Apache Beam for RAG

Ingestion and Enrichment Pipeline

Ayush Pandey

BEAM SUMMIT

September 4-5, 2024

Sunnyvale, CA. USA

Introduction

- Data Science Grad Student
- Google Summer of Code 2023 - Apache Cloudstack
- Google Summer of Code 2024 - Apache Beam
- Currently working as a ML Engineer at MicroStrategy

BEAM SUMMIT

What is RAG ?

What is RAG ?

- AI Framework, which is used for generating relevant and accurate text by combining the Large Language Models with traditional information retrieval system.
- **Retrieval** - Used for retrieval of relevant chunks from the knowledge base based on the semantic search.
- **Augment** - Augmenting the relevant chunks retrieved from knowledge base to the prompt given as input.
- **Generation** - Generating the text using LLM's based on the retrieved chunks and the given prompt.

BEAM SUMMIT

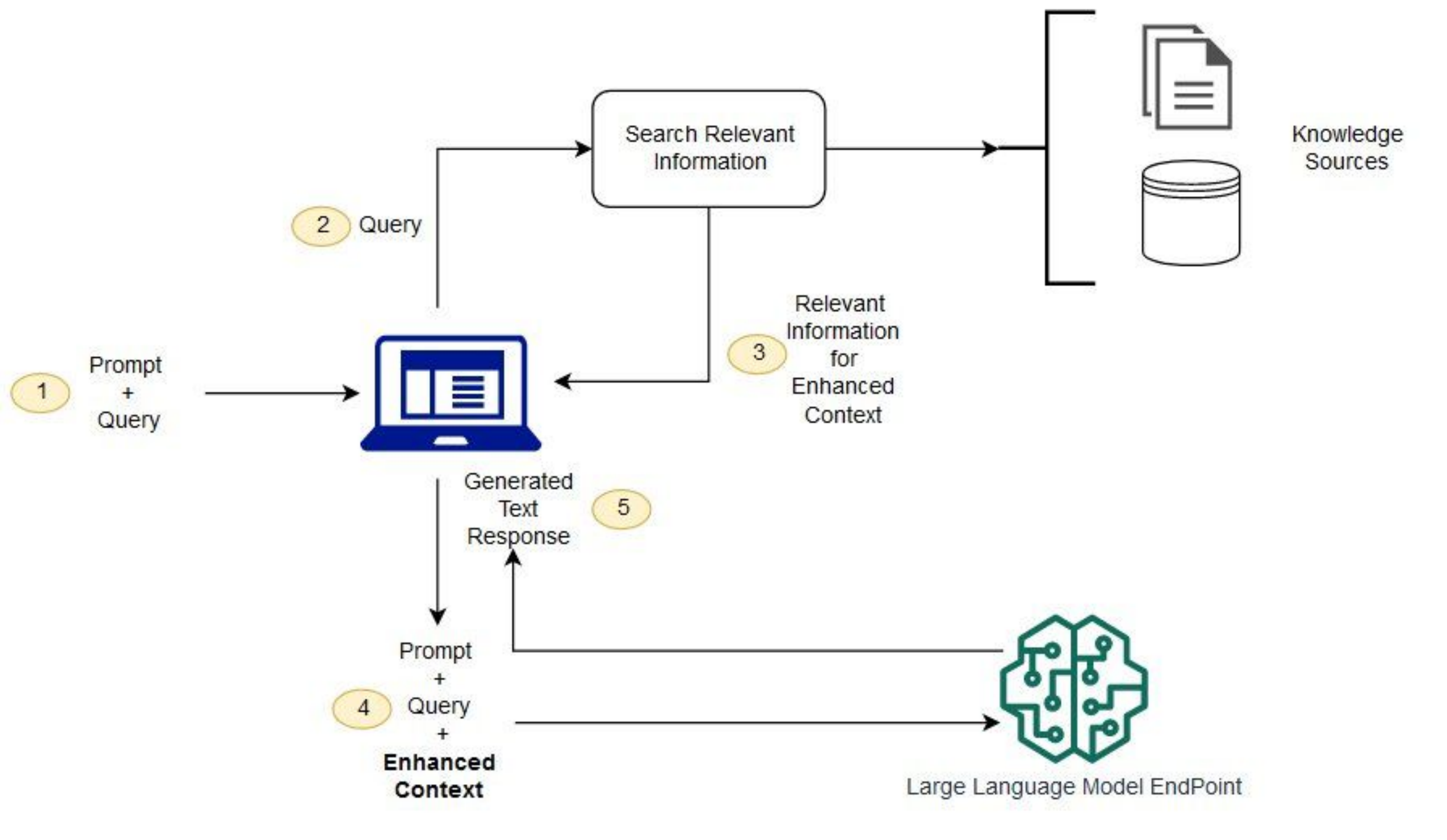
Why RAG and why is it so popular ?

