConfig(
      schema=<BigQuery schema dictionary>,
      chunk_to_dict_fn=chunk_to_dict
  )
)

# ... pipeline code
'Write to BigQuery' >> VectorDatabaseWriteTransform(bigquery_writer_config)
# ... pipeline code
```

Rag Module: apache_beam.ml.rag.enrichment

- Enrichment transform lets you dynamically enrich data in a pipeline by doing querying a remote service
- Backed by RequestResponseIO which provides client-side throttling
- RAG module combines Enrichment transform with Vector Search

Scenario: Online Store

Consider an online store with a product catalog

BigQuery table: product_catalog

id	embedding	description	price
laptop-001	[0.1,0.2...]	Powerful ultralight laptop	1999
desk-001	[0.3,0.4...]	Sleek modern desk	149
desk-002	[0.4,0.5...]	Vintage desk	300

Rag Module: apache_beam.ml.rag.enrichment

Input:

```
{
 'query': 'powerful laptop for video editing',
'max_price': 2000
}
```

Output:

```
Chunk(
 content: 'powerful laptop for video editing',
 metadata: {
   'max_price': 2000,
   'enrichment_data': {
     'id': 'laptop-001',
     'description': 'Powerful ultralight laptop
...',
     'price': 1999
   }
 }
 embedding = [0.12, -0.03, ...]
)
```

Pipeline snippet:

```
from apache_beam.transforms.enrichment import Enrichment
from apache_beam.ml.rag.enrichment.bigquery_vector_search import (
    BigQueryVectorSearchParameters,
    BigQueryVectorSearchEnrichmentHandler
)

vector_search_params = BigQueryVectorSearchParameters(
    project='<project_id>',
    table_name='pduct_catalog',
    embedding_column="embedding",
    columns=["pice", "description"],
    metadata_restriction_template="price <= {max_price}"
    neighbor_count=1
)

pipeline
  | 'Read from PubSub' >> beam.io.ReadFromPubSub()
  | "Convert to Chunk" >> beam.Map(to_chunk)
  | 'Generate Embeddings' >> MLTransform()
    .with_transform(
      HuggingfaceTextEmbeddings(
        model_name="sentence-transformers/all-MiniLM-L6-v2"
      )
    )
  | 'Vector Search' >> Enrichment(
        BigQueryVectorSearchEnrichmentHandler(
        vector_search_parameters=vector_search_params,
        min_batch_size=1,
        max_batch_size=5
      )
  )
```

Recap: Ingestion

```
with beam.Pipeline() as p:
  _ = (
    p
    | 'Read Documents' >> beam.io.ReadFromPubSub(topic=topic_path)
    | 'Chunk and Embed' >> MLTransform()
      .with_transform(
        LangChainChunker(
            chunk_id_fn=chunk_id_fn,
            text_splitter=splitter,
            document_field="content",
            metadata_fields=["source", "language"]
        )
    ).with_transform(
      HuggingfaceTextEmbeddings(
        model_name="sentence-transformers/all-MiniLM-L6-v2")
    )
    | 'Write to Postgres' >> VectorDatabaseWriteTransform(
      CloudSQLPostgresVectorWriterConfig(
        connection_config=LanguageConnectorConfig(
            username="postgres",
            password="****",
            database_name="postgres",
            instance_name="<project>:<region>:<instance>",
          ),
        table_name='embeddings'
        )
      )
  )
```

Retrieval and Generation

```
with beam.Pipeline(options=options) as p:
    results = (
        p
        | 'Read original prompts' >> beam.io.ReadFromPubSub(topic=topic_path)
        | 'Message to Chunk' >> beam.Map(process)
        | 'Generate Embeddings' >> MLTransform()
          .with_transform(HuggingfaceTextEmbeddings(
          model_name="sentence-transformers/all-MiniLM-L6-v2")
          )
        | 'Enrich with Vector Search' >> Enrichment(
            BigQueryVectorSearchEnrichmentHandler(vector_search_parameters)
          )
        | 'Chunk to LLM prompt' >> beam.Map(to_prompt)
        | 'Generate' >> RunInference(VLLMChatModelHandler('google/gemma-2b'))
    )
```

Example use cases

- Bulk embedding generation
- Continuous embedding generation and updates
  - Set up a streaming pipeline with `PostgresVectorWriter` and `ConflictResolution` to continuously update embeddings as embedded content changes
- Combine `RunInference` and Vector Search to create scalable AI agents
- Other interesting ideas:
  - Use LLM `RunInference` to summarize document/images instead of traditional chunking
  - Use LLM to add context about the original document a `Chunk` is extracted from

Conclusion

- Apache beam enables large scale embedding generation and LLM inference
- The RAG module aims to simplify common use cases
- Any interesting use cases to discuss?
- Any feedback or suggestions?
- Find examples at https://github.com/apache/beam/tree/master/examples/notebooks/beam-ml

QUESTIONS?