Why RAG ?

- LLMs are trained on general corpora which makes it hard to generate the relevant text or domain specific text and they hallucinate.
- The foundational models are trained offline, thus becomes hard to incorporate the new data.
- To be able to make the model generate domain specific data, we need to finetune the model which is a costly and time consuming process.

BEAM SUMMIT

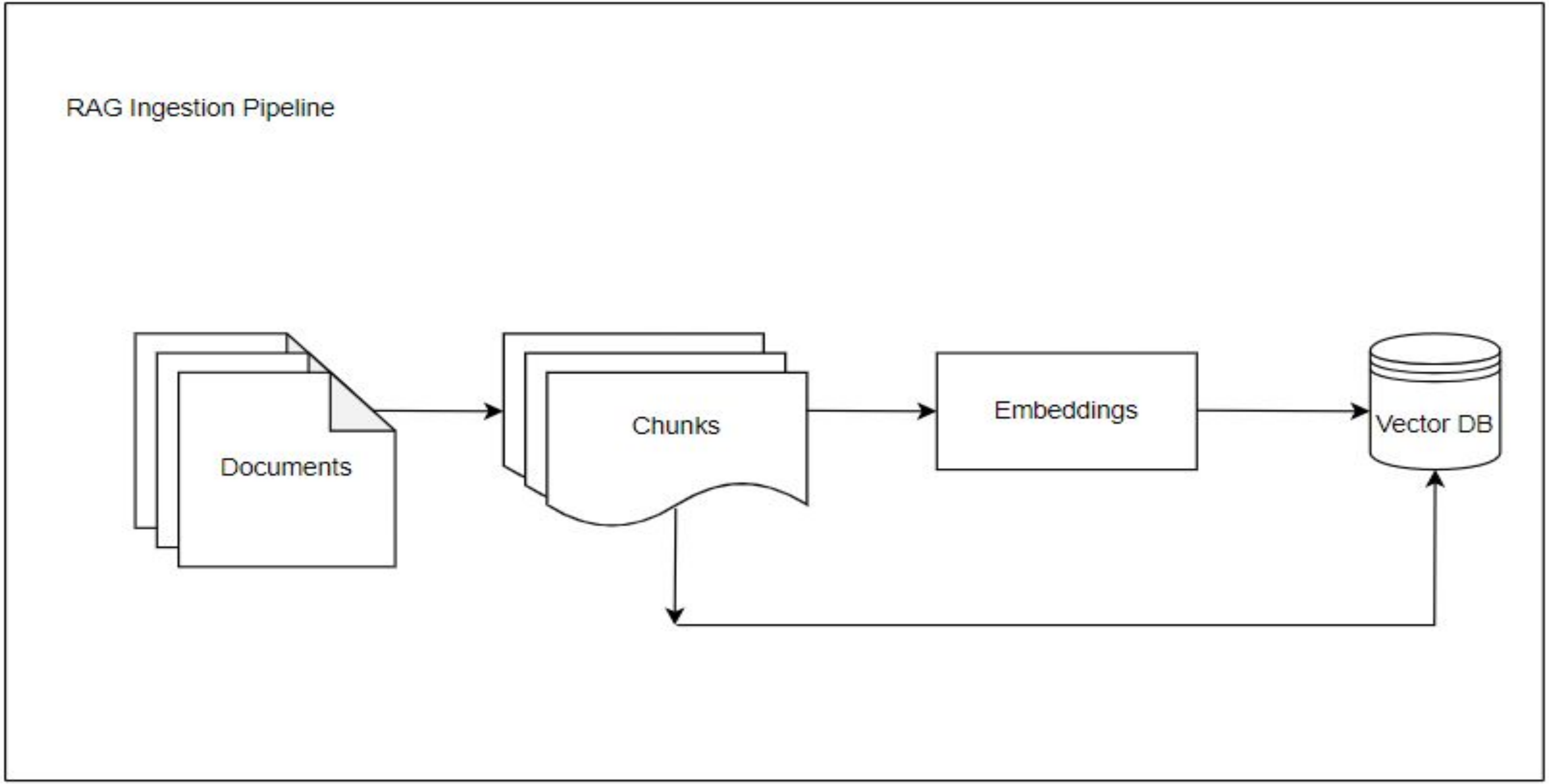
RAG Architecture

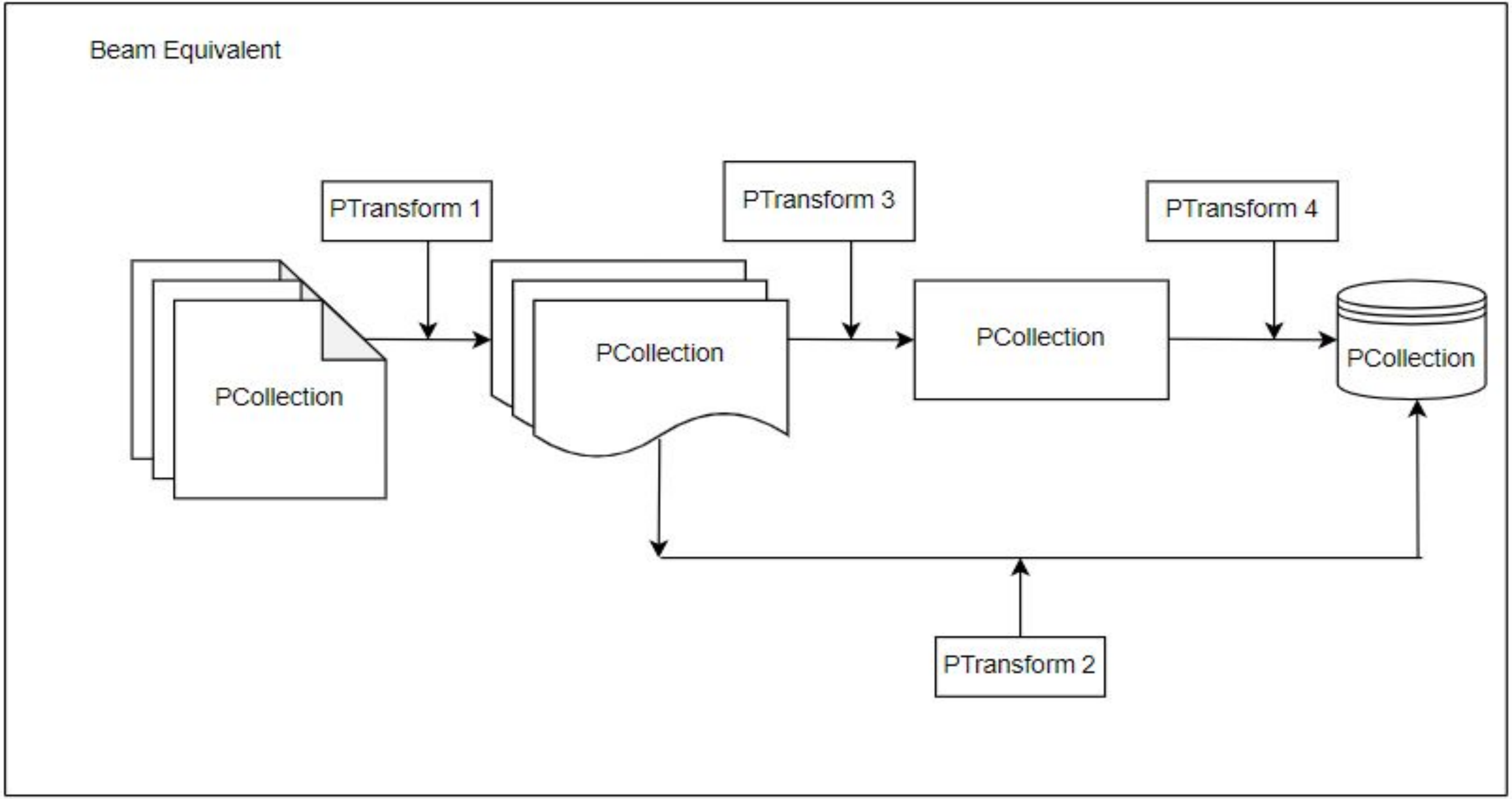

Search Relevant Information
Knowledge Sources
2 Query
Relevant Information for Enhanced Context
3
1
Prompt + Query
Generated Text Response
5
Prompt + Query + Enhanced Context
4
Large Language Model EndPoint

Beam ⇔ RAG

Implementing RAG

- Foundational Beam parts:
  - ML Transform
  - Enrichment
- Writing custom IO class for vector database
- Implementing chunking strategy
- Using Beam's MLTransform module to transform the data and use `SentenceTransformerEmbeddings` to generate embeddings.
- Used beam's `EnrichmentSourceHandler` for searching the relevant chunks in the vector database.


RAG Ingestion Pipeline
Documents
Chunks
Embeddings
Vector DB


Beam Equivalent
PTransform 1
PTransform 3
PTransform 4
PCollection
PCollection
PCollection
PCollection
PTransform 2

Ingestion

PTransform 1: Transforms document in chunks

PTransform 2: a platform aware transform to setup schema and writes document to IO Sink

PTransform 3: MLTransform's SentenceEmbeddingTransform - supports many models

PTransform 4: a platform aware transform to write embeddings into ingested document with transform 3 - nature of MLTransform => 2 inserts

Example Ingestion Pipeline

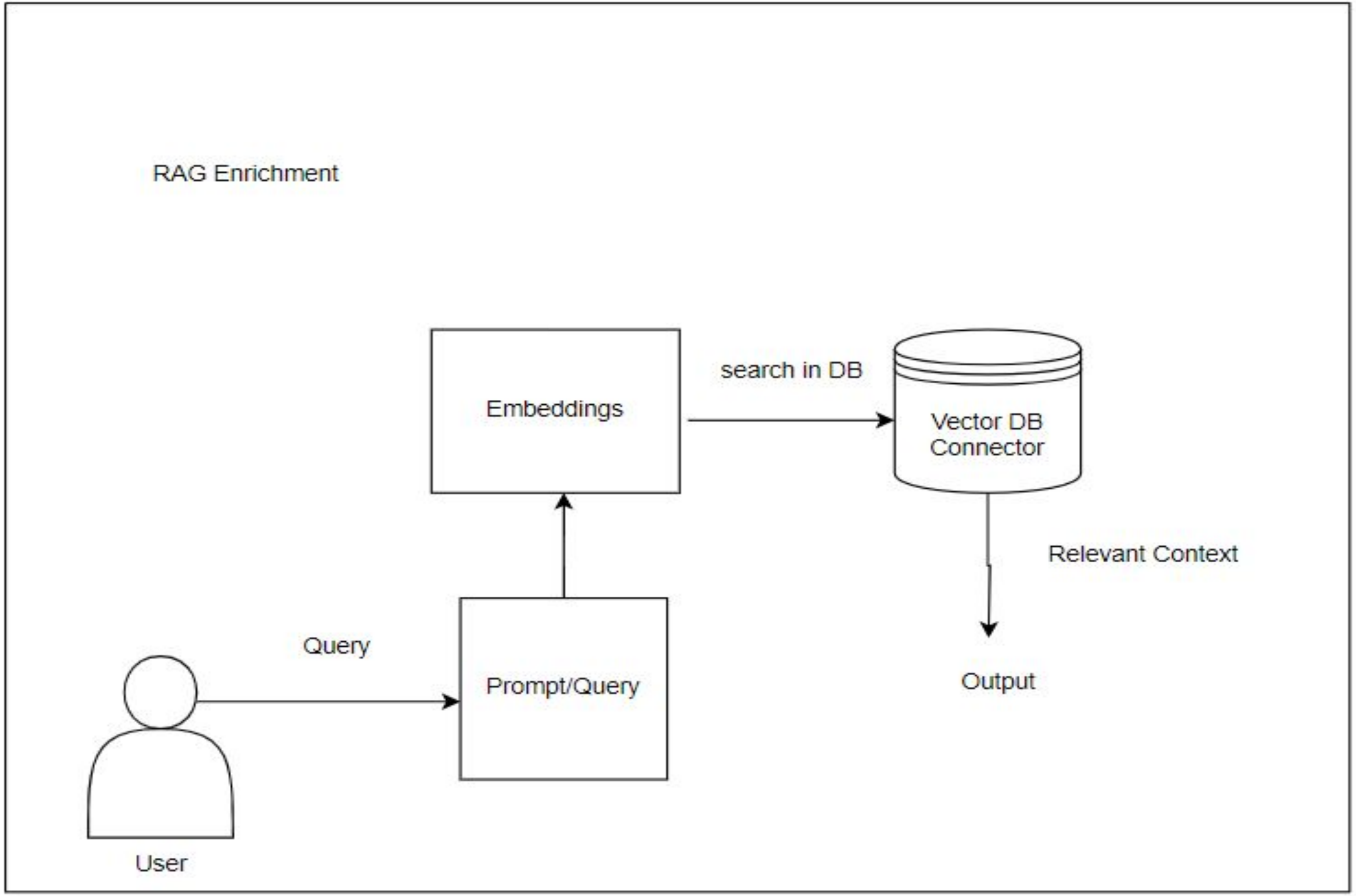
```python
#Insertion Pipeline

artifact_location = tempfile.mkdtemp()
generate_embedding_fn = SentenceTransformerEmbeddings(model_name='all-MiniLM-L6-v2',
                                                       columns=['title','text'])
with beam.Pipeline() as p:
    embeddings = (
        p
        | "Read data" >> beam.Create(contents)
        | "Generate text chunks" >> ChunksGeneration(chunk_size = 500, chunk_overlap = 0, chunking_strategy = ChunkingStrategy.SPLIT_BY_TOKENS)
        | "Insert document in Redis" >> InsertDocInRedis(host='127.0.0.1',port=6379, batch_size=10)
        | "Generate Embeddings" >> MLTransform(write_artifact_location=artifact_location).with_transform(generate_embedding_fn)
        | "Insert Embedding in Redis" >> InsertEmbeddingInRedis(host='127.0.0.1',port=6379, batch_size=10,embedded_columns=['title','text'])
    )
```

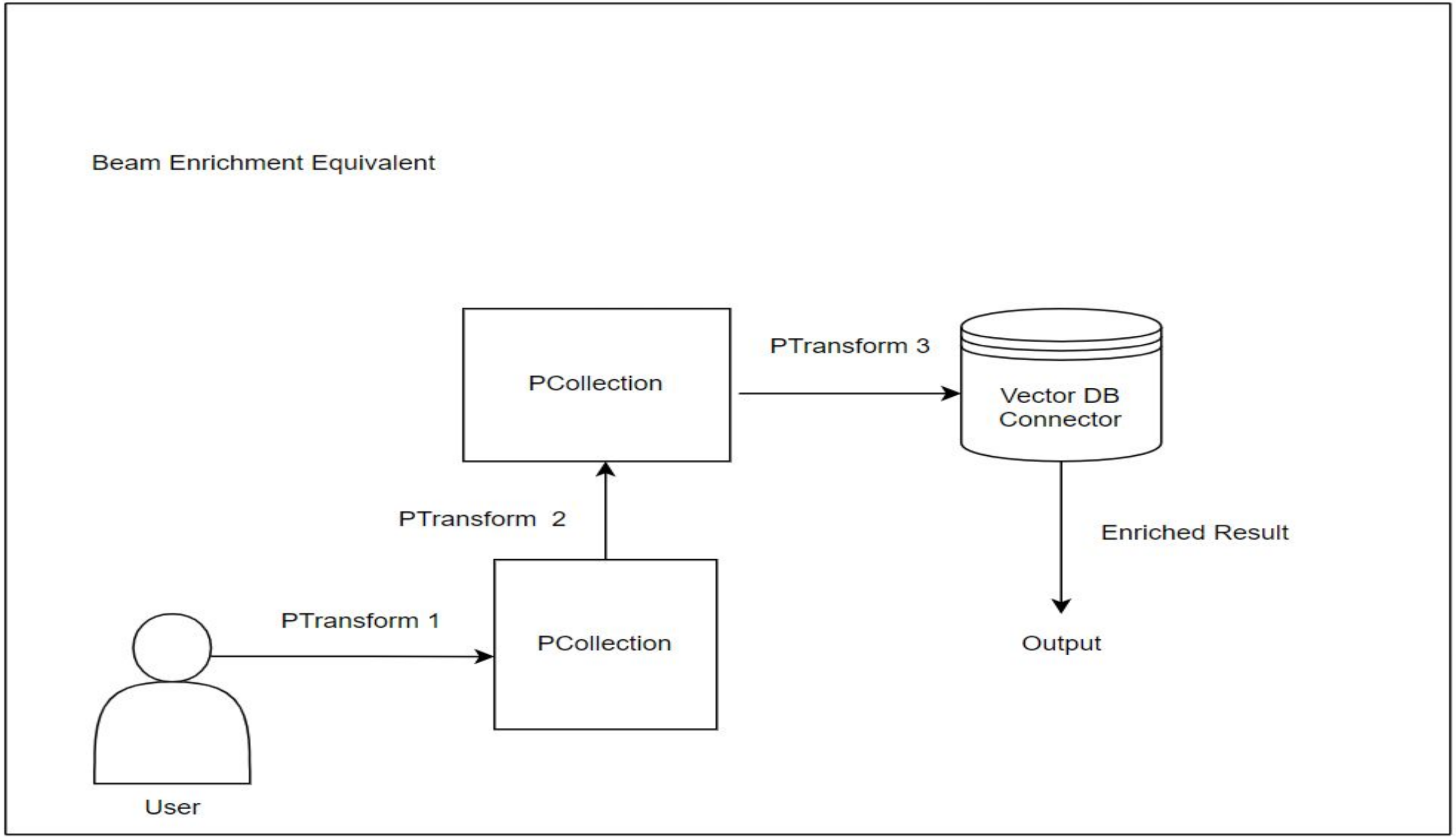
Python

```python
class ChunksGeneration(PTransform):
    """ChunkingStrategy is a ``PTransform`` that takes a ``PCollection`` of
    key, value tuple or 2-element array and generates different chunks for documents.
    """

    def __init__(
            self,
            chunk_size: int,
            chunk_overlap: int,
            chunking_strategy: ChunkingStrategy
    ):
        """

        Args:
        chunk_size : Chunk size is the maximum number of characters that a chunk can contain
        chunk_overlap : the number of characters that should overlap between two adjacent chunks
        chunking_strategy : Defines the way to split text

        Returns:
        :class:`~apache_beam.transforms.ptransform.PTransform`

        """

        self.chunk_size = chunk_size
        self.chunk_overlap = chunk_overlap
        self.chunking_strategy = chunking_strategy

    def expand(self, pcoll):
        return pcoll \
               | "Generate text chunks" >> beam.ParDo(_GenerateChunksFn(self.chunk_size,
                                                                        self.chunk_overlap,
                                                                        self.chunking_strategy))


class _GenerateChunksFn(DoFn):
    """Abstract class that takes in ptransform
    and generate chunks.
    """

    def __init__(
            self,
            chunk_size: int,
            chunk_overlap: int,
            chunking_strategy: ChunkingStrategy
    ):

        self.chunk_size = chunk_size
        self.chunk_overlap = chunk_overlap
        self.chunking_strategy = chunking_strategy

    def process(self, element, *args, **kwargs):

        # For recursive split by character
        if self.chunking_strategy == ChunkingStrategy.RECURSIVE_SPLIT_BY_CHARACTER:
            text_splitter = RecursiveCharacterTextSplitter(
                chunk_size=self.chunk_size,
                chunk_overlap=self.chunk_overlap,
                length_function=len,
                is_separator_regex=False,
            )

        # For  split by character
        elif self.chunking_strategy == ChunkingStrategy.SPLIT_BY_CHARACTER:
            text_splitter = CharacterTextSplitter(
                chunk_size=self.chunk_size,
```

BEAM SUMMIT

Enrichment Pipeline


RAG Enrichment
Embeddings
search in DB
Vector DB Connector
Relevant Context
Query
Prompt/Query
Output
User


Beam Enrichment Equivalent
PCollection
PTransform 3
Vector DB Connector
PTransform 2
Enriched Result
PTransform 1
PCollection
Output
User

Retrieval/Enrichment transforms

PTransform 1: transforming data to PCollection using root PTransform beam.create for loading user questions in batch mode

PTransform 2: transforming prompt questions PCollection to embedding PCollection using MLTransform as before

Enrichment: load additional context from vector DB to enrich user's questions before augmented generation

Example Enrichment Pipeline

```python
#  Enchriment Pipeline


data = [{'text':'What is Anarchy ?'}]

artifact_location = tempfile.mkdtemp()
generate_embedding_fn = SentenceTransformerEmbeddings(model_name='all-MiniLM-L6-v2',
                                                      columns=['text'])

redis_handler = RedisEnrichmentHandler(redis_host='127.0.0.1', redis_port=6379)


with beam.Pipeline() as p:
  _ = (
      p
      | "Create" >> beam.Create(data)
      | "Generate Embedding" >> MLTransform(write_artifact_location=artifact_location).with_transform(generate_embedding_fn)
      | "Enrich W/ Redis" >> Enrichment(redis_handler)
      | "Print" >> beam.Map(print)
  )
```

[10]

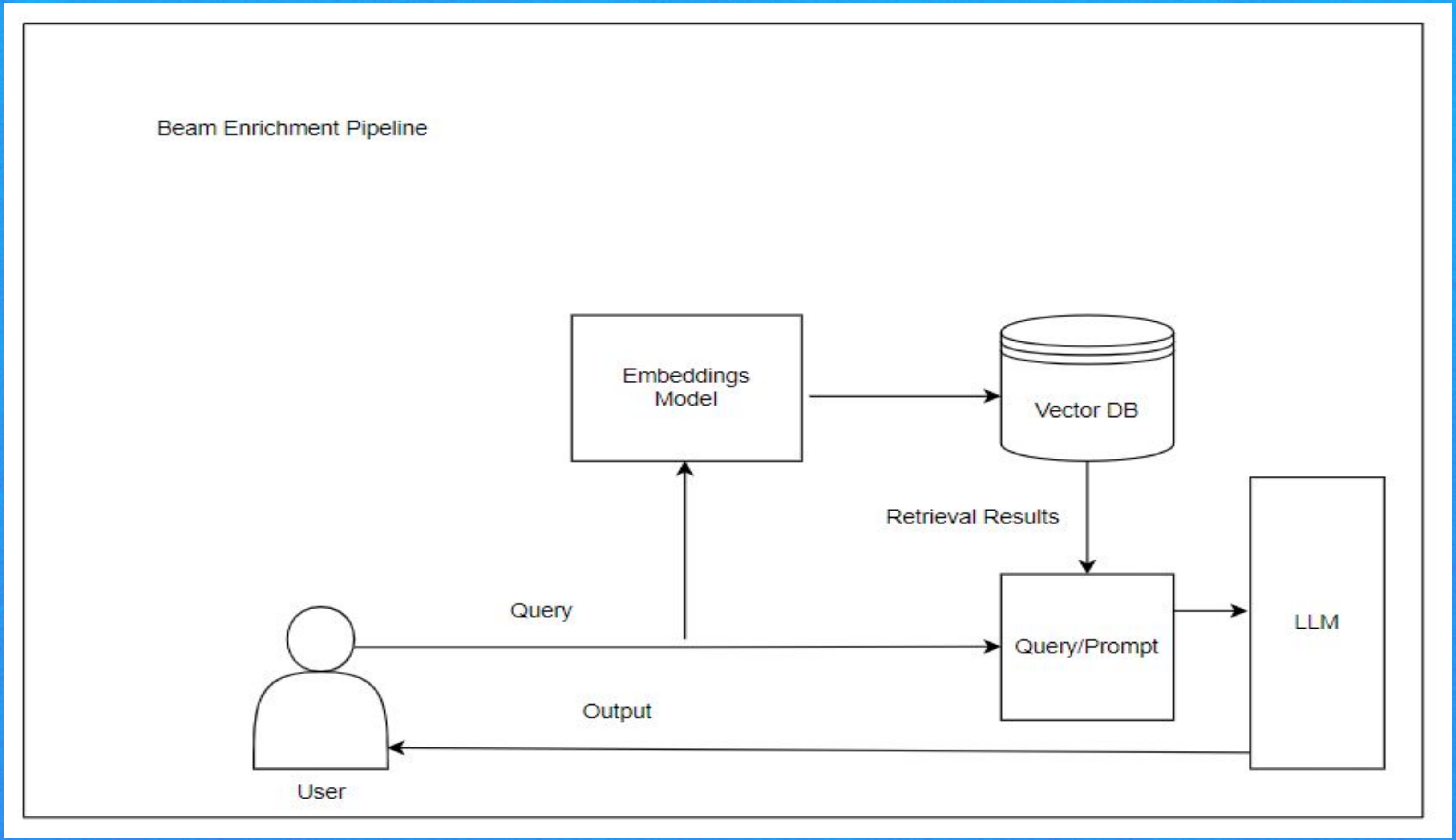
Python

Enrichment Details

- For this firstly we create an index in the vector database for searching of relevant text.
- Using beam's `EnrichmentSourceHandler` we create Vector DB queries for fetching relevant contexts.
- Perform vector search using the search query and return the relevant document chunks and its embeddings to enrich the user's questions before augmented generation.

```python
class RedisEnrichmentHandler(EnrichmentSourceHandler[beam.Row, beam.Row]):
  """A handler for :class:`apache_beam.transforms.enrichment.Enrichment`
  transform to interact with redis vector DB.

  Args:
    redis_host (str): Redis Host to connect to redis DB
    redis_port (int): Redis Port to connect to redis DB
    index_name (str): Index Name created for searching in Redis DB
    vector_field (str): vector field to compute similarity score in vector DB
    return_fields (list): returns list of similar text and its embeddings
    hybrid_fields (str): fields to be selected
    k (int): Value of K in KNN algorithm for searching in redis
  """

    def __init__(
            self,
            redis_host: str,
            redis_port: int,
            index_name: str = "embeddings-index",
            vector_field: str = "text_vector",
            return_fields: list = ["id", "title", "url", "text"],
            hybrid_fields: str = "*",
            k: int = 2,
    ):
        self.redis_host = redis_host
        self.redis_port = redis_port
        self.index_name = index_name
        self.vector_field = vector_field
        self.return_fields = return_fields
        self.hybrid_fields = hybrid_fields
        self.k = k
        self.client = None

    def __enter__(self):
        """connect to the redis DB using redis client."""
        self.client = redis.Redis(host=self.redis_host, port=self.redis_port)

    def __call__(self, request: beam.Row, *args, **kwargs):
        """
    Reads a row from the redis Vector DB and returns
    a `Tuple` of request and response.

    Args:
    request: the input `beam.Row` to enrich.
    """

        # read embedding vector for user query

        embedded_query = request[self.vector_field.strip('_vector')]

        # Prepare the Query
        base_query = f'{self.hybrid_fields}=>[KNN {self.k} @{self.vector_field} $vector AS vector_score]'
        query = (
            Query(base_query)
                .return_fields(*self.return_fields)
                .dialect(2)
        )

        params_dict = {"vector": np.array(embedded_query).astype(dtype=np.float32).tobytes()}

        # perform vector search
        results = self.client.ft(self.index_name).search(query, params_dict)

        return beam.Row(text=embedded_query), beam.Row(docs=results.docs)
```

Inference Pipeline


Beam Enrichment Pipeline
Embeddings Model
Vector DB
Retrieval Results
LLM
Query
Query/Prompt
Output
User

Conclusion

Conclusions

- RAG is fairly streamlined to be implemented with Beam's existing tools at scale for rapid prototyping and deployment.
- **Gaps Observed:**
  - MLTransform has been implemented in a way that it will transform data from 1 PCollection to another. This means we have to execute 1 insert and 1 update query instead of 1 insert query.
  - Less native Python support for vector DB queries meant writing the custom IO class or python clients for redis vector database and opensearch vector database.
- **Next Steps:**
  - Integrate custom IO Connectors into Beam foundation packages for reusability
  - Expand augmented generation step with Beam's inference APIs.

Thank you!

Questions?

Github: itsayushpandey

Linkedin : https://www.linkedin.com/in/itsayushpandey/

BEAM SUMMIT